eCosPro Reference Manual

eCosPro Reference Manual

Publication date 18 March 2024

About this reference manual

This reference manual is for eCos and eCosPro. It forms part of the eCosPro Developer's kit and includes documentation of the standard features of eCos as well documentation of eCosPro libraries and runtime features. Not all the eCosPro features or libraries documented in this reference manual may be available in the runtime code due to licensing restrictions. Some eCosPro features are subject to the eCosPro Evaluation License and require seperate licensing exclusively from eCosCentric when included in a product.

Documentation licensing terms

Open Publication License

The document containing or referencing this license was produced in full, or in part if the document contains multiple licensing references, from work that is subject to the terms and conditions set forth in the Open Publication License, v1.0 or later (the latest version is presently available at http://www.opencontent.org/openpub/).

Distribution of the work or derivative of the work in any standard (paper) book form is prohibited unless prior permission is obtained from the copyright holder(s).

eCosPro Non-Commercial Public License

The document containing or referencing these licenses was produced in full, or in part if the document contains multiple licensing references, from work that is subject to the terms and conditions of the eCosPro Non-Commercial Public License.

Distribution of the work or derivative of the work is permitted for Non-Commercial[*] use only.

[*] As defined by the eCosPro Non- Commercial Public License.

eCosPro License

The document containing or referencing these licenses was produced in full, or in part if the document contains multiple licensing references, from work that is subject to the terms and conditions of the eCosPro License.

Distribution of the work or derivative of the work is not permitted.

Apache 2.0 License

The document containing or referencing this license was produced in full, or in part if the document contains multiple licensing references, from work that is subject to the terms and conditions of the Apache 2.0 License.

Microchip “AS IS” License

The document containing or referencing these licenses was produced in full, or in part if the document contains multiple licensing references, from work that is subject to the terms and conditions of the Basic BSD-Style license that accompanies Microchip Software's cryptoauthlib.

See also Modified BSD “2 clause” and “3 clause” Licenses.

Trademarks

Altera® and Excalibur™ are trademarks of Altera Corporation.

AMD® is a registered trademark of Advanced Micro Devices, Inc.

ARM®, Cortex-M®, StrongARM®, Thumb®, ARM7™, ARM9™ are trademarks of Advanced RISC Machines, Ltd.

Apple®, Bonjour® and Safari® are registered trademarks of Apple Inc., registered in the U.S. and other countries.

Cirrus Logic® and Maverick™ are registered trademarks of Cirrus Logic, Inc.

Cogent™ is a trademark of Cogent Computer Systems, Inc.

Compaq® is a registered trademark of the Compaq Computer Corporation.

Debian® is registered trademark of Software in the Public Interest, Inc.

eCos®, eCosCentric® and eCosPro® are registered trademarks of eCosCentric Limited.

Fujitsu® is a registered trademark of Fujitsu Limited.

eCos and RedBoot Warranty

eCos and RedBoot are open source software, covered by a modified version of the GNU General Public Licence, and you are welcome to change it and/or distribute copies of it under certain conditions. See http://ecos.sourceware.org/license-overview.html for more information about the license.

eCos and RedBoot software have NO WARRANTY.

Because this software is licensed free of charge, there are no warranties for it, to the extent permitted by applicable law. Except when otherwise stated in writing, the copyright holders and/or other parties provide the software “as is” without warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The entire risk as to the quality and performance of the software is with you. Should the software prove defective, you assume the cost of all necessary servicing, repair or correction.

In no event, unless required by applicable law or agreed to in writing, will any copyright holder, or any other party who may modify and/or redistribute the program as permitted above, be liable to you for damages, including any general, special, incidental or consequential damages arising out of the use or inability to use the program (including but not limited to loss of data or data being rendered inaccurate or losses sustained by you or third parties or a failure of the program to operate with any other programs), even if such holder or other party has been advised of the possibility of such damages.

eCosPro Warranty

While eCosPro is open source software, it is covered by a commercial license the most recent version of which can be found at http://www.ecoscentric.com/licensing/ecospro-license.shtml.

This software has no warranties associated with it, other than Intellectual Property Rights as stated in section 5 of the eCosPro license agreement. Distribution of eCosPro sources licensed under the eCosPro license in any form is strictly prohibited unless expressly permitted by the copyright holder.

Other copyrights

Documentation on the lwIP TCP/IP stack includes portions derived from documentation distributed with the following license:

Table of Contents

List of Figures

List of Tables

List of Examples

Part I. The eCos Kernel

Name

Kernel — Overview of the eCos Kernel

Description

The kernel is one of the key packages in all of eCos. It provides the core functionality needed for developing multi-threaded applications:

1. The ability to create new threads in the system, either during startup or when the system is already running.
2. Control over the various threads in the system, for example manipulating their priorities.
3. A choice of schedulers, determining which thread should currently be running.
4. A range of synchronization primitives, allowing threads to interact and share data safely.
5. Integration with the system's support for interrupts and exceptions.

In some other operating systems the kernel provides additional functionality. For example the kernel may also provide memory allocation functionality, and device drivers may be part of the kernel as well. This is not the case for eCos. Memory allocation is handled by a separate package. Similarly each device driver will typically be a separate package. Various packages are combined and configured using the eCos configuration technology to meet the requirements of the application.

The eCos kernel package is optional. It is possible to write single-threaded applications which do not use any kernel functionality, for example RedBoot. Typically such applications are based around a central polling loop, continually checking all devices and taking appropriate action when I/O occurs. A small amount of calculation is possible every iteration, at the cost of an increased delay between an I/O event occurring and the polling loop detecting the event. When the requirements are straightforward it may well be easier to develop the application using a polling loop, avoiding the complexities of multiple threads and synchronization between threads. As requirements get more complicated a multi-threaded solution becomes more appropriate, requiring the use of the kernel. In fact some of the more advanced packages in eCos, for example the TCP/IP stack, use multi-threading internally. Therefore if the application uses any of those packages then the kernel becomes a required package, not an optional one.

The kernel functionality can be used in one of two ways. The kernel provides its own C API, with functions like `cyg_thread_create` and `cyg_mutex_lock`. These can be called directly from application code or from other packages. Alternatively there are a number of packages which provide compatibility with existing API's, for example POSIX threads or µITRON. These allow application code to call standard functions such as `pthread_create`, and those functions are implemented using the basic functionality provided by the eCos kernel. Using compatibility packages in an eCos application can make it much easier to reuse code developed in other environments, and to share code.

Although the different compatibility packages have similar requirements on the underlying kernel, for example the ability to create a new thread, there are differences in the exact semantics. For example, strict µITRON compliance requires that kernel timeslicing is disabled. This is achieved largely through the configuration technology. The kernel provides a number of configuration options that control the exact semantics that are provided, and the various compatibility packages require particular settings for those options. This has two important consequences. First, it is not usually possible to have two different compatibility packages in one eCos configuration because they will have conflicting requirements on the underlying kernel. Second, the semantics of the kernel's own API are only loosely defined because of the many configuration options. For example `cyg_mutex_lock` will always attempt to lock a mutex, but various configuration options determine the behaviour when the mutex is already locked and there is a possibility of priority inversion.

The optional nature of the kernel package presents some complications for other code, especially device drivers. Wherever possible a device driver should work whether or not the kernel is present. However there are some parts of the system, especially those related to interrupt handling, which should be implemented differently in multi-threaded environments containing the eCos kernel and in single-threaded environments without the kernel. To cope with both scenarios the common HAL package provides a driver API, with functions such as `cyg_drv_interrupt_attach`. When the kernel package is present these driver API functions map directly on to the equivalent kernel functions such as `cyg_interrupt_attach`, using macros to avoid any overheads. When the kernel is absent the common HAL package implements the driver API directly, but this implementation is simpler than the one in the kernel because it can assume a single-threaded environment.

Schedulers

When a system involves multiple threads, a scheduler is needed to determine which thread should currently be running. The eCos kernel can be configured with one of three schedulers, the bitmap scheduler, the multi-level queue (MLQ) scheduler and the SMP scheduler (MLQSMP). The bitmap scheduler is somewhat more efficient, but has a number of limitations. Most systems will instead use the MLQ scheduler, and MLQSMP in SMP configurations. Other schedulers may be added in the future, either as extensions to the kernel package or in separate packages.

Both the bitmap and the MLQ schedulers use a simple numerical priority to determine which thread should be running. The number of priority levels is configurable via the option `CYGNUM_KERNEL_SCHED_PRIORITIES`, but a typical system will have up to 32 priority levels. Therefore thread priorities will be in the range 0 to 31, with 0 being the highest priority and 31 the lowest. Usually only the system's idle thread will run at the lowest priority. Thread priorities are absolute, so the kernel will only run a lower-priority thread if all higher-priority threads are currently blocked.

The bitmap scheduler only allows one thread per priority level, so if the system is configured with 32 priority levels then it is limited to only 32 threads — still enough for many applications. A simple bitmap can be used to keep track of which threads are currently runnable. Bitmaps can also be used to keep track of threads waiting on a mutex or other synchronization primitive. Identifying the highest-priority runnable or waiting thread involves a simple operation on the bitmap, and an array index operation can then be used to get hold of the thread data structure itself. This makes the bitmap scheduler fast and totally deterministic.

The MLQ schedulers (MLQ and MLQSMP) allows multiple threads to run at the same priority. This means that there is no limit on the number of threads in the system, other than the amount of memory available. However operations such as finding the highest priority runnable thread are a little bit more expensive than for the bitmap scheduler.

Optionally the MLQ schedulers support timeslicing, where the scheduler automatically switches from one runnable thread to another when some number of clock ticks have occurred. Timeslicing only comes into play when there are two runnable threads at the same priority and no higher priority runnable threads. If timeslicing is disabled then a thread will not be preempted by another thread of the same priority, and will continue running until either it explicitly yields the processor or until it blocks by, for example, waiting on a synchronization primitive. The configuration options `CYGSEM_KERNEL_SCHED_TIMESLICE` and `CYGNUM_KERNEL_SCHED_TIMESLICE_TICKS` control timeslicing. The bitmap scheduler does not provide timeslicing support. It only allows one thread per priority level, so it is not possible to preempt the current thread in favour of another one with the same priority.

An experimental timeslicing feature is also available in eCosPro, which is "fair" timeslicing, and can be enabled with the `CYGSEM_KERNEL_SCHED_TIMESLICE_FAIR` configuration option. By default, normal timeslicing does not guarantee that different threads get a similar amount of CPU time. In fact, a thread could use most of one of its timeslice ticks allocated to it, but then block shortly before the tick occurs, at which point as far as the kernel is concerned, it effectively used no time at all from that tick. In some applications where some threads may often be operating on small parts of work that take less than a tick, or where there is a periodic event such as an external interrupt that regularly causes a thread to be pre-empted and this can occasionally happen roughly synchronised to the kernel clock, then this can result in other threads at the same priority being starved. The fair timeslicing option seeks to prevent this by using information from the underlying HAL clock to determine a more accurate view of how much CPU time a thread has used. This is naturally at the expense of a slightly greater context switch time. With this option enabled, threads should become more fairly timesliced, although due to the granularity of the kernel clock, there will always be a small error margin of roughly half a kernel clock tick on average. This feature can be tested with the `timeslice_fair` kernel test.

Another important configuration option that affects the MLQ schedulers is `CYGIMP_KERNEL_SCHED_SORTED_QUEUES`. This determines what happens when a thread blocks, for example by waiting on a semaphore which has no pending events. The default behaviour of the system is last-in-first-out queuing. For example if several threads are waiting on a semaphore and an event is posted, the thread that gets woken up is the last one that called `cyg_semaphore_wait`. This allows for a simple and fast implementation of both the queue and dequeue operations. However if there are several queued threads with different priorities, it may not be the highest priority one that gets woken up. In practice this is rarely a problem: usually there will be at most one thread waiting on a queue, or when there are several threads they will be of the same priority. However if the application does require strict priority queueing then the option `CYGIMP_KERNEL_SCHED_SORTED_QUEUES` should be enabled. There are disadvantages: more work is needed whenever a thread is queued, and the scheduler needs to be locked for this operation so the system's dispatch latency is worse. If the bitmap scheduler is used then priority queueing is automatic and does not involve any penalties.

Some kernel functionality is currently only supported with the MLQ schedulers, not the bitmap scheduler. This includes support for SMP systems, and protection against priority inversion using either mutex priority ceilings or priority inheritance.

The MLQSMP scheduler is a derivative of the MLQ scheduler that has some additional features for controlling thread affinity and CPU activation. The MLQ scheduler can support SMP operation, but does not support the additional features. By default, eCosPro uses the MLQSMP scheduler when configured for SMP operation.

Synchronization Primitives

The eCos kernel provides a number of different synchronization primitives: mutexes, condition variables, counting semaphores, mail boxes and event flags.

Mutexes serve a very different purpose from the other primitives. A mutex allows multiple threads to share a resource safely: a thread locks a mutex, manipulates the shared resource, and then unlocks the mutex again. The other primitives are used to communicate information between threads, or alternatively from a DSR associated with an interrupt handler to a thread.

When a thread that has locked a mutex needs to wait for some condition to become true, it should use a condition variable. A condition variable is essentially just a place for a thread to wait, and which another thread, or DSR, can use to wake it up. When a thread waits on a condition variable it releases the mutex before waiting, and when it wakes up it reacquires it before proceeding. These operations are atomic so that synchronization race conditions cannot be introduced.

A counting semaphore is used to indicate that a particular event has occurred. A consumer thread can wait for this event to occur, and a producer thread or a DSR can post the event. There is a count associated with the semaphore so if the event occurs multiple times in quick succession this information is not lost, and the appropriate number of semaphore wait operations will succeed.

Mail boxes are also used to indicate that a particular event has occurred, and allows for one item of data to be exchanged per event. Typically this item of data would be a pointer to some data structure. Because of the need to store this extra data, mail boxes have a finite capacity. If a producer thread generates mail box events faster than they can be consumed then, to avoid overflow, it will be blocked until space is again available in the mail box. This means that mail boxes usually cannot be used by a DSR to wake up a thread. Instead mail boxes are typically only used between threads.

Event flags can be used to wait on some number of different events, and to signal that one or several of these events have occurred. This is achieved by associating bits in a bit mask with the different events. Unlike a counting semaphore no attempt is made to keep track of the number of events that have occurred, only the fact that an event has occurred at least once. Unlike a mail box it is not possible to send additional data with the event, but this does mean that there is no possibility of an overflow and hence event flags can be used between a DSR and a thread as well as between threads.

The eCos common HAL package provides its own device driver API which contains some of the above synchronization primitives. These allow the DSR for an interrupt handler to signal events to higher-level code. If the configuration includes the eCos kernel package then the driver API routines map directly on to the equivalent kernel routines, allowing interrupt handlers to interact with threads. If the kernel package is not included and the application consists of just a single thread running in polled mode then the driver API is implemented entirely within the common HAL, and with no need to worry about multiple threads the implementation can obviously be rather simpler.

Threads and Interrupt Handling

During normal operation the processor will be running one of the threads in the system. This may be an application thread, a system thread running inside say the TCP/IP stack, or the idle thread. From time to time a hardware interrupt will occur, causing control to be transferred briefly to an interrupt handler. When the interrupt has been completed the system's scheduler will decide whether to return control to the interrupted thread or to some other runnable thread.

Threads and interrupt handlers must be able to interact. If a thread is waiting for some I/O operation to complete, the interrupt handler associated with that I/O must be able to inform the thread that the operation has completed. This can be achieved in a number of ways. One very simple approach is for the interrupt handler to set a volatile variable. A thread can then poll continuously until this flag is set, possibly sleeping for a clock tick in between. Polling continuously means that the CPU time is not available for other activities, which may be acceptable for some but not all applications. Polling once every clock tick imposes much less overhead, but means that the thread may not detect that the I/O event has occurred until an entire clock tick has elapsed. In typical systems this could be as long as 10 milliseconds. Such a delay might be acceptable for some applications, but not all.

A better solution would be to use one of the synchronization primitives. The interrupt handler could signal a condition variable, post to a semaphore, or use one of the other primitives. The thread would perform a wait operation on the same primitive. It would not consume any CPU cycles until the I/O event had occurred, and when the event does occur the thread can start running again immediately (subject to any higher priority threads that might also be runnable).

Synchronization primitives constitute shared data, so care must be taken to avoid problems with concurrent access. If the thread that was interrupted was just performing some calculations then the interrupt handler could manipulate the synchronization primitive quite safely. However if the interrupted thread happened to be inside some kernel call then there is a real possibility that some kernel data structure will be corrupted.

One way of avoiding such problems would be for the kernel functions to disable interrupts when executing any critical region. On most architectures this would be simple to implement and very fast, but it would mean that interrupts would be disabled often and for quite a long time. For some applications that might not matter, but many embedded applications require that the interrupt handler run as soon as possible after the hardware interrupt has occurred. If the kernel relied on disabling interrupts then it would not be able to support such applications.

Instead the kernel uses a two-level approach to interrupt handling. Associated with every interrupt vector is an Interrupt Service Routine or ISR, which will run as quickly as possible so that it can service the hardware. However an ISR can make only a small number of kernel calls, mostly related to the interrupt subsystem, and it cannot make any call that would cause a thread to wake up. If an ISR detects that an I/O operation has completed and hence that a thread should be woken up, it can cause the associated Deferred Service Routine or DSR to run. A DSR is allowed to make more kernel calls, for example it can signal a condition variable or post to a semaphore.

Disabling interrupts prevents ISRs from running, but very few parts of the system disable interrupts and then only for short periods of time. The main reason for a thread to disable interrupts is to manipulate some state that is shared with an ISR. For example if a thread needs to add another buffer to a linked list of free buffers and the ISR may remove a buffer from this list at any time, the thread would need to disable interrupts for the few instructions needed to manipulate the list. If the hardware raises an interrupt at this time, it remains pending until interrupts are reenabled.

Analogous to interrupts being disabled or enabled, the kernel has a scheduler lock. The various kernel functions such as `cyg_mutex_lock` and `cyg_semaphore_post` will claim the scheduler lock, manipulate the kernel data structures, and then release the scheduler lock. If an interrupt results in a DSR being requested and the scheduler is currently locked, the DSR remains pending. When the scheduler lock is released any pending DSRs will run. These may post events to synchronization primitives, causing other higher priority threads to be woken up.

For an example, consider the following scenario. The system has a high priority thread A, responsible for processing some data coming from an external device. This device will raise an interrupt when data is available. There are two other threads B and C which spend their time performing calculations and occasionally writing results to a display of some sort. This display is a shared resource so a mutex is used to control access.

At a particular moment in time thread A is likely to be blocked, waiting on a semaphore or another synchronization primitive until data is available. Thread B might be running performing some calculations, and thread C is runnable waiting for its next timeslice. Interrupts are enabled, and the scheduler is unlocked because none of the threads are in the middle of a kernel operation. At this point the device raises an interrupt. The hardware transfers control to a low-level interrupt handler provided by eCos which works out exactly which interrupt occurs, and then the corresponding ISR is run. This ISR manipulates the hardware as appropriate, determines that there is now data available, and wants to wake up thread A by posting to the semaphore. However ISR's are not allowed to call `cyg_semaphore_post` directly, so instead the ISR requests that its associated DSR be run and returns. There are no more interrupts to be processed, so the kernel next checks for DSR's. One DSR is pending and the scheduler is currently unlocked, so the DSR can run immediately and post the semaphore. This will have the effect of making thread A runnable again, so the scheduler's data structures are adjusted accordingly. When the DSR returns thread B is no longer the highest priority runnable thread so it will be suspended, and instead thread A gains control over the CPU.

In the above example no kernel data structures were being manipulated at the exact moment that the interrupt happened. However that cannot be assumed. Suppose that thread B had finished its current set of calculations and wanted to write the results to the display. It would claim the appropriate mutex and manipulate the display. Now suppose that thread B was timesliced in favour of thread C, and that thread C also finished its calculations and wanted to write the results to the display. It would call `cyg_mutex_lock`. This kernel call locks the scheduler, examines the current state of the mutex, discovers that the mutex is already owned by another thread, suspends the current thread, and switches control to another runnable thread. Another interrupt happens in the middle of this `cyg_mutex_lock` call, causing the ISR to run immediately. The ISR decides that thread A should be woken up so it requests that its DSR be run and returns back to the kernel. At this point there is a pending DSR, but the scheduler is still locked by the call to `cyg_mutex_lock` so the DSR cannot run immediately. Instead the call to `cyg_mutex_lock` is allowed to continue, which at some point involves unlocking the scheduler. The pending DSR can now run, safely post the semaphore, and thus wake up thread A.

If the ISR had called `cyg_semaphore_post` directly rather than leaving it to a DSR, it is likely that there would have been some sort of corruption of a kernel data structure. For example the kernel might have completely lost track of one of the threads, and that thread would never have run again. The two-level approach to interrupt handling, ISR's and DSR's, prevents such problems with no need to disable interrupts.

Calling Contexts

eCos defines a number of contexts. Only certain calls are allowed from inside each context, for example most operations on threads or synchronization primitives are not allowed from ISR context. The different contexts are initialization, thread, ISR and DSR.

When eCos starts up it goes through a number of phases, including setting up the hardware and invoking C++ static constructors. During this time interrupts are disabled and the scheduler is locked. When a configuration includes the kernel package the final operation is a call to `cyg_scheduler_start`. At this point interrupts are enabled, the scheduler is unlocked, and control is transferred to the highest priority runnable thread. If the configuration also includes the C library package then usually the C library startup package will have created a thread which will call the application's `main` entry point.

Some application code can also run before the scheduler is started, and this code runs in initialization context. If the application is written partly or completely in C++ then the constructors for any static objects will be run. Alternatively application code can define a function `cyg_user_start` which gets called after any C++ static constructors. This allows applications to be written entirely in C.

```
void
cyg_user_start(void)
{
    /* Perform application-specific initialization here */
}
```

It is not necessary for applications to provide a `cyg_user_start` function since the system will provide a default implementation which does nothing.

Typical operations that are performed from inside static constructors or `cyg_user_start` include creating threads, synchronization primitives, setting up alarms, and registering application-specific interrupt handlers. In fact for many applications all such creation operations happen at this time, using statically allocated data, avoiding any need for dynamic memory allocation or other overheads.

Code running in initialization context runs with interrupts disabled and the scheduler locked. It is not permitted to reenable interrupts or unlock the scheduler because the system is not guaranteed to be in a totally consistent state at this point. A consequence is that initialization code cannot use synchronization primitives such as `cyg_semaphore_wait` to wait for an external event. It is permitted to lock and unlock a mutex: there are no other threads running so it is guaranteed that the mutex is not yet locked, and therefore the lock operation will never block; this is useful when making library calls that may use a mutex internally.

At the end of the startup sequence the system will call `cyg_scheduler_start` and the various threads will start running. In thread context nearly all of the kernel functions are available. There may be some restrictions on interrupt-related operations, depending on the target hardware. For example the hardware may require that interrupts be acknowledged in the ISR or DSR before control returns to thread context, in which case `cyg_interrupt_acknowledge` should not be called by a thread.

At any time the processor may receive an external interrupt, causing control to be transferred from the current thread. Typically a VSR provided by eCos will run and determine exactly which interrupt occurred. Then the VSR will switch to the appropriate ISR, which can be provided by a HAL package, a device driver, or by the application. During this time the system is running at ISR context, and most of the kernel function calls are disallowed. This includes the various synchronization primitives, so for example an ISR is not allowed to post to a semaphore to indicate that an event has happened. Usually the only operations that should be performed from inside an ISR are ones related to the interrupt subsystem itself, for example masking an interrupt or acknowledging that an interrupt has been processed. On SMP systems it is also possible to use spinlocks from ISR context.

When an ISR returns it can request that the corresponding DSR be run as soon as it is safe to do so, and that will run in DSR context. This context is also used for running alarm functions, and threads can switch temporarily to DSR context by locking the scheduler. Only certain kernel functions can be called from DSR context, although more than in ISR context. In particular it is possible to use any synchronization primitives which cannot block. These include `cyg_semaphore_post`, `cyg_cond_signal`, `cyg_cond_broadcast`, `cyg_flag_setbits`, and `cyg_mbox_tryput`. It is not possible to use any primitives that may block such as `cyg_semaphore_wait`, `cyg_mutex_lock`, or `cyg_mbox_put`. Calling such functions from inside a DSR may cause the system to hang.

The specific documentation for the various kernel functions gives more details about valid contexts.

Error Handling and Assertions

In many APIs each function is expected to perform some validation of its parameters and possibly of the current state of the system. This is supposed to ensure that each function is used correctly, and that application code is not attempting to perform a semaphore operation on a mutex or anything like that. If an error is detected then a suitable error code is returned, for example the POSIX function `pthread_mutex_lock` can return various error codes including `EINVAL` and `EDEADLK`. There are a number of problems with this approach, especially in the context of deeply embedded systems:

1. Performing these checks inside the mutex lock and all the other functions requires extra CPU cycles and adds significantly to the code size. Even if the application is written correctly and only makes system function calls with sensible arguments and under the right conditions, these overheads still exist.

2. Returning an error code is only useful if the calling code detects these error codes and takes appropriate action. In practice the calling code will often ignore any errors because the programmer *“knows”* that the function is being used correctly. If the programmer is mistaken then an error condition may be detected and reported, but the application continues running anyway and is likely to fail some time later in mysterious ways.

3. If the calling code does always check for error codes, that adds yet more CPU cycles and code size overhead.

4. Usually there will be no way to recover from certain errors, so if the application code detected an error such as `EINVAL` then all it could do is abort the application somehow.

The approach taken within the eCos kernel is different. Functions such as `cyg_mutex_lock` will not return an error code. Instead they contain various assertions, which can be enabled or disabled. During the development process assertions are normally left enabled, and the various kernel functions will perform parameter checks and other system consistency checks. If a problem is detected then an assertion failure will be reported and the application will be terminated. In a typical debug session a suitable breakpoint will have been installed and the developer can now examine the state of the system and work out exactly what is going on. Towards the end of the development cycle assertions will be disabled by manipulating configuration options within the eCos infrastructure package, and all assertions will be eliminated at compile-time. The assumption is that by this time the application code has been mostly debugged: the initial version of the code might have tried to perform a semaphore operation on a mutex, but any problems like that will have been fixed some time ago. This approach has a number of advantages:

1. In the final application there will be no overheads for checking parameters and other conditions. All that code will have been eliminated at compile-time.

2. Because the final application will not suffer any overheads, it is reasonable for the system to do more work during the development process. In particular the various assertions can test for more error conditions and more complicated errors. When an error is detected it is possible to give a text message describing the error rather than just return an error code.

3. There is no need for application programmers to handle error codes returned by various kernel function calls. This simplifies the application code.

4. If an error is detected then an assertion failure will be reported immediately and the application will be halted. There is no possibility of an error condition being ignored because application code did not check for an error code.

Although none of the kernel functions return an error code, many of them do return a status condition. For example the function `cyg_semaphore_timed_wait` waits until either an event has been posted to a semaphore, or until a certain number of clock ticks have occurred. Usually the calling code will need to know whether the wait operation succeeded or whether a timeout occurred. `cyg_semaphore_timed_wait` returns a boolean: a return value of zero or false indicates a timeout, a non-zero return value indicates that the wait succeeded.

In conventional APIs one common error conditions is lack of memory. For example the POSIX function `pthread_create` usually has to allocate some memory dynamically for the thread stack and other per-thread data. If the target hardware does not have enough memory to meet all demands, or more commonly if the application contains a memory leak, then there may not be enough memory available and the function call would fail. The eCos kernel avoids such problems by never performing any dynamic memory allocation. Instead it is the responsibility of the application code to provide all the memory required for kernel data structures and other needs. In the case of `cyg_thread_create` this means a cyg_thread data structure to hold the thread details, and a char array for the thread stack.

In many applications this approach results in all data structures being allocated statically rather than dynamically. This has several advantages. If the application is in fact too large for the target hardware's memory then there will be an error at link-time rather than at run-time, making the problem much easier to diagnose. Static allocation does not involve any of the usual overheads associated with dynamic allocation, for example there is no need to keep track of the various free blocks in the system, and it may be possible to eliminate `malloc` from the system completely. Problems such as fragmentation and memory leaks cannot occur if all data is allocated statically. However, some applications are sufficiently complicated that dynamic memory allocation is required, and the various kernel functions do not distinguish between statically and dynamically allocated memory. It still remains the responsibility of the calling code to ensure that sufficient memory is available, and passing null pointers to the kernel will result in assertions or system failure.

Name

SMP — Support Symmetric Multiprocessing Systems

Description

eCos contains support for limited Symmetric Multi-Processing (SMP). This is only available on selected architectures and platforms. The implementation has a number of restrictions on the kind of hardware supported. These are described in the section called "SMP Support".

The aim for eCos SMP is to support embedded and real time applications on the class of hardware that is the likely target. This means being able to allocate threads to specific CPUs and manage the CPUs that are active. eCos does not support the kind of load balancing scheduler epitomized by the Linux Fair Scheduler, which is oriented to running massively parallel servers. Instead eCos allows deliberately unbalanced scheduling to improve real time latency.

The following sections describe the changes that have been made to the eCos kernel to support SMP operation.

System Startup

The system startup sequence needs to be somewhat different on an SMP system, although this is largely transparent to application code. The main startup takes place on only one CPU, called the primary CPU. All other CPUs, the secondary CPUs, are either placed in suspended state at reset, or are captured by the HAL and put into a spin as they start up. The primary CPU is responsible for copying the DATA segment and zeroing the BSS (if required), calling HAL variant and platform initialization routines and invoking constructors. It then calls `cyg_start` to enter the application. The application may then create extra threads and other objects.

It is only when the application calls `cyg_scheduler_start` that the secondary CPUs are initialized. This routine scans the list of available secondary CPUs and invokes `HAL_SMP_CPU_START` to start each CPU. Finally it calls an internal function `Cyg_Scheduler::start_cpu` to enter the scheduler for the primary CPU.

Each secondary CPU starts in the HAL, where it completes any per-CPU initialization before calling into the kernel at `cyg_kernel_cpu_startup`. Here it claims the scheduler lock and calls `Cyg_Scheduler::start_cpu`.

`Cyg_Scheduler::start_cpu` is common to both the primary and secondary CPUs. The first thing this code does is to install an interrupt object for this CPU's inter-CPU interrupt. From this point on the code is the same as for the single CPU case: an initial thread is chosen and entered.

From this point on the CPUs are all equal, eCos makes no further distinction between the primary and secondary CPUs. However, the hardware may still distinguish between them as far as interrupt delivery is concerned.

Scheduling

To function correctly an operating system kernel must protect its vital data structures, such as the run queues, from concurrent access. In a single CPU system the only concurrent activities to worry about are asynchronous interrupts. The kernel can easily guard its data structures against these by disabling interrupts. However, in a multi-CPU system, this is inadequate since it does not block access by other CPUs.

The eCos kernel protects its vital data structures using the scheduler lock. In single CPU systems this is a simple counter that is atomically incremented to acquire the lock and decremented to release it. If the lock is decremented to zero then the scheduler may be invoked to choose a different thread to run. Because interrupts may continue to be serviced while the scheduler lock is claimed, ISRs are not allowed to access kernel data structures, or call kernel routines that can. Instead all such operations are deferred to an associated DSR routine that is run during the lock release operation, when the data structures are in a consistent state.

By choosing a kernel locking mechanism that does not rely on interrupt manipulation to protect data structures, it is easier to convert eCos to SMP than would otherwise be the case. The principal change needed to make eCos SMP-safe is to convert the scheduler lock into a nestable spin lock. This is done by adding a spinlock and a CPU id to the original counter.

The algorithm for acquiring the scheduler lock is very simple. If the scheduler lock's CPU id matches the current CPU then it can just increment the counter and continue. If it does not match, the CPU must spin on the spinlock, after which it may increment the counter and store its own identity in the CPU id.

To release the lock, the counter is decremented. If it goes to zero the CPU id value must be set to NONE and the spinlock cleared.

To protect these sequences against interrupts, they must be performed with interrupts disabled. However, since these are very short code sequences, they will not have an adverse effect on the interrupt latency.

Beyond converting the scheduler lock, further preparing the kernel for SMP is a relatively minor matter. The main changes are to convert various scalar housekeeping variables into arrays indexed by CPU id. These include the current thread pointer, the need_reschedule flag and the timeslice counter.

At present only the Multi-Level Queue (MLQ) schedulers are capable of supporting SMP configurations. The main change made to this scheduler is to cope with having several threads in execution at the same time. Running threads are marked with the CPU that they are executing on. When scheduling a thread, the scheduler skips past any running threads until it finds a thread that is pending. While not a constant-time algorithm, as in the single CPU case, this is still deterministic, since the worst case time is bounded by the number of CPUs in the system.

A second change to the scheduler is in the code used to decide when the scheduler should be called to choose a new thread. The scheduler attempts to keep the n CPUs running the n highest priority threads. Since an event or interrupt on one CPU may require a reschedule on another CPU, there must be a mechanism for deciding this. The algorithm currently implemented is very simple. Given a thread that has just been awakened (or had its priority changed), the scheduler scans the CPUs, starting with the one it is currently running on, for a current thread that is of lower priority than the new one. If one is found then a reschedule interrupt is sent to that CPU and the scan continues, but now using the current thread of the rescheduled CPU as the candidate thread. In this way the new thread gets to run as quickly as possible, hopefully on the current CPU, and the remaining CPUs will pick up the remaining highest priority threads as a consequence of processing the reschedule interrupt.

The final change to the scheduler is in the handling of timeslicing. Only one CPU receives timer interrupts, although all CPUs must handle timeslicing. To make this work, the CPU that receives the timer interrupt decrements the timeslice counter for all CPUs, not just its own. If the counter for a CPU reaches zero, then it sends a timeslice interrupt to that CPU. On receiving the interrupt the destination CPU enters the scheduler and looks for another thread at the same priority to run. This is somewhat more efficient than distributing clock ticks to all CPUs, since the interrupt is only needed when a timeslice occurs.

In addition to the standard MLQ scheduler, eCosPro also contains an MLQSMP scheduler. This is a derivative of the MLQ scheduler that has some additional features. The main change is to implement a CPU affinity mechanism. This is implemented by adding a CPU affinity map to each thread, indicating which CPUs this thread is allowed to run on. When choosing which thread to run a CPU will only look for threads that have its bit set in their affinity maps. In the future this scheduler will be extended with support for CPU activation and deactivation. By default, eCosPro uses the MLQSMP scheduler when configured for SMP operation.

All existing synchronization mechanisms work as before in an SMP system. Additional synchronization mechanisms have been added to provide explicit synchronization for SMP, in the form of spinlocks.

New functions have also been added to support CPU affinity.

SMP Interrupt Handling

The main area where the SMP nature of a system requires special attention is in device drivers and especially interrupt handling. It is quite possible for the ISR, DSR and thread components of a device driver to execute on different CPUs. For this reason it is much more important that SMP-capable device drivers use the interrupt-related functions correctly. Typically a device driver would use the driver API rather than call the kernel directly, but it is unlikely that anybody would attempt to use a multiprocessor system without the kernel package.

Two new functions have been added to the Kernel API to do interrupt routing: `cyg_interrupt_set_cpu` and `cyg_interrupt_get_cpu`. Once a vector has been routed to a new CPU, all other interrupt masking and configuration operations are relative to that CPU, where relevant.

There are more details of how interrupts should be handled in SMP systems in the section called “SMP Support”.

Name

cyg_thread_create — Create a new thread

Synopsis

#include <cyg/kernel/kapi.h>

`void` **`cyg_thread_create`** `(`*`sched_info`*`,` *`entry`*`,` *`entry_data`*`,` *`name`*`,` *`stack_base`*`,` *`stack_size`*`,` *`handle`*`,` *`thread`*`);`

Description

The `cyg_thread_create` function allows application code and eCos packages to create new threads. In many applications this only happens during system initialization and all required data is allocated statically. However additional threads can be created at any time, if necessary. A newly created thread is always in suspended state and will not start running until it has been resumed via a call to `cyg_thread_resume`. Also, if threads are created during system initialization then they will not start running until the eCos scheduler has been started.

The *`name`* argument is used primarily for debugging purposes, making it easier to keep track of which cyg_thread structure is associated with which application-level thread. The kernel configuration option `CYGVAR_KERNEL_THREADS_NAME` controls whether or not this name is actually used.

On creation each thread is assigned a unique handle, and this will be stored in the location pointed at by the *`handle`* argument. Subsequent operations on this thread including the required `cyg_thread_resume` should use this handle to identify the thread.

The kernel requires a small amount of space for each thread, in the form of a cyg_thread data structure, to hold information such as the current state of that thread. To avoid any need for dynamic memory allocation within the kernel this space has to be provided by higher-level code, typically in the form of a static variable. The *`thread`* argument provides this space.

Thread Entry Point

The entry point for a thread takes the form:

```
void thread_entry_function(cyg_addrword_t data)
{
    …
}
```

The second argument to `cyg_thread_create` is a pointer to such a function. The third argument *`entry_data`* is used to pass additional data to the function. Typically this takes the form of a pointer to some static data, or a small integer, or `0` if the thread does not require any additional data.

If the thread entry function ever returns then this is equivalent to the thread calling `cyg_thread_exit`. Even though the thread will no longer run again, it remains registered with the scheduler. If the application needs to re-use the cyg_thread data structure then a call to `cyg_thread_delete` is required first.

Thread Priorities

The *`sched_info`* argument provides additional information to the scheduler. The exact details depend on the scheduler being used. For the bitmap and mlqueue schedulers it is a small integer, typically in the range 0 to 31, with 0 being the highest priority. The lowest priority is normally used only by the system's idle thread. The exact number of priorities is controlled by the kernel configuration option `CYGNUM_KERNEL_SCHED_PRIORITIES`.

It is the responsibility of the application developer to be aware of the various threads in the system, including those created by eCos packages, and to ensure that all threads run at suitable priorities. For threads created by other packages the documentation provided by those packages should indicate any requirements.

The functions `cyg_thread_set_priority`, `cyg_thread_get_priority`, and `cyg_thread_get_current_priority` can be used to manipulate a thread's priority.

Stacks and Stack Sizes

Each thread needs its own stack for local variables and to keep track of function calls and returns. Again it is expected that this stack is provided by the calling code, usually in the form of static data, so that the kernel does not need any dynamic memory allocation facilities. `cyg_thread_create` takes two arguments related to the stack, a pointer to the base of the stack and the total size of this stack. On many processors stacks actually descend from the top down, so the kernel will add the stack size to the base address to determine the starting location.

The exact stack size requirements for any given thread depend on a number of factors. The most important is of course the code that will be executed in the context of this code: if this involves significant nesting of function calls, recursion, or large local arrays, then the stack size needs to be set to a suitably high value. There are some architectural issues, for example the number of CPU registers and the calling conventions will have some effect on stack usage. Also, depending on the configuration, it is possible that some other code such as interrupt handlers will occasionally run on the current thread's stack. This depends in part on configuration options such as `CYGIMP_HAL_COMMON_INTERRUPTS_USE_INTERRUPT_STACK` and `CYGSEM_HAL_COMMON_INTERRUPTS_ALLOW_NESTING`.

Determining an application's actual stack size requirements is the responsibility of the application developer, since the kernel cannot know in advance what code a given thread will run. However, the system does provide some hints about reasonable stack sizes in the form of two constants: `CYGNUM_HAL_STACK_SIZE_MINIMUM` and `CYGNUM_HAL_STACK_SIZE_TYPICAL`. These are defined by the appropriate HAL package. The `MINIMUM` value is appropriate for a thread that just runs a single function and makes very simple system calls. Trying to create a thread with a smaller stack than this is illegal. The `TYPICAL` value is appropriate for applications where application calls are nested no more than half a dozen or so levels, and there are no large arrays on the stack.

If the stack sizes are not estimated correctly and a stack overflow occurs, the probably result is some form of memory corruption. This can be very hard to track down. The kernel does contain some code to help detect stack overflows, controlled by the configuration option `CYGFUN_KERNEL_THREADS_STACK_CHECKING`: a small amount of space is reserved at the stack limit and filled with a special signature: every time a thread context switch occurs this signature is checked, and if invalid that is a good indication (but not absolute proof) that a stack overflow has occurred. This form of stack checking is enabled by default when the system is built with debugging enabled. A related configuration option is `CYGFUN_KERNEL_THREADS_STACK_MEASUREMENT`: enabling this option means that a thread can call the function `cyg_thread_measure_stack_usage` to find out the maximum stack usage to date. Note that this is not necessarily the true maximum because, for example, it is possible that in the current run no interrupt occurred at the worst possible moment.

Valid contexts

`cyg_thread_create` may be called during initialization and from within thread context. It may not be called from inside a DSR.

Example

A simple example of thread creation is shown below. This involves creating five threads, one producer and four consumers or workers. The threads are created in the system's `cyg_user_start`: depending on the configuration it might be more appropriate to do this elsewhere, for example inside `main`.

```
#include <cyg/hal/hal_arch.h>
#include <cyg/kernel/kapi.h>

// These numbers depend entirely on your application
#define NUMBER_OF_WORKERS    4
```

```
#define PRODUCER_PRIORITY   10
#define WORKER_PRIORITY     11
#define PRODUCER_STACKSIZE  CYGNUM_HAL_STACK_SIZE_TYPICAL
#define WORKER_STACKSIZE    (CYGNUM_HAL_STACK_SIZE_MINIMUM + 1024)

static unsigned char producer_stack[PRODUCER_STACKSIZE];
static unsigned char worker_stacks[NUMBER_OF_WORKERS][WORKER_STACKSIZE];
static cyg_handle_t producer_handle, worker_handles[NUMBER_OF_WORKERS];
static cyg_thread   producer_thread, worker_threads[NUMBER_OF_WORKERS];

static void
producer(cyg_addrword_t data)
{
    …
}

static void
worker(cyg_addrword_t data)
{
    …
}

void
cyg_user_start(void)
{
    int i;

    cyg_thread_create(PRODUCER_PRIORITY, &producer, 0, "producer",
    producer_stack, PRODUCER_STACKSIZE,
    &producer_handle, &producer_thread);
    cyg_thread_resume(producer_handle);
    for (i = 0; i < NUMBER_OF_WORKERS; i++) {
        cyg_thread_create(WORKER_PRIORITY, &worker, i, "worker",
        worker_stacks[i], WORKER_STACKSIZE,
        &(worker_handles[i]), &(worker_threads[i]));
        cyg_thread_resume(worker_handles[i]);
    }
}
```

Thread Entry Points and C++

For code written in C++ the thread entry function must be either a static member function of a class or an ordinary function outside any class. It cannot be a normal member function of a class because such member functions take an implicit additional argument `this`, and the kernel has no way of knowing what value to use for this argument. One way around this problem is to make use of a special static member function, for example:

```
class fred {
    public:
        void thread_function();
        static void static_thread_aux(cyg_addrword_t);
};

void
fred::static_thread_aux(cyg_addrword_t objptr)
{
    fred* object = static_cast<fred*>(objptr);
    object->thread_function();
}

static fred instance;

extern "C" void
cyg_start( void )
{
    …
    cyg_thread_create(  …,
                        &fred::static_thread_aux,
                        reinterpret_cast<cyg_addrword_t>(&instance),
                        …);
    …
}
```

Effectively this uses the *`entry_data`* argument to `cyg_thread_create` to hold the `this` pointer. Unfortunately this approach does require the use of some C++ casts, so some of the type safety that can be achieved when programming in C++ is lost.

Name

cyg_thread_self, cyg_thread_idle_thread, cyg_thread_get_stack_base, cyg_thread_get_stack_size, cyg_thread_measure_stack_usage, cyg_thread_get_next, cyg_thread_get_info, cyg_thread_get_id and cyg_thread_find — Get basic thread information

Synopsis

#include <cyg/kernel/kapi.h>

`cyg_handle_t` **`cyg_thread_self`** `();`

`cyg_handle_t` **`cyg_thread_idle_thread`** `();`

`cyg_addrword_t` **`cyg_thread_get_stack_base`** `(`*`thread`*`);`

`cyg_uint32` **`cyg_thread_get_stack_size`** `(`*`thread`*`);`

`cyg_uint32` **`cyg_thread_measure_stack_usage`** `(`*`thread`*`);`

`cyg_bool` **`cyg_thread_get_next`** `(`*`thread`*`, `*`id`*`);`

`cyg_bool` **`cyg_thread_get_info`** `(`*`thread`*`, `*`id`*`, `*`info`*`);`

`cyg_uint16` **`cyg_thread_get_id`** `(`*`thread`*`);`

`cyg_handle_t` **`cyg_thread_find`** `(`*`id`*`);`

Description

These functions can be used to obtain some basic information about various threads in the system. Typically they serve little or no purpose in real applications, but they can be useful during debugging.

`cyg_thread_self` returns a handle corresponding to the current thread. It will be the same as the value filled in by `cyg_thread_create` when the current thread was created. This handle can then be passed to other functions such as `cyg_thread_get_priority`.

`cyg_thread_idle_thread` returns the handle corresponding to the idle thread. This thread is created automatically by the kernel, so application-code has no other way of getting hold of this information.

`cyg_thread_get_stack_base` and `cyg_thread_get_stack_size` return information about a specific thread's stack. The values returned will match the values passed to `cyg_thread_create` when this thread was created.

`cyg_thread_measure_stack_usage` is only available if the configuration option `CYGFUN_KERNEL_THREADS_STACK_MEASUREMENT` is enabled. The return value is the maximum number of bytes of stack space used so far by the specified thread. Note that this should not be considered a true upper bound, for example it is possible that in the current test run the specified thread has not yet been interrupted at the deepest point in the function call graph. Never the less the value returned can give some useful indication of the thread's stack requirements.

`cyg_thread_get_next` is used to enumerate all the current threads in the system. It should be called initially with the locations pointed to by *`thread`* and *`id`* set to zero. On return these will be set to the handle and ID of the first thread. On subsequent calls, these parameters should be left set to the values returned by the previous call. The handle and ID of the next thread in the system will be installed each time, until a `false` return value indicates the end of the list.

`cyg_thread_get_info` fills in the cyg_thread_info structure with information about the thread described by the *`thread`* and *`id`* arguments. The information returned includes the thread's handle and id, its state and name, priorities and stack parameters. If the thread does not exist the function returns `false`.

The cyg_thread_info structure is defined as follows by <cyg/kernel/kapi.h>, but may be extended in future with additional members, and so its size should not be relied upon:

```
typedef struct
{
    cyg_handle_t        handle;
    cyg_uint16          id;
    cyg_uint32          state;
    char                *name;
    cyg_priority_t      set_pri;
    cyg_priority_t      cur_pri;
    cyg_addrword_t      stack_base;
    cyg_uint32          stack_size;
    cyg_uint32          stack_used;
} cyg_thread_info;
```

cyg_thread_get_id returns the unique thread ID for the thread identified by *thread*.

cyg_thread_find returns a handle for the thread whose ID is *id*. If no such thread exists, a zero handle is returned.

Valid contexts

cyg_thread_self may only be called from thread context. cyg_thread_idle_thread may be called from thread or DSR context, but only after the system has been initialized. cyg_thread_get_stack_base, cyg_thread_get_stack_size and cyg_thread_measure_stack_usage may be called any time after the specified thread has been created, but measuring stack usage involves looping over at least part of the thread's stack so this should normally only be done from thread context. cyg_thread_get_id may be called from any context as long as the caller can guarantee that the supplied thread handle remains valid.

Examples

A simple example of the use of the cyg_thread_get_next and cyg_thread_get_info follows:

```
#include <cyg/kernel/kapi.h>
#include <stdio.h>

void show_threads(void)
{
    cyg_handle_t thread = 0;
    cyg_uint16 id = 0;

    while( cyg_thread_get_next( &thread, &id ) )
    {
        cyg_thread_info info;

        if( !cyg_thread_get_info( thread, id, &info ) )
            break;

        printf("ID: %04x name: %10s pri: %d\n",
        info.id, info.name?info.name:"----", info.set_pri );
    }
}
```

Name

cyg_thread_yield, cyg_thread_delay, cyg_thread_suspend, cyg_thread_resume, cyg_thread_release, cyg_thread_set_affinity and cyg_thread_get_affinity — Control whether or not a thread is running

Synopsis

```
#include <cyg/kernel/kapi.h>
```

void **cyg_thread_yield** ();

void **cyg_thread_delay** (*delay*);

void **cyg_thread_suspend** (*thread*);

void **cyg_thread_resume** (*thread*);

void **cyg_thread_release** (*thread*);

void **cyg_thread_set_affinity** (*thread*, *mask*);

void **cyg_thread_get_affinity** (*thread*, **mask*);

Description

These functions provide some control over whether or not a particular thread can run. Apart from the required use of `cyg_thread_resume` to start a newly-created thread, application code should normally use proper synchronization primitives such as condition variables or mail boxes.

Yield

`cyg_thread_yield` allows a thread to relinquish control of the processor to some other runnable thread which has the same priority. This can have no effect on any higher-priority thread since, if such a thread were runnable, the current thread would have been preempted in its favour. Similarly it can have no effect on any lower-priority thread because the current thread will always be run in preference to those. As a consequence this function is only useful in configurations with a scheduler that allows multiple threads to run at the same priority, for example the mlqueue scheduler. If instead the bitmap scheduler was being used then `cyg_thread_yield()` would serve no purpose.

Even if a suitable scheduler such as the mlqueue scheduler has been configured, `cyg_thread_yield` will still rarely prove useful: instead timeslicing will be used to ensure that all threads of a given priority get a fair slice of the available processor time. However it is possible to disable timeslicing via the configuration option `CYGSEM_KERNEL_SCHED_TIMESLICE`, in which case `cyg_thread_yield` can be used to implement a form of cooperative multitasking.

Delay

`cyg_thread_delay` allows a thread to suspend until the specified number of clock ticks have occurred. For example, if a value of 1 is used and the system clock runs at a frequency of 100Hz then the thread will sleep for up to 10 milliseconds. This functionality depends on the presence of a real-time system clock, as controlled by the configuration option `CYGVAR_KERNEL_COUNTERS_CLOCK`.

If the application requires delays measured in milliseconds or similar units rather than in clock ticks, some calculations are needed to convert between these units as described in Clocks. Usually these calculations can be done by the application developer, or at compile-time. Performing such calculations prior to every call to `cyg_thread_delay` adds unnecessary overhead to the system.

Suspend and Resume

Associated with each thread is a suspend counter. When a thread is first created this counter is initialized to 1. cyg_thread_suspend can be used to increment the suspend counter, and cyg_thread_resume decrements it. The scheduler will never run a thread with a non-zero suspend counter. Therefore a newly created thread will not run until it has been resumed.

An occasional problem with the use of suspend and resume functionality is that a thread gets suspended more times than it is resumed and hence never becomes runnable again. This can lead to very confusing behaviour. To help with debugging such problems the kernel provides a configuration option CYGNUM_KERNEL_MAX_SUSPEND_COUNT_ASSERT which imposes an upper bound on the number of suspend calls without matching resumes, with a reasonable default value. This functionality depends on infrastructure assertions being enabled.

Releasing a Blocked Thread

When a thread is blocked on a synchronization primitive such as a semaphore or a mutex, or when it is waiting for an alarm to trigger, it can be forcibly woken up using cyg_thread_release. Typically this will call the affected synchronization primitive to return false, indicating that the operation was not completed successfully. This function has to be used with great care, and in particular it should only be used on threads that have been designed appropriately and check all return codes. If instead it were to be used on, say, an arbitrary thread that is attempting to claim a mutex then that thread might not bother to check the result of the mutex lock operation - usually there would be no reason to do so. Therefore the thread will now continue running in the false belief that it has successfully claimed a mutex lock, and the resulting behaviour is undefined. If the system has been built with assertions enabled then it is possible that an assertion will trigger when the thread tries to release the mutex it does not actually own.

The main use of cyg_thread_release is in the POSIX compatibility layer, where it is used in the implementation of per-thread signals and cancellation handlers.

Thread Affinity

In SMP configurations, using the MLQSMP scheduler, it is possible to control the set of CPUs on which a thread can be run. This can be controlled by using cyg_thread_set_affinity. In addition to the thread handle, this function takes a bitmask that has bit n set if that thread can run on CPU n and clear if it cannot. Bits corresponding to CPUs that are not present are ignored. A mask of all ones allows the thread to run on any CPU and is the default setting. A value of all zeroes will prevent the thread running at all. The function cyg_thread_get_affinity returns the current setting of the thread's affinity mask.

These functions are also present in non-SMP configurations to retain compatibility. In this case cyg_thread_set_affinity is a no-op, and cyg_thread_get_affinity returns a mask with just the bit for CPU 0 set.

Valid contexts

cyg_thread_yield can only be called from thread context, A DSR must always run to completion and cannot yield the processor to some thread. cyg_thread_suspend, cyg_thread_resume, and cyg_thread_release may be called from thread or DSR context.

Name

cyg_thread_exit, cyg_thread_kill and cyg_thread_delete — Allow threads to terminate

Synopsis

#include <cyg/kernel/kapi.h>

void **cyg_thread_exit** ();

void **cyg_thread_kill** (*thread*);

cyg_bool_t **cyg_thread_delete** (*thread*);

Description

In many embedded systems the various threads are allocated statically, created during initialization, and never need to terminate. This avoids any need for dynamic memory allocation or other resource management facilities. However if a given application does have a requirement that some threads be created dynamically, must terminate, and their resources such as the stack be reclaimed, then the kernel provides the functions `cyg_thread_exit`, `cyg_thread_kill`, and `cyg_thread_delete`.

`cyg_thread_exit` allows a thread to terminate itself, thus ensuring that it will not be run again by the scheduler. However the cyg_thread data structure passed to `cyg_thread_create` remains in use, and the handle returned by `cyg_thread_create` remains valid. This allows other threads to perform certain operations on the terminated thread, for example to determine its stack usage via `cyg_thread_measure_stack_usage`. When the handle and cyg_thread structure are no longer required, `cyg_thread_delete` should be called to release these resources. If the stack was dynamically allocated then this should not be freed until after the call to `cyg_thread_delete`.

Alternatively, one thread may use `cyg_thread_kill` on another This has much the same effect as the affected thread calling `cyg_thread_exit`. However killing a thread is generally rather dangerous because no attempt is made to unlock any synchronization primitives currently owned by that thread or release any other resources that thread may have claimed. Therefore use of this function should be avoided, and `cyg_thread_exit` is preferred. `cyg_thread_kill` cannot be used by a thread to kill itself.

`cyg_thread_delete` should be used on a thread after it has exited and is no longer required. After this call the thread handle is no longer valid, and both the cyg_thread structure and the thread stack can be re-used or freed. If `cyg_thread_delete` is invoked on a thread that is still running then there is an implicit call to `cyg_thread_kill`. This function returns `true` if the delete was successful, and `false` if the delete did not happen. The delete may not happen for example if the thread being destroyed is a lower priority thread than the running thread, and will thus not wake up in order to exit until it is rescheduled.

Valid contexts

`cyg_thread_exit`, `cyg_thread_kill` and `cyg_thread_delete` can only be called from thread context.

Name

cyg_thread_get_priority, cyg_thread_get_current_priority and cyg_thread_set_priority — Examine and manipulate thread priorities

Synopsis

#include <cyg/kernel/kapi.h>

```
cyg_priority_t cyg_thread_get_priority (thread);
```

```
cyg_priority_t cyg_thread_get_current_priority (thread);
```

```
void cyg_thread_set_priority (thread, priority);
```

Description

Typical schedulers use the concept of a thread priority to determine which thread should run next. Exactly what this priority consists of will depend on the scheduler, but a typical implementation would be a small integer in the range 0 to 31, with 0 being the highest priority. Usually only the idle thread will run at the lowest priority. The exact number of priority levels available depends on the configuration, typically the option `CYGNUM_KERNEL_SCHED_PRIORITIES`.

`cyg_thread_get_priority` can be used to determine the priority of a thread, or more correctly the value last used in a `cyg_thread_set_priority` call or when the thread was first created. In some circumstances it is possible that the thread is actually running at a higher priority. For example, if it owns a mutex and priority ceilings or inheritance is being used to prevent priority inversion problems, then the thread's priority may have been boosted temporarily. `cyg_thread_get_current_priority` returns the real current priority.

In many applications appropriate thread priorities can be determined and allocated statically. However, if it is necessary for a thread's priority to change at run-time then the `cyg_thread_set_priority` function provides this functionality.

Valid contexts

`cyg_thread_get_priority` and `cyg_thread_get_current_priority` can be called from thread or DSR context, although the latter is rarely useful. `cyg_thread_set_priority` should also only be called from thread context.

Name

cyg_thread_new_data_index, cyg_thread_free_data_index, cyg_thread_get_data, cyg_thread_get_data_ptr and cyg_thread_set_data — Manipulate per-thread data

Synopsis

#include <cyg/kernel/kapi.h>

cyg_ucount32 **cyg_thread_new_data_index** ();

void **cyg_thread_free_data_index** (*index*);

cyg_addrword_t **cyg_thread_get_data** (*index*);

cyg_addrword_t* **cyg_thread_get_data_ptr** (*index*);

void **cyg_thread_set_data** (*index*, *data*);

Description

In some applications and libraries it is useful to have some data that is specific to each thread. For example, many of the functions in the POSIX compatibility package return -1 to indicate an error and store additional information in what appears to be a global variable errno. However, if multiple threads make concurrent calls into the POSIX library and if errno were really a global variable then a thread would have no way of knowing whether the current errno value really corresponded to the last POSIX call it made, or whether some other thread had run in the meantime and made a different POSIX call which updated the variable. To avoid such confusion errno is instead implemented as a per-thread variable, and each thread has its own instance.

The support for per-thread data can be disabled via the configuration option CYGVAR_KERNEL_THREADS_DATA. If enabled, each cyg_thread data structure holds a small array of words. The size of this array is determined by the configuration option CYGNUM_KERNEL_THREADS_DATA_MAX. When a thread is created the array is filled with zeroes.

If an application needs to use per-thread data then it needs an index into this array which has not yet been allocated to other code. This index can be obtained by calling cyg_thread_new_data_index, and then used in subsequent calls to cyg_thread_get_data. Typically indices are allocated during system initialization and stored in static variables. If for some reason a slot in the array is no longer required and can be re-used then it can be released by calling cyg_thread_free_data_index. When a slot index is allocated, then if the CYGVAR_KERNEL_THREADS_LIST option is enabled, the corresponding array entry for all threads will be reset back to zero in case that slot had been previously used.

The current per-thread data in a given slot can be obtained using cyg_thread_get_data. This implicitly operates on the current thread, and its single argument should be an index as returned by cyg_thread_new_data_index. The per-thread data can be updated using cyg_thread_set_data. If a particular item of per-thread data is needed repeatedly then cyg_thread_get_data_ptr can be used to obtain the address of the data, and indirecting through this pointer allows the data to be examined and updated efficiently.

Some packages, for example the error and POSIX packages, have pre-allocated slots in the array of per-thread data. These slots should not normally be used by application code, and instead slots should be allocated during initialization by a call to cyg_thread_new_data_index. If it is known that, for example, the configuration will never include the POSIX compatibility package then application code may instead decide to re-use the slot allocated to that package, CYGNUM_KERNEL_THREADS_DATA_POSIX, but obviously this does involve a risk of strange and subtle bugs if the application's requirements ever change.

Valid contexts

Typically cyg_thread_new_data_index is only called during initialization, but may also be called at any time in thread context. cyg_thread_free_data_index, if used at all, can also be called during initialization or from thread context.

`cyg_thread_get_data`, `cyg_thread_get_data_ptr`, and `cyg_thread_set_data` may only be called from thread context because they implicitly operate on the current thread.

Name

cyg_thread_add_destructor and cyg_thread_rem_destructor — Call functions on thread termination

Synopsis

#include <cyg/kernel/kapi.h> typedef void (*cyg_thread_destructor_fn)(cyg_addrword_t);

`cyg_bool_t` **`cyg_thread_add_destructor`** `(`*`fn`*`, `*`data`*`);`

`cyg_bool_t` **`cyg_thread_rem_destructor`** `(`*`fn`*`, `*`data`*`);`

Description

These functions are provided for cases when an application requires a function to be automatically called when a thread exits. This is often useful when, for example, freeing up resources allocated by the thread.

This support must be enabled with the configuration option `CYGPKG_KERNEL_THREADS_DESTRUCTORS`. When enabled, you may register a function of type cyg_thread_destructor_fn to be called on thread termination using `cyg_thread_add_destructor`. You may also provide it with a piece of arbitrary information in the *`data`* argument which will be passed to the destructor function *`fn`* when the thread terminates. If you no longer wish to call a function previous registered with `cyg_thread_add_destructor`, you may call `cyg_thread_rem_destructor` with the same parameters used to register the destructor function. Both these functions return `true` on success and `false` on failure.

By default, thread destructors are per-thread, which means that registering a destructor function only registers that function for the current thread. In other words, each thread has its own list of destructors. Alternatively you may disable the configuration option `CYGSEM_KERNEL_THREADS_DESTRUCTORS_PER_THREAD` in which case any registered destructors will be run when *any* threads exit. In other words, the thread destructor list is global and all threads have the same destructors.

There is a limit to the number of destructors which may be registered, which can be controlled with the `CYGNUM_KERNEL_THREADS_DESTRUCTORS` configuration option. Increasing this value will very slightly increase the amount of memory in use, and when `CYGSEM_KERNEL_THREADS_DESTRUCTORS_PER_THREAD` is enabled, the amount of memory used per thread will increase. When the limit has been reached, `cyg_thread_add_destructor` will return `false`.

Valid contexts

When `CYGSEM_KERNEL_THREADS_DESTRUCTORS_PER_THREAD` is enabled, these functions must only be called from a thread context as they implicitly operate on the current thread. When `CYGSEM_KERNEL_THREADS_DESTRUCTORS_PER_THREAD` is disabled, these functions may be called from thread or DSR context, or at initialization time.

Name

cyg_exception_set_handler, cyg_exception_clear_handler and cyg_exception_call_handler — Handle processor exceptions

Synopsis

#include <cyg/kernel/kapi.h>

`void` **`cyg_exception_set_handler`** `(`*`exception_number`*`,` *`new_handler`*`,` *`new_data`*`,` *`old_handler`*`,` *`old_data`*`);`

`void` **`cyg_exception_clear_handler`** `(`*`exception_number`*`);`

`void` **`cyg_exception_call_handler`** `(`*`thread`*`,` *`exception_number`*`,` *`exception_info`*`);`

Description

Sometimes code attempts operations that are not legal on the current hardware, for example dividing by zero, or accessing data through a pointer that is not properly aligned. When this happens the hardware will raise an exception. This is very similar to an interrupt, but happens synchronously with code execution rather than asynchronously and hence can be tied to the thread that is currently running.

The exceptions that can be raised depend very much on the hardware, especially the processor. The corresponding documentation should be consulted for more details. Alternatively the architectural HAL header file `hal_intr.h`, or one of the variant or platform header files it includes, will contain appropriate definitions. The details of how to handle exceptions, including whether or not it is possible to recover from them, also depend on the hardware.

Exception handling is optional, and can be disabled through the configuration option `CYGPKG_KERNEL_EXCEPTIONS`. If an application has been exhaustively tested and is trusted never to raise a hardware exception then this option can be disabled and code and data sizes will be reduced somewhat. If exceptions are left enabled then the system will provide default handlers for the various exceptions, but these do nothing. Even the specific type of exception is ignored, so there is no point in attempting to decode this and distinguish between say a divide-by-zero and an unaligned access. If the application installs its own handlers and wants details of the specific exception being raised then the configuration option `CYGSEM_KERNEL_EXCEPTIONS_DECODE` has to be enabled.

An alternative handler can be installed using `cyg_exception_set_handler`. This requires a code for the exception, a function pointer for the new exception handler, and a parameter to be passed to this handler. Details of the previously installed exception handler will be returned via the remaining two arguments, allowing that handler to be reinstated, or null pointers can be used if this information is of no interest. An exception handling function should take the following form:

```
void
my_exception_handler(cyg_addrword_t data, cyg_code_t exception, cyg_addrword_t info)
{
    …
}
```

The data argument corresponds to the *`new_data`* parameter supplied to `cyg_exception_set_handler`. The exception code is provided as well, in case a single handler is expected to support multiple exceptions. The *`info`* argument will depend on the hardware and on the specific exception.

`cyg_exception_clear_handler` can be used to restore the default handler, if desired. It is also possible for software to raise an exception and cause the current handler to be invoked, but generally this is useful only for testing.

By default the system maintains a single set of global exception handlers. However, since exceptions occur synchronously it is sometimes useful to handle them on a per-thread basis, and have a different set of handlers for each thread. This behaviour can be obtained by disabling the configuration option `CYGSEM_KERNEL_EXCEPTIONS_GLOBAL`. If per-thread exception handlers are being used then `cyg_exception_set_handler` and `cyg_exception_clear_handler` apply to the current thread. Otherwise they apply to the global set of handlers.

Caution

In the current implementation `cyg_exception_call_handler` can only be used on the current thread. There is no support for delivering an exception to another thread.

Note

Exceptions at the eCos kernel level refer specifically to hardware-related events such as unaligned accesses to memory or division by zero. There is no relation with other concepts that are also known as exceptions, for example the `throw` and `catch` facilities associated with C++.

Valid contexts

If the system is configured with a single set of global exception handlers then `cyg_exception_set_handler` and `cyg_exception_clear_handler` may be called during initialization or from thread context. If instead per-thread exception handlers are being used then it is not possible to install new handlers during initialization because the functions operate implicitly on the current thread, so they can only be called from thread context. `cyg_exception_call_handler` should only be called from thread context.

Name

cyg_counter_create, cyg_counter_delete, cyg_counter_current_value, cyg_counter_set_value and cyg_counter_tick — Count event occurrences

Synopsis

#include <cyg/kernel/kapi.h>

```
void cyg_counter_create (handle, counter);

void cyg_counter_delete (counter);

cyg_tick_count_t cyg_counter_current_value (counter);

void cyg_counter_set_value (counter, new_value);

void cyg_counter_tick (counter);
```

Description

Kernel counters can be used to keep track of how many times a particular event has occurred. Usually this event is an external signal of some sort. The most common use of counters is in the implementation of clocks, but they can be useful with other event sources as well. Application code can attach alarms to counters, causing a function to be called when some number of events have occurred.

A new counter is initialized by a call to `cyg_counter_create`. The first argument is used to return a handle to the new counter which can be used for subsequent operations. The second argument allows the application to provide the memory needed for the object, thus eliminating any need for dynamic memory allocation within the kernel. If a counter is no longer required and does not have any alarms attached then `cyg_counter_delete` can be used to release the resources, allowing the cyg_counter data structure to be re-used.

Initializing a counter does not automatically attach it to any source of events. Instead some other code needs to call `cyg_counter_tick` whenever a suitable event occurs, which will cause the counter to be incremented and may cause alarms to trigger. The current value associated with the counter can be retrieved using `cyg_counter_current_value` and modified with `cyg_counter_set_value`. Typically the latter function is only used during initialization, for example to set a clock to wallclock time, but it can be used to reset a counter if necessary. However `cyg_counter_set_value` will never trigger any alarms. A newly initialized counter has a starting value of 0.

The kernel provides two different implementations of counters. The default is `CYGIMP_KERNEL_COUNTERS_SINGLE_LIST` which stores all alarms attached to the counter on a single list. This is simple and usually efficient. However when a tick occurs the kernel code has to traverse this list, typically at DSR level, so if there are a significant number of alarms attached to a single counter this will affect the system's dispatch latency. The alternative implementation, `CYGIMP_KERNEL_COUNTERS_MULTI_LIST`, stores each alarm in one of an array of lists such that at most one of the lists needs to be searched per clock tick. This involves extra code and data, but can improve real-time responsiveness in some circumstances. Another configuration option that is relevant here is `CYGIMP_KERNEL_COUNTERS_SORT_LIST`, which is disabled by default. This provides a trade off between doing work whenever a new alarm is added to a counter and doing work whenever a tick occurs. It is application-dependent which of these is more appropriate.

Valid contexts

`cyg_counter_create` is typically called during system initialization but may also be called in thread context. Similarly `cyg_counter_delete` may be called during initialization or in thread context. `cyg_counter_current_value`, `cyg_counter_set_value` and `cyg_counter_tick` may be called during initialization or from thread or DSR context. In fact, `cyg_counter_tick` is usually called from inside a DSR in response to an external event of some sort.

Name

cyg_clock_create, cyg_clock_delete, cyg_clock_to_counter, cyg_clock_set_resolution, cyg_clock_get_resolution, cyg_real_time_clock and cyg_current_time — Provide system clocks

Synopsis

#include <cyg/kernel/kapi.h>

```
void cyg_clock_create (resolution, handle, clock);

void cyg_clock_delete (clock);

void cyg_clock_to_counter (clock, counter);

void cyg_clock_set_resolution (clock, resolution);

cyg_resolution_t cyg_clock_get_resolution (clock);

cyg_handle_t cyg_real_time_clock ();

cyg_tick_count_t cyg_current_time ();
```

Description

In the eCos kernel clock objects are a special form of counter objects. They are attached to a specific type of hardware, clocks that generate ticks at very specific time intervals, whereas counters can be used with any event source.

In a default configuration the kernel provides a single clock instance, the real-time clock. This gets used for timeslicing and for operations that involve a timeout, for example `cyg_semaphore_timed_wait`. If this functionality is not required it can be removed from the system using the configuration option `CYGVAR_KERNEL_COUNTERS_CLOCK`. Otherwise the real-time clock can be accessed by a call to `cyg_real_time_clock`, allowing applications to attach alarms, and the current counter value can be obtained using `cyg_current_time`.

Applications can create and destroy additional clocks if desired, using `cyg_clock_create` and `cyg_clock_delete`. The first argument to `cyg_clock_create` specifies the resolution this clock will run at. The second argument is used to return a handle for this clock object, and the third argument provides the kernel with the memory needed to hold this object. This clock will not actually tick by itself. Instead it is the responsibility of application code to initialize a suitable hardware timer to generate interrupts at the appropriate frequency, install an interrupt handler for this, and call `cyg_counter_tick` from inside the DSR. Associated with each clock is a kernel counter, a handle for which can be obtained using `cyg_clock_to_counter`.

Clock Resolutions and Ticks

At the kernel level all clock-related operations including delays, timeouts and alarms work in units of clock ticks, rather than in units of seconds or milliseconds. If the calling code, whether the application or some other package, needs to operate using units such as milliseconds then it has to convert from these units to clock ticks.

The main reason for this is that it accurately reflects the hardware: calling something like `nanosleep` with a delay of ten nanoseconds will not work as intended on any real hardware because timer interrupts simply will not happen that frequently; instead calling `cyg_thread_delay` with the equivalent delay of 0 ticks gives a much clearer indication that the application is attempting something inappropriate for the target hardware. Similarly, passing a delay of five ticks to `cyg_thread_delay` makes it fairly obvious that the current thread will be suspended for somewhere between four and five clock periods, as opposed to passing 50000000 to `nanosleep` which suggests a granularity that is not actually provided.

A secondary reason is that conversion between clock ticks and units such as milliseconds can be somewhat expensive, and whenever possible should be done at compile-time or by the application developer rather than at run-time. This saves code size and CPU cycles.

The information needed to perform these conversions is the clock resolution. This is a structure with two fields, a dividend and a divisor, and specifies the number of nanoseconds between clock ticks. For example a clock that runs at 100Hz will have 10 milliseconds between clock ticks, or 10000000 nanoseconds. The ratio between the resolution's dividend and divisor will therefore be 10000000 to 1, and typical values for these might be 1000000000 and 100. If the clock runs at a different frequency, say 60Hz, the numbers could be 1000000000 and 60 respectively. Given a delay in nanoseconds, this can be converted to clock ticks by multiplying with the the divisor and then dividing by the dividend. For example a delay of 50 milliseconds corresponds to 50000000 nanoseconds, and with a clock frequency of 100Hz this can be converted to ((50000000 * 100) / 1000000000) = 5 clock ticks. Given the large numbers involved this arithmetic normally has to be done using 64-bit precision and the long long data type, but allows code to run on hardware with unusual clock frequencies.

The default frequency for the real-time clock on any platform is usually about 100Hz, but platform-specific documentation should be consulted for this information. Usually it is possible to override this default by configuration options, but again this depends on the capabilities of the underlying hardware. The resolution for any clock can be obtained using `cyg_clock_get_resolution`. For clocks created by application code, there is also a function `cyg_clock_set_resolution`. This does not affect the underlying hardware timer in any way, it merely updates the information that will be returned in subsequent calls to `cyg_clock_get_resolution`: changing the actual underlying clock frequency will require appropriate manipulation of the timer hardware.

Valid contexts

`cyg_clock_create` is usually only called during system initialization (if at all), but may also be called from thread context. The same applies to `cyg_clock_delete`. The remaining functions may be called during initialization, from thread context, or from DSR context, although it should be noted that there is no locking between `cyg_clock_get_resolution` and `cyg_clock_set_resolution` so theoretically it is possible that the former returns an inconsistent data structure.

Name

cyg_alarm_create, cyg_alarm_delete, cyg_alarm_initialize, cyg_alarm_enable and cyg_alarm_disable — Run an alarm function when a number of events have occurred

Synopsis

#include <cyg/kernel/kapi.h>

void **cyg_alarm_create** (*counter*, *alarmfn*, *data*, *handle*, *alarm*);

void **cyg_alarm_delete** (*alarm*);

void **cyg_alarm_initialize** (*alarm*, *trigger*, *interval*);

void **cyg_alarm_enable** (*alarm*);

void **cyg_alarm_disable** (*alarm*);

Description

Kernel alarms are used together with counters and allow for action to be taken when a certain number of events have occurred. If the counter is associated with a clock then the alarm action happens when the appropriate number of clock ticks have occurred, in other words after a certain period of time.

Setting up an alarm involves a two-step process. First the alarm must be created with a call to `cyg_alarm_create`. This takes five arguments. The first identifies the counter to which the alarm should be attached. If the alarm should be attached to the system's real-time clock then `cyg_real_time_clock` and `cyg_clock_to_counter` can be used to get hold of the appropriate handle. The next two arguments specify the action to be taken when the alarm is triggered, in the form of a function pointer and some data. This function should take the form:

```
void
alarm_handler(cyg_handle_t alarm, cyg_addrword_t data)
{
    …
}
```

The data argument passed to the alarm function corresponds to the third argument passed to `cyg_alarm_create`. The fourth argument to `cyg_alarm_create` is used to return a handle to the newly-created alarm object, and the final argument provides the memory needed for the alarm object and thus avoids any need for dynamic memory allocation within the kernel.

Once an alarm has been created a further call to `cyg_alarm_initialize` is needed to activate it. The first argument specifies the alarm. The second argument indicates the number of events, for example clock ticks, that need to occur before the alarm triggers. If the third argument is 0 then the alarm will only trigger once. A non-zero value specifies that the alarm should trigger repeatedly, with an interval of the specified number of events.

Alarms can be temporarily disabled and reenabled using `cyg_alarm_disable` and `cyg_alarm_enable`. Alternatively another call to `cyg_alarm_initialize` can be used to modify the behaviour of an existing alarm. If an alarm is no longer required then the associated resources can be released using `cyg_alarm_delete`.

If two or more alarms are registered for precisely the same counter tick, the order of execution of the alarm functions is unspecified.

Handler context

The alarm function is invoked when a counter tick occurs, in other words when there is a call to `cyg_counter_tick`, and will happen in the same context. If the alarm is associated with the system's real-time clock then by default this will be DSR context, following a clock interrupt. If the alarm is associated with some other application-specific counter then the details will depend on how that counter is updated.

It is also possible to configure the kernel to call kernel RTC alarms in a thread context, instead of a DSR context. This is enabled with the option "Call RTC events from a thread" (`CYGIMP_KERNEL_COUNTERS_RTC_TICK_THREAD`). When enabled, a dedicated thread is created for running kernel alarm handlers. This can be useful in improving deterministic real-time behaviour as lengthy alarm handlers in a DSR context could disrupt normal scheduling.

As an additional variation, normally the scheduler is locked while running all the alarm handlers, preventing DSRs (and any higher priority threads) from running. However if the "Reduce DSR latency" option is enabled, the scheduler will briefly be unlocked at a safe point between each alarm handler call, in order to allow DSRs to run. This reduces the worst case DSR latency to that of the longest single running alarm handler. However note that if enabled, this option effects *all* counters and clocks in the system, not just those associated with the kernel RTC.

Valid calling contexts

`cyg_alarm_createcyg_alarm_initialize` is typically called during system initialization but may also be called in thread context. The same applies to `cyg_alarm_delete`. `cyg_alarm_initialize`, `cyg_alarm_disable` and `cyg_alarm_enable` may be called during initialization or from thread or DSR context, but `cyg_alarm_enable` and `cyg_alarm_initialize` may be expensive operations and should only be called when necessary.

Name

cyg_mutex_init, cyg_mutex_destroy, cyg_mutex_lock, cyg_mutex_timed_lock, cyg_mutex_trylock, cyg_mutex_unlock, cyg_mutex_release, cyg_mutex_set_ceiling and cyg_mutex_set_protocol — Synchronization primitive

Synopsis

#include <cyg/kernel/kapi.h>

`void` **`cyg_mutex_init`** `(`*`mutex`*`);`

`void` **`cyg_mutex_destroy`** `(`*`mutex`*`);`

`cyg_bool_t` **`cyg_mutex_lock`** `(`*`mutex`*`);`

`cyg_bool_t` **`cyg_mutex_timed_lock`** `(`*`mutex`*`,` *`abstime`*`);`

`cyg_bool_t` **`cyg_mutex_trylock`** `(`*`mutex`*`);`

`void` **`cyg_mutex_unlock`** `(`*`mutex`*`);`

`void` **`cyg_mutex_release`** `(`*`mutex`*`);`

`void` **`cyg_mutex_set_ceiling`** `(`*`mutex`*`,` *`priority`*`);`

`void` **`cyg_mutex_set_protocol`** `(`*`mutex`*`,` *`protocol/`*`);`

Description

The purpose of mutexes is to let threads share resources safely. If two or more threads attempt to manipulate a data structure with no locking between them then the system may run for quite some time without apparent problems, but sooner or later the data structure will become inconsistent and the application will start behaving strangely and is quite likely to crash. The same can apply even when manipulating a single variable or some other resource. For example, consider:

```
static volatile int counter = 0;

void
process_event(void)
{
    …

    counter++;
}
```

Assume that after a certain period of time `counter` has a value of 42, and two threads A and B running at the same priority call `process_event`. Typically thread A will read the value of `counter` into a register, increment this register to 43, and write this updated value back to memory. Thread B will do the same, so usually `counter` will end up with a value of 44. However if thread A is timesliced after reading the old value 42 but before writing back 43, thread B will still read back the old value and will also write back 43. The net result is that the counter only gets incremented once, not twice, which depending on the application may prove disastrous.

Sections of code like the above which involve manipulating shared data are generally known as critical regions. Code should claim a lock before entering a critical region and release the lock when leaving. Mutexes provide an appropriate synchronization primitive for this.

```
static volatile int counter = 0;
static cyg_mutex_t  lock;

void
```

```
process_event(void)
{
    …

    cyg_mutex_lock(&lock);
    counter++;
    cyg_mutex_unlock(&lock);
}
```

A mutex must be initialized before it can be used, by calling `cyg_mutex_init`. This takes a pointer to a cyg_mutex_t data structure which is typically statically allocated, and may be part of a larger data structure. If a mutex is no longer required and there are no threads waiting on it then `cyg_mutex_destroy` can be used.

The main functions for using a mutex are `cyg_mutex_lock` and `cyg_mutex_unlock`. In normal operation `cyg_mutex_lock` will return success after claiming the mutex lock, blocking if another thread currently owns the mutex. However the lock operation may fail if other code calls `cyg_mutex_release` or `cyg_thread_release`, so if these functions may get used then it is important to check the return value. The current owner of a mutex should call `cyg_mutex_unlock` when a lock is no longer required. This operation must be performed by the owner, not by another thread.

The kernel supplies a variant of `cyg_mutex_lock`, `cyg_mutex_timed_wait`, which can be used to wait for the lock or until some number of clock ticks have passed. The number of ticks is specified as an absolute, not relative, tick count and so in order to wait for a relative number of ticks, the return value of the `cyg_current_time()` function should be added to determine the absolute number of ticks. If this function returns `true` then the mutex has been claimed, if it returns `false` then either a timeout has occurred or the thread has been released.

`cyg_mutex_trylock` is a variant of `cyg_mutex_lock` that will always return immediately, returning success or failure as appropriate. This function is rarely useful. Typical code locks a mutex just before entering a critical region, so if the lock cannot be claimed then there may be nothing else for the current thread to do. Use of this function may also cause a form of priority inversion if the owner runs at a lower priority, because the priority inheritance code will not be triggered. Instead the current thread continues running, preventing the owner from getting any CPU time, completing the critical region, and releasing the mutex.

`cyg_mutex_release` can be used to wake up all threads that are currently blocked inside a call to `cyg_mutex_lock` for a specific mutex. These lock calls will return failure. The current mutex owner is not affected.

Priority Inversion

The use of mutexes gives rise to a problem known as priority inversion. In a typical scenario this requires three threads A, B, and C, running at high, medium and low priority respectively. Thread A and thread B are temporarily blocked waiting for some event, so thread C gets a chance to run, needs to enter a critical region, and locks a mutex. At this point threads A and B are woken up - the exact order does not matter. Thread A needs to claim the same mutex but has to wait until C has left the critical region and can release the mutex. Meanwhile thread B works on something completely different and can continue running without problems. Because thread C is running a lower priority than B it will not get a chance to run until B blocks for some reason, and hence thread A cannot run either. The overall effect is that a high-priority thread A cannot proceed because of a lower priority thread B, and priority inversion has occurred.

In simple applications it may be possible to arrange the code such that priority inversion cannot occur, for example by ensuring that a given mutex is never shared by threads running at different priority levels. However this may not always be possible even at the application level. In addition mutexes may be used internally by underlying code, for example the memory allocation package, so careful analysis of the whole system would be needed to be sure that priority inversion cannot occur. Instead it is common practice to use one of two techniques: priority ceilings and priority inheritance.

Priority ceilings involve associating a priority with each mutex. Usually this will match the highest priority thread that will ever lock the mutex. When a thread running at a lower priority makes a successful call to `cyg_mutex_lock` or `cyg_mutex_trylock` its priority will be boosted to that of the mutex. For example, given the previous example the priority associated with the mutex would be that of thread A, so for as long as it owns the mutex thread C will run in preference to thread B. When C releases the mutex its priority drops to the normal value again, allowing A to run and claim the mutex. Setting the priority for a mutex involves a call to `cyg_mutex_set_ceiling`, which is typically called during initialization. It is possible to change the ceiling dynamically but this will only affect subsequent lock operations, not the current owner of the mutex.

Priority ceilings are very suitable for simple applications, where for every thread in the system it is possible to work out which mutexes will be accessed. For more complicated applications this may prove difficult, especially if thread priorities change at run-time. An additional problem occurs for any mutexes outside the application, for example used internally within eCos packages. A typical eCos package will be unaware of the details of the various threads in the system, so it will have no way of setting suitable ceilings for its internal mutexes. If those mutexes are not exported to application code then using priority ceilings may not be viable. The kernel does provide a configuration option `CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_DEFAULT_PRIORITY` that can be used to set the default priority ceiling for all mutexes, which may prove sufficient.

The alternative approach is to use priority inheritance: if a thread calls `cyg_mutex_lock` for a mutex that it currently owned by a lower-priority thread, then the owner will have its priority raised to that of the current thread. Often this is more efficient than priority ceilings because priority boosting only happens when necessary, not for every lock operation, and the required priority is determined at run-time rather than by static analysis. However there are complications when multiple threads running at different priorities try to lock a single mutex, or when the current owner of a mutex then tries to lock additional mutexes, and this makes the implementation significantly more complicated than priority ceilings.

There are a number of configuration options associated with priority inversion. First, if after careful analysis it is known that priority inversion cannot arise then the component `CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL` can be disabled. More commonly this component will be enabled, and one of either `CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_INHERIT` or `CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_CEILING` will be selected, so that one of the two protocols is available for all mutexes. It is possible to select multiple protocols, so that some mutexes can have priority ceilings while others use priority inheritance or no priority inversion protection at all. Obviously this flexibility will add to the code size and to the cost of mutex operations. The default for all mutexes will be controlled by `CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_DEFAULT`, and can be changed at run-time using `cyg_mutex_set_protocol`.

Priority inversion problems can also occur with other synchronization primitives such as semaphores. For example there could be a situation where a high-priority thread A is waiting on a semaphore, a low-priority thread C needs to do just a little bit more work before posting the semaphore, but a medium priority thread B is running and preventing C from making progress. However a semaphore does not have the concept of an owner, so there is no way for the system to know that it is thread C which would next post to the semaphore. Hence there is no way for the system to boost the priority of C automatically and prevent the priority inversion. Instead situations like this have to be detected by application developers and appropriate precautions have to be taken, for example making sure that all the threads run at suitable priorities at all times.

Warning

The default implementation of priority inheritance within the eCos kernel has been simplified in a way that may cause behaviour which is unexpected for developers. Problems will only arise if a thread owns one mutex, then attempts to claim another mutex, and there are other threads attempting to lock these same mutexes. Although the system will continue running, the current owners of the various mutexes involved may not run at the priority they should.

The reason for this is that, with the default implementation of priority inheritance, a thread which has its priority boosted due to it having locked *two or more* mutexes will not have its priority reduced until *both* mutexes are unlocked. In other words, that thread will continue running at the highest priority of any of the threads waiting for a mutex it holds, and will keep running at that priority until it has unlocked all the mutexes which it holds.

This situation rarely arises in real-world code because a mutex should generally only be locked for a small critical region, and there is no need to manipulate other shared resources inside this region. However eCosPro offers an alternative implementation which does allow priorities to be reduced in a fair and accurate way when mutexes are unlocked. This alternative is not enabled by default, but can be enabled with the "Fair priority inheritance semantics" (`CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_INHERIT_ACCURATE`) CDL configuration option. However the trade-off of providing a fair and accurate behaviour of priority inheritance is that mutex unlock operations then have a non-deterministic element as each mutex held by a thread must be examined to determine the priority of the highest waiting thread. Fortunately, it is unlikely in real-world applications that more than a few mutexes will be held simultaneously, so that list should be short; and therefore this trade-off may be acceptable for many developers.

Warning

Support for priority ceilings and priority inheritance is not implemented for all schedulers. In particular neither priority ceilings nor priority inheritance are currently available for the bitmap scheduler.

Alternatives

In nearly all circumstances, if two or more threads need to share some data then protecting this data with a mutex is the correct thing to do. Mutexes are the only primitive that combine a locking mechanism and protection against priority inversion problems. However this functionality is achieved at a cost, and in exceptional circumstances such as an application's most critical inner loop it may be desirable to use some other means of locking.

When a critical region is very very small it is possible to lock the scheduler, thus ensuring that no other thread can run until the scheduler is unlocked again. This is achieved with calls to `cyg_scheduler_lock` and `cyg_scheduler_unlock`. If the critical region is sufficiently small then this can actually improve both performance and dispatch latency because `cyg_mutex_lock` also locks the scheduler for a brief period of time. This approach will not work on SMP systems because another thread may already be running on a different processor and accessing the critical region.

Another way of avoiding the use of mutexes is to make sure that all threads that access a particular critical region run at the same priority and configure the system with timeslicing disabled (`CYGSEM_KERNEL_SCHED_TIMESLICE`). Without timeslicing a thread can only be preempted by a higher-priority one, or if it performs some operation that can block. This approach requires that none of the operations in the critical region can block, so for example it is not legal to call `cyg_semaphore_wait`. It is also vulnerable to any changes in the configuration or to the various thread priorities: any such changes may now have unexpected side effects. It will not work on SMP systems.

Recursive Mutexes

The implementation of mutexes within the eCos kernel does not support recursive locks. If a thread has locked a mutex and then attempts to lock the mutex again, typically as a result of some recursive call in a complicated call graph, then either an assertion failure will be reported or the thread will deadlock. This behaviour is deliberate. When a thread has just locked a mutex associated with some data structure, it can assume that that data structure is in a consistent state. Before unlocking the mutex again it must ensure that the data structure is again in a consistent state. Recursive mutexes allow a thread to make arbitrary changes to a data structure, then in a recursive call lock the mutex again while the data structure is still inconsistent. The net result is that code can no longer make any assumptions about data structure consistency, which defeats the purpose of using mutexes.

Valid contexts

`cyg_mutex_init`, `cyg_mutex_set_ceiling` and `cyg_mutex_set_protocol` are normally called during initialization but may also be called from thread context. The remaining functions should only be called from thread context. Mutexes serve as a mutual exclusion mechanism between threads, and cannot be used to synchronize between threads and the interrupt handling subsystem. If a critical region is shared between a thread and a DSR then it must be protected using `cyg_scheduler_lock` and `cyg_scheduler_unlock`. If a critical region is shared between a thread and an ISR, it must be protected by disabling or masking interrupts. Obviously these operations must be used with care because they can affect dispatch and interrupt latencies.

Name

cyg_cond_init, cyg_cond_destroy, cyg_cond_wait, cyg_cond_timed_wait, cyg_cond_signal and cyg_cond_broadcast — Synchronization primitive

Synopsis

#include <cyg/kernel/kapi.h>

void **cyg_cond_init** (*cond*, *mutex*);

void **cyg_cond_destroy** (*cond*);

cyg_bool_t **cyg_cond_wait** (*cond*);

cyg_bool_t **cyg_cond_timed_wait** (*cond*, *abstime*);

void **cyg_cond_signal** (*cond*);

void **cyg_cond_broadcast** (*cond*);

Description

Condition variables are used in conjunction with mutexes to implement long-term waits for some condition to become true. For example consider a set of functions that control access to a pool of resources:

```
cyg_mutex_t res_lock;
res_t res_pool[RES_MAX];
int res_count = RES_MAX;

void res_init(void)
{
    cyg_mutex_init(&res_lock);
    <fill pool with resources>
}

res_t res_allocate(void)
{
    res_t res;

    cyg_mutex_lock(&res_lock);               // lock the mutex

    if( res_count == 0 )                     // check for free resource
        res = RES_NONE;                      // return RES_NONE if none
    else
    {
        res_count--;                         // allocate a resources
        res = res_pool[res_count];
    }

    cyg_mutex_unlock(&res_lock);             // unlock the mutex

    return res;
}

void res_free(res_t res)
{
    cyg_mutex_lock(&res_lock);               // lock the mutex

    res_pool[res_count] = res;               // free the resource
    res_count++;
    cyg_mutex_unlock(&res_lock);             // unlock the mutex
}
```

These routines use the variable `res_count` to keep track of the resources available. If there are none then `res_allocate` returns `RES_NONE`, which the caller must check for and take appropriate error handling actions.

Now suppose that we do not want to return `RES_NONE` when there are no resources, but want to wait for one to become available. This is where a condition variable can be used:

```
cyg_mutex_t res_lock;
cyg_cond_t res_wait;
res_t res_pool[RES_MAX];
int res_count = RES_MAX;

void res_init(void)
{
    cyg_mutex_init(&res_lock);
    cyg_cond_init(&res_wait, &res_lock);
    <fill pool with resources>
}

res_t res_allocate(void)
{
    res_t res;

    cyg_mutex_lock(&res_lock);               // lock the mutex

    while( res_count == 0 )                  // wait for a resources
        cyg_cond_wait(&res_wait);

    res_count--;                             // allocate a resource
    res = res_pool[res_count];

    cyg_mutex_unlock(&res_lock);             // unlock the mutex

    return res;
}

void res_free(res_t res)
{
    cyg_mutex_lock(&res_lock);               // lock the mutex

    res_pool[res_count] = res;               // free the resource
    res_count++;

    cyg_cond_signal(&res_wait);              // wake up any waiting allocators

    cyg_mutex_unlock(&res_lock);             // unlock the mutex
}
```

In this version of the code, when `res_allocate` detects that there are no resources it calls `cyg_cond_wait`. This does two things: it unlocks the mutex, and puts the calling thread to sleep on the condition variable. When `res_free` is eventually called, it puts a resource back into the pool and calls `cyg_cond_signal` to wake up any thread waiting on the condition variable. When the waiting thread eventually gets to run again, it will re-lock the mutex before returning from `cyg_cond_wait`.

There are two important things to note about the way in which this code works. The first is that the mutex unlock and wait in `cyg_cond_wait` are atomic: no other thread can run between the unlock and the wait. If this were not the case then a call to `res_free` by that thread would release the resource but the call to `cyg_cond_signal` would be lost, and the first thread would end up waiting when there were resources available.

The second feature is that the call to `cyg_cond_wait` is in a `while` loop and not a simple `if` statement. This is because of the need to re-lock the mutex in `cyg_cond_wait` when the signalled thread reawakens. If there are other threads already queued to claim the lock then this thread must wait. Depending on the scheduler and the queue order, many other threads may have entered the critical section before this one gets to run. So the condition that it was waiting for may have been rendered false. Using a loop around all condition variable wait operations is the only way to guarantee that the condition being waited for is still true after waiting.

Before a condition variable can be used it must be initialized with a call to `cyg_cond_init`. This requires two arguments, memory for the data structure and a pointer to an existing mutex. This mutex will not be initialized by `cyg_cond_init`, instead a separate call to `cyg_mutex_init` is required. If a condition variable is no longer required and there are no threads waiting on it then `cyg_cond_destroy` can be used.

When a thread needs to wait for a condition to be satisfied it can call `cyg_cond_wait`. The thread must have already locked the mutex that was specified in the `cyg_cond_init` call. This mutex will be unlocked and the current thread will be suspended in an atomic operation. When some other thread performs a signal or broadcast operation the current thread will be woken up and automatically reclaim ownership of the mutex again, allowing it to examine global state and determine whether or not the condition is now satisfied.

The kernel supplies a variant of this function, `cyg_cond_timed_wait`, which can be used to wait on the condition variable or until some number of clock ticks have occurred. The number of ticks is specified as an absolute, not relative tick count, and so in order to wait for a relative number of ticks, the return value of the `cyg_current_time()` function should be added to determine the absolute number of ticks. The mutex will always be reclaimed before `cyg_cond_timed_wait` returns, regardless of whether it was a result of a signal operation or a timeout.

There is no `cyg_cond_trywait` function because this would not serve any purpose. If a thread has locked the mutex and determined that the condition is satisfied, it can just release the mutex and return. There is no need to perform any operation on the condition variable.

When a thread changes shared state that may affect some other thread blocked on a condition variable, it should call either `cyg_cond_signal` or `cyg_cond_broadcast`. These calls do not require ownership of the mutex, but usually the mutex will have been claimed before updating the shared state. A signal operation only wakes up the first thread that is waiting on the condition variable, while a broadcast wakes up all the threads. If there are no threads waiting on the condition variable at the time, then the signal or broadcast will have no effect: past signals are not counted up or remembered in any way. Typically a signal should be used when all threads will check the same condition and at most one thread can continue running. A broadcast should be used if threads check slightly different conditions, or if the change to the global state might allow multiple threads to proceed.

Valid contexts

`cyg_cond_init` is typically called during system initialization but may also be called in thread context. The same applies to `cyg_cond_delete`. `cyg_cond_wait` and `cyg_cond_timedwait` may only be called from thread context since they may block. `cyg_cond_signal` and `cyg_cond_broadcast` may be called from thread or DSR context.

Name

cyg_semaphore_init, cyg_semaphore_destroy, cyg_semaphore_wait, cyg_semaphore_timed_wait, cyg_semaphore_post and cyg_semaphore_peek — Synchronization primitive

Synopsis

#include <cyg/kernel/kapi.h>

```
void cyg_semaphore_init (sem, val);

void cyg_semaphore_destroy (sem);

cyg_bool_t cyg_semaphore_wait (sem);

cyg_bool_t cyg_semaphore_timed_wait (sem, abstime);

cyg_bool_t cyg_semaphore_trywait (sem);

void cyg_semaphore_post (sem);

void cyg_semaphore_peek (sem, val);
```

Description

Counting semaphores are a synchronization primitive that allow threads to wait until an event has occurred. The event may be generated by a producer thread, or by a DSR in response to a hardware interrupt. Associated with each semaphore is an integer counter that keeps track of the number of events that have not yet been processed. If this counter is zero, an attempt by a consumer thread to wait on the semaphore will block until some other thread or a DSR posts a new event to the semaphore. If the counter is greater than zero then an attempt to wait on the semaphore will consume one event, in other words decrement the counter, and return immediately. Posting to a semaphore will wake up the first thread that is currently waiting, which will then resume inside the semaphore wait operation and decrement the counter again.

Another use of semaphores is for certain forms of resource management. The counter would correspond to how many of a certain type of resource are currently available, with threads waiting on the semaphore to claim a resource and posting to release the resource again. In practice condition variables are usually much better suited for operations like this.

`cyg_semaphore_init` is used to initialize a semaphore. It takes two arguments, a pointer to a cyg_sem_t structure and an initial value for the counter. Note that semaphore operations, unlike some other parts of the kernel API, use pointers to data structures rather than handles. This makes it easier to embed semaphores in a larger data structure. The initial counter value can be any number, zero, positive or negative, but typically a value of zero is used to indicate that no events have occurred yet.

`cyg_semaphore_wait` is used by a consumer thread to wait for an event. If the current counter is greater than 0, in other words if the event has already occurred in the past, then the counter will be decremented and the call will return immediately. Otherwise the current thread will be blocked until there is a `cyg_semaphore_post` call.

`cyg_semaphore_post` is called when an event has occurs. This increments the counter and wakes up the first thread waiting on the semaphore (if any). Usually that thread will then continue running inside `cyg_semaphore_wait` and decrement the counter again. However other scenarios are possible. For example the thread calling `cyg_semaphore_post` may be running at high priority, some other thread running at medium priority may be about to call `cyg_semaphore_wait` when it next gets a chance to run, and a low priority thread may be waiting on the semaphore. What will happen is that the current high priority thread continues running until it is descheduled for some reason, then the medium priority thread runs and its call to `cyg_semaphore_wait` succeeds immediately, and later on the low priority thread runs again, discovers a counter value of 0, and blocks until another event is posted. If there are multiple threads blocked on a semaphore then the configuration option `CYGIMP_KERNEL_SCHED_SORTED_QUEUES` determines which one will be woken up by a post operation.

cyg_semaphore_wait returns a boolean. Normally it will block until it has successfully decremented the counter, retrying as necessary, and return success. However the wait operation may be aborted by a call to cyg_thread_release, and cyg_semaphore_wait will then return false.

cyg_semaphore_timed_wait is a variant of cyg_semaphore_wait. It can be used to wait until either an event has occurred or a number of clock ticks have happened. The number of ticks is specified as an absolute, not relative tick count, and so in order to wait for a relative number of ticks, the return value of the cyg_current_time() function should be added to determine the absolute number of ticks. The function returns success if the semaphore wait operation succeeded, or false if the operation timed out or was aborted by cyg_thread_release. If support for the real-time clock has been removed from the current configuration then this function will not be available. cyg_semaphore_trywait is another variant which will always return immediately rather than block, again returning success or failure. If cyg_semaphore_timedwait is given a timeout in the past, it operates like cyg_semaphore_trywait.

cyg_semaphore_peek can be used to get hold of the current counter value. This function is rarely useful except for debugging purposes since the counter value may change at any time if some other thread or a DSR performs a semaphore operation.

Valid contexts

cyg_semaphore_init is normally called during initialization but may also be called from thread context. cyg_semaphore_wait and cyg_semaphore_timed_wait may only be called from thread context because these operations may block. cyg_semaphore_trywait, cyg_semaphore_post and cyg_semaphore_peek may be called from thread or DSR context.

Name

cyg_mbox_create, cyg_mbox_delete, cyg_mbox_get, cyg_mbox_timed_get, cyg_mbox_tryget, cyg_mbox_peek_item, cyg_mbox_put, cyg_mbox_timed_put, cyg_mbox_tryput, cyg_mbox_peek, cyg_mbox_waiting_to_get and cyg_mbox_waiting_to_put — Synchronization primitive

Synopsis

#include <cyg/kernel/kapi.h>

void **cyg_mbox_create** (*handle*, *mbox*);

void **cyg_mbox_delete** (*mbox*);

void* **cyg_mbox_get** (*mbox*);

void* **cyg_mbox_timed_get** (*mbox*, *abstime*);

void* **cyg_mbox_tryget** (*mbox*);

cyg_count32 **cyg_mbox_peek** (*mbox*);

void* **cyg_mbox_peek_item** (*mbox*);

cyg_bool_t **cyg_mbox_put** (*mbox*, *item*);

cyg_bool_t **cyg_mbox_timed_put** (*mbox*, *item*, *abstime*);

cyg_bool_t **cyg_mbox_tryput** (*mbox*, *item*);

cyg_bool_t **cyg_mbox_waiting_to_get** (*mbox*);

cyg_bool_t **cyg_mbox_waiting_to_put** (*mbox*);

Description

Mail boxes are a synchronization primitive. Like semaphores they can be used by a consumer thread to wait until a certain event has occurred, but the producer also has the ability to transmit some data along with each event. This data, the message, is normally a pointer to some data structure. It is stored in the mail box itself, so the producer thread that generates the event and provides the data usually does not have to block until some consumer thread is ready to receive the event. However a mail box will only have a finite capacity, typically ten slots. Even if the system is balanced and events are typically consumed at least as fast as they are generated, a burst of events can cause the mail box to fill up and the generating thread will block until space is available again. This behaviour is very different from semaphores, where it is only necessary to maintain a counter and hence an overflow is unlikely.

Before a mail box can be used it must be created with a call to `cyg_mbox_create`. Each mail box has a unique handle which will be returned via the first argument and which should be used for subsequent operations. `cyg_mbox_create` also requires an area of memory for the kernel structure, which is provided by the cyg_mbox second argument. If a mail box is no longer required then `cyg_mbox_delete` can be used. This will simply discard any messages that remain posted.

The main function for waiting on a mail box is `cyg_mbox_get`. If there is a pending message because of a call to `cyg_mbox_put` then `cyg_mbox_get` will return immediately with the message that was put into the mail box. Otherwise this function will block until there is a put operation. Exceptionally the thread can instead be unblocked by a call to `cyg_thread_release`, in which case `cyg_mbox_get` will return a null pointer. It is assumed that there will never be a call to `cyg_mbox_put` with a null pointer, because it would not be possible to distinguish between that and a release operation. Messages are always retrieved in the order in which they were put into the mail box, and there is no support for messages with different priorities.

There are two variants of cyg_mbox_get. The first, cyg_mbox_timed_get will wait until either a message is available or until a number of clock ticks have occurred. The number of ticks is specified as an absolute, not relative tick count, and so in order to wait for a relative number of ticks, the return value of the cyg_current_time() function should be added to determine the absolute number of ticks. If no message is posted within the timeout then a null pointer will be returned. cyg_mbox_tryget is a non-blocking operation which will either return a message if one is available or a null pointer.

New messages are placed in the mail box by calling cyg_mbox_put or one of its variants. The main put function takes two arguments, a handle to the mail box and a pointer for the message itself. If there is a spare slot in the mail box then the new message can be placed there immediately, and if there is a waiting thread it will be woken up so that it can receive the message. If the mail box is currently full then cyg_mbox_put will block until there has been a get operation and a slot is available. The cyg_mbox_timed_put variant imposes a time limit on the put operation, returning false if the operation cannot be completed within the specified number of clock ticks and as for cyg_mbox_timed_get this is an absolute tick count. The cyg_mbox_tryput variant is non-blocking, returning false if there are no free slots available and the message cannot be posted without blocking.

There are a further four functions available for examining the current state of a mailbox. The results of these functions must be used with care because usually the state can change at any time as a result of activity within other threads, but they may prove occasionally useful during debugging or in special situations. cyg_mbox_peek returns a count of the number of messages currently stored in the mail box. cyg_mbox_peek_item retrieves the first message, but it remains in the mail box until a get operation is performed. cyg_mbox_waiting_to_get and cyg_mbox_waiting_to_put indicate whether or not there are currently threads blocked in a get or a put operation on a given mail box.

The number of slots in each mail box is controlled by a configuration option CYGNUM_KERNEL_SYNCH_MBOX_QUEUE_SIZE, with a default value of 10. All mail boxes are the same size.

Valid contexts

cyg_mbox_create is typically called during system initialization but may also be called in thread context. The remaining functions are normally called only during thread context. Of special note is cyg_mbox_put which can be a blocking operation when the mail box is full, and which therefore must never be called from DSR context. It is permitted to call cyg_mbox_tryput, cyg_mbox_tryget, and the information functions from DSR context but this is rarely useful.

Name

cyg_flag_init, cyg_flag_destroy, cyg_flag_setbits, cyg_flag_maskbits, cyg_flag_wait, cyg_flag_timed_wait, cyg_flag_poll, cyg_flag_peek and cyg_flag_waiting — Synchronization primitive

Synopsis

#include <cyg/kernel/kapi.h>

`void` **`cyg_flag_init`** `(`*`flag`*`);`

`void` **`cyg_flag_destroy`** `(`*`flag`*`);`

`void` **`cyg_flag_setbits`** `(`*`flag`*`,` *`value`*`);`

`void` **`cyg_flag_maskbits`** `(`*`flag`*`,` *`value`*`);`

`cyg_flag_value_t` **`cyg_flag_wait`** `(`*`flag`*`,` *`pattern`*`,` *`mode`*`);`

`cyg_flag_value_t` **`cyg_flag_timed_wait`** `(`*`flag`*`,` *`pattern`*`,` *`mode`*`,` *`abstime`*`);`

`cyg_flag_value_t` **`cyg_flag_poll`** `(`*`flag`*`,` *`pattern`*`,` *`mode`*`);`

`cyg_flag_value_t` **`cyg_flag_peek`** `(`*`flag`*`);`

`cyg_bool_t` **`cyg_flag_waiting`** `(`*`flag`*`);`

Description

Event flags allow a consumer thread to wait for one of several different types of event to occur. Alternatively it is possible to wait for some combination of events. The implementation is relatively straightforward. Each event flag contains a 32-bit integer. Application code associates these bits with specific events, so for example bit 0 could indicate that an I/O operation has completed and data is available, while bit 1 could indicate that the user has pressed a start button. A producer thread or a DSR can cause one or more of the bits to be set, and a consumer thread currently waiting for these bits will be woken up.

Unlike semaphores no attempt is made to keep track of event counts. It does not matter whether a given event occurs once or multiple times before being consumed, the corresponding bit in the event flag will change only once. However semaphores cannot easily be used to handle multiple event sources. Event flags can often be used as an alternative to condition variables, although they cannot be used for completely arbitrary conditions and they only support the equivalent of condition variable broadcasts, not signals.

Before an event flag can be used it must be initialized by a call to `cyg_flag_init`. This takes a pointer to a cyg_flag_t data structure, which can be part of a larger structure. All 32 bits in the event flag will be set to 0, indicating that no events have yet occurred. If an event flag is no longer required it can be cleaned up with a call to `cyg_flag_destroy`, allowing the memory for the *`cyg_flag_t`* structure to be re-used.

A consumer thread can wait for one or more events by calling `cyg_flag_wait`. This takes three arguments. The first identifies a particular event flag. The second is some combination of bits, indicating which events are of interest. The final argument should be one of the following:

`CYG_FLAG_WAITMODE_AND`

> The call to `cyg_flag_wait` will block until all the specified event bits are set. The event flag is not cleared when the wait succeeds, in other words all the bits remain set.

`CYG_FLAG_WAITMODE_OR`

> The call will block until at least one of the specified event bits is set. The event flag is not cleared on return.

CYG_FLAG_WAITMODE_AND | CYG_FLAG_WAITMODE_CLR

The call will block until all the specified event bits are set, and the entire event flag is cleared when the call succeeds. Note that if this mode of operation is used then a single event flag cannot be used to store disjoint sets of events, even though enough bits might be available. Instead each disjoint set of events requires its own event flag.

CYG_FLAG_WAITMODE_OR | CYG_FLAG_WAITMODE_CLR

The call will block until at least one of the specified event bits is set, and the entire flag is cleared when the call succeeds.

A call to `cyg_flag_wait` normally blocks until the required condition is satisfied. It will return the value of the event flag at the point that the operation succeeded, which may be a superset of the requested events. If `cyg_thread_release` is used to unblock a thread that is currently in a wait operation, the `cyg_flag_wait` call will instead return 0.

`cyg_flag_timed_wait` is a variant of `cyg_flag_wait` which adds a timeout: the wait operation must succeed within the specified number of ticks, or it will fail with a return value of 0. The number of ticks is specified as an absolute, not relative tick count, and so in order to wait for a relative number of ticks, the return value of the `cyg_current_time()` function should be added to determine the absolute number of ticks. `cyg_flag_poll` is a non-blocking variant: if the wait operation can succeed immediately it acts like `cyg_flag_wait`, otherwise it returns immediately with a value of 0.

`cyg_flag_setbits` is called by a producer thread or from inside a DSR when an event occurs. The specified bits are or'd into the current event flag value. This may cause one or more waiting threads to be woken up, if their conditions are now satisfied. How many threads are awoken depends on the use of `CYG_FLAG_WAITMODE_CLR`. The queue of threads waiting on the flag is walked to find threads which now have their wake condition fulfilled. If the awoken thread has passed `CYG_FLAG_WAITMODE_CLR` the walking of the queue is terminated, otherwise the walk continues. Thus if no threads have passed `CYG_FLAG_WAITMORE_CLR` all threads with fulfilled conditions will be awoken. If `CYG_FLAG_WAITMODE_CLR` is passed by threads with fulfilled conditions, the number of awoken threads will depend on the order the threads are in the queue.

`cyg_flag_maskbits` can be used to clear one or more bits in the event flag. This can be called from a producer when a particular condition is no longer satisfied, for example when the user is no longer pressing a particular button. It can also be used by a consumer thread if `CYG_FLAG_WAITMODE_CLR` was not used as part of the wait operation, to indicate that some but not all of the active events have been consumed. If there are multiple consumer threads performing wait operations without using `CYG_FLAG_WAITMODE_CLR` then typically some additional synchronization such as a mutex is needed to prevent multiple threads consuming the same event.

Two additional functions are provided to query the current state of an event flag. `cyg_flag_peek` returns the current value of the event flag, and `cyg_flag_waiting` can be used to find out whether or not there are any threads currently blocked on the event flag. Both of these functions must be used with care because other threads may be operating on the event flag.

Valid contexts

`cyg_flag_init` is typically called during system initialization but may also be called in thread context. The same applies to `cyg_flag_destroy`. `cyg_flag_wait` and `cyg_flag_timed_wait` may only be called from thread context. The remaining functions may be called from thread or DSR context.

Name

cyg_spinlock_create, cyg_spinlock_destroy, cyg_spinlock_spin, cyg_spinlock_clear, cyg_spinlock_test, cyg_spinlock_spin_intsave and cyg_spinlock_clear_intsave — Low-level Synchronization Primitive

Synopsis

#include <cyg/kernel/kapi.h>

void **cyg_spinlock_init** (*lock*, *locked*);

void **cyg_spinlock_destroy** (*lock*);

void **cyg_spinlock_spin** (*lock*);

void **cyg_spinlock_clear** (*lock*);

cyg_bool_t **cyg_spinlock_try** (*lock*);

cyg_bool_t **cyg_spinlock_test** (*lock*);

void **cyg_spinlock_spin_intsave** (*lock*, *istate*);

void **cyg_spinlock_clear_intsave** (*lock*, *istate*);

Description

Spinlocks provide an additional synchronization primitive for applications running on SMP systems. They operate at a lower level than the other primitives such as mutexes, and for most purposes the higher-level primitives should be preferred. However there are some circumstances where a spinlock is appropriate, especially when interrupt handlers and threads need to share access to hardware, and on SMP systems the kernel implementation itself depends on spinlocks.

Essentially a spinlock is just a simple flag. When code tries to claim a spinlock it checks whether or not the flag is already set. If not then the flag is set and the operation succeeds immediately. The exact implementation of this is hardware-specific, for example it may use a test-and-set instruction to guarantee the desired behaviour even if several processors try to access the spinlock at the exact same time. If it is not possible to claim a spinlock then the current thread spins in a tight loop, repeatedly checking the flag until it is clear. This behaviour is very different from other synchronization primitives such as mutexes, where contention would cause a thread to be suspended. The assumption is that a spinlock will only be held for a very short time. If claiming a spinlock could cause the current thread to be suspended then spinlocks could not be used inside interrupt handlers, which is not acceptable.

This does impose a constraint on any code which uses spinlocks. Specifically it is important that spinlocks are held only for a short period of time, typically just some dozens of instructions. Otherwise another processor could be blocked on the spinlock for a long time, unable to do any useful work. It is also important that a thread which owns a spinlock does not get preempted because that might cause another processor to spin for a whole timeslice period, or longer. One way of achieving this is to disable interrupts on the current processor, and the function `cyg_spinlock_spin_intsave` is provided to facilitate this.

Spinlocks should not be used on single-processor systems. Consider a high priority thread which attempts to claim a spinlock already held by a lower priority thread: it will just loop forever and the lower priority thread will never get another chance to run and release the spinlock. Even if the two threads were running at the same priority, the one attempting to claim the spinlock would spin until it was timesliced and a lot of CPU time would be wasted. If an interrupt handler tried to claim a spinlock owned by a thread, the interrupt handler would loop forever. Therefore spinlocks are only appropriate for SMP systems where the current owner of a spinlock can continue running on a different processor.

Before a spinlock can be used it must be initialized by a call to `cyg_spinlock_init`. This takes two arguments, a pointer to a `cyg_spinlock_t` data structure, and a flag to specify whether the spinlock starts off locked or unlocked. If a spinlock is no longer required then it can be destroyed by a call to `cyg_spinlock_destroy`.

There are two routines for claiming a spinlock: `cyg_spinlock_spin` and `cyg_spinlock_spin_intsave`. The former can be used when it is known the current code will not be preempted, for example because it is running in an interrupt handler or because interrupts are disabled. The latter will disable interrupts in addition to claiming the spinlock, so is safe to use in all circumstances. The previous interrupt state is returned via the second argument, and should be used in a subsequent call to `cyg_spinlock_clear_intsave`.

Similarly there are two routines for releasing a spinlock: `cyg_spinlock_clear` and `cyg_spinlock_clear_intsave`. Typically the former will be used if the spinlock was claimed by a call to `cyg_spinlock_spin`, and the latter when `cyg_spinlock_intsave` was used.

There are two additional routines. `cyg_spinlock_try` is a non-blocking version of `cyg_spinlock_spin`: if possible the lock will be claimed and the function will return `true`; otherwise the function will return immediately with failure. `cyg_spinlock_test` can be used to find out whether or not the spinlock is currently locked. This function must be used with care because, especially on a multiprocessor system, the state of the spinlock can change at any time.

Spinlocks should only be held for a short period of time, and attempting to claim a spinlock will never cause a thread to be suspended. This means that there is no need to worry about priority inversion problems, and concepts such as priority ceilings and inheritance do not apply.

Valid contexts

All of the spinlock functions can be called from any context, including ISR and DSR context. Typically `cyg_spinlock_init` is only called during system initialization.

Name

cyg_scheduler_start, cyg_scheduler_lock, cyg_scheduler_unlock, cyg_scheduler_safe_lock, cyg_scheduler_read_lock, cyg_thread_lock_preemption, cyg_thread_unlock_preemption and cyg_thread_get_preemption_lock — Control the state of the scheduler

Synopsis

#include <cyg/kernel/kapi.h>

void **cyg_scheduler_start** ();

void **cyg_scheduler_lock** ();

void **cyg_scheduler_unlock** ();

cyg_ucount32 **cyg_scheduler_read_lock** ();

void **cyg_thread_lock_preemption** ();

void **cyg_thread_unlock_preemption** ();

cyg_ucount32 **cyg_thread_get_preemption_lock** ();

Description

`cyg_scheduler_start` should only be called once, to mark the end of system initialization. In typical configurations it is called automatically by the system startup, but some applications may bypass the standard startup in which case `cyg_scheduler_start` will have to be called explicitly. The call will enable system interrupts, allowing I/O operations to commence. Then the scheduler will be invoked and control will be transferred to the highest priority runnable thread. The call will never return.

The various data structures inside the eCos kernel must be protected against concurrent updates. Consider a call to `cyg_semaphore_post` which causes a thread to be woken up: the semaphore data structure must be updated to remove the thread from its queue; the scheduler data structure must also be updated to mark the thread as runnable; it is possible that the newly runnable thread has a higher priority than the current one, in which case preemption is required. If in the middle of the semaphore post call an interrupt occurred and the interrupt handler tried to manipulate the same data structures, for example by making another thread runnable, then it is likely that the structures will be left in an inconsistent state and the system will fail.

To prevent such problems the kernel contains a special lock known as the scheduler lock. A typical kernel function such as `cyg_semaphore_post` will claim the scheduler lock, do all its manipulation of kernel data structures, and then release the scheduler lock. The current thread cannot be preempted while it holds the scheduler lock. If an interrupt occurs and a DSR is supposed to run to signal that some event has occurred, that DSR is postponed until the scheduler unlock operation. This prevents concurrent updates of kernel data structures.

The kernel exports three routines for manipulating the scheduler lock. `cyg_scheduler_lock` can be called to claim the lock. On return it is guaranteed that the current thread will not be preempted, and that no other code is manipulating any kernel data structures. `cyg_scheduler_unlock` can be used to release the lock, which may cause the current thread to be preempted. `cyg_scheduler_read_lock` can be used to query the current state of the scheduler lock. This function should never be needed because well-written code should always know whether or not the scheduler is currently locked, but may prove useful during debugging.

The implementation of the scheduler lock involves a simple counter. Code can call `cyg_scheduler_lock` multiple times, causing the counter to be incremented each time, as long as `cyg_scheduler_unlock` is called the same number of times. This behaviour is different from mutexes where an attempt by a thread to lock a mutex multiple times will result in deadlock or an assertion failure.

Typical application code should not use the scheduler lock. Instead other synchronization primitives such as mutexes and semaphores should be used. While the scheduler is locked the current thread cannot be preempted, so any higher priority threads will not be able to run. Also no DSRs can run, so device drivers may not be able to service I/O requests. However there is one situation where locking the scheduler is appropriate: if some data structure needs to be shared between an application thread and a DSR associated with some interrupt source, the thread can use the scheduler lock to prevent concurrent invocations of the DSR and then safely manipulate the structure. It is desirable that the scheduler lock is held for only a short period of time, typically some tens of instructions. In exceptional cases there may also be some performance-critical code where it is more appropriate to use the scheduler lock rather than a mutex, because the former is more efficient.

Valid contexts

`cyg_scheduler_start` can only be called during system initialization, since it marks the end of that phase. The remaining functions may be called from thread or DSR context. Locking the scheduler from inside the DSR has no practical effect because the lock is claimed automatically by the interrupt subsystem before running DSRs, but allows functions to be shared between normal thread code and DSRs.

Pre-emption lock

Separate but similar to the scheduler lock is the thread pre-emption lock. This provides a more light-weight method of preventing pre-emption by higher priority threads. So even if a higher priority thread becomes runnable (on this CPU, if SMP) the current thread will not be descheduled until the pre-emption lock has been released. While the pre-emption lock is held, DSRs (and ISRs) are still permitted to run.

Like the scheduler lock, the pre-emption lock is a count and so it can be called multiple times, and only when the final level is unlocked will pre-emption become possible again. A thread should only invoke the lock for itself, not on another thread, as the lock is not protected from access by multiple threads. If a thread blocks or goes to sleep, then it will be descheduled in the normal way and another thread can run. Although, once it is rescheduled, its pre-emption lock state will be preserved; so if pre-emption was disabled before, it will become disabled again when rescheduled. In an SMP system, locking pre-emption will only affect operations on the current CPU, it will not affect threads on other CPUs.

If timeslicing is enabled and the current thread has locked pre-emption, then even if the thread has run out of time in its timeslice, it will still not be descheduled. Instead, it will run until it unlocks pre-emption.

Clearly, as with the scheduler lock, the pre-emption lock can severely affect proper real-time operation and so it should be avoided if other synchronization methods, such as mutexes, semaphores etc. can be used instead. If it is used, the length of time pre-emption is disabled should be kept to a minimum.

The availability of pre-emption locking can be controlled with the CDL configuration option `CYGFUN_KERNEL_THREAD_PREEMPTION_LOCK`, which defaults to enabled.

Name

cyg_interrupt_create, cyg_interrupt_delete, cyg_interrupt_attach, cyg_interrupt_detach, cyg_interrupt_configure, cyg_interrupt_acknowledge, cyg_interrupt_enable, cyg_interrupt_disable, cyg_interrupt_dsr_count, cyg_interrupt_mask, cyg_interrupt_mask_intunsafe, cyg_interrupt_unmask, cyg_interrupt_unmask_intunsafe, cyg_interrupt_set_cpu, cyg_interrupt_get_cpu, cyg_interrupt_get_vsr and cyg_interrupt_set_vsr — Manage interrupt handlers

Synopsis

#include <cyg/kernel/kapi.h>

`void` **`cyg_interrupt_create`** `(`*`vector`*`,` *`priority`*`,` *`data`*`,` *`isr`*`,` *`dsr`*`,` *`handle`*`,` *`intr`*`);`

`void` **`cyg_interrupt_delete`** `(`*`interrupt`*`);`

`void` **`cyg_interrupt_attach`** `(`*`interrupt`*`);`

`void` **`cyg_interrupt_detach`** `(`*`interrupt`*`);`

`void` **`cyg_interrupt_configure`** `(`*`vector`*`,` *`level`*`,` *`up`*`);`

`void` **`cyg_interrupt_acknowledge`** `(`*`vector`*`);`

`void` **`cyg_interrupt_disable`** `();`

`void` **`cyg_interrupt_enable`** `();`

`cyg_ucount32` **`cyg_interrupt_dsr_count`** `(`*`interrupt`*`);`

`void` **`cyg_interrupt_mask`** `(`*`vector`*`);`

`void` **`cyg_interrupt_mask_intunsafe`** `(`*`vector`*`);`

`void` **`cyg_interrupt_unmask`** `(`*`vector`*`);`

`void` **`cyg_interrupt_unmask_intunsafe`** `(`*`vector`*`);`

`void` **`cyg_interrupt_set_cpu`** `(`*`vector`*`,` *`cpu`*`);`

`cyg_cpu_t` **`cyg_interrupt_get_cpu`** `(`*`vector`*`);`

`void` **`cyg_interrupt_get_vsr`** `(`*`vector`*`,` *`vsr`*`);`

`void` **`cyg_interrupt_set_vsr`** `(`*`vector`*`,` *`vsr`*`);`

Description

The kernel provides an interface for installing interrupt handlers and controlling when interrupts occur. This functionality is used primarily by eCos device drivers and by any application code that interacts directly with hardware. However in most cases it is better to avoid using this kernel functionality directly, and instead the device driver API provided by the common HAL package should be used. Use of the kernel package is optional, and some applications such as RedBoot work with no need for multiple threads or synchronization primitives. Any code which calls the kernel directly rather than the device driver API will not function in such a configuration. When the kernel package is present the device driver API is implemented as `#define`'s to the equivalent kernel calls, otherwise it is implemented inside the common HAL package. The latter implementation can be simpler than the kernel one because there is no need to consider thread preemption and similar issues.

The exact details of interrupt handling vary widely between architectures. The functionality provided by the kernel abstracts away from many of the details of the underlying hardware, thus simplifying application development. However this is not always successful. For example, if some hardware does not provide any support at all for masking specific interrupts then calling `cyg_interrupt_mask` may not behave as intended: instead of masking just the one interrupt source it might disable all interrupts, because that is as close to the desired behaviour as is possible given the hardware restrictions. Another possibility is that masking a given interrupt source also affects all lower-priority interrupts, but still allows higher-priority ones. The documentation for the appropriate HAL packages should be consulted for more information about exactly how interrupts are handled on any given hardware. The HAL header files will also contain useful information.

Interrupt Handlers

Interrupt handlers are created by a call to `cyg_interrupt_create`. This takes the following arguments:

cyg_vector_t *`vector`*

The interrupt vector, a small integer, identifies the specific interrupt source. The appropriate hardware documentation or HAL header files should be consulted for details of which vector corresponds to which device.

cyg_priority_t *`priority`*

Some hardware may support interrupt priorities, where a low priority interrupt handler can in turn be interrupted by a higher priority one. Again hardware-specific documentation should be consulted for details about what the valid interrupt priority levels are.

cyg_addrword_t *`data`*

When an interrupt occurs eCos will first call the associated interrupt service routine or ISR, then optionally a deferred service routine or DSR. The *`data`* argument to `cyg_interrupt_create` will be passed to both these functions. Typically it will be a pointer to some data structure.

cyg_ISR_t *`isr`*

When an interrupt occurs the hardware will transfer control to the appropriate vector service routine or VSR, which is usually provided by eCos. This performs any appropriate processing, for example to work out exactly which interrupt occurred, and then as quickly as possible transfers control the installed ISR. An ISR is a C function which takes the following form:

```
cyg_uint32
isr_function(cyg_vector_t vector, cyg_addrword_t data)
{
    cyg_bool_t dsr_required = 0;

    …

    return dsr_required ?
        (CYG_ISR_CALL_DSR | CYG_ISR_HANDLED) :
        CYG_ISR_HANDLED;
}
```

The first argument identifies the particular interrupt source, especially useful if there multiple instances of a given device and a single ISR can be used for several different interrupt vectors. The second argument is the *`data`* field passed to `cyg_interrupt_create`, usually a pointer to some data structure. The exact conditions under which an ISR runs will depend partly on the hardware and partly on configuration options. Interrupts may currently be disabled globally, especially if the hardware does not support interrupt priorities. Alternatively interrupts may be enabled such that higher priority interrupts are allowed through. The ISR may be running on a separate interrupt stack, or on the stack of whichever thread was running at the time the interrupt happened.

A typical ISR will do as little work as possible, just enough to meet the needs of the hardware and then acknowledge the interrupt by calling `cyg_interrupt_acknowledge`. This ensures that interrupts will be quickly reenabled, so higher priority devices can be serviced. For some applications there may be one device which is especially important and whose ISR can take much longer than normal. However eCos device drivers usually will not assume that they are especially important, so their ISRs will be as short as possible.

The return value of an ISR is normally a bit mask containing zero, one or both of the following bits: `CYG_ISR_CALL_DSR` or `CYG_ISR_HANDLED`. The former indicates that further processing is required at DSR level, and the interrupt handler's DSR will be run as soon as possible. The latter indicates that the interrupt was handled by this ISR so there is no need to call other interrupt handlers which might be chained on this interrupt vector. If this ISR did not handle the interrupt it should not set the CYG_ISR_HANDLED bit so that other chained interrupt handlers may handle the interrupt.

An ISR is allowed to make very few kernel calls. It can manipulate the interrupt mask, and on SMP systems it can use spinlocks. However an ISR must not make higher-level kernel calls such as posting to a semaphore, instead any such calls must be made from the DSR. This avoids having to disable interrupts throughout the kernel and thus improves interrupt latency.

cyg_DSR_t *`dsr`*

If an interrupt has occurred and the ISR has returned a value with `CYG_ISR_CALL_DSR` bit being set, the system will call the DSR associated with this interrupt handler. If the scheduler is not currently locked then the DSR will run immediately. However if the interrupted thread was in the middle of a kernel call and had locked the scheduler, then the DSR will be deferred until the scheduler is again unlocked. This allows the DSR to make certain kernel calls safely, for example posting to a semaphore or signalling a condition variable. A DSR is a C function which takes the following form:

```
void
dsr_function(cyg_vector_t vector,
    cyg_ucount32 count,
    cyg_addrword_t data)
{
    …
}
```

The first argument identifies the specific interrupt that has caused the DSR to run. The second argument indicates the number of these interrupts that have occurred and for which the ISR requested a DSR. Usually this will be `1`, unless the system is suffering from a very heavy load. The third argument is the *`data`* field passed to `cyg_interrupt_create`.

cyg_handle_t* *`handle`*

The kernel will return a handle to the newly created interrupt handler via this argument. Subsequent operations on the interrupt handler such as attaching it to the interrupt source will use this handle.

cyg_interrupt* *`intr`*

This provides the kernel with an area of memory for holding this interrupt handler and associated data.

The call to `cyg_interrupt_create` simply fills in a kernel data structure. A typical next step is to call `cyg_interrupt_attach` using the handle returned by the create operation. This makes it possible to have several different interrupt handlers for a given vector, attaching whichever one is currently appropriate. Replacing an interrupt handler requires a call to `cyg_interrupt_detach`, followed by another call to `cyg_interrupt_attach` for the replacement handler. `cyg_interrupt_delete` can be used if an interrupt handler is no longer required.

Some hardware may allow for further control over specific interrupts, for example whether an interrupt is level or edge triggered. Any such hardware functionality can be accessed using `cyg_interrupt_configure`: the *`level`* argument selects between level versus edge triggered; the *`up`* argument selects between high and low level, or between rising and falling edges.

Usually interrupt handlers are created, attached and configured during system initialization, while global interrupts are still disabled. On most hardware it will also be necessary to call `cyg_interrupt_unmask`, since the sensible default for interrupt masking is to ignore any interrupts for which no handler is installed.

Controlling Interrupts

eCos provides two ways of controlling whether or not interrupts happen. It is possible to disable and reenable all interrupts globally, using `cyg_interrupt_disable` and `cyg_interrupt_enable`. Typically this works by manipulating state inside the

CPU itself, for example setting a flag in a status register or executing special instructions. Alternatively it may be possible to mask a specific interrupt source by writing to one or to several interrupt mask registers. Hardware-specific documentation should be consulted for the exact details of how interrupt masking works, because a full implementation is not possible on all hardware.

The primary use for these functions is to allow data to be shared between ISRs and other code such as DSRs or threads. If both a thread and an ISR need to manipulate either a data structure or the hardware itself, there is a possible conflict if an interrupt happens just when the thread is doing such manipulation. Problems can be avoided by the thread either disabling or masking interrupts during the critical region. If this critical region requires only a few instructions then usually it is more efficient to disable interrupts. For larger critical regions it may be more appropriate to use interrupt masking, allowing other interrupts to occur. There are other uses for interrupt masking. For example if a device is not currently being used by the application then it may be desirable to mask all interrupts generated by that device.

There are two functions for masking a specific interrupt source, `cyg_interrupt_mask` and `cyg_interrupt_mask_intunsafe`. On typical hardware masking an interrupt is not an atomic operation, so if two threads were to perform interrupt masking operations at the same time there could be problems. `cyg_interrupt_mask` disables all interrupts while it manipulates the interrupt mask. In situations where interrupts are already known to be disabled, `cyg_interrupt_mask_intunsafe` can be used instead. There are matching functions `cyg_interrupt_unmask` and `cyg_interrupt_unmask_intsafe`.

If an interrupt handler is no longer required, it can be deleted from the interrupt system with `cyg_interrupt_delete`, but it is up to the user to ensure that the interrupt source can no longer be generating interrupts, and there are no as-yet-unhandled pending interrupts or DSRs. Calling `cyg_interrupt_mask` before `cyg_interrupt_delete` will be sufficient to ensure that no more interrupts are delivered. The interrupt may be checked for pending DSRs by calling `cyg_interrupt_dsr_count`, which will return a non-zero result if there are DSRs pending. The application can cause pending DSRs to be delivered by making a kernel call, for example to `cyg_thread_yield`. The following code example shows the sequence that might be used to delete an interrupt handler:

```
        cyg_interrupt_mask( vector );
        while( cyg_interrupt_dsr_count( interrupt ) )
        {
            cyg_thread_yield();
        }
        cyg_interrupt_delete( interrupt );
```

If an interrupt handler is deleted but the interrupt is subsequently raised and is not masked, then the HAL will treat this as a spurious interrupt which, depending on the HAL and the configuration, may result in an assertion failure, an exception or it may simply be ignored albeit wasting the CPU resources to handle the interrupt.

SMP Support

On SMP systems the kernel provides an additional two functions related to interrupt handling. `cyg_interrupt_set_cpu` specifies that a particular hardware interrupt should always be handled on a specified set of processors in the system. In other words when the interrupt triggers it is only one of those processors which detects it, and it is only on those processors that the VSR and ISR will run. If a DSR is requested then it will also run on the same CPU. The function `cyg_interrupt_get_cpu` can be used to find out which interrupts are handled on which processors.

VSR Support

When an interrupt occurs the hardware will transfer control to a piece of code known as the VSR, or Vector Service Routine. By default this code is provided by eCos. Usually it is written in assembler, but on some architectures it may be possible to implement VSRs in C by specifying an interrupt attribute. Compiler documentation should be consulted for more information on this. The default eCos VSR will work out which ISR function should process the interrupt, and set up a C environment suitable for this ISR.

For some applications it may be desirable to replace the default eCos VSR and handle some interrupts directly. This minimizes interrupt latency, but it requires application developers to program at a lower level. Usually the best way to write a custom VSR is to copy the existing one supplied by eCos and then make appropriate modifications. The function `cyg_interrupt_get_vsr` can be used to get hold of the current VSR for a given interrupt vector, allowing it to be restored if the custom VSR is no longer required. `cyg_interrupt_set_vsr` can be used to install a replacement VSR. Usually the *`vsr`* argument will correspond to an exported label in an assembler source file.

Note

On some eCos platforms, possibly only in certain configurations, the table of VSRs resides in read-only memory and `cyg_interrupt_set_vsr` will not be available. Portable code can test for this condition by including the header file `cyg/hal/hal_intr.h` and testing for the macro `HAL_VSR_SET`.

Valid contexts

In a typical configuration interrupt handlers are created and attached during system initialization, and never detached or deleted. However it is possible to perform these operations at thread level, if desired. Similarly `cyg_interrupt_configure`, `cyg_interrupt_set_vsr`, and `cyg_interrupt_set_cpu` are usually called only during system initialization, but on typical hardware may be called at any time. `cyg_interrupt_get_vsr` and `cyg_interrupt_get_cpu` may be called at any time.

The functions for enabling, disabling, masking and unmasking interrupts can be called in any context, when appropriate. It is the responsibility of application developers to determine when the use of these functions is appropriate.

Name

tm_basic — Measure the performance of the eCos kernel

Description

When building a real-time system, care must be taken to ensure that the system will be able to perform properly within the constraints of that system. One of these constraints may be how fast certain operations can be performed. Another might be how deterministic the overall behavior of the system is. Lastly the memory footprint (size) and unit cost may be important.

One of the major problems encountered while evaluating a system will be how to compare it with possible alternatives. Most manufacturers of real-time systems publish performance numbers, ostensibly so that users can compare the different offerings. However, what these numbers mean and how they were gathered is often not clear. The values are typically measured on a particular piece of hardware, so in order to truly compare, one must obtain measurements for exactly the same set of hardware that were gathered in a similar fashion.

Two major items need to be present in any given set of measurements. First, the raw values for the various operations; these are typically quite easy to measure and will be available for most systems. Second, the determinacy of the numbers; in other words how much the value might change depending on other factors within the system. This value is affected by a number of factors: how long interrupts might be masked, whether or not the function can be interrupted, even very hardware-specific effects such as cache locality and pipeline usage. It is very difficult to measure the determinacy of any given operation, but that determinacy is fundamentally important to proper overall characterization of a system.

In the discussion and numbers that follow, three key measurements are provided. The first measurement is an estimate of the interrupt latency: this is the length of time from when a hardware interrupt occurs until its Interrupt Service Routine (ISR) is called. The second measurement is an estimate of overall interrupt overhead: this is the length of time average interrupt processing takes, as measured by the real-time clock interrupt (other interrupt sources will certainly take a different amount of time, but this data cannot be easily gathered). The third measurement consists of the timings for the various kernel primitives.

Methodology

Key operations in the kernel were measured by using a simple test program which exercises the various kernel primitive operations. A hardware timer, normally the one used to drive the real-time clock, was used for these measurements. In most cases this timer can be read with quite high resolution, typically in the range of a few microseconds. For each measurement, the operation was repeated a number of times. Time stamps were obtained directly before and after the operation was performed. The data gathered for the entire set of operations was then analyzed, generating average (mean), maximum and minimum values. The sample variance (a measure of how close most samples are to the mean) was also calculated. The cost of obtaining the real-time clock timer values was also measured, and was subtracted from all other times.

Most kernel functions can be measured separately. In each case, a reasonable number of iterations are performed. Where the test case involves a kernel object, for example creating a task, each iteration is performed on a different object. There is also a set of tests which measures the interactions between multiple tasks and certain kernel primitives. Most functions are tested in such a way as to determine the variations introduced by varying numbers of objects in the system. For example, the mailbox tests measure the cost of a 'peek' operation when the mailbox is empty, has a single item, and has multiple items present. In this way, any effects of the state of the object or how many items it contains can be determined.

There are a few things to consider about these measurements. Firstly, they are quite micro in scale and only measure the operation in question. These measurements do not adequately describe how the timings would be perturbed in a real system with multiple interrupting sources. Secondly, the possible aberration incurred by the real-time clock (system heartbeat tick) is explicitly avoided. Virtually all kernel functions have been designed to be interruptible. Thus the times presented are typical, but best case, since any particular function may be interrupted by the clock tick processing. This number is explicitly calculated so that the value may be included in any deadline calculations required by the end user. Lastly, the reported measurements were obtained from a system built with all options at their default values. Kernel instrumentation and asserts are also disabled for these measurements. Any number of configuration options can change the measured results, sometimes quite dramatically. For example, mutexes are using priority inheritance in these measurements. The numbers will change if the system is built with priority inheritance on mutex variables turned off.

The final value that is measured is an estimate of interrupt latency. This particular value is not explicitly calculated in the test program used, but rather by instrumenting the kernel itself. The raw number of timer ticks that elapse between the time the timer generates an interrupt and the start of the timer ISR is kept in the kernel. These values are printed by the test program after all other operations have been tested. Thus this should be a reasonable estimate of the interrupt latency over time.

Using these Measurements

These measurements can be used in a number of ways. The most typical use will be to compare different real-time kernel offerings on similar hardware, another will be to estimate the cost of implementing a task using eCos (applications can be examined to see what effect the kernel operations will have on the total execution time). Another use would be to observe how the tuning of the kernel affects overall operation.

Influences on Performance

A number of factors can affect real-time performance in a system. One of the most common factors, yet most difficult to characterize, is the effect of device drivers and interrupts on system timings. Different device drivers will have differing requirements as to how long interrupts are suppressed, for example. The eCos system has been designed with this in mind, by separating the management of interrupts (ISR handlers) and the processing required by the interrupt (DSR—Deferred Service Routine— handlers). However, since there is so much variability here, and indeed most device drivers will come from the end users themselves, these effects cannot be reliably measured. Attempts have been made to measure the overhead of the single interrupt that eCos relies on, the real-time clock timer. This should give you a reasonable idea of the cost of executing interrupt handling for devices.

Measured Items

This section describes the various tests and the numbers presented. All tests use the C kernel API (available by way of `cyg/kernel/kapi.h`). There is a single main thread in the system that performs the various tests. Additional threads may be created as part of the testing, but these are short lived and are destroyed between tests unless otherwise noted. The terminology “lower priority” means a priority that is less important, not necessarily lower in numerical value. A higher priority thread will run in preference to a lower priority thread even though the priority value of the higher priority thread may be numerically less than that of the lower priority thread.

Thread Primitives

Create thread

> This test measures the `cyg_thread_create()` call. Each call creates a totally new thread. The set of threads created by this test will be reused in the subsequent thread primitive tests.

Yield thread

> This test measures the `cyg_thread_yield()` call. For this test, there are no other runnable threads, thus the test should just measure the overhead of trying to give up the CPU.

Suspend [suspended] thread

> This test measures the `cyg_thread_suspend()` call. A thread may be suspended multiple times; each thread is already suspended from its initial creation, and is suspended again.

Resume thread

> This test measures the `cyg_thread_resume()` call. All of the threads have a suspend count of 2, thus this call does not make them runnable. This test just measures the overhead of resuming a thread.

Set priority

This test measures the `cyg_thread_set_priority()` call. Each thread, currently suspended, has its priority set to a new value.

Get priority

This test measures the `cyg_thread_get_priority()` call.

Kill [suspended] thread

This test measures the `cyg_thread_kill()` call. Each thread in the set is killed. All threads are known to be suspended before being killed.

Yield [no other] thread

This test measures the `cyg_thread_yield()` call again. This is to demonstrate that the `cyg_thread_yield()` call has a fixed overhead, regardless of whether there are other threads in the system.

Resume [suspended low priority] thread

This test measures the `cyg_thread_resume()` call again. In this case, the thread being resumed is lower priority than the main thread, thus it will simply become ready to run but not be granted the CPU. This test measures the cost of making a thread ready to run.

Resume [runnable low priority] thread

This test measures the `cyg_thread_resume()` call again. In this case, the thread being resumed is lower priority than the main thread and has already been made runnable, so in fact the resume call has no effect.

Suspend [runnable] thread

This test measures the `cyg_thread_suspend()` call again. In this case, each thread has already been made runnable (by previous tests).

Yield [only low priority] thread

This test measures the `cyg_thread_yield()` call. In this case, there are many other runnable threads, but they are all lower priority than the main thread, thus no thread switches will take place.

Suspend [runnable->not runnable] thread

This test measures the `cyg_thread_suspend()` call again. The thread being suspended will become non-runnable by this action.

Kill [runnable] thread

This test measures the `cyg_thread_kill()` call again. In this case, the thread being killed is currently runnable, but lower priority than the main thread.

Resume [high priority] thread

This test measures the `cyg_thread_resume()` call. The thread being resumed is higher priority than the main thread, thus a thread switch will take place on each call. In fact there will be two thread switches; one to the new higher priority thread and a second back to the test thread. The test thread exits immediately.

Thread switch

This test attempts to measure the cost of switching from one thread to another. Two equal priority threads are started and they will each yield to the other for a number of iterations. A time stamp is gathered in one thread before the `cyg_thread_yield()` call and after the call in the other thread.

Scheduler Primitives

Scheduler lock

This test measures the `cyg_scheduler_lock()` call.

Scheduler unlock [0 threads]

This test measures the `cyg_scheduler_unlock()` call. There are no other threads in the system and the unlock happens immediately after a lock so there will be no pending DSR,s to run.

Scheduler unlock [1 suspended thread]

This test measures the `cyg_scheduler_unlock()` call. There is one other thread in the system which is currently suspended.

Scheduler unlock [many suspended threads]

This test measures the `cyg_scheduler_unlock()` call. There are many other threads in the system which are currently suspended. The purpose of this test is to determine the cost of having additional threads in the system when the scheduler is activated by way of `cyg_scheduler_unlock()`.

Scheduler unlock [many low priority threads]

This test measures the `cyg_scheduler_unlock()` call. There are many other threads in the system which are runnable but are lower priority than the main thread. The purpose of this test is to determine the cost of having additional threads in the system when the scheduler is activated by way of `cyg_scheduler_unlock()`.

Mutex Primitives

Init mutex

This test measures the `cyg_mutex_init()` call. A number of separate mutex variables are created. The purpose of this test is to measure the cost of creating a new mutex and introducing it to the system.

Lock [unlocked] mutex

This test measures the `cyg_mutex_lock()` call. The purpose of this test is to measure the cost of locking a mutex which is currently unlocked. There are no other threads executing in the system while this test runs.

Unlock [locked] mutex

This test measures the `cyg_mutex_unlock()` call. The purpose of this test is to measure the cost of unlocking a mutex which is currently locked. There are no other threads executing in the system while this test runs.

Trylock [unlocked] mutex

This test measures the `cyg_mutex_trylock()` call. The purpose of this test is to measure the cost of locking a mutex which is currently unlocked. There are no other threads executing in the system while this test runs.

Trylock [locked] mutex

This test measures the `cyg_mutex_trylock()` call. The purpose of this test is to measure the cost of locking a mutex which is currently locked. There are no other threads executing in the system while this test runs.

Destroy mutex

This test measures the `cyg_mutex_destroy()` call. The purpose of this test is to measure the cost of deleting a mutex from the system. There are no other threads executing in the system while this test runs.

Unlock/Lock mutex

This test attempts to measure the cost of unlocking a mutex for which there is another higher priority thread waiting. When the mutex is unlocked, the higher priority waiting thread will immediately take the lock. The time from when the unlock is issued until after the lock succeeds in the second thread is measured, thus giving the round-trip or circuit time for this type of synchronizer.

Mailbox Primitives

Create mbox

This test measures the `cyg_mbox_create()` call. A number of separate mailboxes is created. The purpose of this test is to measure the cost of creating a new mailbox and introducing it to the system.

Peek [empty] mbox

This test measures the `cyg_mbox_peek()` call. An attempt is made to peek the value in each mailbox, which is currently empty. The purpose of this test is to measure the cost of checking a mailbox for a value without blocking.

Put [first] mbox

This test measures the `cyg_mbox_put()` call. One item is added to a currently empty mailbox. The purpose of this test is to measure the cost of adding an item to a mailbox. There are no other threads currently waiting for mailbox items to arrive.

Peek [1 msg] mbox

This test measures the `cyg_mbox_peek()` call. An attempt is made to peek the value in each mailbox, which contains a single item. The purpose of this test is to measure the cost of checking a mailbox which has data to deliver.

Put [second] mbox

This test measures the `cyg_mbox_put()` call. A second item is added to a mailbox. The purpose of this test is to measure the cost of adding an additional item to a mailbox. There are no other threads currently waiting for mailbox items to arrive.

Peek [2 msgs] mbox

This test measures the `cyg_mbox_peek()` call. An attempt is made to peek the value in each mailbox, which contains two items. The purpose of this test is to measure the cost of checking a mailbox which has data to deliver.

Get [first] mbox

This test measures the `cyg_mbox_get()` call. The first item is removed from a mailbox that currently contains two items. The purpose of this test is to measure the cost of obtaining an item from a mailbox without blocking.

Get [second] mbox

This test measures the `cyg_mbox_get()` call. The last item is removed from a mailbox that currently contains one item. The purpose of this test is to measure the cost of obtaining an item from a mailbox without blocking.

Tryput [first] mbox

This test measures the `cyg_mbox_tryput()` call. A single item is added to a currently empty mailbox. The purpose of this test is to measure the cost of adding an item to a mailbox.

Peek item [non-empty] mbox

This test measures the `cyg_mbox_peek_item()` call. A single item is fetched from a mailbox that contains a single item. The purpose of this test is to measure the cost of obtaining an item without disturbing the mailbox.

Tryget [non-empty] mbox

This test measures the `cyg_mbox_tryget()` call. A single item is removed from a mailbox that contains exactly one item. The purpose of this test is to measure the cost of obtaining one item from a non-empty mailbox.

Peek item [empty] mbox

This test measures the `cyg_mbox_peek_item()` call. An attempt is made to fetch an item from a mailbox that is empty. The purpose of this test is to measure the cost of trying to obtain an item when the mailbox is empty.

Tryget [empty] mbox

This test measures the `cyg_mbox_tryget()` call. An attempt is made to fetch an item from a mailbox that is empty. The purpose of this test is to measure the cost of trying to obtain an item when the mailbox is empty.

Waiting to get mbox

This test measures the `cyg_mbox_waiting_to_get()` call. The purpose of this test is to measure the cost of determining how many threads are waiting to obtain a message from this mailbox.

Waiting to put mbox

This test measures the `cyg_mbox_waiting_to_put()` call. The purpose of this test is to measure the cost of determining how many threads are waiting to put a message into this mailbox.

Delete mbox

This test measures the `cyg_mbox_delete()` call. The purpose of this test is to measure the cost of destroying a mailbox and removing it from the system.

Put/Get mbox

In this round-trip test, one thread is sending data to a mailbox that is being consumed by another thread. The time from when the data is put into the mailbox until it has been delivered to the waiting thread is measured. Note that this time will contain a thread switch.

Semaphore Primitives

Init semaphore

This test measures the `cyg_semaphore_init()` call. A number of separate semaphore objects are created and introduced to the system. The purpose of this test is to measure the cost of creating a new semaphore.

Post [0] semaphore

This test measures the `cyg_semaphore_post()` call. Each semaphore currently has a value of 0 and there are no other threads in the system. The purpose of this test is to measure the overhead cost of posting to a semaphore. This cost will differ if there is a thread waiting for the semaphore.

Wait [1] semaphore

This test measures the `cyg_semaphore_wait()` call. The semaphore has a current value of 1 so the call is non-blocking. The purpose of the test is to measure the overhead of “taking” a semaphore.

Trywait [0] semaphore

This test measures the `cyg_semaphore_trywait()` call. The semaphore has a value of 0 when the call is made. The purpose of this test is to measure the cost of seeing if a semaphore can be “taken” without blocking. In this case, the answer would be no.

Trywait [1] semaphore

This test measures the `cyg_semaphore_trywait()` call. The semaphore has a value of 1 when the call is made. The purpose of this test is to measure the cost of seeing if a semaphore can be “taken” without blocking. In this case, the answer would be yes.

Peek semaphore

This test measures the `cyg_semaphore_peek()` call. The purpose of this test is to measure the cost of obtaining the current semaphore count value.

Destroy semaphore

This test measures the `cyg_semaphore_destroy()` call. The purpose of this test is to measure the cost of deleting a semaphore from the system.

Post/Wait semaphore

In this round-trip test, two threads are passing control back and forth by using a semaphore. The time from when one thread calls `cyg_semaphore_post()` until the other thread completes its `cyg_semaphore_wait()` is measured. Note that each iteration of this test will involve a thread switch.

Counters

Create counter

This test measures the `cyg_counter_create()` call. A number of separate counters are created. The purpose of this test is to measure the cost of creating a new counter and introducing it to the system.

Get counter value

This test measures the `cyg_counter_current_value()` call. The current value of each counter is obtained.

Set counter value

This test measures the `cyg_counter_set_value()` call. Each counter is set to a new value.

Tick counter

This test measures the `cyg_counter_tick()` call. Each counter is “ticked” once.

Delete counter

This test measures the `cyg_counter_delete()` call. Each counter is deleted from the system. The purpose of this test is to measure the cost of deleting a counter object.

Alarms

Create alarm

> This test measures the `cyg_alarm_create()` call. A number of separate alarms are created, all attached to the same counter object. The purpose of this test is to measure the cost of creating a new counter and introducing it to the system.

Initialize alarm

> This test measures the `cyg_alarm_initialize()` call. Each alarm is initialized to a small value.

Disable alarm

> This test measures the `cyg_alarm_disable()` call. Each alarm is explicitly disabled.

Enable alarm

> This test measures the `cyg_alarm_enable()` call. Each alarm is explicitly enabled.

Delete alarm

> This test measures the `cyg_alarm_delete()` call. Each alarm is destroyed. The purpose of this test is to measure the cost of deleting an alarm and removing it from the system.

Tick counter [1 alarm]

> This test measures the `cyg_counter_tick()` call. A counter is created that has a single alarm attached to it. The purpose of this test is to measure the cost of "ticking" a counter when it has a single attached alarm. In this test, the alarm is not activated (fired).

Tick counter [many alarms]

> This test measures the `cyg_counter_tick()` call. A counter is created that has multiple alarms attached to it. The purpose of this test is to measure the cost of "ticking" a counter when it has many attached alarms. In this test, the alarms are not activated (fired).

Tick & fire counter [1 alarm]

> This test measures the `cyg_counter_tick()` call. A counter is created that has a single alarm attached to it. The purpose of this test is to measure the cost of "ticking" a counter when it has a single attached alarm. In this test, the alarm is activated (fired). Thus the measured time will include the overhead of calling the alarm callback function.

Tick & fire counter [many alarms]

> This test measures the `cyg_counter_tick()` call. A counter is created that has multiple alarms attached to it. The purpose of this test is to measure the cost of "ticking" a counter when it has many attached alarms. In this test, the alarms are activated (fired). Thus the measured time will include the overhead of calling the alarm callback function.

Alarm latency [0 threads]

> This test attempts to measure the latency in calling an alarm callback function. The time from the clock interrupt until the alarm function is called is measured. In this test, there are no threads that can be run, other than the system idle thread, when the clock interrupt occurs (all threads are suspended).

Alarm latency [2 threads]

> This test attempts to measure the latency in calling an alarm callback function. The time from the clock interrupt until the alarm function is called is measured. In this test, there are exactly two threads which are running when the clock interrupt occurs.

They are simply passing back and forth by way of the `cyg_thread_yield()` call. The purpose of this test is to measure the variations in the latency when there are executing threads.

Alarm latency [many threads]

This test attempts to measure the latency in calling an alarm callback function. The time from the clock interrupt until the alarm function is called is measured. In this test, there are a number of threads which are running when the clock interrupt occurs. They are simply passing back and forth by way of the `cyg_thread_yield()` call. The purpose of this test is to measure the variations in the latency when there are many executing threads.

Name

Thread Debugging — Overview of eCos Kernel thread-aware debugging

Description

Thread-aware debugging refers to the ability to interrogate the list of threads active within an application when the system is stopped (halted). This is normally when the code has either stopped at a breakpoint, or when execution is interrupted via a hosted debug session (e.g. from GDB).

Helper Symbols

For eCosPro to aid external host-based debug tools, a set of helper symbols are defined to provide information on the size (width) and offset of useful fields, or relevant constant values, instead of addresses. The majority of these symbols are named to avoid possible namespace clashes with applications, but for historical reasons some architecture specific symbols are valid in the C/C++ namespace. Similarly different architectures export their own symbols.

An external tool that wishes to interpret thread information can check for the presence of the specifically named symbol, and add support accordingly.

These symbols are held in the symbol table of the ELF file, but have no cost impact (code or data size) on the actual binary loaded into the target (either via a debugger, or if an application binary is stored on the target). Obviously a stripped executable will lose the helper symbols, but debugging using a stripped ELF file would always pose some restrictions.

Required

These symbols are required for accessing the list of threads and the currently active thread. If they are **not** present in the symbol table then it indicates an eCos build without a thread scheduler, and hence there is no need for thread-aware debug support.

`Cyg_Thread::thread_list`

Pointer to the first thread descriptor in the chain of created threads.

`Cyg_Scheduler_Base::current_thread`

Pointer to the thread context for the currently active thread.

Common

When an eCos scheduler is configured, information describing the thread context is provided to enable generic scanning code to be implemented in an external tool regardless whether some eCos features are enabled or disabled. Since individual eCos configurations can have features present that change the shape of the actual thread descriptor structure, we need the important fields for scanning a list of threads to be available in each ELF file.

`__ecospro_syminfo.size.cyg_thread.list_next`
`__ecospro_syminfo.size.cyg_thread.state`
`__ecospro_syminfo.size.cyg_thread.sleep_reason`
`__ecospro_syminfo.size.cyg_thread.wake_reason`
`__ecospro_syminfo.size.cyg_thread.unique_id`
`__ecospro_syminfo.size.cyg_thread.name`
`__ecospro_syminfo.size.cyg_thread.priority`
`__ecospro_syminfo.size.cyg_thread.stack_ptr`

The **presence** of a field in the thread descriptor structure can be determined by a **non-zero** size symbol being provided. These symbols give the size, in bytes, of the relevant field at the offset specified by the corresponding `__ecospro_syminfo.off.*` symbol. This allows any host tool to provide features based on the conditional presence of fields.

`__ecospro_syminfo.off.cyg_thread.list_next`

Offset to field that points to the next thread descriptor.

`__ecospro_syminfo.off.cyg_thread.state`
`__ecospro_syminfo.off.cyg_thread.sleep_reason`
`__ecospro_syminfo.off.cyg_thread.wake_reason`
`__ecospro_syminfo.off.cyg_thread.unique_id`
`__ecospro_syminfo.off.cyg_thread.name`
`__ecospro_syminfo.off.cyg_thread.priority`

Offsets for useful fields in a thread descriptor structure.

`__ecospro_syminfo.off.cyg_thread.stack_ptr`

This symbol is the offset for the field containing the address of the stacked register context for inactive threads.

Cortex-M

Cortex-M Base

For Cortex-M targets the following symbols describe the main CPU register state. The presence of the symbol `__ecospro_syminfo.cortexm.thread.saved` can be used as indicator of an eCos Cortex-M application.

`__ecospro_syminfo.cortexm.thread.saved`

This symbol provides the actual `PC` address of the point in the code where a switch actually occurs, It may be useful depending on how the external tool interprets the stacked context information.

`__ecospro_syminfo.value.HAL_SAVEDREGISTERS.THREAD`
`__ecospro_syminfo.value.HAL_SAVEDREGISTERS.EXCEPTION`
`__ecospro_syminfo.value.HAL_SAVEDREGISTERS.INTERRUPT`

These symbols provide the values in the type field used to identify the shape of the stacked context. Since only threads will be accessed by the `Cyg_Thread::thread_list` list the host tools should only ever encounter `__ecospro_syminfo.value.HAL_SAVEDREGISTERS.THREAD` type contexts (with the optional FPU register state indicator flag). The values for the other types of Cortex-M contexts are provided for completeness only.

`__ecospro_syminfo.size.HAL_SavedRegisters.Thread`

This symbol provides the total size, in bytes, of a stacked CPU context for contexts of type `__ecospro_syminfo.value.HAL_SAVEDREGISTERS.THREAD`.

`__ecospro_syminfo.size.HAL_SavedRegisters.u.thread.r`

Total size, in bytes, of all the core CPU registers present in a stacked context.

`__ecospro_syminfo.size.HAL_SavedRegisters.u.thread.type`

Size, in bytes, of the field that encodes the type of stacked context. Since for Cortex-M targets the actual context stored (and its size) depends on whether the individual thread has any hardware FPU context saved. This is needed to ensure only valid information is used when dealing with lazy per-thread hardware FPU support.

`__ecospro_syminfo.size.HAL_SavedRegisters.u.thread.basepri`
`__ecospro_syminfo.size.HAL_SavedRegisters.u.thread.sp`
`__ecospro_syminfo.size.HAL_SavedRegisters.u.thread.pc`

Size of individual context fields referenced by the corresponding offset symbol.

`__ecospro_syminfo.off.HAL_SavedRegisters.u.thread.type`
`__ecospro_syminfo.off.HAL_SavedRegisters.u.thread.basepri`
`__ecospro_syminfo.off.HAL_SavedRegisters.u.thread.sp`
`__ecospro_syminfo.off.HAL_SavedRegisters.u.thread.r`
`__ecospro_syminfo.off.HAL_SavedRegisters.u.thread.pc`

Offsets into the stacked context for the core register values.

Cortex-M FPU

When a Cortex-M configuration with hardware FPU support configured is used then the following optional symbols will be present with **non-zero** values where appropriate.

`__ecospro_syminfo.value.HAL_SAVEDREGISTERS.WITH_FPU`

This value provides the bitmask flag `OR`-ed into the type field provided at the `__ecospro_syminfo.off.HAL_SavedRegisters.u.thread.type` offset used to identify individual thread contexts that contain FPU state.

`__ecospro_syminfo.size.HAL_SavedRegisters.u.thread.fpscr`

Size, in bytes, of the `FPSCR` register stacked in the context.

`__ecospro_syminfo.size.HAL_SavedRegisters.u.thread.s`

Total size, in bytes, of the single-precision vector stacked in the context.

`__ecospro_syminfo.off.HAL_SavedRegisters.u.thread.fpscr`

Offset of `FPSCR` register in the stacked context.

`__ecospro_syminfo.off.HAL_SavedRegisters.u.thread.s`

Offset of the single-precision register vector in the stacked context.

ARM

ARM/Cortex-A Base

For ARM/Cortex-A targets the following symbols describe the main CPU register state. The presence of the symbol `ARMREG_SIZE` can be used as indicator of an eCos `arm` architecture application.

`ARMREG_SIZE`

This symbol, if **non-zero**, provides the total size of the stacked context for inactive threads.

armreg_r0
armreg_r1
armreg_r2
armreg_r3
armreg_r4
armreg_r5
armreg_r6
armreg_r7
armreg_r8
armreg_r9
armreg_r10
armreg_fp
armreg_ip
armreg_sp
armreg_lr
armreg_pc
armreg_cpsr

These symbols provide the offset of the corresponding register within the stacked context referenced from the thread object __ecospro_syminfo.off.cyg_thread.stack_ptr field.

ARM/Cortex-A FPU

Optional ARM FPU symbols.

ARMREG_FPUCONTEXT_SIZE

If non-zero then this symbol indicates that the eCos applicaton has been configured with hardware FPU support. The value is the total size of the stacked inactive thread context.

armreg_fpscr

Offset with the stacked context of the FPSCR register.

ARM/Cortex-A FPU Single-Precision

Symbols present when single-precision ARM FPU is configured.

armreg_s_vec

The offset to the start of the stacked single-precision register vector.

ARMREG_S_COUNT

Number of single-precision registers present from the armreg_s_vec offset.

ARM/Cortex-A FPU Double-Precision

Symbols present when doubled-precision ARM FPU is configured.

armreg_vfp_vec

The offset to the start of the stacked double-precision register vector.

ARMREG_VFP_COUNT

Number of single-precision registers present from the armreg_vfp_vec offset.

ColdFire

Symbols provided by eCos ColdFire targets. The presence of the symbol `hal_context_pcsr_offset` can be used as an identifier for ColdFire targets.

`hal_context_size`

The overall stacked context size.

`hal_context_fpu_size`

If hardware FPU support is configured this symbol provides the size of the stacked FPU context.

`hal_context_pcsr_size`
`hal_context_integer_size`

These symbols provide information on the size (width) of individual registers.

`hal_context_pcsr_offset`
`hal_context_integer_d0_offset`
`hal_context_integer_d2_offset`
`hal_context_integer_a0_offset`
`hal_context_integer_a2_offset`
`hal_context_fpu_offset`
`hal_context_other_offset`

The offsets within the stacked context for the processor state.

`hal_context_rte_adjust`

The value, in bytes, if a `PCSR` RTE adjustment is used.

Other Useful Symbols

Some other standard eCos symbols **may** be present that could also be useful for external debug tools.

`idle_thread`

This symbol can be used to identify the idle (background) thread descriptor object if useful to the thread-aware host debug tool.

`cyg_libc_main_thread`

This symbol will not be present in the application symbol table if the relevant object is not defined. When `CYGSEM_LIBC_STARTUP_MAIN_THREAD` is configured this symbol can be used to reference the thread descriptor object for the `main()` C thread created by the run-time.

GDB

The following documentation uses the GDB command-line interface for its examples, although the thread-aware debug support is applicable to applications that access GDB via its programmatic interface, e.g. Eclipse.

When a GDB debug session halts the CPU, either from the code hitting a previously set breakpoint or via the user requesting a halt, it will display the state of the currently active CPU state and select the currently executing thread. When displaying threads via the `info threads` command the currently selected thread is highlighted by an asterisk (`*`) character. Therefore, immediately after a halt this will indicate the active, running, thread. Examining the CPU register state will report the state of this active thread.

With thread-aware debugging for the target application available, GDB will display a list of all known threads when given the command `info threads`. The command `thread` *`id`* can be used to switch context to other threads, providing the ability to examine their CPU register state, call stack, and local variables.

Notes:

1. The documentation for all GDB features is beyond the scope of this reference. Please refer to the website GDB: The GNU Project Debugger for definitive documentation of the GDB thread debug support.
2. All GDB execution operations such as single stepping and return from function call will always apply to the currently active, executing thread, *NOT* the thread currently selected by the developer or user within the debugger.

Depending on the target system being connected, the act of loading an eCos application into memory will not necessarily initialise all the memory and hardware state. The eCos application run-time startup code will normally initialise memory alongside other I/O requirements. Since it can be possible to execute thread interrogation commands before any target code has been executed, it can be useful having helper macros in your `.gdbinit` script to minimise misinformation being displayed. The following `clear_ecos_thread_pointers` GDB macro is an example which could be executed after loading an application and **before** any system initialisation code in the loaded application has been executed. It ensures that debug commands to interrogate thread state will not parse stale/undefined information from uninitialised memory.

```
define clear_ecos_thread_pointers
  set *((unsigned int *)&Cyg_Thread::thread_list) = 0
  set *((unsigned int *)&Cyg_Scheduler_Base::current_thread) = 0
end

document clear_ecos_thread_pointers
When starting a new debug session from application reset the run-time
code that clears BSS will not have been executed, so stale/unitialised
state may be present in memory. For RTOS aware thread debugging as
provided by external tools the GDB server may be confused and report
invalid state if the thread state is interrogated before the initial
eCos run-time initialisation has cleared the BSS area. This macro just
ensures that the relevant eCos pointers are NULL prior to debugging.
end
```

Of course if a hardware debugger is being used to connect to an existing application session (rather than loading and starting a new application session) then the macro should not be called.

It is useful to wrap the steps needed to connect to a target in a helper macro. e.g.:

```
define connocd
  target extended-remote localhost:3333
  load
  break cyg_test_exit
  break cyg_assert_fail
  display/i $pc
  clear_ecos_thread_pointers
end
```

So that all of the normal steps for loading and setting the debug environment for an application can be performed by a single command:

```
(gdb) connocd
0x080016dc in ?? ()

Loading section .rom_vectors, size 0x8 lma 0x90000000
Loading section .text, size 0x7b34 lma 0x90000008
Loading section .rodata, size 0x678 lma 0x90007b40
Loading section .data, size 0x180 lma 0x900081b8
Start address 0x90000008, load size 33588
Transfer rate: 94 KB/sec, 6717 bytes/write.
Breakpoint 1 at 0x900040f4: file ecospro-path/packages/infra/current/src/tcdiag.cxx, line 391.
```

```
Function "cyg_assert_fail" not defined.
Make breakpoint pending on future shared library load? (y or [n]) [answered N; input not from terminal]
(gdb)
```

The use of such macros from a GDB script file can make the task of debugging less cumbersome.

Ronetix PEEDI

Note

The Ronetix PEEDI firmware must be updated to at least version 21.2.0 to ensure the correct operation of the thread-aware debugging support.

The PEEDI `[TARGET]` section option `COREn_OS` can be used to introduce a thread/context description using the generic PEEDI support.

The example PEEDI configuration files supplied with eCosPro releases 4.5.8 and above should already have suitable RTOS support fragments. For example, the file `packages/hal/arm/arm9/sam9g45ek/<version>/misc/peedi.sam9g45ek.cfg` contains a `[OS_ECOS_ARM]` section, referenced from the head of the file via the `CORE0_OS=OS_ECOS_ARM` setting.

With a suitable `COREn_OS` the PEEDI will parse the eCos thread lists and stacked register contents when interrogating threads other than the current thread of executing on the CPU.

OpenOCD

OpenOCD provides the `-rtos eCos` option that can be used to configure thread-aware debug support in the configuration file used for the OpenOCD session.

Note

eCoscentric contributed the previously eCosPro specific eCos thread-aware debug support to the `OpenOCD` project. As of 2023-01-15 the support was merged into the `OpenOCD` mainline.

At its simplest the OpenOCD configuration file just needs to specify:

```
$_TARGETNAME configure -rtos eCos
```

NOTE: When OpenOCD `-rtos` support for eCos is configured the act of executing `target remote` or `target extended-remote` to connect to an OpenOCD instance will cause the configured OpenOCD RTOS support to perform an **update_threads** operation against the current memory state. This is to allow a debug session to be attached to an active system. However, it does mean that for an undefined memory state (power-on, CPU reset with undefined DRAM state, an application with a different thread context shape to the previous application new different-configuration application to be subsequently loaded after connecting to the OpenOCD GDB server) that the GDB server may report spurious thread information upon request.

The following is example output when the application is halted in a thread named "busy". The name, state and priority of the other available threads is also shown:

```
(gdb) info thr
 Id  Target Id         Frame
* 1  Thread 12 (Name: busy, State: Ready Pri: 20) 0x20009f50 in thread_busy (data=30000) at
     ecospro-path/packages/kernel/current/tests/fpint_thread_switch.cxx:566
  2  Thread 1 (Name: Idle Thread, State: Ready Pri: 31) Cyg_Scheduler::unlock_inner (new_lock=0) at
     ecospro-path/packages/kernel/current/src/sched/sched.cxx:233
  3  Thread 2 (Name: Test, State: Sleeping (WAIT) Pri: 3) Cyg_Scheduler::unlock_inner (new_lock=1) at
     ecospro-path/packages/kernel/current/src/sched/sched.cxx:233
  4  Thread 11 (Name: highpri, State: Sleeping (DELAY) Pri: 10) Cyg_Scheduler::unlock_inner (new_lock=0)
```

```
     at ecospro-path/packages/kernel/current/src/sched/sched.cxx:233
```

The OpenOCD GDB server can be left executing between GDB application debug sessions. It does not need to be re-started for every GDB session.

Segger JLink/JTrace

Segger do not allow source distributions of RTOS aware plugins based on their SDK. This policy unfortunately restricts eCos and eCos eCosPro support to pre-built shared library files only. There does not seem to be any obvious technical reason for this binary-only restriction, considering the simplicity of the exposed SDK API and limited feature set required to support RTOS aware debugging (as can be seen by the functionality required for the open-source OpenOCD "-rtos" support, and the generic config-file description approach built into the PEEDI (closed source) firmware.

Currently only 64-bit Linux `x86_64` is available via the file `libRTOSPlugin_eCosPro.so`. Please contact eCosCentric to discuss the options available if other host platforms are required.

For example, the following is used to start a JLink GDB server session connected to a STM32F429I-DISCO board with the eCosPro RTOS aware plugin selected:

```
$ JLinkGDBServer -device stm32f429zi -if swd -rtos libRTOSPlugin_eCosPro.so
```

The JLinkGDBServer can be left executing between GDB application debug sessions. It does not need to be re-started for every GDB session.

GDB stubs

Unlike the hardware debug approaches described above which benefit from bare-metal hardware support (SWD, JTAG, BDM, etc.), eCos also supports the use of GDBstubs which can be built into the application and accessed via an I/O channel (e.g. serial, Ethernet, etc.), or provided via a boot monitor/loader (e.g. RedBoot covering in Chapter 224, *Getting Started with RedBoot*) that provides an environment for executing applications. Such support has a run-time cost (code+data space as well as CPU cycles), and is only usable after some level of system initialisation has occurred. A hardware debug solution is therefore preferred.

However, if GDBstubs is the only available/possible solution, the eCos GDBstubs implementation supports thread aware debugging.

Name

Kernel Instrumentation — Overview of eCos Kernel and Infrastructure instrumentation

Description

The kernel implements a simple macro based mechanism for tracing the flow of execution. It is designed for embedding many trace points, each with an optional small amount of associated data (e.g. that can be encoded in two 32-bit “argument” fields). The mechanism also allows for extending the instrumentation with further package specific event generation as required.

Instrumentation records will only be generated if the `CYGPKG_KERNEL_INSTRUMENT)` option is enabled, and then only if the relevant individual kernel event code sub-options are also enabled. The default state is for all the kernel instrumentation sub-options to be enabled.

Warning

Some options, when enabled, will generate a **large** quantity of instrumentation records in a heavily loaded system and so care may need to be taken regarding the instrumentation that is enabled vs the instrumentation recording mechanism being used to avoid missing events. Depending on why the kernel instrumentation framework is being enabled (debugging, timing validation, etc.) the user can choose which events they wish to record by enabling only the specific CDL options required.

The tuning of the amount of instrumentation generated, and any buffering required to hold event records, is always a consideration when investigating systems and it may **not** be possible to have every instrumentation option enabled all of the time.

At its simplest, default, the kernel instrumentation stores event records in a memory buffer for subsequent extraction and post-processing.

However the instrumentation code generated can be over-ridden by a configuration providing a suitable `CYGBLD_KERNEL_INSTRUMENT_WRAPPER_H` header to override the default kernel implementation. This allows for other eCos packages, or for application specific support, to provide their own implementation.

The `CYGPKG_INFRA` infrastructure package also provides for other tracing mechanisms to be enabled. The provision of multiple “tracing” solutions reflects the different feature sets they provided. The user has the option for using the mechanism that best suits their needs, e.g. system level events, high-level application logic or performance, etc.

As an adjunct to the main kernel instrumentation support the infrastructure `cyg_systrace.h` header defines some macros used to instrument specific points in the eCos source (INFRA, KERNEL, HAL, etc.). The `CYG_SYSTRACE_*` support has been driven by the support required for specific features of 3rd-party trace tools.

The infrastructure package also provides the `cyg_trac.h` header which implements an alternative approach to tracing the execution path of code. The documentation for that trace mechanism is provided in the actual header file.

Part II. The eCos Hardware Abstraction Layer (HAL)

Table of Contents

Chapter 1. Introduction

This is an initial specification of the *eCos* Hardware Abstraction Layer (HAL). The HAL abstracts the underlying hardware of a processor architecture and/or the platform to a level sufficient for the eCos kernel to be ported onto that platform.

Caveat

This document is an informal description of the HAL capabilities and is not intended to be full documentation, although it may be used as a source for such. It also describes the HAL as it is currently implemented for the architectures targeted in this release. It most closely describes the HALs for the MIPS, I386 and PowerPC HALs. Other architectures are similar but may not be organized precisely as described here.

Chapter 2. Architecture, Variant and Platform

We have identified three levels at which the HAL must operate.

- The *architecture HAL* abstracts the basic CPU architecture and includes things like interrupt delivery, context switching, CPU startup etc.
- The *variant HAL* encapsulates features of the CPU variant such as caches, MMU and FPU features. It also deals with any on-chip peripherals such as memory and interrupt controllers. For architectural variations, the actual implementation of the variation is often in the architectural HAL, and the variant HAL simply provides the correct configuration definitions.
- The *platform HAL* abstracts the properties of the current platform and includes things like platform startup, timer devices, I/O register access and interrupt controllers.

The boundaries between these three HAL levels are necessarily blurred since functionality shifts between levels on a target-by-target basis. For example caches and MMU may be either an architecture feature or a variant feature. Similarly, memory and interrupt controllers may be on-chip and in the variant HAL, or off-chip and in the platform HAL.

Generally there is a separate package for each of the architecture, variant and package HALs for a target. For some of the older targets, or where it would be essentially empty, the variant HAL is omitted.

Chapter 3. General principles

The HAL has been implemented according to the following general principles:

1. The HAL is implemented in C and assembler, although the eCos kernel is largely implemented in C++. This is to permit the HAL the widest possible applicability.

2. All interfaces to the HAL are implemented by CPP macros. This allows them to be implemented as inline C code, inline assembler or function calls to external C or assembler code. This allows the most efficient implementation to be selected without affecting the interface. It also allows them to be redefined if the platform or variant HAL needs to replace or enhance a definition from the architecture HAL.

3. The HAL provides simple, portable mechanisms for dealing with the hardware of a wide range of architectures and platforms. It is always possible to bypass the HAL and program the hardware directly, but this may lead to a loss of portability.

Chapter 4. HAL Interfaces

This section describes the main HAL interfaces.

Base Definitions

These are definitions that characterize the properties of the base architecture that are used to compile the portable parts of the kernel. They are concerned with such things a portable type definitions, endianness, and labeling.

These definitions are supplied by the `cyg/hal/basetype.h` header file which is supplied by the architecture HAL. It is included automatically by `cyg/infra/cyg_type.h`.

Byte order

`CYG_BYTEORDER`

This defines the byte order of the target and must be set to either `CYG_LSBFIRST` or `CYG_MSBFIRST`.

Label Translation

`CYG_LABEL_NAME(name)`

This is a wrapper used in some C and C++ files which use labels defined in assembly code or the linker script. It need only be defined if the default implementation in `cyg/infra/cyg_type.h`, which passes the name argument unaltered, is inadequate. It should be paired with `CYG_LABEL_DEFN()`.

`CYG_LABEL_DEFN(name)`

This is a wrapper used in assembler sources and linker scripts which define labels. It need only be defined if the default implementation in `cyg/infra/cyg_type.h`, which passes the name argument unaltered, is inadequate. The most usual alternative definition of this macro prepends an underscore to the label name.

Base types

```
    cyg_halint8
    cyg_halint16
    cyg_halint32
    cyg_halint64
    cyg_halcount8
    cyg_halcount16
    cyg_halcount32
    cyg_halcount64
    cyg_halbool
```

These macros define the C base types that should be used to define variables of the given size. They only need to be defined if the default types specified in `cyg/infra/cyg_type.h` cannot be used. Note that these are only the base types, they will be composed with `signed` and `unsigned` to form full type specifications.

Atomic types

```
    cyg_halatomic CYG_ATOMIC
```

These types are guaranteed to be read or written in a single uninterruptible operation. It is architecture defined what size this type is, but it will be at least a byte.

Architecture Characterization

These are definition that are related to the basic architecture of the CPU. These include the CPU context save format, context switching, bit twiddling, breakpoints, stack sizes and address translation.

Most of these definition are found in `cyg/hal/hal_arch.h`. This file is supplied by the architecture HAL. If there are variant or platform specific definitions then these will be found in `cyg/hal/var_arch.h` or `cyg/hal/plf_arch.h`. These files are include automatically by this header, so need not be included explicitly.

Register Save Format

```
typedef struct HAL_SavedRegisters
{
    /* architecture-dependent list of registers to be saved */
} HAL_SavedRegisters;
```

This structure describes the layout of a saved machine state on the stack. Such states are saved during thread context switches, interrupts and exceptions. Different quantities of state may be saved during each of these, but usually a thread context state is a subset of the interrupt state which is itself a subset of an exception state. For debugging purposes, the same structure is used for all three purposes, but where these states are significantly different, this structure may contain a union of the three states.

Thread Context Initialization

```
HAL_THREAD_INIT_CONTEXT( sp, arg, entry, id )
```

This macro initializes a thread's context so that it may be switched to by `HAL_THREAD_SWITCH_CONTEXT()`. The arguments are:

sp
: A location containing the current value of the thread's stack pointer. This should be a variable or a structure field. The SP value will be read out of here and an adjusted value written back.

arg
: A value that is passed as the first argument to the entry point function.

entry
: The address of an entry point function. This will be called according the C calling conventions, and the value of *`arg`* will be passed as the first argument. This function should have the following type signature `void entry(CYG_ADDRWORD arg)`.

id
: A thread id value. This is only used for debugging purposes, it is ORed into the initialization pattern for unused registers and may be used to help identify the thread from its register dump. The least significant 16 bits of this value should be zero to allow space for a register identifier.

Thread Context Switching

```
HAL_THREAD_LOAD_CONTEXT( to )
HAL_THREAD_SWITCH_CONTEXT( from, to )
```

These macros implement the thread switch code. The arguments are:

from
: A pointer to a location where the stack pointer of the current thread will be stored.

to
: A pointer to a location from where the stack pointer of the next thread will be read.

For `HAL_THREAD_LOAD_CONTEXT()` the current CPU state is discarded and the state of the destination thread is loaded. This is only used once, to load the first thread when the scheduler is started.

For `HAL_THREAD_SWITCH_CONTEXT()` the state of the current thread is saved onto its stack, using the current value of the stack pointer, and the address of the saved state placed in *`*from`*. The value in *`*to`* is then read and the state of the new thread is loaded from it.

While these two operations may be implemented with inline assembler, they are normally implemented as calls to assembly code functions in the HAL. There are two advantages to doing it this way. First, the return link of the call provides a convenient PC value to be used in the saved context. Second, the calling conventions mean that the compiler will have already saved the caller-saved registers before the call, so the HAL need only save the callee-saved registers.

The implementation of `HAL_THREAD_SWITCH_CONTEXT()` saves the current CPU state on the stack, including the current interrupt state (or at least the register that contains it). For debugging purposes it is useful to save the entire register set, but for performance only the ABI-defined callee-saved registers need be saved. If it is implemented, the option `CYGDBG_HAL_COMMON_CONTEXT_SAVE_MINIMUM` controls how many registers are saved.

The implementation of `HAL_THREAD_LOAD_CONTEXT()` loads a thread context, destroying the current context. With a little care this can be implemented by sharing code with `HAL_THREAD_SWITCH_CONTEXT()`. To load a thread context simply requires the saved registers to be restored from the stack and a jump or return made back to the saved PC.

Note that interrupts are not disabled during this process, any interrupts that occur will be delivered onto the stack to which the current CPU stack pointer points. Hence the stack pointer should never be invalid, or loaded with a value that might cause the saved state to become corrupted by an interrupt. However, the current interrupt state is saved and restored as part of the thread context. If a thread disables interrupts and does something to cause a context switch, interrupts may be re-enabled on switching to another thread. Interrupts will be disabled again when the original thread regains control.

Bit indexing

```
HAL_LSBIT_INDEX( index, mask )
HAL_MSBIT_INDEX( index, mask )
```

These macros place in *`index`* the bit index of the least significant bit in *`mask`*. Some architectures have instruction level support for one or other of these operations. If no architectural support is available, then these macros may call C functions to do the job.

Idle thread activity

```
HAL_IDLE_THREAD_ACTION( count )
```

It may be necessary under some circumstances for the HAL to execute code in the kernel idle thread's loop. An example might be to execute a processor halt instruction. This macro provides a portable way of doing this. The argument is a copy of the idle thread's loop counter, and may be used to trigger actions at longer intervals than every loop.

Reorder barrier

```
HAL_REORDER_BARRIER()
```

When optimizing the compiler can reorder code. In some parts of multi-threaded systems, where the order of actions is vital, this can sometimes cause problems. This macro may be inserted into places where reordering should not happen and prevents code being migrated across it by the compiler optimizer. It should be placed between statements that must be executed in the order written in the code.

Breakpoint support

```
HAL_BREAKPOINT( label )
HAL_BREAKINST
HAL_BREAKINST_SIZE
```

These macros provide support for breakpoints.

HAL_BREAKPOINT() executes a breakpoint instruction. The label is defined at the breakpoint instruction so that exception code can detect which breakpoint was executed.

HAL_BREAKINST contains the breakpoint instruction code as an integer value. HAL_BREAKINST_SIZE is the size of that breakpoint instruction in bytes. Together these may be used to place a breakpoint in any code.

GDB support

```
HAL_THREAD_GET_SAVED_REGISTERS( sp, regs )
HAL_GET_GDB_REGISTERS( regval, regs )
HAL_SET_GDB_REGISTERS( regs, regval )
```

These macros provide support for interfacing GDB to the HAL.

HAL_THREAD_GET_SAVED_REGISTERS() extracts a pointer to a HAL_SavedRegisters structure from a stack pointer value. The stack pointer passed in should be the value saved by the thread context macros. The macro will assign a pointer to the HAL_SavedRegisters structure to the variable passed as the second argument.

HAL_GET_GDB_REGISTERS() translates a register state as saved by the HAL and into a register dump in the format expected by GDB. It takes a pointer to a HAL_SavedRegisters structure in the *regs* argument and a pointer to the memory to contain the GDB register dump in the *regval* argument.

HAL_SET_GDB_REGISTERS() translates a GDB format register dump into a the format expected by the HAL. It takes a pointer to the memory containing the GDB register dump in the *regval* argument and a pointer to a HAL_SavedRegisters structure in the *regs* argument.

Setjmp and longjmp support

```
CYGARC_JMP_BUF_SIZE
hal_jmp_buf[CYGARC_JMP_BUF_SIZE]
hal_setjmp( hal_jmp_buf env )
hal_longjmp( hal_jmp_buf env, int val )
```

These functions provide support for the C setjmp() and longjmp() functions. Refer to the C library for further information.

Stack Sizes

```
CYGNUM_HAL_STACK_SIZE_MINIMUM
CYGNUM_HAL_STACK_SIZE_TYPICAL
```

The values of these macros define the minimum and typical sizes of thread stacks.

CYGNUM_HAL_STACK_SIZE_MINIMUM defines the minimum size of a thread stack. This is enough for the thread to function correctly within eCos and allows it to take interrupts and context switches. There should also be enough space for a simple thread entry function to execute and call basic kernel operations on objects like mutexes and semaphores. However there will not be enough room for much more than this. When creating stacks for their own threads, applications should determine the stack usage needed for application purposes and then add CYGNUM_HAL_STACK_SIZE_MINIMUM.

CYGNUM_HAL_STACK_SIZE_TYPICAL is a reasonable increment over CYGNUM_HAL_STACK_SIZE_MINIMUM, usually about 1kB. This should be adequate for most modest thread needs. Only threads that need to define significant amounts of local data, or have very deep call trees should need to use a larger stack size.

Address Translation

```
CYGARC_CACHED_ADDRESS(addr)
CYGARC_UNCACHED_ADDRESS(addr)
```

```
CYGARC_PHYSICAL_ADDRESS(addr)
```

These macros provide address translation between different views of memory. In many architectures a given memory location may be visible at different addresses in both cached and uncached forms. It is also possible that the MMU or some other address translation unit in the CPU presents memory to the program at a different virtual address to its physical address on the bus.

`CYGARC_CACHED_ADDRESS()` translates the given address to its location in cached memory. This is typically where the application will access the memory.

`CYGARC_UNCACHED_ADDRESS()` translates the given address to its location in uncached memory. This is typically where device drivers will access the memory to avoid cache problems. It may additionally be necessary for the cache to be flushed before the contents of this location is fully valid.

`CYGARC_PHYSICAL_ADDRESS()` translates the given address to its location in the physical address space. This is typically the address that needs to be passed to device hardware such as a DMA engine, Ethernet device or PCI bus bridge. The physical address may not be directly accessible to the program, it may be re-mapped by address translation.

Global Pointer

```
CYGARC_HAL_SAVE_GP()
CYGARC_HAL_RESTORE_GP()
```

These macros insert code to save and restore any global data pointer that the ABI uses. These are necessary when switching context between two eCos instances - for example between an eCos application and RedBoot.

Interrupt Handling

These interfaces contain definitions related to interrupt handling. They include definitions of exception and interrupt numbers, interrupt enabling and masking.

These definitions are normally found in `cyg/hal/hal_intr.h`. This file is supplied by the architecture HAL. Any variant or platform specific definitions will be found in `cyg/hal/var_intr.h`, `cyg/hal/plf_intr.h` or `cyg/hal/hal_platform_ints.h` in the variant or platform HAL, depending on the exact target. These files are include automatically by this header, so need not be included explicitly.

Vector numbers

```
CYGNUM_HAL_VECTOR_XXXX
CYGNUM_HAL_VSR_MIN
CYGNUM_HAL_VSR_MAX
CYGNUM_HAL_VSR_COUNT

CYGNUM_HAL_INTERRUPT_XXXX
CYGNUM_HAL_ISR_MIN
CYGNUM_HAL_ISR_MAX
CYGNUM_HAL_ISR_COUNT

CYGNUM_HAL_EXCEPTION_XXXX
CYGNUM_HAL_EXCEPTION_MIN
CYGNUM_HAL_EXCEPTION_MAX
CYGNUM_HAL_EXCEPTION_COUNT
```

All possible VSR, interrupt and exception vectors are specified here, together with maximum and minimum values for range checking. While the VSR and exception numbers will be defined in this file, the interrupt numbers will normally be defined in the variant or platform HAL file that is included by this header.

There are two ranges of numbers, those for the vector service routines and those for the interrupt service routines. The relationship between these two ranges is undefined, and no equivalence should be assumed if vectors from the two ranges coincide.

The VSR vectors correspond to the set of exception vectors that can be delivered by the CPU architecture, many of these will be internal exception traps. The ISR vectors correspond to the set of external interrupts that can be delivered and are usually determined by extra decoding of the interrupt controller by the interrupt VSR.

Where a CPU supports synchronous exceptions, the range of such exceptions allowed are defined by CYGNUM_HAL_EXCEPTION_MIN and CYGNUM_HAL_EXCEPTION_MAX. The CYGNUM_HAL_EXCEPTION_XXXX definitions are standard names used by target independent code to test for the presence of particular exceptions in the architecture. The actual exception numbers will normally correspond to the VSR exception range. In future other exceptions generated by the system software (such as stack overflow) may be added.

CYGNUM_HAL_ISR_COUNT, CYGNUM_HAL_VSR_COUNT and CYGNUM_HAL_EXCEPTION_COUNT define the number of ISRs, VSRs and EXCEPTIONs respectively for the purposes of defining arrays etc. There might be a translation from the supplied vector numbers into array offsets. Hence CYGNUM_HAL_XXX_COUNT may not simply be CYGNUM_HAL_XXX_MAX - CYGNUM_HAL_XXX_MIN or CYGNUM_HAL_XXX_MAX+1.

Interrupt state control

```
CYG_INTERRUPT_STATE
HAL_DISABLE_INTERRUPTS( old )
HAL_RESTORE_INTERRUPTS( old )
HAL_ENABLE_INTERRUPTS()
HAL_QUERY_INTERRUPTS( state )
```

These macros provide control over the state of the CPUs interrupt mask mechanism. They should normally manipulate a CPU status register to enable and disable interrupt delivery. They should not access an interrupt controller.

CYG_INTERRUPT_STATE is a data type that should be used to store the interrupt state returned by HAL_DISABLE_INTERRUPTS() and HAL_QUERY_INTERRUPTS() and passed to HAL_RESTORE_INTERRUPTS().

HAL_DISABLE_INTERRUPTS() disables the delivery of interrupts and stores the original state of the interrupt mask in the variable passed in the *old* argument.

HAL_RESTORE_INTERRUPTS() restores the state of the interrupt mask to that recorded in *old*.

HAL_ENABLE_INTERRUPTS() simply enables interrupts regardless of the current state of the mask.

HAL_QUERY_INTERRUPTS() stores the state of the interrupt mask in the variable passed in the *state* argument. The state stored here should also be capable of being passed to HAL_RESTORE_INTERRUPTS() at a later point.

It is at the HAL implementer,s discretion exactly which interrupts are masked by this mechanism. Where a CPU has more than one interrupt type that may be masked separately (e.g. the ARM's IRQ and FIQ) only those that can raise DSRs need to be masked here. A separate architecture specific mechanism may then be used to control the other interrupt types.

ISR and VSR management

```
HAL_INTERRUPT_IN_USE( vector, state )
HAL_INTERRUPT_ATTACH( vector, isr, data, object )
HAL_INTERRUPT_DETACH( vector, isr )
HAL_VSR_SET( vector, vsr, poldvsr )
HAL_VSR_GET( vector, pvsr )
HAL_VSR_SET_TO_ECOS_HANDLER( vector, poldvsr )
```

These macros manage the attachment of interrupt and vector service routines to interrupt and exception vectors respectively.

HAL_INTERRUPT_IN_USE() tests the state of the supplied interrupt vector and sets the value of the state parameter to either 1 or 0 depending on whether there is already an ISR attached to the vector. The HAL will only allow one ISR to be attached to each vector, so it is a good idea to use this function before using HAL_INTERRUPT_ATTACH().

`HAL_INTERRUPT_ATTACH()` attaches the ISR, data pointer and object pointer to the given *`vector`*. When an interrupt occurs on this vector the ISR is called using the C calling convention and the vector number and data pointer are passed to it as the first and second arguments respectively.

`HAL_INTERRUPT_DETACH()` detaches the ISR from the vector.

`HAL_VSR_SET()` replaces the VSR attached to the *`vector`* with the replacement supplied in *`vsr`*. The old VSR is returned in the location pointed to by *`pvsr`*. On some platforms, possibly only in certain configurations, the table of VSRs will be in read-only memory. If so then this macro should be left undefined.

`HAL_VSR_GET()` assigns a copy of the VSR to the location pointed to by *`pvsr`*.

`HAL_VSR_SET_TO_ECOS_HANDLER()` ensures that the VSR for a specific exception is pointing at the eCos exception VSR and not one for RedBoot or some other ROM monitor. The default when running under RedBoot is for exceptions to be handled by RedBoot and passed to GDB. This macro diverts the exception to eCos so that it may be handled by application code. The arguments are the VSR vector to be replaces, and a location in which to store the old VSR pointer, so that it may be replaced at a later point. On some platforms, possibly only in certain configurations, the table of VSRs will be in read-only memory. If so then this macro should be left undefined.

Interrupt controller management

```
HAL_INTERRUPT_MASK( vector )
HAL_INTERRUPT_UNMASK( vector )
HAL_INTERRUPT_ACKNOWLEDGE( vector )
HAL_INTERRUPT_CONFIGURE( vector, level, up )
HAL_INTERRUPT_SET_LEVEL( vector, level )
```

These macros exert control over any prioritized interrupt controller that is present. If no priority controller exists, then these macros should be empty.

Note

These macros may not be reentrant, so care should be taken to prevent them being called while interrupts are enabled. This means that they can be safely used in initialization code before interrupts are enabled, and in ISRs. In DSRs, ASRs and thread code, however, interrupts must be disabled before these macros are called. Here is an example for use in a DSR where the interrupt source is unmasked after data processing:

```
…
HAL_DISABLE_INTERRUPTS(old);
HAL_INTERRUPT_UNMASK(CYGNUM_HAL_INTERRUPT_ETH);
HAL_RESTORE_INTERRUPTS(old);
…
```

`HAL_INTERRUPT_MASK()` causes the interrupt associated with the given vector to be blocked.

`HAL_INTERRUPT_UNMASK()` causes the interrupt associated with the given vector to be unblocked.

`HAL_INTERRUPT_ACKNOWLEDGE()` acknowledges the current interrupt from the given vector. This is usually executed from the ISR for this vector when it is prepared to allow further interrupts. Most interrupt controllers need some form of acknowledge action before the next interrupt is allowed through. Executing this macro may cause another interrupt to be delivered. Whether this interrupts the current code depends on the state of the CPU interrupt mask.

`HAL_INTERRUPT_CONFIGURE()` provides control over how an interrupt signal is detected. The arguments are:

vector
: The interrupt vector to be configured.

level
: Set to `true` if the interrupt is detected by level, and `false` if it is edge triggered.

up
: If the interrupt is set to level detect, then if this is `true` it is detected by a high signal level, and if `false` by a low signal level. If the interrupt is set to edge triggered, then if this is `true` it is triggered by a rising edge and if `false` by a falling edge.

`HAL_INTERRUPT_SET_LEVEL()` provides control over the hardware priority of the interrupt. The arguments are:

vector
: The interrupt whose level is to be set.

level
: The priority level to which the interrupt is to set. In some architectures the masking of an interrupt is achieved by changing its priority level. Hence this function, `HAL_INTERRUPT_MASK()` and `HAL_INTERRUPT_UNMASK()` may interfere with each other.

Clocks and Timers

These interfaces contain definitions related to clock and timer handling. They include interfaces to initialize and read a clock for generating regular interrupts, definitions for setting the frequency of the clock, and support for short timed delays.

Clock Control

```
HAL_CLOCK_INITIALIZE( period )
HAL_CLOCK_RESET( vector, period )
HAL_CLOCK_READ( pvalue )
```

These macros provide control over a clock or timer device that may be used by the kernel to provide time-out, delay and scheduling services. The clock is assumed to be implemented by some form of counter that is incremented or decremented by some external source and which raises an interrupt when it reaches a predetermined value.

`HAL_CLOCK_INITIALIZE()` initializes the timer device to interrupt at the given period. The period is essentially the value used to initialize the timer counter and must be calculated from the timer frequency and the desired interrupt rate. The timer device should generate an interrupt every `period` cycles.

`HAL_CLOCK_RESET()` re-initializes the timer to provoke the next interrupt. This macro is only really necessary when the timer device needs to be reset in some way after each interrupt.

`HAL_CLOCK_READ()` reads the current value of the timer counter and puts the value in the location pointed to by *`pvalue`*. The value stored will always be the number of timer cycles since the last interrupt, and hence ranges between zero and the initial period value. If this is a count-down cyclic timer, some arithmetic may be necessary to generate this value.

Microsecond Delay

```
HAL_DELAY_US(us)
```

This macro provides a busy loop delay for the given number of microseconds. It is intended mainly for controlling hardware that needs short delays between operations. Code which needs longer delays, of the order of milliseconds, should instead use higher-level functions such as `cyg_thread_delay`. The macro implementation should be thread-safe. It can also be used in ISRs or DSRs, although such usage is undesirable because of the impact on interrupt and dispatch latency.

The macro should never delay for less than the specified amount of time. It may delay for somewhat longer, although since the macro uses a busy loop this is a waste of CPU cycles. Of course the code invoking `HAL_DELAY_US` may get interrupted or timesliced, in which case the delay may be much longer than intended. If this is unacceptable then the calling code must take preventative action such as disabling interrupts or locking the scheduler.

There are three main ways of implementing the macro:

1. a counting loop, typically written in inline assembler, using an outer loop for the microseconds and an inner loop that consumes approximately 1us. This implementation is automatically thread-safe and does not impose any dependencies on the rest of the

system, for example it does not depend on the system clock having been started. However it assumes that the CPU clock speed is known at compile-time or can be easily determined at run-time.

2. monitor one of the hardware clocks, usually the system clock. Usually this clock ticks at a rate independent of the CPU so calibration is easier. However the implementation relies on the system clock having been started, and assumes that no other code is manipulating the clock hardware. There can also be complications when the system clock wraps around.

3. a combination of the previous two. The system clock is used during system initialization to determine the CPU clock speed, and the result is then used to calibrate a counting loop. This has the disadvantage of significantly increasing the system startup time, which may be unacceptable to some applications. There are also complications if the system startup code normally runs with the cache disabled because the instruction cache will greatly affect any calibration loop.

Clock Frequency Definition

```
CYGNUM_HAL_RTC_NUMERATOR
CYGNUM_HAL_RTC_DENOMINATOR
CYGNUM_HAL_RTC_PERIOD
```

These macros are defined in the CDL for each platform and supply the necessary parameters to specify the frequency at which the clock interrupts. These parameters are usually found in the CDL definitions for the target platform, or in some cases the CPU variant.

`CYGNUM_HAL_RTC_NUMERATOR` and `CYGNUM_HAL_RTC_DENOMINATOR` specify the resolution of the clock interrupt. This resolution involves two separate values, the numerator and the denominator. The result of dividing the numerator by the denominator should correspond to the number of nanoseconds between clock interrupts. For example a numerator of 1000000000 and a denominator of 100 means that there are 10000000 nanoseconds (or 10 milliseconds) between clock interrupts. Expressing the resolution as a fraction minimizes clock drift even for frequencies that cannot be expressed as a simple integer. For example a frequency of 60Hz corresponds to a clock resolution of 16666666.66… nanoseconds. This can be expressed accurately as 1000000000 over 60.

`CYGNUM_HAL_RTC_PERIOD` specifies the exact value used to initialize the clock hardware, it is the value passed as a parameter to `HAL_CLOCK_INITIALIZE()` and `HAL_CLOCK_RESET()`. The exact meaning of the value and the range of legal values therefore depends on the target hardware, and the hardware documentation should be consulted for further details.

The default values for these macros in all HALs are calculated to give a clock interrupt frequency of 100Hz, or 10ms between interrupts. To change the clock frequency, the period needs to be changed, and the resolution needs to be adjusted accordingly. As an example consider the i386 PC target. The default values for these macros are:

```
CYGNUM_HAL_RTC_NUMERATOR     1000000000
CYGNUM_HAL_RTC_DENOMINATOR   100
CYGNUM_HAL_RTC_PERIOD        11932
```

To change to, say, a 200Hz clock the period needs to be halved to 5966, and to compensate the denominator needs to be doubled to 200. To change to a 1KHz interrupt rate change the period to 1193 and the denominator to 1000.

Some HALs make this process a little easier by deriving the period arithmetically from the denominator. This calculation may also involve the CPU clock frequency and possibly other factors. For example in the ARM AT91 variant HAL the period is defined by the following expression:

```
((CYGNUM_HAL_ARM_AT91_CLOCK_SPEED/32) / CYGNUM_HAL_RTC_DENOMINATOR)
```

In this case it is not necessary to change the period at all, just change the denominator to select the desired clock frequency. However, note that for certain choices of frequency, rounding errors in this calculation may result in a small clock drift over time. This is usually negligible, but if perfect accuracy is required, it may be necessary to adjust the frequency or period by hand.

HAL I/O

This section contains definitions for supporting access to device control registers in an architecture neutral fashion.

These definitions are normally found in the header file `cyg/hal/hal_io.h`. This file itself contains macros that are generic to the architecture. If there are variant or platform specific IO access macros then these will be found in `cyg/hal/var_io.h` and `cyg/hal/plf_io.h` in the variant or platform HALs respectively. These files are included automatically by this header, so need not be included explicitly.

This header (or more likely `cyg/hal/plf_io.h`) also defines the PCI access macros. For more information on these see the eCos PCI library reference documentation.

Register address

```
HAL_IO_REGISTER
```

This type is used to store the address of an I/O register. It will normally be a memory address, an integer port address or an offset into an I/O space. More complex architectures may need to code an address space plus offset pair into a single word, or may represent it as a structure.

Values of variables and constants of this type will usually be supplied by configuration mechanisms or in target specific headers.

Register read

```
HAL_READ_XXX( register, value )
HAL_READ_XXX_VECTOR( register, buffer, count, stride )
```

These macros support the reading of I/O registers in various sizes. The *XXX* component of the name may be `UINT8`, `UINT16`, `UINT32`.

`HAL_READ_XXX()` reads the appropriately sized value from the register and stores it in the variable passed as the second argument.

`HAL_READ_XXX_VECTOR()` reads *count* values of the appropriate size into *buffer*. The *stride* controls how the pointer advances through the register space. A stride of zero will read the same register repeatedly, and a stride of one will read adjacent registers of the given size. Greater strides will step by larger amounts, to allow for sparsely mapped registers for example.

Register write

```
HAL_WRITE_XXX( register, value )
HAL_WRITE_XXX_VECTOR( register, buffer,count, stride )
```

These macros support the writing of I/O registers in various sizes. The *XXX* component of the name may be `UINT8`, `UINT16`, `UINT32`.

`HAL_WRITE_XXX()` writes the appropriately sized value from the variable passed as the second argument stored it in the register.

`HAL_WRITE_XXX_VECTOR()` writes *count* values of the appropriate size from *buffer*. The *stride* controls how the pointer advances through the register space. A stride of zero will write the same register repeatedly, and a stride of one will write adjacent registers of the given size. Greater strides will step by larger amounts, to allow for sparsely mapped registers for example.

HAL Unique-ID

This section contains definitions for supporting the optional Unique-ID access in an architecture neutral fashion. Not all variants, or platforms, will provide a mechanism for accessing device-specific Unique-ID data, in which case the macros as documented in this section will not be defined.

The required definitions are normally referenced via the header file `cyg/hal/hal_io.h`. This file itself contains macros that are generic to the configured architecture. If there are variant or platform specific Unique-ID access macros then these will be found in `cyg/hal/var_io.h` and `cyg/hal/plf_io.h` in the variant or platform HALs respectively. These files are included automatically by the architecture header, so need not be included explicitly.

HAL_UNIQUE_ID_LEN

```
HAL_UNIQUE_ID_LEN( CYG_WORD32 maxlen )
```

This macro, when defined, provides a mechanism for ascertaining the maximum number of bytes of Unique-ID data available.

For most implementations this macro will return a build-time constant value in the passed *`maxlen`* parameter, but some systems may have a run-time calculated limit.

HAL_UNIQUE_ID

```
HAL_UNIQUE_ID( CYG_BYTE *buffer, CYG_WORD32 bufflen )
```

T his macro, when defined, provides a mechanism for filling the passed *`buffer`* parameter with upto *`bufflen`* bytes of Unique-ID data. If the implementation provides fewer that *`bufflen`* bytes of unique information then only the available data will be copied to the destination *`buffer`*.

The use of returned Unique-ID data is application specific, but examples may include use for USB device serial# identification, Ethernet MAC addresses, cryptography seeds, etc.

When a valid *`buffer`* parameter is passed then the *`bufflen`* parameter indicates how many bytes are available. This *`bufflen`* size may be less than the total amount of Unique-ID information available, with the variant/platform implementation only copying the requested amount.

Note

This “always-copy” model is used since it allows the same API to be used for systems where the ID information is *not* held in CPU addressable memory, or where multiple sources are concatenated by an implementation to provide a larger Unique-ID value.

Cache Control

This section contains definitions for supporting control of the caches on the CPU.

These definitions are usually found in the header file `cyg/hal/hal_cache.h`. This file may be defined in the architecture, variant or platform HAL, depending on where the caches are implemented for the target. Often there will be a generic implementation of the cache control macros in the architecture HAL with the ability to override or undefine them in the variant or platform HAL. Even when the implementation of the cache macros is in the architecture HAL, the cache dimensions will be defined in the variant or platform HAL. As with other files, the variant or platform specific definitions are usually found in `cyg/hal/var_cache.h` and `cyg/hal/plf_cache.h` respectively. These files are include automatically by this header, so need not be included explicitly.

There are versions of the macros defined here for both the Data and Instruction caches. these are distinguished by the use of either `DCACHE` or `ICACHE` in the macro names. Some architectures have a unified cache, where both data and instruction share the same cache. In these cases the control macros use `UCACHE` and the `DCACHE` and `ICACHE` macros will just be calls to the `UCACHE` version. In the following descriptions, `XCACHE` is used to stand for any of these. Where there are issues specific to a particular cache, this will be explained in the text.

There might be target specific restrictions on the use of some of the macros which it is the user's responsibility to comply with. Such restrictions are documented in the header file with the macro definition.

Note that destructive cache macros should be used with caution. Preceding a cache invalidation with a cache synchronization is not safe in itself since an interrupt may happen after the synchronization but before the invalidation. This might cause the state of dirty data lines created during the interrupt to be lost.

Depending on the architecture's capabilities, it may be possible to temporarily disable the cache while doing the synchronization and invalidation which solves the problem (no new data would be cached during an interrupt). Otherwise it is necessary to disable interrupts while manipulating the cache which may take a long time.

Some platform HALs now support a pair of cache state query macros: `HAL_ICACHE_IS_ENABLED( x )` and `HAL_DCACHE_IS_ENABLED( x )` which set the argument to true if the instruction or data cache is enabled, respectively. Like most cache control macros, these are optional, because the capabilities of different targets and boards can vary considerably. Code which uses them, if it is to be considered portable, should test for their existence first by means of `#ifdef`. Be sure to include `<cyg/hal/hal_cache.h>` in order to do this test and (maybe) use the macros.

Cache Dimensions

```
HAL_XCACHE_SIZE
HAL_XCACHE_LINE_SIZE
HAL_XCACHE_WAYS
HAL_XCACHE_SETS
```

These macros define the size and dimensions of the Instruction and Data caches.

HAL_XCACHE_SIZE

Defines the total size of the cache in bytes.

HAL_XCACHE_LINE_SIZE

Defines the cache line size in bytes.

HAL_XCACHE_WAYS

Defines the number of ways in each set and defines its level of associativity. This would be 1 for a direct mapped cache, 2 for a 2-way cache, 4 for 4-way and so on.

HAL_XCACHE_SETS

Defines the number of sets in the cache, and is calculated from the previous values.

Global Cache Control

```
HAL_XCACHE_ENABLE()
HAL_XCACHE_DISABLE()
HAL_XCACHE_INVALIDATE_ALL()
HAL_XCACHE_SYNC()
HAL_XCACHE_BURST_SIZE( size )
HAL_DCACHE_WRITE_MODE( mode )
HAL_XCACHE_LOCK( base, size )
HAL_XCACHE_UNLOCK( base, size )
HAL_XCACHE_UNLOCK_ALL()
```

These macros affect the state of the entire cache, or a large part of it.

HAL_XCACHE_ENABLE() and HAL_XCACHE_DISABLE()

Enable and disable the cache.

HAL_XCACHE_INVALIDATE_ALL()

Causes the entire contents of the cache to be invalidated. Depending on the hardware, this may require the cache to be disabled during the invalidation process. If so, the implementation must use `HAL_XCACHE_IS_ENABLED()` to save and restore the previous state.

Note

If this macro is called after `HAL_XCACHE_SYNC()` with the intention of clearing the cache (invalidating the cache after writing dirty data back to memory), you must prevent interrupts from happening between the two calls:

```
…
HAL_DISABLE_INTERRUPTS(old);
HAL_XCACHE_SYNC();
HAL_XCACHE_INVALIDATE_ALL();
HAL_RESTORE_INTERRUPTS(old);
…
```

Since the operation may take a very long time, real-time responsiveness could be affected, so only do this when it is absolutely required and you know the delay will not interfere with the operation of drivers or the application.

HAL_XCACHE_SYNC()

Causes the contents of the cache to be brought into synchronization with the contents of memory. In some implementations this may be equivalent to `HAL_XCACHE_INVALIDATE_ALL()`.

HAL_XCACHE_BURST_SIZE()

Allows the size of cache to/from memory bursts to be controlled. This macro will only be defined if this functionality is available.

HAL_DCACHE_WRITE_MODE()

Controls the way in which data cache lines are written back to memory. There will be definitions for the possible modes. Typical definitions are `HAL_DCACHE_WRITEBACK_MODE` and `HAL_DCACHE_WRITETHRU_MODE`. This macro will only be defined if this functionality is available.

HAL_XCACHE_LOCK()

Causes data to be locked into the cache. The base and size arguments define the memory region that will be locked into the cache. It is architecture dependent whether more than one locked region is allowed at any one time, and whether this operation causes the cache to cease acting as a cache for addresses outside the region during the duration of the lock. This macro will only be defined if this functionality is available.

HAL_XCACHE_UNLOCK()

Cancels the locking of the memory region given. This should normally correspond to a region supplied in a matching lock call. This macro will only be defined if this functionality is available.

HAL_XCACHE_UNLOCK_ALL()

Cancels all existing locked memory regions. This may be required as part of the cache initialization on some architectures. This macro will only be defined if this functionality is available.

Cache Line Control

```
HAL_DCACHE_ALLOCATE( base , size )
HAL_DCACHE_FLUSH( base , size )
HAL_XCACHE_INVALIDATE( base , size )
```

```
HAL_DCACHE_STORE( base , size )
HAL_DCACHE_READ_HINT( base , size )
HAL_DCACHE_WRITE_HINT( base , size )
HAL_DCACHE_ZERO( base , size )
```

All of these macros apply a cache operation to all cache lines that match the memory address region defined by the base and size arguments. These macros will only be defined if the described functionality is available. Also, it is not guaranteed that the cache function will only be applied to just the described regions, in some architectures it may be applied to the whole cache.

HAL_DCACHE_ALLOCATE()

Allocates lines in the cache for the given region without reading their contents from memory, hence the contents of the lines is undefined. This is useful for preallocating lines which are to be completely overwritten, for example in a block copy operation.

HAL_DCACHE_FLUSH()

Invalidates all cache lines in the region after writing any dirty lines to memory.

HAL_XCACHE_INVALIDATE()

Invalidates all cache lines in the region. Any dirty lines are invalidated without being written to memory.

HAL_DCACHE_STORE()

Writes all dirty lines in the region to memory, but does not invalidate any lines.

HAL_DCACHE_READ_HINT()

Hints to the cache that the region is going to be read from in the near future. This may cause the region to be speculatively read into the cache.

HAL_DCACHE_WRITE_HINT()

Hints to the cache that the region is going to be written to in the near future. This may have the identical behavior to HAL_DCACHE_READ_HINT().

HAL_DCACHE_ZERO()

Allocates and zeroes lines in the cache for the given region without reading memory. This is useful if a large area of memory is to be cleared.

Linker Scripts

When an eCos application is linked it must be done under the control of a linker script. This script defines the memory areas, addresses and sized, into which the code and data are to be put, and allocates the various sections generated by the compiler to these.

The linker script actually used is in `lib/target.ld` in the install directory. This is actually manufactured out of two other files: a base linker script and an `.ldi` file that was generated by the memory layout tool.

The base linker script is usually supplied either by the architecture HAL or the variant HAL. It consists of a set of linker script fragments, in the form of C preprocessor macros, that define the major output sections to be generated by the link operation. The `.ldi` file, which is `#include'ed` by the base linker script, uses these macro definitions to assign the output sections to the required memory areas and link addresses.

The `.ldi` file is supplied by the platform HAL, and contains knowledge of the memory layout of the target platform. These files generally conform to a standard naming convention, each file being of the form:

```
pkgconf/mlt_<architecture>_<variant>_<platform>_<startup>.ldi
```

where <architecture>, <variant> and <platform> are the respective HAL package names and <startup> is the startup type which is usually one of ROM, RAM or ROMRAM.

In addition to the .ldi file, there is also a congruously name .h file. This may be used by the application to access information defined in the .ldi file. Specifically it contains the memory layout defined there, together with any additional section names defined by the user. Examples of the latter are heap areas or PCI bus memory access windows.

The .ldi is manufactured by the Memory Layout Tool (MLT). The MLT saves the memory configuration into a file named

include/pkgconf/mlt_<architecture>_<variant>_<platform>_<startup>.mlt

in the platform HAL. This file is used by the MLT to manufacture both the .ldi and .h files. Users should beware that direct edits the either of these files may be overwritten if the MLT is run and regenerates them from the .mlt file.

The names of the .ldi and .h files are defined by macro definitions in pkgconf/system.h. These are CYGHWR_MEMORY_LAYOUT_LDI and CYGHWR_MEMORY_LAYOUT_H respectively. While there will be little need for the application to refer to the .ldi file directly, it may include the .h file as follows:

```
#include CYGHWR_MEMORY_LAYOUT_H
```

Diagnostic Support

The HAL provides support for low level diagnostic IO. This is particularly useful during early development as an aid to bringing up a new platform. Usually this diagnostic channel is a UART or some other serial IO device, but it may equally be a a memory buffer, a simulator supported output channel, a ROM emulator virtual UART, and LCD panel, a memory mapped video buffer or any other output device.

HAL_DIAG_INIT() performs any initialization required on the device being used to generate diagnostic output. This may include, for a UART, setting baud rate, and stop, parity and character bits. For other devices it may include initializing a controller or establishing contact with a remote device.

HAL_DIAG_WRITE_CHAR(c) writes the character supplied to the diagnostic output device.

HAL_DIAG_READ_CHAR(c) reads a character from the diagnostic device into the supplied variable. This is not supported for all diagnostic devices.

These macros are defined in the header file cyg/hal/hal_diag.h. This file is usually supplied by the variant or platform HAL, depending on where the IO device being used is located. For example for on-chip UARTs it would be in the variant HAL, but for a board-level LCD panel it would be in the platform HAL.

SMP Support

eCos contains support for limited Symmetric Multi-Processing (SMP). This is only available on selected architectures and platforms.

Target Hardware Limitations

To allow a reasonable implementation of SMP, and to reduce the disruption to the existing source base, a number of assumptions have been made about the features of the target hardware.

- Modest multiprocessing. The typical number of CPUs supported is two to four, with an upper limit around eight. While there are no inherent limits in the code, hardware and algorithmic limitations will probably become significant beyond this point.
- SMP synchronization support. The hardware must supply a mechanism to allow software on two CPUs to synchronize. This is normally provided as part of the instruction set in the form of test-and-set, compare-and-swap or load-link/store-conditional

instructions. An alternative approach is the provision of hardware semaphore registers which can be used to serialize implementations of these operations. Whatever hardware facilities are available, they are used in eCos to implement spinlocks.

- Coherent caches. It is assumed that no extra effort will be required to access shared memory from any processor. This means that either there are no caches, they are shared by all processors, or are maintained in a coherent state by the hardware. It would be too disruptive to the eCos sources if every memory access had to be bracketed by cache load/flush operations. Any hardware that requires this is not supported.

- Uniform addressing. It is assumed that all memory that is shared between CPUs is addressed at the same location from all CPUs. Like non-coherent caches, dealing with CPU-specific address translation is considered too disruptive to the eCos source base. This does not, however, preclude systems with non-uniform access costs for different CPUs.

- Uniform device addressing. As with access to memory, it is assumed that all devices are equally accessible to all CPUs. Since device access is often made from thread contexts, it is not possible to restrict access to device control registers to certain CPUs.

- Interrupt routing. The target hardware must have an interrupt controller that can route interrupts to specific CPUs. It is acceptable for all interrupts to be delivered to just one CPU, or for some interrupts to be bound to specific CPUs, or for some interrupts to be local to each CPU. At present dynamic routing, where a different CPU may be chosen each time an interrupt is delivered, is not supported. ECos cannot support hardware where all interrupts are delivered to all CPUs simultaneously with the expectation that software will resolve any conflicts.

- Inter-CPU interrupts. A mechanism to allow one CPU to interrupt another is needed. This is necessary so that events on one CPU can cause rescheduling on other CPUs.

- CPU Identifiers. Code running on a CPU must be able to determine which CPU it is running on. The CPU Id is usually provided either in a CPU status register, or in a register associated with the inter-CPU interrupt delivery subsystem. ECos expects CPU Ids to be small positive integers, although alternative representations, such as bitmaps, can be converted relatively easily. Complex mechanisms for getting the CPU Id cannot be supported. Getting the CPU Id must be a cheap operation, since it is done often, and in performance critical places such as interrupt handlers and the scheduler.

HAL Support

SMP support in any platform depends on the HAL supplying the appropriate operations. All HAL SMP support is defined in the `cyg/hal/hal_smp.h` header. Variant and platform specific definitions will be in `cyg/hal/var_smp.h` and `cyg/hal/plf_smp.h` respectively. These files are include automatically by this header, so need not be included explicitly.

SMP support falls into a number of functional groups.

CPU Control

This group consists of descriptive and control macros for managing the CPUs in an SMP system.

`HAL_SMP_CPU_TYPE`

A type that can contain a CPU id. A CPU id is usually a small integer that is used to index arrays of variables that are managed on an per-CPU basis.

`HAL_SMP_CPU_MASK`

A type that can contain a bitmask of all CPUs in the system. In this mask, bit n corresponds to CPU n.

`HAL_SMP_CPU_COUNT`

The maximum number of CPUs that can be supported. This is used to provide the size of any arrays that have an element per CPU.

`HAL_SMP_CPU_MAX`

The maximum possible CPU ID. This should normally be one less that `HAL_SMP_CPU_COUNT`.

`HAL_SMP_CPU_THIS()`

Returns the CPU id of the current CPU.

`HAL_SMP_CPU_NONE`

A value that does not match any real CPU id. This is uses where a CPU type variable must be set to a null value.

`HAL_SMP_CPU_MASK_ALL`

A value for the HAL_SMP_CPU_MASK type that has a bit set for each CPU supported. This value can be derived from HAL_SMP_CPU_COUNT.

`HAL_SMP_CPU_START( cpu )`

Starts the given CPU executing at a defined HAL entry point. After performing any HAL level initialization, the CPU calls up into the kernel at `cyg_kernel_cpu_startup()`.

`HAL_SMP_CPU_RESCHEDULE_INTERRUPT( cpu, wait )`

Sends the CPU a reschedule interrupt, and if *`wait`* is non-zero, waits for an acknowledgment. The interrupted CPU should call `cyg_scheduler_set_need_reschedule()` in its DSR to cause the reschedule to occur.

`HAL_SMP_CPU_TIMESLICE_INTERRUPT( cpu, wait )`

Sends the CPU a timeslice interrupt, and if *`wait`* is non-zero, waits for an acknowledgment. The interrupted CPU should call `cyg_scheduler_timeslice_cpu()` to cause the timeslice event to be processed.

Test-and-set Support

Test-and-set is the foundation of the SMP synchronization mechanisms.

`HAL_TAS_TYPE`

The type for all test-and-set variables. The test-and-set macros only support operations on a single bit (usually the least significant bit) of this location. This allows for maximum flexibility in the implementation.

`HAL_TAS_SET( tas, oldb )`

Performs a test and set operation on the location *`tas`*. *`oldb`* will contain `true` if the location was already set, and `false` if it was clear.

`HAL_TAS_CLEAR( tas, oldb )`

Performs a test and clear operation on the location *`tas`*. *`oldb`* will contain `true` if the location was already set, and `false` if it was clear.

Spinlocks

Spinlocks provide inter-CPU locking. Normally they will be implemented on top of the test-and-set mechanism above, but may also be implemented by other means if, for example, the hardware has more direct support for spinlocks.

`HAL_SPINLOCK_TYPE`

The type for all spinlock variables.

`HAL_SPINLOCK_INIT_CLEAR`

A value that may be assigned to a spinlock variable to initialize it to clear.

`HAL_SPINLOCK_INIT_SET`

A value that may be assigned to a spinlock variable to initialize it to set.

`HAL_SPINLOCK_INIT( lock, val )`

A macro to initialize a spinlock at runtime. The current state of the spinlock is set according to `val`: zero for clear, non-zero for set.

`HAL_SPINLOCK_SPIN( lock )`

The caller spins in a busy loop waiting for the lock to become clear. It then sets it and continues. This is all handled atomically, so that there are no race conditions between CPUs.

`HAL_SPINLOCK_CLEAR( lock )`

The caller clears the lock. One of any waiting spinners will then be able to proceed.

`HAL_SPINLOCK_TRY( lock, val )`

Attempts to set the lock. The value put in *`val`* will be `true` if the lock was claimed successfully, and `false` if it was not.

`HAL_SPINLOCK_TEST( lock, val )`

Tests the current value of the lock. The value put in *`val`* will be `true` if the lock is claimed and `false` of it is clear.

Scheduler Lock

The scheduler lock is the main protection for all kernel data structures. By default the kernel implements the scheduler lock itself using a spinlock. However, if spinlocks cannot be supported by the hardware, or there is a more efficient implementation available, the HAL may provide macros to implement the scheduler lock.

`HAL_SMP_SCHEDLOCK_DATA_TYPE`

A data type, possibly a structure, that contains any data items needed by the scheduler lock implementation. A variable of this type will be instantiated as a static member of the Cyg_Scheduler_SchedLock class and passed to all the following macros.

`HAL_SMP_SCHEDLOCK_INIT( lock, data )`

Initialize the scheduler lock. The *`lock`* argument is the scheduler lock counter and the *`data`* argument is a variable of HAL_SMP_SCHEDLOCK_DATA_TYPE type.

`HAL_SMP_SCHEDLOCK_INC( lock, data )`

Increment the scheduler lock. The first increment of the lock from zero to one for any CPU may cause it to wait until the lock is zeroed by another CPU. Subsequent increments should be less expensive since this CPU already holds the lock.

`HAL_SMP_SCHEDLOCK_ZERO( lock, data )`

Zero the scheduler lock. This operation will also clear the lock so that other CPUs may claim it.

`HAL_SMP_SCHEDLOCK_SET( lock, data, new )`

Set the lock to a different value, in *`new`*. This is only called when the lock is already known to be owned by the current CPU. It is never called to zero the lock, or to increment it from zero.

Interrupt Routing

The routing of interrupts to different CPUs is supported by two new interfaces in hal_intr.h.

Once an interrupt has been routed to a new CPU, the existing vector masking and configuration operations should take account of the CPU routing. For example, if the operation is not invoked on the destination CPU itself, then the HAL may need to arrange to transfer the operation to the destination CPU for correct application.

`HAL_INTERRUPT_SET_CPU( vector, mask )`

Route the interrupt for the given *`vector`* to any of the CPUs whose bit is set in *`mask`*.

`HAL_INTERRUPT_GET_CPU( vector, mask )`

Set *`mask`* to the set of CPUs to which this vector is routed.

Chapter 5. Exception Handling

Most of the HAL consists of simple macros or functions that are called via the interfaces described in the previous section. These just perform whatever operation is required by accessing the hardware and then return. The exception to this is the handling of exceptions: either synchronous hardware traps or asynchronous device interrupts. Here control is passed first to the HAL, which then passed it on to eCos or the application. After eCos has finished with it, control is then passed back to the HAL for it to tidy up the CPU state and resume processing from the point at which the exception occurred.

The HAL exceptions handling code is usually found in the file `vectors.S` in the architecture HAL. Since the reset entry point is usually implemented as one of these it also deals with system startup.

The exact implementation of this code is under the control of the HAL implementer. So long as it interacts correctly with the interfaces defined previously it may take any form. However, all current implementation follow the same pattern, and there should be a very good reason to break with this. The rest of this section describes these operate.

Exception handling normally deals with the following broad areas of functionality:

- Startup and initialization.
- Hardware exception delivery.
- Default handling of synchronous exceptions.
- Default handling of asynchronous interrupts.

HAL Startup

Execution normally begins at the reset vector with the machine in a minimal startup state. From here the HAL needs to get the machine running, set up the execution environment for the application, and finally invoke its entry point.

The following is a list of the jobs that need to be done in approximately the order in which they should be accomplished. Many of these will not be needed in some configurations.

- Initialize the hardware. This may involve initializing several subsystems in both the architecture, variant and platform HALs. These include:
  - Initialize various CPU status registers. Most importantly, the CPU interrupt mask should be set to disable interrupts.
  - Initialize the MMU, if it is used. On many platforms it is only possible to control the cacheability of address ranges via the MMU. Also, it may be necessary to remap RAM and device registers to locations other than their defaults. However, for simplicity, the mapping should be kept as close to one-to-one physical-to-virtual as possible.
  - Set up the memory controller to access RAM, ROM and I/O devices correctly. Until this is done it may not be possible to access RAM. If this is a ROMRAM startup then the program code can now be copied to its RAM address and control transferred to it.
  - Set up any bus bridges and support chips. Often access to device registers needs to go through various bus bridges and other intermediary devices. In many systems these are combined with the memory controller, so it makes sense to set these up together. This is particularly important if early diagnostic output needs to go through one of these devices.
  - Set up diagnostic mechanisms. If the platform includes an LED or LCD output device, it often makes sense to output progress indications on this during startup. This helps with diagnosing hardware and software errors.
  - Initialize floating point and other extensions such as SIMD and multimedia engines. It is usually necessary to enable these and maybe initialize control and exception registers for these extensions.

  - Initialize interrupt controller. At the very least, it should be configured to mask all interrupts. It may also be necessary to set up the mapping from the interrupt controller's vector number space to the CPU's exception number space. Similar mappings may need to be set up between primary and secondary interrupt controllers.
  - Disable and initialize the caches. The caches should not normally be enabled at this point, but it may be necessary to clear or initialize them so that they can be enabled later. Some architectures require that the caches be explicitly reinitialized after a power-on reset.
  - Initialize the timer, clock etc. While the timer used for RTC interrupts will be initialized later, it may be necessary to set up the clocks that drive it here.

  The exact order in which these initializations is done is architecture or variant specific. It is also often not necessary to do anything at all for some of these options. These fragments of code should concentrate on getting the target up and running so that C function calls can be made and code can be run. More complex initializations that cannot be done in assembly code may be postponed until calls to `hal_variant_init()` or `hal_platform_init()` are made.

  Not all of these initializations need to be done for all startup types. In particular, RAM startups can reasonably assume that the ROM monitor or loader has already done most of this work.

- Set up the stack pointer, this allows subsequent initialization code to make proper procedure calls. Usually the interrupt stack is used for this purpose since it is available, large enough, and will be reused for other purposes later.
- Initialize any global pointer register needed for access to globally defined variables. This allows subsequent initialization code to access global variables.
- If the system is starting from ROM, copy the ROM template of the `.data` section out to its correct position in RAM. (the section called “Linker Scripts”).
- Zero the `.bss` section.
- Create a suitable C call stack frame. This may involve making stack space for call frames, and arguments, and initializing the back pointers to halt a GDB backtrace operation.
- Call `hal_variant_init()` and `hal_platform_init()`. These will perform any additional initialization needed by the variant and platform. This typically includes further initialization of the interrupt controller, PCI bus bridges, basic IO devices and enabling the caches.
- Call `cyg_hal_invoke_constructors()` to run any static constructors.
- Call `cyg_start()`. If `cyg_start()` returns, drop into an infinite loop.

Vectors and VSRs

The CPU delivers all exceptions, whether synchronous faults or asynchronous interrupts, to a set of hardware defined vectors. Depending on the architecture, these may be implemented in a number of different ways. Examples of existing mechanisms are:

PowerPC

Exceptions are vectored to locations 256 bytes apart starting at either zero or `0xFFF00000`. There are 16 such vectors defined by the basic architecture and extra vectors may be defined by specific variants. One of the base vectors is for all external interrupts, and another is for the architecture defined timer.

MIPS

Most exceptions and all interrupts are vectored to a single address at either `0x80000000` or `0xBFC00180`. Software is responsible for reading the exception code from the CPU `cause` register to discover its true source. Some TLB and debug exceptions are delivered to different vector addresses, but these are not used currently by eCos. One of the exception codes in the `cause` register indicates an external interrupt. Additional bits in the `cause` register provide a first-level decode for the interrupt source, one of which represents an architecture defined timer.

IA32 Exceptions are delivered via an Interrupt Descriptor Table (IDT) which is essentially an indirection table indexed by exception number. The IDT may be placed anywhere in memory. In PC hardware the standard interrupt controller can be programmed to deliver the external interrupts to a block of 16 vectors at any offset in the IDT. There is no hardware supplied mechanism for determining the vector taken, other than from the address jumped to.

ARM All exceptions, including the FIQ and IRQ interrupts, are vectored to locations four bytes apart starting at zero. There is only room for one instruction here, which must immediately jump out to handling code higher in memory. Interrupt sources have to be decoded entirely from the interrupt controller.

With such a wide variety of hardware approaches, it is not possible to provide a generic mechanism for the substitution of exception vectors directly. Therefore, eCos translates all of these mechanisms in to a common approach that can be used by portable code on all platforms.

The mechanism implemented is to attach to each hardware vector a short piece of trampoline code that makes an indirect jump via a table to the actual handler for the exception. This handler is called the Vector Service Routine (VSR) and the table is called the VSR table.

The trampoline code performs the absolute minimum processing necessary to identify the exception source, and jump to the VSR. The VSR is then responsible for saving the CPU state and taking the necessary actions to handle the exception or interrupt. The entry conditions for the VSR are as close to the raw hardware exception entry state as possible - although on some platforms the trampoline will have had to move or reorganize some registers to do its job.

To make this more concrete, consider how the trampoline code operates in each of the architectures described above:

PowerPC A separate trampoline is contained in each of the vector locations. This code saves a few work registers away to the special purposes registers available, loads the exception number into a register and then uses that to index the VSR table and jump to the VSR. The VSR is entered with some registers move to the SPRs, and one of the data register containing the number of the vector taken.

MIPS A single trampoline routine attached to the common vector reads the exception code out of the `cause` register and uses that value to index the VSR table and jump to the VSR. The trampoline uses the two registers defined in the ABI for kernel use to do this, one of these will contain the exception vector number for the VSR.

IA32 There is a separate 3 or 4 instruction trampoline pointed to by each active IDT table entry. The trampoline for exceptions that also have an error code pop it from the stack and put it into a memory location. Trampolines for non-error-code exceptions just zero the memory location. Then all trampolines push an interrupt/exception number onto the stack, and take an indirect jump through a precalculated offset in the VSR table. This is all done without saving any registers, using memory-only operations. The VSR is entered with the vector number pushed onto the stack on top of the standard hardware saved state.

ARM The trampoline consists solely of the single instruction at the exception entry point. This is an indirect jump via a location 32 bytes higher in memory. These locations, from `0x20` up, form the VSR table. Since each VSR is entered in a different CPU mode (`SVC,UNDEF,ABORT,IRQ or FIQ`) there has to be a different VSR for each exception that knows how to save the CPU state correctly.

Default Synchronous Exception Handling

Most synchronous exception VSR table entries will point to a default exception VSR which is responsible for handling all exceptions in a generic manner. The default VSR simply saves the CPU state, makes any adjustments to the CPU state that is necessary, and calls `cyg_hal_exception_handler()`.

`cyg_hal_exception_handler()` needs to pass the exception on to some handling code. There are two basic destinations: enter GDB or pass the exception up to eCos. Exactly which destination is taken depends on the configuration. When the GDB stubs are included then the exception is passed to them, otherwise it is passed to eCos.

If an eCos application has been loaded by RedBoot then the VSR table entries will all point into RedBoot's exception VSR, and will therefore enter GDB if an exception occurs. If the eCos application wants to handle an exception itself, it needs to replace the the VSR table entry with one pointing to its own VSR. It can do this with the `HAL_VSR_SET_TO_ECOS_HANDLER()` macro.

Default Interrupt Handling

Most asynchronous external interrupt vectors will point to a default interrupt VSR which decodes the actual interrupt being delivered from the interrupt controller and invokes the appropriate ISR.

The default interrupt VSR has a number of responsibilities if it is going to interact with the Kernel cleanly and allow interrupts to cause thread preemption.

To support this VSR an ISR vector table is needed. For each valid vector three pointers need to be stored: the ISR, its data pointer and an opaque (to the HAL) interrupt object pointer needed by the kernel. It is implementation defined whether these are stored in a single table of triples, or in three separate tables.

The VSR follows the following approximate plan:

1. Save the CPU state. In non-debug configurations, it may be possible to get away with saving less than the entire machine state. The option `CYGDBG_HAL_COMMON_INTERRUPTS_SAVE_MINIMUM_CONTEXT` is supported in some targets to do this.

2. Increment the kernel scheduler lock. This is a static member of the Cyg_Scheduler class, however it has also been aliased to `cyg_scheduler_sched_lock` so that it can be accessed from assembly code.

3. (Optional) Switch to an interrupt stack if not already running on it. This allows nested interrupts to be delivered without needing every thread to have a stack large enough to take the maximum possible nesting. It is implementation defined how to detect whether this is a nested interrupt but there are two basic techniques. The first is to inspect the stack pointer and switch only if it is not currently within the interrupt stack range; the second is to maintain a counter of the interrupt nesting level and switch only if it is zero. The option `CYGIMP_HAL_COMMON_INTERRUPTS_USE_INTERRUPT_STACK` controls whether this happens.

4. Decode the actual external interrupt being delivered from the interrupt controller. This will yield the ISR vector number. The code to do this usually needs to come from the variant or platform HAL, so is usually present in the form of a macro or procedure callout.

5. (Optional) Re-enable interrupts to permit nesting. At this point we can potentially allow higher priority interrupts to occur. It depends on the interrupt architecture of the CPU and platform whether more interrupts will occur at this point, or whether they will only be delivered after the current interrupt has been acknowledged (by a call to `HAL_INTERRUPT_ACKNOWLEDGE()` in the ISR).

6. Using the ISR vector number as an index, retrieve the ISR pointer and its data pointer from the ISR vector table.

7. Construct a C call stack frame. This may involve making stack space for call frames, and arguments, and initializing the back pointers to halt a GDB backtrace operation.

8. Call the ISR, passing the vector number and data pointer. The vector number and a pointer to the saved state should be preserved across this call, preferably by storing them in registers that are defined to be callee-saved by the calling conventions.

9. If this is an un-nested interrupt and a separate interrupt stack is being used, switch back to the interrupted thread's own stack.

10. Use the saved ISR vector number to get the interrupt object pointer from the ISR vector table.

11. Call `interrupt_end()` passing it the return value from the ISR, the interrupt object pointer and a pointer to the saved CPU state. This function is implemented by the Kernel and is responsible for finishing off the interrupt handling. Specifically, it may post a DSR depending on the ISR return value, and will decrement the scheduler lock. If the lock is zeroed by this operation then any posted DSRs may be called and may in turn result in a thread context switch.

12. The return from `interrupt_end()` may occur some time after the call. Many other threads may have executed in the meantime. So here all we may do is restore the machine state and resume execution of the interrupted thread. Depending on the architecture, it may be necessary to disable interrupts again for part of this.

The detailed order of these steps may vary slightly depending on the architecture, in particular where interrupts are enabled and disabled.

Chapter 6. HAL GDB File I/O Routines

Name

`hal_gdb_fileio` — access host file system

Synopsis

#include <cyg/hal/hal_gdb_fileio.h>

int **hal_gdb_fileio_open** (*path*, *flags*, *mode*);

int **hal_gdb_fileio_close** (*fd*);

int **hal_gdb_fileio_read** (*fd*, *buffer*, *count*);

int **hal_gdb_fileio_write** (*fd*, *buffer*, *count*);

cyg_int32 **hal_gdb_fileio_lseek** (*fd*, *offset*, *whence*);

int **hal_gdb_fileio_rename** (*oldpath*, *newpath*);

int **hal_gdb_fileio_unlink** (*path*);

int **hal_gdb_fileio_stat** (*path*, *stat*);

int **hal_gdb_fileio_fstat** (*fd*, *stat*);

int **hal_gdb_fileio_gettimeofday** (*tv*, *tz*);

int **hal_gdb_fileio_isatty** (*fd*);

int **hal_gdb_fileio_system** (*command*);

Description

In some configurations an eCos application can perform a number of file I/O and other operations on the host by interacting with gdb. For example the application can open a log file to the host and write very large amounts of debug data to that file over a period of time while consuming minimal target-side resources. However, the application will be completely blocked for the duration of the I/O operation with interrupts globally disabled. The functionality uses the gdb File I/O Remote Protocol Extension, described in the Remote Protocol appendix of the gdb documentation.

The gdb file I/O support is only available when the configuration option `CYGFUN_HAL_GDB_FILEIO` is enabled. In turn that option will have dependencies on other parts of the HAL, and the required functionality will not be available for all targets.

When debugging involves a hardware debug solution such as jtag or BDM, typically gdb will interact with a remote protocol server running inside or controlling the hardware debug unit. That server will implement the core parts of the remote protocol such as accessing memory, but typically there will be no way for the eCos application to get the server to send specific requests such as for file I/O. Instead a different approach is used. From inside the gdb session the command **set hwdebug on** should be used after attaching to the target. The next time the eCos application attempts a file I/O operation it will cause execution to halt at `_gdb_hwdebug_breakpoint`. Code inside gdb recognises that address, retrieves details of the I/O request from the target's memory, and then acts as if the request had come in a remote protocol message. The target resumes execution automatically once the I/O operation has been performed. If the `hwdebug` flag has not been set or if the application is running outside a gdb debug session, the behaviour of eCos is dependant on the `CYGFUN_HAL_DIAG_VIA_GDB_FILEIO` and `CYGSEM_HAL_DIAG_VIA_GDB_FILEIO_IMMEDIATE` configuration options. If both options are set, the target will halt at `_gdb_hwdebug_breakpoint` until a gdb session is established and the command **set hwdebug on** issued. Otherwise the file I/O operations will fail with error code `HAL_GDB_FILEIO_ENOSYS`.

This behaviour is useful during initial debugging using Eclipse as it allows diagnostic messages that would otherwise be discarded to be captured by a gdb session. A typical example is where an application is located in flash and diagnostic messages are issued by eCos before the application's main entry point is reached. During this initial stage of target bring-up eCos's internal data structures may not be adequately initialized so as to allow eCos to determine whether file I/O operations should either: (1) fail with diagnostic messages to `gdb_hwdebug_fileio` discarded (See HAL GDB File I/O Diagnostics Support); or (2) be passed to an underlying gdb session. See Use with the Eclipse CDT extensions for eCosPro application development for additional information regarding the use of Eclipse for debugging.

The Functions

Full details of the functions, parameters, data structures and error codes can be found in the header file `cyg/hal/hal_gdb_fileio.h`. The I/O calls are loosely modelled after the equivalent POSIX calls. They return 0 or a positive number for success, a negative number to indicate an error. The specific error code is the absolute value of the return value, so for example `hal_gdb_fileio_open` will return `-HAL_GDB_FILEIO_ENOENT` when attempting to open a file that does not exist. Some example code can be found in the testcase `gdb_fileio.c`.

`hal_gdb_fileio_open` is used to open a file on the host file system. It should only be used on files, not on special devices such as serial port or Unix-domain sockets: the gdb file I/O functionality is limited and has no support for `select`, non-blocking I/O, `ioctl`-style control, and so on. Hence if the eCos application does attempt to open and read from a serial port that will cause gdb to block and both the application and the debug session will freeze. Valid flags include `HAL_GDB_FILEIO_O_RDONLY`, `HAL_GDB_FILEIO_O_WRONLY`, and `HAL_GDB_FILEIO_O_CREAT`. The *`mode`* argument is used only when creating a new file and is used to set the access rights, for example `HAL_GDB_FILEIO_S_IURSR+HAL_GDB_FILEIO_S_IWUSR`.

The return value of `hal_gdb_fileio_open` is an integer file descriptor. Note that this is distinct from the file descriptor returned by the eCos `open` call. The two types of file descriptor are not interchangeable. For example `hal_gdb_fileio_read` should only be used with a file descriptor returned from `hal_gdb_fileio_open`, not with the return value of `open`.

`hal_gdb_fileio_read`, `hal_gdb_fileio_write`, `hal_gdb_fileio_lseek`, `hal_gdb_fileio_fstat` and `hal_gdb_fileio_isatty` perform operations on a file opened with `hal_gdb_fileio_open`. `hal_gdb_fileio_write` can also be used with the predefined file descriptor `HAL_GDB_FILEIO_STDOUT`, corresponding to gdb's standard output. For `hal_gdb_fileio_lseek` valid *`whence`* parameters are `HAL_GDB_FILEIO_SEEK_SET`, `HAL_GDB_FILEIO_SEEK_CUR` and `HAL_GDB_FILEIO_SEEK_END`. Due to limitations within the protocol and the implementation `hal_gdb_fileio_lseek` cannot fully support files of 2GB or larger. In other words offsets are limited to 31 bits.

The `hal_gdb_fileio_stat` and `hal_gdb_fileio_fstat` functions should be called with a struct hal_gdb-fileio_stat buffer:

```
struct hal_gdb_fileio_stat
{
    cyg_uint32 st_dev;
    cyg_uint32 st_ino;
    cyg_uint32 st_mode;
    cyg_uint32 st_nlink;
    cyg_uint32 st_uid;
    cyg_uint32 st_gid;
    cyg_uint32 st_rdev;
    cyg_uint64 st_size;
    cyg_uint64 st_blksize;
    cyg_uint64 st_blocks;
    cyg_uint32 st_atime;
    cyg_uint32 st_mtime;
    cyg_uint32 st_ctime;
};
```

Warning

The time values obtained are not year 2038 safe, as only space for a 32-bit `time_t` has been allocated. It is hoped that newlib and GDB will update the protocol in order to support 64-bit `time_t`, at which point eCos will be able to conform to whatever mechanism they use to supply the updated time, but this has not yet happened. It is therefore recommended that the time values are not used if this function is intended to be used after year 2038.

The first argument to `hal_gdb_fileio_gettimeofday` should be hal_gdb_fileio_timeval structure:

```
struct hal_gdb_fileio_timeval
{
    cyg_uint32 tv_sec;
    cyg_uint32 tv_usec;
};
```

The second argument to `hal_gdb_fileio_gettimeofday` is not currently used and application code should use a NULL pointer for this.

Warning

Again, the time values obtained are not year 2038 safe, as only space for a 32-bit `time_t` has been allocated. eCos will be updated when the GDB maintainers update the protocol to exchange a 64-bit time value. It is therefore strongly recommended that this function is not used if it may still be in use after year 2038.

`hal_gdb_fileio_system` can be used to invoke an arbitrary command on the host. For obvious security reasons this functionality is disabled within gdb by default. It must be explicitly enabled within gdb using a **set remote system-call-allowed** command.

Diagnostics Support

When the eCos application is built stand-alone and will be debugged via a hardware debug solution such as jtag or BDM, some platforms will allow HAL diagnostics to be sent a destination `gdb_hwdebug_fileio`. This output will end up being written to the gdb console via `hal_gdb_fileio_write` if the application is running inside a gdb session and the **set hwdebug on** command has been used after connecting to the target. Otherwise, if both `CYGFUN_HAL_DIAG_VIA_GDB_FILEIO` and `CYGSEM_HAL_DIAG_VIA_GDB_FILEIO_IMMEDIATE` configuration options are set, the eCos application will halt at a simulated breakpoint at `_gdb_hwdebug_breakpoint` until a gdb session is established. If `CYGSEM_HAL_DIAG_VIA_GDB_FILEIO_IMMEDIATE` is not set, the eCos application will not halt and HAL diagnostics will be discarded. Upon establishing a gdb session, the GDB console command **set hwdebug on** command must be used. On resuming the target and eCos application with the GDB command **continue**, the original HAL diagnostics message will appear within the gdb console and execution of the eCos application continue. Subsequent HAL diagnostics messages will appear without further interaction as GDB automatically resumes the application once `hwdebug` has been set. (See HAL GDB File I/O Description)

If the eCos application is resumed by a gdb session without the **set hwdebug on** command being issued, this output will be discarded and execution resumed until the next time the application performs a GDB file I/O operation and halts at the simulated breakpoint at `_gdb_hwdebug_breakpoint`. This behaviour will continue until **set hwdebug on** is issued as automatic resumption of execution is only performed by GDB once it has been issued. If subsequent HAL diagnostics are to be discarded, the **set hwdebug off** can be issued although this will also disable any further GDB file I/O operations as well.

Use with the Eclipse CDT extensions for eCosPro application development

The eCosCentric CDT extensions for eCosPro application development for the Eclipse Kepler release and above always effectively test if an eCos application is configured with option `CYGFUN_HAL_GDB_FILEIO` enabled and automatically issues the command **set hwdebug on** when enabled. This ensures that all HAL diagnostics messages will be visible within the GDB console window prior to and after the initial application entry point is reached (normally either `main` or `cyg_user_start`). The eCos application developer therefore does not need to be concerned with issuing these GDB commands when using eCosCentric's Eclipse CDT extensions for the Kepler release and above. They only need to configure eCos appropriately when they wish to make use of GDB file I/O operations or capture HAL diagnostics messages when debugging through either jtag or BDM when no alternative stream is available for HAL diagnostics messages (e.g. serial). (See HAL GDB File I/O Diagnostics Support and HAL GDB File I/O Description)

Chapter 7. Porting Guide

Introduction

eCos has been designed to be fairly easy to port to new targets. A target is a specific platform (board) using a given architecture (CPU type). The porting is facilitated by the hierarchical layering of the eCos sources - all architecture and platform specific code is implemented in a HAL (hardware abstraction layer).

By porting the eCos HAL to a new target the core functionality of eCos (infra, kernel, uITRON, etc) will be able to run on the target. It may be necessary to add further platform specific code such as serial drivers, display drivers, ethernet drivers, etc. to get a fully capable system.

This document is intended as a help to the HAL porting process. Due to the nature of a porting job, it is impossible to give a complete description of what has to be done for each and every potential target. This should not be considered a clear-cut recipe - you will probably need to make some implementation decisions, tweak a few things, and just plain have to rely on common sense.

However, what is covered here should be a large part of the process. If you get stuck, you are advised to read the ecos-discuss archive where you may find discussions which apply to the problem at hand. You are also invited to ask questions on the ecos-discuss mailing list to help you resolve problems - but as is always the case with community lists, do not consider it an oracle for any and all questions. Use common sense - if you ask too many questions which could have been answered by reading the documentation, FAQ or source code, you are likely to be ignored.

This document will be continually improved by Red Hat engineers as time allows. Feedback and help with improving the document is sought, so if you have any comments at all, please do not hesitate to post them on ecos-discuss (please prefix the subject with [porting]).

At the moment this document is mostly an outline. There are many details to fill in before it becomes complete. Many places you'll just find a list of keywords / concepts that should be described (please post on ecos-discuss if there are areas you think are not covered).

All pages or sections where the caption ends in [TBD] contain little more than key words and/or random thoughts - there has been no work done as such on the content. The word FIXME may appear in the text to highlight places where information is missing.

HAL Structure

In order to write an eCos HAL it's a good idea to have at least a passing understanding of how the HAL interacts with the rest of the system.

HAL Classes

The eCos HAL consists of four HAL sub-classes. This table gives a brief description of each class and partly reiterates the description in Chapter 2, *Architecture, Variant and Platform*. The links refer to the on-line CVS tree (specifically to the sub-HALs used by the PowerPC MBX target).

HAL type	**Description**	**Functionality Overview**
Common HAL (hal/common)	Configuration options and functionality shared by all HALs.	Generic debugging functionality, driver API, eCos/ROM monitor calling interface, and tests.
Architecture HAL (hal/<architecture>/arch)	Functionality specific to the given architecture. Also default implementations of some functionality which can be overridden by variant or platform HALs.	Architecture specific debugger functionality (handles single stepping, exception-to-signal conversion, etc.), exception/interrupt vector definitions and

HAL type	Description	Functionality Overview
		handlers, cache definition and control macros, context switching code, assembler functions for early system initialization, configuration options, and possibly tests.
Variant HAL (hal/<architecture>/<variant>)	Some CPU architectures consist of a number variants, for example MIPS CPUs come in both 32 and 64 bit versions, and some variants have embedded features additional to the CPU core.	Variant extensions to the architecture code (cache, exception/interrupt), configuration options, possibly drivers for variant on-core devices, and possibly tests.
Platform HAL (hal/<architecture>/<platform>)	Contains functionality and configuration options specific to the platform.	Early platform initialization code, platform memory layout specification, configuration options (processor speed, compiler options), diagnostic IO functions, debugger IO functions, platform specific extensions to architecture or variant code (off-core interrupt controller), and possibly tests.
Auxiliary HAL (hal/<architecture>/<module>)	Some variants share common modules on the core. Motorola's PowerPC QUICC is an example of such a module.	Module specific functionality (interrupt controller, simple device drivers), possibly tests.

File Descriptions

Listed below are the files found in various HALs, with a short description of what each file contains. When looking in existing HALs beware that they do not necessarily follow this naming scheme. If you are writing a new HAL, please try to follow it as closely as possible. Still, no two targets are the same, so sometimes it makes sense to use additional files.

Common HAL

File	Description
`include/dbg-thread-syscall.h`	Defines the thread debugging syscall function. This is used by the ROM monitor to access the thread debugging API in the RAM application. .
`include/dbg-threads-api.h`	Defines the thread debugging API. .
`include/drv_api.h`	Defines the driver API.
`include/generic-stub.h`	Defines the generic stub features.
`include/hal_if.h`	Defines the ROM/RAM calling interface API.
`include/hal_misc.h`	Defines miscellaneous helper functions shared by all HALs.
`include/hal_stub.h`	Defines eCos mappings of GDB stub features.
`src/dbg-threads-syscall.c`	Thread debugging implementation.
`src/drv_api.c`	Driver API implementation. Depending on configuration this provides either wrappers for the kernel API, or a minimal implementation of these features. This allows drivers to be written relying only on HAL features.
`src/dummy.c`	Empty dummy file ensuring creation of libtarget.a.
`src/generic-stub.c`	Generic GDB stub implementation. This provides the communication protocol used to communicate with GDB over a serial device or via the network.

File	Description
`src/hal_if.c`	ROM/RAM calling interface implementation. Provides wrappers from the calling interface API to the eCos features used for the implementation.
`src/hal_misc.c`	Various helper functions shared by all platforms and architectures.
`src/hal_stub.c`	Wrappers from eCos HAL features to the features required by the generic GDB stub.
`src/stubrom/stubrom.c`	The file used to build eCos GDB stub images. Basically a cyg_start function with a hard coded breakpoint.
`src/thread-packets.c`	More thread debugging related functions.
`src/thread-pkts.h`	Defines more thread debugging related function.

Architecture HAL

Some architecture HALs may add extra files for architecture specific serial drivers, or for handling interrupts and exceptions if it makes sense.

Note that many of the definitions in these files are only conditionally defined - if the equivalent variant or platform headers provide the definitions, those override the generic architecture definitions.

File	Description
`include/arch.inc`	Various assembly macros used during system initialization.
`include/basetype.h`	Endian, label, alignment, and type size definitions. These override common defaults in CYGPKG_INFRA.
`include/hal_arch.h`	Saved register frame format, various thread, register and stack related macros.
`include/hal_cache.h`	Cache definitions and cache control macros.
`include/hal_intr.h`	Exception and interrupt definitions. Macros for configuring and controlling interrupts. eCos real-time clock control macros.
`include/hal_io.h`	Macros for accessing IO devices.
`include/<arch>_regs.h`	Architecture register definitions.
`include/<arch>_stub.h`	Architecture stub definitions. In particular the register frame layout used by GDB. This may differ from the one used by eCos.
`include/<arch>.inc`	Architecture convenience assembly macros.
`src/<arch>.ld`	Linker macros.
`src/context.S`	Functions handling context switching and setjmp/longjmp.
`src/hal_misc.c`	Exception and interrupt handlers in C. Various other utility functions.
`src/hal_mk_defs.c`	Used to export definitions from C header files to assembler header files.
`src/hal_intr.c`	Any necessary interrupt handling functions.
`src/<arch>stub.c`	Architecture stub code. Contains functions for translating eCos exceptions to UNIX signals and functions for single-stepping.
`src/vectors.S`	Exception, interrupt and early initialization code.

Variant HAL

Some variant HALs may add extra files for variant specific serial drivers, or for handling interrupts/exceptions if it makes sense.

Note that these files may be mostly empty if the CPU variant can be controlled by the generic architecture macros. The definitions present are only conditionally defined - if the equivalent platform headers provide the definitions, those override the variant definitions.

File	Description
`include/var_arch.h`	Saved register frame format, various thread, register and stack related macros.
`include/var_cache.h`	Cache related macros.
`include/var_intr.h`	Interrupt related macros.
`include/var_regs.h`	Extra register definitions for the CPU variant.
`include/variant.inc`	Various assembly macros used during system initialization.
`src/var_intr.c`	Interrupt functions if necessary.
`src/var_misc.c`	hal_variant_init function and any necessary extra functions.
`src/variant.S`	Interrupt handler table definition.
`src/<arch>_<variant>.ld`	Linker macros.

Platform HAL

Extras files may be added for platform specific serial drivers. Extra files for handling interrupts and exceptions will be present if it makes sense.

File	Description
`include/hal_diag.h`	Defines functions used for HAL diagnostics output. This would normally be the ROM calling interface wrappers, but may also be the low-level IO functions themselves, saving a little overhead.
`include/platform.inc`	Platform initialization code. This includes memory controller, vectors, and monitor initialization. Depending on the architecture, other things may need defining here as well: interrupt decoding, status register initialization value, etc.
`include/plf_cache.h`	Platform specific cache handling.
`include/plf_intr.h`	Platform specific interrupt handling.
`include/plf_io.h`	PCI IO definitions and macros. May also be used to override generic HAL IO macros if the platform endianness differs from that of the CPU.
`include/plf_stub.h`	Defines stub initializer and board reset details.
`src/hal_diag.c`	May contain the low-level device drivers. But these may also reside in plf_stub.c
`src/platform.S`	Memory controller setup macro, and if necessary interrupt springboard code.
`src/plf_misc.c`	Platform initialization code.
`src/plf_mk_defs.c`	Used to export definitions from C header files to assembler header files.
`src/plf_stub.c`	Platform specific stub initialization and possibly the low-level device driver.

The platform HAL also contains files specifying the platform's memory layout. These files are located in `include/pkgconf`.

Auxiliary HAL

Auxiliary HALs contain whatever files are necessary to provide the required functionality. There are no predefined set of files required in an auxiliary HAL.

Virtual Vectors (eCos/ROM Monitor Calling Interface)

Virtually all eCos platforms provide full debugging capabilities via RedBoot. This environment contains not only debug stubs based on GDB, but also rich I/O support which can be exported to loaded programs. Such programs can take advantage of the I/O capabilities using a special ROM/RAM calling interface (also referred to as virtual vector table). eCos programs make use of the virtual vector mechanism implicitly. Non-eCos programs can access these functions using the support from the *newlib* library.

Virtual Vectors

What are virtual vectors, what do they do, and why are they needed?

"Virtual vectors" is the name of a table located at a static location in the target address space. This table contains 64 vectors that point to *service* functions or data.

The fact that the vectors are always placed at the same location in the address space means that both ROM and RAM startup configurations can access these and thus the services pointed to.

The primary goal is to allow services to be provided by ROM configurations (ROM monitors such as RedBoot in particular) with *clients* in RAM configurations being able to use these services.

Without the table of pointers this would be impossible since the ROM and RAM applications would be linked separately - in effect having separate name spaces - preventing direct references from one to the other.

This decoupling of service from client is needed by RedBoot, allowing among other things debugging of applications which do not contain debugging client code (stubs).

Initialization (or Mechanism vs. Policy)

Virtual vectors are a *mechanism* for decoupling services from clients in the address space.

The mechanism allows services to be implemented by a ROM monitor, a RAM application, to be switched out at run-time, to be disabled by installing pointers to dummy functions, etc.

The appropriate use of the mechanism is specified loosely by a *policy*. The general policy dictates that the vectors are initialized in whole by ROM monitors (built for ROM or RAM), or by stand-alone applications.

For configurations relying on a ROM monitor environment, the policy is to allow initialization on a service by service basis. The default is to initialize all services, except COMMS services since these are presumed to already be carrying a communication session to the debugger / console which was used for launching the application. This means that the bulk of the code gets tested in normal builds, and not just once in a blue moon when building new stubs or a ROM configuration.

The configuration options are written to comply with this policy by default, but can be overridden by the user if desired. Defaults are:

- For application development: the ROM monitor provides debugging and diagnostic IO services, the RAM application relies on these by default.
- For production systems: the application contains all the necessary services.

Pros and Cons of Virtual Vectors

There are pros and cons associated with the use of virtual vectors. We do believe that the pros generally outweigh the cons by a great margin, but there may be situations where the opposite is true.

The use of the services are implemented by way of macros, meaning that it is possible to circumvent the virtual vectors if desired. There is (as yet) no implementation for doing this, but it is possible.

Here is a list of pros and cons:

Pro: Allows debugging without including stubs

> This is the primary reason for using virtual vectors. It allows the ROM monitor to provide most of the debugging infrastructure, requiring only the application to provide hooks for asynchronous debugger interrupts and for accessing kernel thread information.

Pro: Allows debugging to be initiated from arbitrary channel

> While this is only true where the application does not actively override the debugging channel setup, it is a very nice feature during development. In particular it makes it possible to launch (and/or debug) applications via Ethernet even though the application configuration does not contain networking support.

Pro: Image smaller due to services being provided by ROM monitor

> All service functions except HAL IO are included in the default configuration. But if these are all disabled the image for download will be a little smaller. Probably doesn't matter much for regular development, but it is a worthwhile saving for the 20000 daily tests run in the Red Hat eCos test farm.

Con: The vectors add a layer of indirection, increasing application size and reducing performance.

> The size increase is a fraction of what is required to implement the services. So for RAM configurations there is a net saving, while for ROM configurations there is a small overhead.
>
> The performance loss means little for most of the services (of which the most commonly used is diagnostic IO which happens via polled routines anyway).

Con: The layer of indirection is another point of failure.

> The concern primarily being that of vectors being trashed by rogue writes from bad code, causing a complete loss of the service and possibly a crash. But this does not differ much from a rogue write to anywhere else in the address space which could cause the same amount of mayhem. But it is arguably an additional point of failure for the service in question.

Con: All the indirection stuff makes it harder to bring a HAL up

> This is a valid concern. However, seeing as most of the code in question is shared between all HALs and should remain unchanged over time, the risk of it being broken when a new HAL is being worked on should be minimal.
>
> When starting a new port, be sure to implement the HAL IO drivers according to the scheme used in other drivers, and there should be no problem.
>
> However, it is still possible to circumvent the vectors if they are suspect of causing problems: simply change the HAL_DIAG_INIT and HAL_DIAG_WRITE_CHAR macros to use the raw IO functions.

Available services

The `hal_if.h` file in the common HAL defines the complete list of available services. A few worth mentioning in particular:

- COMMS services. All HAL IO happens via the communication channels.

- uS delay. Fine granularity (busy wait) delay function.
- Reset. Allows a software initiated reset of the board.

The COMMS channels

As all HAL IO happens via the COMMS channels these deserve to be described in a little more detail. In particular the controls of where diagnostic output is routed and how it is treated to allow for display in debuggers.

Console and Debugging Channels

There are two COMMS channels - one for console IO and one for debugging IO. They can be individually configured to use any of the actual IO ports (serial or Ethernet) available on the platform.

The console channel is used for any IO initiated by calling the `diag_*()` functions. Note that these should only be used during development for debugging, assertion and possibly tracing messages. All proper IO should happen via proper devices. This means it should be possible to remove the HAL device drivers from production configurations where assertions are disabled.

The debugging channel is used for communication between the debugger and the stub which remotely controls the target for the debugger (the stub runs on the target). This usually happens via some protocol, encoding commands and replies in some suitable form.

Having two separate channels allows, e.g., for simple logging without conflicts with the debugger or interactive IO which some debuggers do not allow.

Mangling

As debuggers usually have a protocol using specialized commands when communicating with the stub on the target, sending out text as raw ASCII from the target on the same channel will either result in protocol errors (with loss of control over the target) or the text may just be ignored as junk by the debugger.

To get around this, some debuggers have a special command for text output. Mangling is the process of encoding diagnostic ASCII text output in the form specified by the debugger protocol.

When it is necessary to use mangling, i.e. when writing console output to the same port used for debugging, a mangler function is installed on the console channel which mangles the text and passes it on to the debugger channel.

Controlling the Console Channel

Console output configuration is either inherited from the ROM monitor launching the application, or it is specified by the application. This is controlled by the new option `CYGSEM_HAL_VIRTUAL_VECTOR_INHERIT_CONSOLE` which defaults to enabled when the configuration is set to use a ROM monitor.

If the user wants to specify the console configuration in the application image, there are two new options that are used for this.

Defaults are to direct diagnostic output via a mangler to the debugging channel (`CYGDBG_HAL_DIAG_TO_DEBUG_CHAN` enabled). The mangler type is controlled by the option `CYGSEM_HAL_DIAG_MANGLER`. At present there are only two mangler types:

GDB
: This causes a mangler appropriate for debugging with GDB to be installed on the console channel.

None
: This causes a NULL mangler to be installed on the console channel. It will redirect the IO to/from the debug channel without mangling of the data. This option differs from setting the console channel to the same IO port as the debugging channel in that it will keep redirecting data to the debugging channel even if that is changed to some other port.

Finally, by disabling `CYGDBG_HAL_DIAG_TO_DEBUG_CHAN`, the diagnostic output is directed in raw form to the specified console IO port.

In summary this results in the following common configuration scenarios for RAM startup configurations:

- For regular debugging with diagnostic output appearing in the debugger, mangling is enabled and stubs disabled.

  Diagnostic output appears via the debugging channel as initiated by the ROM monitor, allowing for correct behavior whether the application was launched via serial or Ethernet, from the RedBoot command line or from a debugger.

- For debugging with raw diagnostic output, mangling is disabled.

  Debugging session continues as initiated by the ROM monitor, whether the application was launched via serial or Ethernet. Diagnostic output is directed at the IO port configured in the application configuration.

Note:

There is one caveat to be aware of. If the application uses proper devices (be it serial or Ethernet) on the same ports as those used by the ROM monitor, the connections initiated by the ROM monitor will be terminated.

And for ROM startup configurations:

- Production configuration with raw output and no debugging features (configured for RAM or ROM), mangling is disabled, no stubs are included.

  Diagnostic output appears (in unmangled form) on the specified IO port.

- RedBoot configuration, includes debugging features and necessary mangling.

  Diagnostic and debugging output port is auto-selected by the first connection to any of the supported IO ports. Can change from interactive mode to debugging mode when a debugger is detected - when this happens a mangler will be installed as required.

- GDB stubs configuration (obsoleted by RedBoot configuration), includes debugging features, mangling is hardwired to GDB protocol.

  Diagnostic and debugging output is hardwired to configured IO ports, mangling is hardwired.

Footnote: Design Reasoning for Control of Console Channel

The current code for controlling the console channel is a replacement for an older implementation which had some shortcomings which addressed by the new implementation.

This is what the old implementation did: on initialization it would check if the CDL configured console channel differed from the active debug channel - and if so, set the console channel, thereby disabling mangling.

The idea was that whatever channel was configured to be used for console (i.e., diagnostic output) in the application was what should be used. Also, it meant that if debug and console channels were normally the same, a changed console channel would imply a request for unmangled output.

But this prevented at least two things:

- It was impossible to inherit the existing connection by which the application was launched (either by RedBoot commands via telnet, or by via a debugger).

  This was mostly a problem on targets supporting Ethernet access since the diagnostic output would not be returned via the Ethernet connection, but on the configured serial port.

The problem also occurred on any targets with multiple serial ports where the ROM monitor was configured to use a different port than the CDL defaults.

- Proper control of when to mangle or just write out raw ASCII text.

  Sometimes it's desirable to disable mangling, even if the channel specified is the same as that used for debugging. This usually happens if GDB is used to download the application, but direct interaction with the application on the same channel is desired (GDB protocol only allows output from the target, no input).

The calling Interface API

The calling interface API is defined by hal_if.h and hal_if.c in hal/common.

The API provides a set of services. Different platforms, or different versions of the ROM monitor for a single platform, may implement fewer or extra service. The table has room for growth, and any entries which are not supported map to a NOP-service (when called it returns 0 (`false`)).

A client of a service should either be selected by configuration, or have suitable fall back alternatives in case the feature is not implemented by the ROM monitor.

Note:

Checking for unimplemented service when this may be a data field/pointer instead of a function: suggest reserving the last entry in the table as the NOP-service pointer. Then clients can compare a service entry with this pointer to determine whether it's initialized or not.

The header file `cyg/hal/hal_if.h` defines the table layout and accessor macros (allowing primitive type checking and alternative implementations should it become necessary).

The source file `hal_if.c` defines the table initialization function. All HALs should call this during platform initialization - the table will get initialized according to configuration. Also defined here are wrapper functions which map between the calling interface API and the API of the used eCos functions.

Implemented Services

This is a brief description of the services, some of which are described in further detail below.

`VERSION`
Version of table. Serves as a way to check for how many features are available in the table. This is the index of the last service in the table.

`KILL_VECTOR`
[Presently unused by the stub code, but initialized] This vector defines a function to execute when the system receives a kill signal from the debugger. It is initialized with the reset function (see below), but the application (or eCos) can override it if necessary.

`CONSOLE_PROCS`
The communication procedure table used for console IO (see the section called “IO channels”.

`DEBUG_PROCS`
The communication procedure table used for debugger IO (see the section called “IO channels”).

`FLUSH_DCACHE`
Flushes the data cache for the specified region. Some implementations may flush the entire data cache.

`FLUSH_ICACHE`
Flushes (invalidates) the instruction cache for the specified region. Some implementations may flush the entire instruction cache.

`SET_DEBUG_COMM` Change debugging communication channel.

`SET_CONSOLE_COMM` Change console communication channel.

`DBG_SYSCALL` Vector used to communication between debugger functions in ROM and in RAM. RAM eCos configurations may install a function pointer here which the ROM monitor uses to get thread information from the kernel running in RAM.

`RESET` Resets the board on call. If it is not possible to reset the board from software, it will jump to the ROM entry point which will perform a "software" reset of the board.

`CONSOLE_INTERRUPT_FLAG` Set if a debugger interrupt request was detected while processing console IO. Allows the actual breakpoint action to be handled after return to RAM, ensuring proper backtraces etc.

`DELAY_US` Will delay the specified number of microseconds. The precision is platform dependent to some extend - a small value (<100us) is likely to cause bigger delays than requested.

`FLASH_CFG_OP` For accessing configuration settings kept in flash memory.

`INSTALL_BPT_FN` Installs a breakpoint at the specified address. This is used by the asynchronous breakpoint support (see).

Compatibility

When a platform is changed to support the calling interface, applications will use it if so configured. That means that if an application is run on a platform with an older ROM monitor, the service is almost guaranteed to fail.

For this reason, applications should only use Console Comm for HAL diagnostics output if explicitly configured to do so (`CYGSEM_HAL_VIRTUAL_VECTOR_DIAG`).

As for asynchronous GDB interrupts, the service will always be used. This is likely to cause a crash under older ROM monitors, but this crash may be caught by the debugger. The old workaround still applies: if you need asynchronous breakpoints or thread debugging under older ROM monitors, you may have to include the debugging support when configuring eCos.

Implementation details

During the startup of a ROM monitor, the calling table will be initialized. This also happens if eCos is configured *not* to rely on a ROM monitor.

Note:

There is reserved space (256 bytes) for the vector table whether it gets used or not. This may be something that we want to change if we ever have to shave off every last byte for a given target.

If thread debugging features are enabled, the function for accessing the thread information gets registered in the table during startup of a RAM startup configuration.

Further implementation details are described where the service itself is described.

New Platform Ports

The `hal_platform_init()` function must call `hal_if_init()`.

The HAL serial driver must, when called via `cyg_hal_plf_comms_init()` must initialize the communication channels.

The `reset()` function defined in `hal_if.c` will attempt to do a hardware reset, but if this fails it will fall back to simply jumping to the reset entry-point. On most platforms the startup initialization will go a long way to reset the target to a sane state (there will be exceptions, of course). For this reason, make sure to define `HAL_STUB_PLATFORM_RESET_ENTRY` in plf_stub.h.

All debugging features must be in place in order for the debugging services to be functional. See general platform porting notes.

New architecture ports

There are no specific requirements for a new architecture port in order to support the calling interface, but the basic debugging features must be in place. See general architecture porting notes.

IO channels

The calling interface provides procedure tables for all IO channels on the platform. These are used for console (diagnostic) and debugger IO, allowing a ROM monitor to provided all the needed IO routines. At the same time, this makes it easy to switch console/debugger channels at run-time (the old implementation had hardwired drivers for console and debugger IO, preventing these to change at run-time).

The hal_if provides wrappers which interface these services to the eCos infrastructure diagnostics routines. This is done in a way which ensures proper string mangling of the diagnostics output when required (e.g. O-packetization when using a GDB compatible ROM monitor).

Available Procedures

This is a brief description of the procedures

`CH_DATA`

Pointer to the controller IO base (or a pointer to a per-device structure if more data than the IO base is required). All the procedures below are called with this data item as the first argument.

`WRITE`

Writes the buffer to the device.

`READ`

Fills a buffer from the device.

`PUTC`

Write a character to the device.

`GETC`

Read a character from the device.

`CONTROL`

Device feature control. Second argument specifies function:

`SETBAUD`	Changes baud rate.
`GETBAUD`	Returns the current baud rate.
`INSTALL_DBG_ISR`	[Unused]
`REMOVE_DBG_ISR`	[Unused]
`IRQ_DISABLE`	Disable debugging receive interrupts on the device.
`IRQ_ENABLE`	Enable debugging receive interrupts on the device.

`DBG_ISR_VECTOR` Returns the ISR vector used by the device for debugging receive interrupts.

`SET_TIMEOUT` Set GETC timeout in milliseconds.

`FLUSH_OUTPUT` Forces driver to flush data in its buffers. Note that this may not affect hardware buffers (e.g. FIFOs).

`DBG_ISR`

ISR used to handle receive interrupts from the device (see).

`GETC_TIMEOUT`

Read a character from the device with timeout.

Usage

The standard eCos diagnostics IO functions use the channel procedure table when `CYGSEM_HAL_VIRTUAL_VECTOR_DIAG` is enabled. That means that when you use diag_printf (or the libc printf function) the stream goes through the selected console procedure table. If you use the virtual vector function SET_CONSOLE_COMM you can change the device which the diagnostics output goes to at run-time.

You can also use the table functions directly if desired (regardless of the `CYGSEM_HAL_VIRTUAL_VECTOR_DIAG` setting - assuming the ROM monitor provides the services). Here is a small example which changes the console to use channel 2, fetches the comm procs pointer and calls the write function from that table, then restores the console to the original channel:

```
#define T "Hello World!\n"

int
main(void)
{
    hal_virtual_comm_table_t* comm;
    int cur = CYGACC_CALL_IF_SET_CONSOLE_COMM(CYGNUM_CALL_IF_SET_COMM_ID_QUERY_CURRENT);

    CYGACC_CALL_IF_SET_CONSOLE_COMM(2);

    comm = CYGACC_CALL_IF_CONSOLE_PROCS();
    CYGACC_COMM_IF_WRITE(*comm, T, strlen(T));

    CYGACC_CALL_IF_SET_CONSOLE_COMM(cur);
}
```

Beware that if doing something like the above, you should only do it to a channel which does not have GDB at the other end: GDB ignores raw data, so you would not see the output.

Compatibility

The use of this service is controlled by the option `CYGSEM_HAL_VIRTUAL_VECTOR_DIAG` which is disabled per default on most older platforms (thus preserving backwards compatibility with older stubs). On newer ports, this option should always be set.

Implementation Details

There is an array of procedure tables (raw comm channels) for each IO device of the platform which get initialized by the ROM monitor, or optionally by a RAM startup configuration (allowing the RAM configuration to take full control of the target). In addition to this, there's a special table which is used to hold mangler procedures.

The vector table defines which of these channels are selected for console and debugging IO respectively: console entry can be empty, point to mangler channel, or point to a raw channel. The debugger entry should always point to a raw channel.

During normal console output (i.e., diagnostic output) the console table will be used to handle IO if defined. If not defined, the debug table will be used.

This means that debuggers (such as GDB) which require text streams to be mangled (O-packetized in the case of GDB), can rely on the ROM monitor install mangling IO routines in the special mangler table and select this for console output. The mangler will pass the mangled data on to the selected debugging channel.

If the eCos configuration specifies a different console channel from that used by the debugger, the console entry will point to the selected raw channel, thus overriding any mangler provided by the ROM monitor.

See hal_if_diag_* routines in hal_if.c for more details of the stream path of diagnostic output. See `cyg_hal_gdb_diag_*()` routines in `hal_stub.c` for the mangler used for GDB communication.

New Platform Ports

Define CDL options `CYGNUM_HAL_VIRTUAL_VECTOR_COMM_CHANNELS`, `CYGNUM_HAL_VIRTUAL_VECTOR_DEBUG_CHANNEL`, and `CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL`.

If `CYGSEM_HAL_VIRTUAL_VECTOR_DIAG` is set, make sure the infra diag code uses the hal_if diag functions:

```
#define HAL_DIAG_INIT() hal_if_diag_init()
#define HAL_DIAG_WRITE_CHAR(_c_) hal_if_diag_write_char(_c_)
#define HAL_DIAG_READ_CHAR(_c_) hal_if_diag_read_char(&_c_)
```

In addition to the above functions, the platform HAL must also provide a function cyg_hal_plf_comms_init which initializes the drivers and the channel procedure tables.

Most of the other functionality in the table is more or less possible to copy unchanged from existing ports. Some care is necessary though to ensure the proper handling of interrupt vectors and timeouts for various devices handled by the same driver. See PowerPC/Cogent platform HAL for an example implementation.

Note:

When vector table console code is *not* used, the platform HAL must map the HAL_DIAG_INIT, HAL_DIAG_WRITE_CHAR and HAL_DIAG_READ_CHAR macros directly to the low-level IO functions, hardwired to use a compile-time configured channel.

Note:

On old ports the hardwired `HAL_DIAG_INIT`, `HAL_DIAG_WRITE_CHAR` and `HAL_DIAG_READ_CHAR` implementations will also contain code to O-packetize the output for GDB. This should *not* be adopted for new ports! On new ports the ROM monitor is guaranteed to provide the necessary mangling via the vector table. The hardwired configuration should be reserved for ROM startups where achieving minimal image size is crucial.

HAL Coding Conventions

To get changes and larger submissions included into the eCos source repository, we ask that you adhere to a set of coding conventions. The conventions are defined as an attempt to make a consistent tree. Consistency makes it easier for people to read, understand and maintain the code, which is important when many people work on the same project.

The below is only a brief, and probably incomplete, summary of the rules. Please look through files in the area where you are making changes to get a feel for any additional conventions. Also feel free to ask on the list if you have specific questions.

Implementation issues

There are a few implementation issues that should be kept in mind:

HALs

HALs must be written in C and assembly only. C++ must not be used. This is in part to keep the HALs simple since this is usually the first part of eCos a newcomer will see, and in part to maintain the existing de facto standard.

IO access

Use HAL IO access macros for code that might be reused on different platforms than the one you are writing it for.

MMU

If it is necessary to use the MMU (e.g., to prevent caching of IO areas), use a simple 1-1 mapping of memory if possible. On most platforms where using the MMU is necessary, it will be possible to achieve the 1-1 mapping using the MMU's provision for mapping large continuous areas (hardwired TLBs or BATs). This reduces the footprint (no MMU table) and avoids execution overhead (no MMU-related exceptions).

Assertions

The code should contain assertions to validate argument values, state information and any assumptions the code may be making. Assertions are not enabled in production builds, so liberally sprinkling assertions throughout the code is good.

Testing

The ability to test your code is very important. In general, do not add new code to the eCos runtime unless you also add a new test to exercise that code. The test also serves as an example of how to use the new code.

Source code details

Line length

Keep line length below 78 columns whenever possible.

Comments

Whenever possible, use // comments instead of /**/.

Indentation

Use spaces instead of TABs. Indentation level is 4. Braces start on the same line as the expression. See below for emacs mode details.

```
;;=================================================================
;; eCos C/C++ mode Setup.
;;
;; bsd mode: indent = 4
;; tail comments are at col 40.
;; uses spaces not tabs in C;

(defun ecos-c-mode ()
  "C mode with adjusted defaults for use with the eCos sources."
  (interactive)
  (c++-mode)
  (c-set-style "bsd")
  (setq comment-column 40)
  (setq indent-tabs-mode nil)
  (show-paren-mode 1)
  (setq c-basic-offset 4)

  (set-variable 'add-log-full-name "Your Name")
  (set-variable 'add-log-mailing-address "Your email address"))

(defun ecos-asm-mode ()
  "ASM mode with adjusted defaults for use with the eCos sources."
  (interactive)
  (setq comment-column 40)
  (setq indent-tabs-mode nil)
  (asm-mode)
  (setq c-basic-offset 4)

  (set-variable 'add-log-full-name "Your Name")
  (set-variable 'add-log-mailing-address "Your email address"))

(setq auto-mode-alist
  (append '(("/local/ecc/.*\\.C$"   . ecos-c-mode)
  ("/local/ecc/.*\\.cc$"  . ecos-c-mode)
  ("/local/ecc/.*\\.cpp$" . ecos-c-mode)
  ("/local/ecc/.*\\.inl$" . ecos-c-mode)
  ("/local/ecc/.*\\.c$"   . ecos-c-mode)
  ("/local/ecc/.*\\.h$"   . ecos-c-mode)
  ("/local/ecc/.*\\.S$"   . ecos-asm-mode)
  ("/local/ecc/.*\\.inc$" . ecos-asm-mode)
  ("/local/ecc/.*\\.cdl$" . tcl-mode)
) auto-mode-alist))
```

Nested Headers

In order to allow platforms to define all necessary details, while still maintaining the ability to share code between common platforms, all HAL headers are included in a nested fashion.

The architecture header (usually `hal_XXX.h`) includes the variant equivalent of the header (`var_XXX.h`) which in turn includes the platform equivalent of the header (`plf_XXX.h`).

All definitions that may need to be overridden by a platform are then only conditionally defined, depending on whether a lower layer has already made the definition:

```
hal_intr.h:     #include <var_intr.h>

                #ifndef MACRO_DEFINED
                # define MACRO ...
                # define MACRO_DEFINED
                #endif



var_intr.h:     #include <plf_intr.h>

                #ifndef MACRO_DEFINED
                # define MACRO ...
                # define MACRO_DEFINED
                #endif


plf_intr.h:

                # define MACRO ...
                # define MACRO_DEFINED
```

This means a platform can opt to rely on the variant or architecture implementation of a feature, or implement it itself.

Platform HAL Porting

This is the type of port that takes the least effort. It basically consists of describing the platform (board) for the HAL: memory layout, early platform initialization, interrupt controllers, and a simple serial device driver.

Doing a platform port requires a preexisting architecture and possibly a variant HAL port.

HAL Platform Porting Process

Brief overview

The easiest way to make a new platform HAL is simply to copy an existing platform HAL of the same architecture/variant and change all the files to match the new one. In case this is the first platform for the architecture/variant, a platform HAL from another architecture should be used as a template.

The best way to start a platform port is to concentrate on getting RedBoot to run. RedBoot is a simpler environment than full eCos, it does not use interrupts or threads, but covers most of the basic startup requirements.

RedBoot normally runs out of FLASH or ROM and provides program loading and debugging facilities. This allows further HAL development to happen using RAM startup configurations, which is desirable for the simple reason that downloading an image which you need to test is often many times faster than either updating a flash part, or indeed, erasing and reprogramming an EPROM.

There are two approaches to getting to this first goal:

1. The board is equipped with a ROM monitor which allows "load and go" of ELF, binary, S-record or some other image type which can be created using objcopy. This allows you to develop RedBoot by downloading and running the code (saving time).

   When the stub is running it is a good idea to examine the various hardware registers to help you write the platform initialization code.

   Then you may have to fiddle a bit going through step two (getting it to run from ROM startup). If at all possible, preserve the original ROM monitor so you can revert to it if necessary.

2. The board has no ROM monitor. You need to get the platform initialization and stub working by repeatedly making changes, updating flash or EPROM and testing the changes. If you are lucky, you have a JTAG or similar CPU debugger to help you. If not, you will probably learn to appreciate LEDs. This approach may also be needed during the initial phase of moving RedBoot from RAM startup to ROM, since it is very unlikely to work first time.

Step-by-step

Given that no two platforms are exactly the same, you may have to deviate from the below. Also, you should expect a fair amount of fiddling - things almost never go right the first time. See the hints section below for some suggestions that might help debugging.

The description below is based on the HAL layout used in the MIPS, PC and MN10300 HALs. Eventually all HALs should be converted to look like these - but in a transition period there will be other HALs which look substantially different. Please try to adhere to the following as much is possible without causing yourself too much grief integrating with a HAL which does not follow this layout.

Minimal requirements

These are the changes you must make before you attempt to build RedBoot. You are advised to read all the sources though.

1. Copy an existing platform HAL from the same or another architecture. Rename the files as necessary to follow the standard: CDL and MLT related files should contain the <arch>_<variant>_<platform> triplet.

2. Adjust CDL options. Primarily option naming, real-time clock/counter, and CYGHWR_MEMORY_LAYOUT variables, but also other options may need editing. Look through the architecture/variant CDL files to see if there are any requirements/features which where not used on the platform you copied. If so, add appropriate ones. See the section called “HAL Platform CDL” for more details.

3. Add the necessary packages and target descriptions to the top-level `ecos.db` file. See the section called “eCos Database”. Initially, the target entry should only contain the HAL packages. Other hardware support packages will be added later.

4. Adjust the MLT files in `include/pkgconf` to match the memory layout on the platform. For initial testing it should be enough to just hand edit .h and .ldi files, but eventually you should generate all files using the memory layout editor in the configuration tool. See the section called “Platform Memory Layout” for more details.

5. Edit the `misc/redboot_<STARTUP>.ecm` for the startup type you have chosen to begin with. Rename any platform specific options and remove any that do not apply. In the `cdl_configuration` section, comment out any extra packages that are added, particularly packages such as `CYGPKG_IO_FLASH` and `CYGPKG_IO_ETH_DRIVERS`. These are not needed for initial porting and will be added back later.

6. If the default IO macros are not correct, override them in plf_io.h. This may be necessary if the platform uses a different endianness from the default for the CPU.

7. Leave out/comment out code that enables caches and/or MMU if possible. Execution speed will not be a concern until the port is feature complete.

8. Implement a simple serial driver (polled mode only). Make sure the initialization function properly hooks the procedures up in the virtual vector IO channel tables. RedBoot will call the serial driver via these tables.

   By copying an existing platform HAL most of this code will be already done, and will only need the platform specific hardware access code to be written.

9. Adjust/implement necessary platform initialization. This can be found in `platform.inc` and `platform.S` files (ARM: `hal_platform_setup.h` and `<platform>_misc.c`, PowerPC: `<platform>.S`). This step can be postponed if you are doing a RAM startup RedBoot first and the existing ROM monitor handles board initialization.

10. Define `HAL_STUB_PLATFORM_RESET` (optionally empty) and `HAL_STUB_PLATFORM_RESET_ENTRY` so that RedBoot can reset-on-detach - this is very handy, often removing the need for physically resetting the board between downloads.

You should now be able to build RedBoot. For ROM startup:

```
% ecosconfig new <target_name> redboot
% ecosconfig import $(ECOS_REPOSITORY)/hal/<architecture>/<platform>/<version>/misc/redboot_ROM.ecm
% ecosconfig tree
% make
```

You may have to make further changes than suggested above to get the make command to succeed. But when it does, you should find a RedBoot image in install/bin. To program this image into flash or EPROM, you may need to convert to some other file type, and possibly adjust the start address. When you have the correct objcopy command to do this, add it to the `CYGBLD_BUILD_GDB_STUBS` custom build rule in the platform CDL file.

Having updated the flash/EPROM on the board, you should see output on the serial port looking like this when powering on the board:

```
RedBoot(tm) bootstrap and debug environment [ROMRAM]
Non-certified release, version UNKNOWN - built 15:42:24, Mar 14 2002

Platform: <PLATFORM> (<ARCHITECTURE> <VARIANT>)
Copyright (C) 2000, 2001, 2002, Free Software Foundation, Inc.

RAM: 0x00000000-0x01000000, 0x000293e8-0x00ed1000 available
FLASH: 0x24000000 - 0x26000000, 256 blocks of 0x00020000 bytes each.
RedBoot>
```

If you do not see this output, you need to go through all your changes and figure out what's wrong. If there's a user programmable LED or LCD on the board it may help you figure out how far RedBoot gets before it hangs. Unfortunately there's no good way to describe what to do in this situation - other than that you have to play with the code and the board.

Adding features

Now you should have a basic RedBoot running on the board. This means you have a the correct board initialization and a working serial driver. It's time to flesh out the remaining HAL features.

1. Reset. As mentioned above it is desirable to get the board to reset when GDB disconnects. When GDB disconnects it sends RedBoot a kill-packet, and RedBoot first calls `HAL_STUB_PLATFORM_RESET()`, attempting to perform a software-invoked reset. Most embedded CPUs/boards have a watchdog which is capable of triggering a reset. If your target does not have a watchdog, leave `HAL_STUB_PLATFORM_RESET()` empty and rely on the fallback approach.

   If `HAL_STUB_PLATFORM_RESET()` did not cause a reset, RedBoot will jump to `HAL_STUB_PLATFORM_RESET_ENTRY` - this should be the address where the CPU will start execution after a reset. Re-initializing the board and drivers will *usually* be good enough to make a hardware reset unnecessary.

After the reset caused by the kill-packet, the target will be ready for GDB to connect again. During a days work, this will save you from pressing the reset button many times.

Note that it is possible to disconnect from the board without causing it to reset by using the GDB command "detach".

2. Single-stepping is necessary for both instruction-level debugging and for breakpoint support. Single-stepping support should already be in place as part of the architecture/variant HAL, but you want to give it a quick test since you will come to rely on it.
3. Real-time clock interrupts drive the eCos scheduler clock. Many embedded CPUs have an on-core timer (e.g. SH) or decrementer (e.g. MIPS, PPC) that can be used, and in this case it will already be supported by the architecture/variant HAL. You only have to calculate and enter the proper `CYGNUM_HAL_RTC_CONSTANTS` definitions in the platform CDL file.

   On some targets it may be necessary to use a platform-specific timer source for driving the real-time clock. In this case you also have to enter the proper CDL definitions, but must also define suitable versions of the `HAL_CLOCK_XXXX` macros.
4. Interrupt decoding usually differs between platforms because the number and type of devices on the board differ. In `plf_intr.h` (ARM: `hal_platform_ints.h`) you must either extend or replace the default vector definitions provided by the architecture or variant interrupt headers. You may also have to define `HAL_INTERRUPT_XXXX` control macros.
5. Caching may also differ from architecture/variant definitions. This maybe just the cache sizes, but there can also be bigger differences for example if the platform supports 2nd level caches.

   When cache definitions are in place, enable the caches on startup. First verify that the system is stable for RAM startups, then build a new RedBoot and install it. This will test if caching, and in particular the cache sync/flush operations, also work for ROM startup.
6. Asynchronous breakpoints allow you to stop application execution and enter the debugger. Asynchronous breakpoint details are described in .

You should now have a completed platform HAL port. Verify its stability and completeness by running all the eCos tests and fix any problems that show up (you have a working RedBoot now, remember! That means you can debug the code to see why it fails).

Given the many configuration options in eCos, there may be hidden bugs or missing features that do not show up even if you run all the tests successfully with a default configuration. A comprehensive test of the entire system will take many configuration permutations and many many thousands of tests executed.

Hints

- JTAG or similar CPU debugging hardware can greatly reduce the time it takes to write a HAL port since you always have full visibility of what the CPU is doing.
- LEDs can be your friends if you don't have a JTAG device. Especially in the start of the porting effort if you don't already have a working ROM monitor on the target. Then you have to get a basic RedBoot working while basically being blindfolded. The LED can make it little easier, as you'll be able to do limited tracking of program flow and behavior by switching the LED on and off. If the board has multiple LEDs you can show a number (using binary notation with the LEDs) and sprinkle code which sets different numbers throughout the code.
- Debugging the interrupt processing is possible if you are careful with the way you program the very early interrupt entry handling. Write it so that as soon as possible in the interrupt path, taking a trap (exception) does not harm execution. See the SH vectors.S code for an example. Look for `cyg_hal_default_interrupt_vsr` and the label `cyg_hal_default_interrupt_vsr_bp_safe`, which marks the point after which traps/single-stepping is safe.

  Being able to display memory content, CPU registers, interrupt controller details at the time of an interrupt can save a lot of time.
- Using assertions is a good idea. They can sometimes reveal subtle bugs or missing features long before you would otherwise have found them, let alone notice them.

The default eCos configuration does not use assertions, so you have to enable them by switching on the option `CYGPKG_INFRA_DEBUG` in the infra package.

- The idle loop can be used to help debug the system.

  Triggering clock from the idle loop is a neat trick for examining system behavior either before interrupts are fully working, or to speed up "the clock".

  Use the idle loop to monitor and/or print out variables or hardware registers.

- hal_mk_defs is used in some of the HALs (ARM, SH) as a way to generate assembler symbol definitions from C header files without imposing an assembler/C syntax separation in the C++ header files.

HAL Platform CDL

The platform CDL both contains details necessary for the building of eCos, and platform-specific configuration options. For this reason the options differ between platforms, and the below is just a brief description of the most common options.

See Components Writers Guide for more details on CDL. Also have a quick look around in existing platform CDL files to get an idea of what is possible and how various configuration issues can be represented with CDL.

eCos Database

The eCos configuration system is made aware of a package by adding a package description in `ecos.db`. As an example we use the `TX39/JMR3904` platform:

```
package CYGPKG_HAL_MIPS_TX39_JMR3904 {
    alias     { "Toshiba JMR-TX3904 board" hal_tx39_jmr3904 tx39_jmr3904_hal }
    directory hal/mips/jmr3904
    script    hal_mips_tx39_jmr3904.cdl
    hardware
    description "
        The JMR3904 HAL package should be used when targeting the
        actual hardware. The same package can also be used when
        running on the full simulator, since this provides an
        accurate simulation of the hardware including I/O devices.
        To use the simulator in this mode the command
        `target sim --board=jmr3904' should be used from inside gdb."
}
```

This contains the title and description presented in the Configuration Tool when the package is selected. It also specifies where in the tree the package files can be found (`directory`) and the name of the CDL file which contains the package details (`script`).

To be able to build and test a configuration for the new target, there also needs to be a `target` entry in the `ecos.db` file.

```
target jmr3904 {
    alias       { "Toshiba JMR-TX3904 board" jmr tx39 }
    packages    { CYGPKG_HAL_MIPS
                  CYGPKG_HAL_MIPS_TX39
                  CYGPKG_HAL_MIPS_TX39_JMR3904
    }
    description "
        The jmr3904 target provides the packages needed to run
        eCos on a Toshiba JMR-TX3904 board. This target can also
        be used when running in the full simulator, since the simulator provides an
        accurate simulation of the hardware including I/O devices.
        To use the simulator in this mode the command
        `target sim --board=jmr3904' should be used from inside gdb."
}
```

The important part here is the packages section which defines the various hardware specific packages that contribute to support for this target. In this case the MIPS architecture package, the TX39 variant package, and the JMR-TX3904 platform packages are selected. Other packages, for serial drivers, ethernet drivers and FLASH memory drivers may also appear here.

CDL File Layout

All the platform options are contained in a CDL package named CYGPKG_HAL_<architecture>_<variant>_<platform>. They all share more or less the same cdl_package details:

```
cdl_package CYGPKG_HAL_MIPS_TX39_JMR3904 {
    display       "JMR3904 evaluation board"
    parent        CYGPKG_HAL_MIPS
    requires      CYGPKG_HAL_MIPS_TX39
    define_header hal_mips_tx39_jmr3904.h
    include_dir   cyg/hal
    description   "
        The JMR3904 HAL package should be used when targeting the
        actual hardware. The same package can also be used when
        running on the full simulator, since this provides an
        accurate simulation of the hardware including I/O devices.
        To use the simulator in this mode the command
        `target sim --board=jmr3904' should be used from inside gdb."

    compile       platform.S plf_misc.c plf_stub.c

    define_proc {
        puts $::cdl_system_header "#define CYGBLD_HAL_TARGET_H   <pkgconf/hal_mips_tx39.h>"
        puts $::cdl_system_header "#define CYGBLD_HAL_PLATFORM_H <pkgconf/hal_mips_tx39_jmr3904.h>"
    }

    …
}
```

This specifies that the platform package should be parented under the MIPS packages, requires the TX39 variant HAL and all configuration settings should be saved in cyg/hal/hal_mips_tx39_jmt3904.h.

The compile line specifies which files should be built when this package is enabled, and the define_proc defines some macros that are used to access the variant or architecture (the _TARGET_ name is a bit of a misnomer) and platform configuration options.

Startup Type

eCos uses an option to select between a set of valid startup configurations. These are normally RAM, ROM and possibly ROMRAM. This setting is used to select which linker map to use (i.e., where to link eCos and the application in the memory space), and how the startup code should behave.

```
cdl_component CYG_HAL_STARTUP {
    display       "Startup type"
    flavor        data
    legal_values  {"RAM" "ROM"}
    default_value {"RAM"}
    no_define
    define -file system.h CYG_HAL_STARTUP
    description   "
        When targeting the JMR3904 board it is possible to build
        the system for either RAM bootstrap, ROM bootstrap, or STUB
        bootstrap. RAM bootstrap generally requires that the board
        is equipped with ROMs containing a suitable ROM monitor or
        equivalent software that allows GDB to download the eCos
        application on to the board. The ROM bootstrap typically
        requires that the eCos application be blown into EPROMs or
        equivalent technology."
}
```

The no_define and define pair is used to make the setting of this option appear in the file system.h instead of the default specified in the header.

Build options

A set of options under the components CYGBLD_GLOBAL_OPTIONS and CYGHWR_MEMORY_LAYOUT specify how eCos should be built: what tools and compiler options should be used, and which linker fragments should be used.

```
cdl_component CYGBLD_GLOBAL_OPTIONS {
    display "Global build options"
    flavor  none
    parent  CYGPKG_NONE
    description   "
        Global build options including control over
        compiler flags, linker flags and choice of toolchain."


    cdl_option CYGBLD_GLOBAL_COMMAND_PREFIX {
        display "Global command prefix"
        flavor  data
        no_define
        default_value { "mips-tx39-elf" }
        description "
            This option specifies the command prefix used when
            invoking the build tools."
    }

    cdl_option CYGBLD_GLOBAL_CFLAGS {
        display "Global compiler flags"
        flavor  data
        no_define
        default_value { "-Wall -Wpointer-arith -Wstrict-prototypes -Winline -Wundef -Woverloaded-virtual " .
            "-g -O2 -ffunction-sections -fdata-sections -fno-rtti -fno-exceptions" }
        description   "
            This option controls the global compiler flags which
            are used to compile all packages by
            default. Individual packages may define
            options which override these global flags."
    }

    cdl_option CYGBLD_GLOBAL_LDFLAGS {
        display "Global linker flags"
        flavor  data
        no_define
        default_value { "-g -nostdlib -Wl,--gc-sections -Wl,-static" }
        description   "
            This option controls the global linker flags. Individual
            packages may define options which override these global flags."
        }
    }

    cdl_component CYGHWR_MEMORY_LAYOUT {
        display "Memory layout"
        flavor data
        no_define
        calculated { CYG_HAL_STARTUP == "RAM" ? "mips_tx39_jmr3904_ram" : \
                                                "mips_tx39_jmr3904_rom" }

        cdl_option CYGHWR_MEMORY_LAYOUT_LDI {
        display "Memory layout linker script fragment"
        flavor data
        no_define
        define -file system.h CYGHWR_MEMORY_LAYOUT_LDI
        calculated { CYG_HAL_STARTUP == "RAM" ? "<pkgconf/mlt_mips_tx39_jmr3904_ram.ldi>" : \
                                                "<pkgconf/mlt_mips_tx39_jmr3904_rom.ldi>" }
    }
    cdl_option CYGHWR_MEMORY_LAYOUT_H {
        display "Memory layout header file"
        flavor data
        no_define
        define -file system.h CYGHWR_MEMORY_LAYOUT_H
        calculated { CYG_HAL_STARTUP == "RAM" ? "<pkgconf/mlt_mips_tx39_jmr3904_ram.h>" : \
                                                "<pkgconf/mlt_mips_tx39_jmr3904_rom.h>" }
    }
}
```

Common Target Options

All platforms also specify real-time clock details:

```
# Real-time clock/counter specifics
cdl_component CYGNUM_HAL_RTC_CONSTANTS {
    display       "Real-time clock constants."
    flavor        none

    cdl_option CYGNUM_HAL_RTC_NUMERATOR {
        display       "Real-time clock numerator"
        flavor        data
        calculated    1000000000
    }
    cdl_option CYGNUM_HAL_RTC_DENOMINATOR {
        display       "Real-time clock denominator"
        flavor        data
        calculated    100
    }
    # Isn't a nice way to handle freq requirement!
    cdl_option CYGNUM_HAL_RTC_PERIOD {
        display       "Real-time clock period"
        flavor        data
        legal_values  { 15360 20736 }
        calculated     { CYGHWR_HAL_MIPS_CPU_FREQ == 50 ? 15360 : \
        CYGHWR_HAL_MIPS_CPU_FREQ == 66 ? 20736 : 0 }
    }
}
```

The `NUMERATOR` divided by the `DENOMINATOR` gives the number of nanoseconds per tick. The `PERIOD` is the divider to be programmed into a hardware timer that is driven from an appropriate hardware clock, such that the timer overflows once per tick (normally generating a CPU interrupt to mark the end of a tick). The tick default rate is typically 100Hz.

Platforms that make use of the virtual vector ROM calling interface (see the section called “Virtual Vectors (eCos/ROM Monitor Calling Interface)”) will also specify details necessary to define configuration channels (these options are from the SH/EDK7707 HAL) :

```
cdl_option CYGNUM_HAL_VIRTUAL_VECTOR_COMM_CHANNELS {
    display      "Number of communication channels on the board"
    flavor       data
    calculated   1
}

cdl_option CYGNUM_HAL_VIRTUAL_VECTOR_DEBUG_CHANNEL {
    display          "Debug serial port"
    flavor data
    legal_values     0 to CYGNUM_HAL_VIRTUAL_VECTOR_COMM_CHANNELS-1
    default_value    0
    description      "
        The EDK/7708 board has only one serial port. This option
        chooses which port will be used to connect to a host
        running GDB."
}
```

```
cdl_option CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL {
    display          "Diagnostic serial port"
    flavor data
    legal_values     0 to CYGNUM_HAL_VIRTUAL_VECTOR_COMM_CHANNELS-1
    default_value    0
    description      "
        The EDK/7708 board has only one serial port.  This option
        chooses which port will be used for diagnostic output."
}
```

The platform usually also specify an option controlling the ability to co-exist with a ROM monitor:

```
cdl_option CYGSEM_HAL_USE_ROM_MONITOR {
    display       "Work with a ROM monitor"
    flavor        booldata
    legal_values  { "Generic" "CygMon" "GDB_stubs" }
    default_value { CYG_HAL_STARTUP == "RAM" ? "CygMon" : 0 }
    parent        CYGPKG_HAL_ROM_MONITOR
    requires      { CYG_HAL_STARTUP == "RAM" }
    description   "
        Support can be enabled for three different varieties of ROM monitor.
        This support changes various eCos semantics such as the encoding
        of diagnostic output, or the overriding of hardware interrupt
        vectors.
        Firstly there is \"Generic\" support which prevents the HAL
        from overriding the hardware vectors that it does not use, to
        instead allow an installed ROM monitor to handle them. This is
        the most basic support which is likely to be common to most
        implementations of ROM monitor.
        \"CygMon\" provides support for the Cygnus ROM Monitor.
        And finally, \"GDB_stubs\" provides support when GDB stubs are
        included in the ROM monitor or boot ROM."
}
```

Or the ability to be configured as a ROM monitor:

```
cdl_option CYGSEM_HAL_ROM_MONITOR {
    display       "Behave as a ROM monitor"
    flavor        bool
    default_value 0
    parent        CYGPKG_HAL_ROM_MONITOR
    requires      { CYG_HAL_STARTUP == "ROM" }
    description   "
        Enable this option if this program is to be used as a ROM monitor,
        i.e. applications will be loaded into RAM on the board, and this
        ROM monitor may process exceptions or interrupts generated from the
        application. This enables features such as utilizing a separate
        interrupt stack when exceptions are generated."
}
```

The latter option is accompanied by a special build rule that extends the generic ROM monitor build rule in the common HAL:

```
cdl_option CYGBLD_BUILD_GDB_STUBS {
    display "Build GDB stub ROM image"
    default_value 0
    requires { CYG_HAL_STARTUP == "ROM" }
    requires CYGSEM_HAL_ROM_MONITOR
    requires CYGBLD_BUILD_COMMON_GDB_STUBS
    requires CYGDBG_HAL_DEBUG_GDB_INCLUDE_STUBS
    requires ! CYGDBG_HAL_DEBUG_GDB_BREAK_SUPPORT
    requires ! CYGDBG_HAL_DEBUG_GDB_THREAD_SUPPORT
    requires ! CYGDBG_HAL_COMMON_INTERRUPTS_SAVE_MINIMUM_CONTEXT
    requires ! CYGDBG_HAL_COMMON_CONTEXT_SAVE_MINIMUM
    no_define
    description "
        This option enables the building of the GDB stubs for the
        board. The common HAL controls takes care of most of the
        build process, but the final conversion from ELF image to
        binary data is handled by the platform CDL, allowing
        relocation of the data if necessary."

    make -priority 320 {
        <PREFIX>/bin/gdb_module.bin : <PREFIX>/bin/gdb_module.img
        $(OBJCOPY) -O binary $< $@
    }
}
```

Most platforms support RedBoot, and some options are needed to configure for RedBoot.

```
    cdl_component CYGPKG_REDBOOT_HAL_OPTIONS {
    display       "Redboot HAL options"
    flavor        none
    no_define
    parent        CYGPKG_REDBOOT
    active_if     CYGPKG_REDBOOT
    description   "
        This option lists the target's requirements for a valid Redboot
        configuration."

    cdl_option CYGBLD_BUILD_REDBOOT_BIN {
        display       "Build Redboot ROM binary image"
        active_if     CYGBLD_BUILD_REDBOOT
        default_value 1
        no_define
        description "This option enables the conversion of the Redboot ELF
            image to a binary image suitable for ROM programming."

        make -priority 325 {
            <PREFIX>/bin/redboot.bin : <PREFIX>/bin/redboot.elf
            $(OBJCOPY) --strip-debug $< $(@:.bin=.img)
            $(OBJCOPY) -O srec $< $(@:.bin=.srec)
            $(OBJCOPY) -O binary $< $@
        }
    }
}
```

The important part here is the `make` command in the `CYGBLD_BUILD_REDBOOT_BIN` option which emits makefile commands to translate the `.elf` file generated by the link phase into both a binary file and an S-Record file. If a different format is required by a PROM programmer or ROM monitor, then different output formats would need to be generated here.

Platform Memory Layout

The platform memory layout is defined using the Memory Configuration Window in the Configuration Tool.

Note

If you do not have access to a Windows machine, you can hand edit the `.h` and `.ldi` files to match the properties of your platform. If you want to contribute your port back to the eCos community, ask someone on the list to make proper memory map files for you.

Layout Files

The memory configuration details are saved in three files:

`.mlt` This is the Configuration Tool save-file. It is only used by the Configuration Tool.

`.ldi` This is the linker script fragment. It defines the memory and location of sections by way of macros defined in the architecture or variant linker script.

`.h` This file describes some of the memory region details as C macros, allowing eCos or the application adapt the memory layout of a specific configuration.

These three files are generated for each startup-type, since the memory details usually differ.

Reserved Regions

Some areas of the memory space are reserved for specific purposes, making room for exception vectors and various tables. RAM startup configurations also need to reserve some space at the bottom of the memory map for the ROM monitor.

These reserved areas are named with the prefix "reserved_" which is handled specially by the Configuration Tool: instead of referring to a linker macro, the start of the area is labeled and a gap left in the memory map.

Platform Serial Device Support

The first step is to set up the CDL definitions. The configuration options that need to be set are the following:

`CYGNUM_HAL_VIRTUAL_VECTOR_COMM_CHANNELS`

The number of channels, usually 0, 1 or 2.

`CYGNUM_HAL_VIRTUAL_VECTOR_DEBUG_CHANNEL`

The channel to use for GDB.

`CYGNUM_HAL_VIRTUAL_VECTOR_DEBUG_CHANNEL_BAUD`

Initial baud rate for debug channel.

`CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL`

The channel to use for the console.

`CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL_BAUD`

The initial baud rate for the console channel.

`CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL_DEFAULT`

The default console channel.

The code in `hal_diag.c` need to be converted to support the new serial device. If this the same as a device already supported, copy that.

The following functions and types need to be rewritten to support a new serial device.

`struct channel_data_t;`

Structure containing base address, timeout and ISR vector number for each serial device supported. Extra fields my be added if necessary for the device. For example some devices have write-only control registers, so keeping a shadow of the last value written here can be useful.

`xxxx_ser_channels[];`

Array of `channel_data_t`, initialized with parameters of each channel. The index into this array is the channel number used in the CDL options above and is used by the virtual vector mechanism to refer to each channel.

`void cyg_hal_plf_serial_init_channel(void *__ch_data)`

Initialize the serial device. The parameter is actually a pointer to a `channel_data_t` and should be cast back to this type before use. This function should use the CDL definition for the baud rate for the channel it is initializing.

`void cyg_hal_plf_serial_putc(void * __ch_data, char *c)`

Send a character to the serial device. This function should poll for the device being ready to send and then write the character. Since this is intended to be a diagnostic/debug channel, it is often also a good idea to poll for end of transmission too. This ensures that as much data gets out of the system as possible.

`bool cyg_hal_plf_serial_getc_nonblock(void* __ch_data, cyg_uint8* ch)`

This function tests the device and if a character is available, places it in *`*ch`* and returns `TRUE`. If no character is available, then the function returns `FALSE` immediately.

`int cyg_hal_plf_serial_control(void *__ch_data, __comm_control_cmd_t __func, ...)`

This is an IOCTL-like function for controlling various aspects of the serial device. The only part in which you may need to do some work initially is in the `__COMMCTL_IRQ_ENABLE` and `__COMMCTL_IRQ_DISABLE` cases to enable/disable interrupts.

`int cyg_hal_plf_serial_isr(void *__ch_data, int* __ctrlc, CYG_ADDRWORD __vector, CYG_ADDRWORD __data)`

This interrupt handler, called from the spurious interrupt vector, is specifically for dealing with `Ctrl-C` interrupts from GDB. When called this function should do the following:

1. Check for an incoming character. The code here is very similar to that in `cyg_hal_plf_serial_getc_nonblock()`.
2. Read the character and call `cyg_hal_is_break()`.
3. If result is true, set *`*__ctrlc`* to `1`.
4. Return `CYG_ISR_HANDLED`.

`void cyg_hal_plf_serial_init()`

Initialize each of the serial channels. First call `cyg_hal_plf_serial_init_channel()` for each channel. Then call the `CYGACC_COMM_IF_*` macros for each channel. This latter set of calls are identical for all channels, so the best way to do this is to copy and edit an existing example.

Variant HAL Porting

A variant port can be a fairly limited job, but can also require quite a lot of work. A variant HAL describes how a specific CPU variant differs from the generic CPU architecture. The variant HAL can re-define cache, MMU, interrupt, and other features which override the default implementation provided by the architecture HAL.

Doing a variant port requires a preexisting architecture HAL port. It is also likely that a platform port will have to be done at the same time if it is to be tested.

HAL Variant Porting Process

The easiest way to make a new variant HAL is simply to copy an existing variant HAL and change all the files to match the new variant. If this is the first variant for an architecture, it may be hard to decide which parts should be put in the variant - knowledge of other variants of the architecture is required.

Looking at existing variant HALs (e.g., MIPS tx39, tx49) may be a help - usually things such as caching, interrupt and exception handling differ between variants. Initialization code, and code for handling various core components (FPU, DSP, MMU, etc.) may also differ or be missing altogether on some variants. Linker scripts may also require specific variant versions.

Note

Some CPU variants may require specific compiler support. That support must be in place before you can undertake the eCos variant port.

HAL Variant CDL

The CDL in a variant HAL tends to depend on the exact functionality supported by the variant. If it implements some of the devices described in the platform HAL, then the CDL for those will be here rather than there (for example the real-time clock).

There may also be CDL to select options in the architecture HAL to configure it to a particular architectural variant.

Each variant needs an entry in the `ecos.db` file. This is the one for the SH3:

```
package CYGPKG_HAL_SH_SH3 {
    alias         { "SH3 architecture" hal_sh_sh3 }
    directory     hal/sh/sh3
    script        hal_sh_sh3.cdl
    hardware
    description   "
    The SH3 (SuperH 3) variant HAL package provides generic
    support for SH3 variant CPUs."
}
```

As you can see, it is very similar to the platform entry.

The variant CDL file will contain a package entry named for the architecture and variant, matching the package name in the `ecos.db` file. Here is the initial part of the MIPS VR4300 CDL file:

```
cdl_package CYGPKG_HAL_MIPS_VR4300 {
    display       "VR4300 variant"
    parent        CYGPKG_HAL_MIPS
    implements    CYGINT_HAL_MIPS_VARIANT
    hardware
    include_dir   cyg/hal
    define_header hal_mips_vr4300.h
    description   "
        The VR4300 variant HAL package provides generic support
        for this processor architecture. It is also necessary to
        select a specific target platform HAL package."
}
```

This defines the package, placing it under the MIPS architecture package in the hierarchy. The `implements` line indicates that this is a MIPS variant. The architecture package uses this to check that exactly one variant is configured in.

The variant defines some options that cause the architecture HAL to configure itself to support this variant.

```
    cdl_option CYGHWR_HAL_MIPS_64BIT {
    display    "Variant 64 bit architecture support"
    calculated 1
}

cdl_option CYGHWR_HAL_MIPS_FPU {
    display    "Variant FPU support"
    calculated 1
}

cdl_option CYGHWR_HAL_MIPS_FPU_64BIT {
    display    "Variant 64 bit FPU support"
    calculated 1
}
```

These tell the architecture that this is a 64 bit MIPS architecture, that it has a floating point unit, and that we are going to use it in 64 bit mode rather than 32 bit mode.

The CDL file finishes off with some build options.

```
    define_proc {
        puts $::cdl_header "#include <pkgconf/hal_mips.h>"
    }

    compile       var_misc.c

    make {
        <PREFIX>/lib/target.ld: <PACKAGE>/src/mips_vr4300.ld
        $(CC) -E -P -Wp,-MD,target.tmp -DEXTRAS=1 -xc $(INCLUDE_PATH) $(CFLAGS) -o $@ $<
        @echo $@ ": \\" > $(notdir $@).deps
        @tail +2 target.tmp >> $(notdir $@).deps
        @echo >> $(notdir $@).deps
        @rm target.tmp
    }

    cdl_option CYGBLD_LINKER_SCRIPT {
        display "Linker script"
        flavor data
        no_define
        calculated  { "src/mips_vr4300.ld" }
    }

}
```

The `define_proc` causes the architecture configuration file to be included into the configuration file for the variant. The `compile` causes the single source file for this variant, `var_misc.c` to be compiled. The `make` command emits makefile rules to combine the linker script with the `.ldi` file to generate `target.ld`. Finally, in the MIPS HALs, the main linker script is defined in the variant, rather than the architecture, so `CYGBLD_LINKER_SCRIPT` is defined here.

Cache Support

The main area where the variant is likely to be involved is in cache support. Often the only thing that distinguishes one CPU variant from another is the size of its caches.

In architectures such as the MIPS and PowerPC where cache instructions are part of the ISA, most of the actual cache operations are implemented in the architecture HAL. In this case the variant HAL only needs to define the cache dimensions. The following are the cache dimensions defined in the MIPS VR4300 variant `var_cache.h`.

```
// Data cache
#define HAL_DCACHE_SIZE                 (8*1024)        // Size of data cache in bytes
#define HAL_DCACHE_LINE_SIZE            16              // Size of a data cache line
#define HAL_DCACHE_WAYS                 1               // Associativity of the cache

// Instruction cache
#define HAL_ICACHE_SIZE                 (16*1024)       // Size of cache in bytes
#define HAL_ICACHE_LINE_SIZE            32              // Size of a cache line
#define HAL_ICACHE_WAYS                 1               // Associativity of the cache

#define HAL_DCACHE_SETS (HAL_DCACHE_SIZE/(HAL_DCACHE_LINE_SIZE*HAL_DCACHE_WAYS))
#define HAL_ICACHE_SETS (HAL_ICACHE_SIZE/(HAL_ICACHE_LINE_SIZE*HAL_ICACHE_WAYS))
```

Additional cache macros, or overrides for the defaults, may also appear in here. While some architectures have instructions for managing cache lines, overall enable/disable operations may be handled via variant specific registers. If so then `var_cache.h` should also define the `HAL_XCACHE_ENABLE()` and `HAL_XCACHE_DISABLE()` macros.

If there are any generic features that the variant does not support (cache locking is a typical example) then `var_cache.h` may need to disable definitions of certain operations. It is architecture dependent exactly how this is done.

Architecture HAL Porting

A new architecture HAL is the most complex HAL to write, and it the least easily described. Hence this section is presently nothing more than a place holder for the future.

HAL Architecture Porting Process

The easiest way to make a new architecture HAL is simply to copy an existing architecture HAL of an, if possible, closely matching architecture and change all the files to match the new architecture. The MIPS architecture HAL should be used if possible, as it has the appropriate layout and coding conventions. Other HALs may deviate from that norm in various ways.

Note

eCos is written for GCC. It requires C and C++ compiler support as well as a few compiler features introduced during eCos development - so compilers older than eCos may not provide these features. Note that there is no C++ support for any 8 or 16 bit CPUs. Before you can undertake an eCos port, you need the required compiler support.

The following gives a rough outline of the steps needed to create a new architecture HAL. The exact order and set of steps needed will vary greatly from architecture to architecture, so a lot of flexibility is required. And of course, if the architecture HAL is to be tested, it is necessary to do variant and platform ports for the initial target simultaneously.

1. Make a new directory for the new architecture under the `hal` directory in the source repository. Make an `arch` directory under this and populate this with the standard set of package directories.

2. Copy the CDL file from an example HAL changing its name to match the new HAL. Edit the file, changing option names as appropriate. Delete any options that are specific to the original HAL, and and any new options that are necessary for the new architecture. This is likely to be a continuing process during the development of the HAL. See the section called “CDL Requirements” for more details.

3. Copy the `hal_arch.h` file from an example HAL. Within this file you need to change or define the following:

   - Define the HAL_SavedRegisters structure. This may need to reflect the save order of any group register save/restore instructions, the interrupt and exception save and restore formats, and the procedure calling conventions. It may also need to cater for optional FPUs and other functional units. It can be quite difficult to develop a layout that copes with all requirements.

   - Define the bit manipulation routines, `HAL_LSBIT_INDEX()` and `HAL_MSBIT_INDEX()`. If the architecture contains instructions to perform these, or related, operations, then these should be defined as inline assembler fragments. Otherwise make them calls to functions.

   - Define `HAL_THREAD_INIT_CONTEXT()`. This initializes a restorable CPU context onto a stack pointer so that a later call to `HAL_THREAD_LOAD_CONTEXT()` or `HAL_THREAD_SWITCH_CONTEXT()` will execute it correctly. This macro needs to take account of the same optional features of the architecture as the definition of HAL_SavedRegisters.

   - Define `HAL_THREAD_LOAD_CONTEXT()` and `HAL_THREAD_SWITCH_CONTEXT()`. These should just be calls to functions in `context.S`.

   - Define `HAL_REORDER_BARRIER()`. This prevents code being moved by the compiler and is necessary in some order-sensitive code. This macro is actually defined identically in all architecture, so it can just be copied.

   - Define breakpoint support. The macro `HAL_BREAKPOINT(label)` needs to be an inline assembly fragment that invokes a breakpoint. The breakpoint instruction should be labeled with the *`label`* argument. `HAL_BREAKINST` and `HAL_BREAKINST_SIZE` define the breakpoint instruction for debugging purposes.

- Optionally provide a macro HAL_HWDEBUG_BREAKPOINT. This is used by the common HAL's gdb file I/O support to get the attention of gdb when using hardware debug technology such as jtag or BDM. The macro may involve a dedicated breakpoint instruction or a processor exception or trap of some sort. Only one instance of this macro will ever be invoked. It should define either one or two labels. _gdb_hwdebug_break should correspond to the address that will be reported to gdb. If that address is the same as the breakpoint instruction or trap, or if the instruction has side effects like pushing exception data onto the stack, then the macro should also define a label _gdb_hwdebug_continue. When the application is resumed gdb will transfer control to that label if defined, allowing any necessary clean-up operations to be performed.

- Define GDB support. GDB views the registers of the target as a linear array, with each register having a well defined offset. This array may differ from the ordering defined in HAL_SavedRegisters. The macros HAL_GET_GDB_REGISTERS() and HAL_SET_GDB_REGISTERS() translate between the GDB array and the HAL_SavedRegisters structure. The HAL_THREAD_GET_SAVED_REGISTERS() translates a stack pointer saved by the context switch macros into a pointer to a HAL_SavedRegisters structure. Usually this is a one-to-one translation, but this macro allows it to differ if necessary.

- Define long jump support. The type hal_jmp_buf and the functions hal_setjmp() and hal_longjmp() provide the underlying implementation of the C library setjmp() and longjmp().

- Define idle thread action. Generally the macro HAL_IDLE_THREAD_ACTION() is defined to call a function in hal_misc.c.

- Define stack sizes. The macros CYGNUM_HAL_STACK_SIZE_MINIMUM and CYGNUM_HAL_STACK_SIZE_TYPICAL should be defined to the minimum size for any thread stack and a reasonable default for most threads respectively. It is usually best to construct these out of component sizes for the CPU save state and procedure call stack usage. These definitions should not use anything other than numerical values since they can be used from assembly code in some HALs.

- Define memory access macros. These macros provide translation between cached and uncached and physical memory spaces. They usually consist of masking out bits of the supplied address and ORing in alternative address bits.

- Define global pointer save/restore macros. These really only need defining if the calling conventions of the architecture require a global pointer (as does the MIPS architecture), they may be empty otherwise. If it is necessary to define these, then take a look at the MIPS implementation for an example.

4. Copy hal_intr.h from an example HAL. Within this file you should change or define the following:

   - Define the exception vectors. These should be detailed in the architecture specification. Essentially for each exception entry point defined by the architecture there should be an entry in the VSR table. The offsets of these VSR table entries should be defined here by CYGNUM_HAL_VECTOR_* definitions. The size of the VSR table also needs to be defined here.

   - Map any hardware exceptions to standard names. There is a group of exception vector name of the form CYGNUM_HAL_EXCEPTION_* that define a wide variety of possible exceptions that many architectures raise. Generic code detects whether the architecture can raise a given exception by testing whether a given CYGNUM_HAL_EXCEPTION_* definition is present. If it is present then its value is the vector that raises that exception. This does not need to be a one-to-one correspondence, and several CYGNUM_HAL_EXCEPTION_* definitions may have the same value.

     Interrupt vectors are usually defined in the variant or platform HALs. The interrupt number space may either be continuous with the VSR number space, where they share a vector table (as in the i386) or may be a separate space where a separate decode stage is used (as in MIPS or PowerPC).

   - Declare any static data used by the HAL to handle interrupts and exceptions. This is usually three vectors for interrupts: hal_interrupt_handlers[], hal_interrupt_data[] and hal_interrupt_objects[], which are sized according to the interrupt vector definitions. In addition a definition for the VSR table, hal_vsr_table[] should be made. These vectors are normally defined in either vectors.S or hal_misc.c.

   - Define interrupt enable/disable macros. These are normally inline assembly fragments to execute the instructions, or manipulate the CPU register, that contains the CPU interrupt enable bit.

- A feature that many HALs support is the ability to execute DSRs on the interrupt stack. This is not an essential feature, and is better left unimplemented in the initial porting effort. If this is required, then the macro `HAL_INTERRUPT_STACK_CALL_PENDING_DSRS()` should be defined to call a function in `vectors.S`.
- Define the interrupt and VSR attachment macros. If the same arrays as for other HALs have been used for VSR and interrupt vectors, then these macro can be copied across unchanged.

5. A number of other header files also need to be filled in:

   - `basetype.h`. This file defines the basic types used by eCos, together with the endianness and some other characteristics. This file only really needs to contain definitions if the architecture differs significantly from the defaults defined in `cyg_type.h`
   - `hal_io.h`. This file contains macros for accessing device IO registers. If the architecture uses memory mapped IO, then these can be copied unchanged from an existing HAL such as MIPS. If the architecture uses special IO instructions, then these macros must be defined as inline assembler fragments. See the I386 HAL for an example. PCI bus access macros are usually defined in the variant or platform HALs.

     This file may also provide further macro definitions, if relevant for the underlying hardware:

     `HAL_MEMORY_BARRIER()`

     This causes any memory writes pending within the CPU to be flushed to memory before continuing. Frequently there is a specific instruction, such as **sync** on MIPS, to cause write buffers to be flushed. This macro is generally not relevant to be called if you also have a writeback data cache, as that needs separate treatment. However this macro is relevant for systems with no data cache, a writethrough data cache, or in code running with the data cache disabled. For the latter reason this macro should be implemented if the facility exists, irrespective of the cache properties.

     `HAL_IO_BARRIER()`

     This causes any I/O writes pending within the CPU to be flushed to the I/O space before continuing. Frequently there is a specific instruction, such as **eieio** on PowerPC, to cause such pending writes to be guaranteed to be committed. On systems with no separate I/O space, such that all device access is instead memory-mapped, then this function may be defined to be the same as `HAL_MEMORY_BARRIER()`.
   - `hal_cache.h`. This file contains cache access macros. If the architecture defines cache instructions, or control registers, then the access macros should be defined here. Otherwise they must be defined in the variant or platform HAL. Usually the cache dimensions (total size, line size, ways etc.) are defined in the variant HAL.
   - `arch.inc` and `<architecture>.inc`. These files are assembler headers used by `vectors.S` and `context.S`. `<architecture>.inc` is a general purpose header that should contain things like register aliases, ABI definitions and macros useful to general assembly code. If there are no such definitions, then this file need not be provided. `arch.inc` contains macros for performing various eCos related operations such as initializing the CPU, caches, FPU etc. The definitions here may often be configured or overridden by definitions in the variant or platform HALs. See the MIPS HAL for an example of this.

6. Write `vectors.S`. This is the most important file in the HAL. It contains the CPU initialization code, exception and interrupt handlers. While other HALs should be consulted for structures and techniques, there is very little here that can be copied over without major edits.

   The main pieces of code that need to be defined here are:

   - Reset vector. This usually need to be positioned at the start of the ROM or FLASH, so should be in a linker section of its own. It can then be placed correctly by the linker script. Normally this code is little more than a jump to the label `_start`.
   - Exception vectors. These are the trampoline routines connected to the hardware exception entry points that vector through the VSR table. In many architectures these are adjacent to the reset vector, and should occupy the same linker section. If

the architecture allow the vectors to be moved then it may be necessary for these trampolines to be position independent so they can be relocated at runtime.

The trampolines should do the minimum necessary to transfer control from the hardware vector to the VSR pointed to by the matching table entry. Exactly how this is done depends on the architecture. Usually the trampoline needs to get some working registers by either saving them to CPU special registers (e.g. PowerPC SPRs), using reserved general registers (MIPS K0 and K1), using only memory based operations (IA32), or just jumping directly (ARM). The VSR table index to be used is either implicit in the entry point taken (PowerPC, IA32, ARM), or must be determined from a CPU register (MIPS).

- Write kernel startup code. This is the location the reset vector jumps to, and can be in the main text section of the executable, rather than a special section. The code here should first initialize the CPU and other hardware subsystems. The best approach is to use a set of macro calls that are defined either in `arch.inc` or overridden in the variant or platform HALs. Other jobs that this code should do are: initialize stack pointer; copy the data section from ROM to RAM if necessary; zero the BSS; call variant and platform initializers; call `cyg_hal_invoke_constructors()`; call `initialize_stub()` if necessary. Finally it should call `cyg_start()`. See the section called “HAL Startup” for details.
- Write the default exception VSR. This VSR is installed in the VSR table for all synchronous exception vectors. See the section called “Default Synchronous Exception Handling” for details of what this VSR does.
- Write the default interrupt VSR. This is installed in all VSR table entries that correspond to external interrupts. See the section called “Default Synchronous Exception Handling” for details of what this VSR does.
- Write `hal_interrupt_stack_call_pending_dsrs()`. If this function is defined in `hal_arch.h` then it should appear here. The purpose of this function is to call DSRs on the interrupt stack rather than the current thread's stack. This is not an essential feature, and may be left until later. However it interacts with the stack switching that goes on in the interrupt VSR, so it may make sense to write these pieces of code at the same time to ensure consistency.

  When this function is implemented it should do the following:

  - Take a copy of the current SP and then switch to the interrupt stack.
  - Save the old SP, together with the CPU status register (or whatever register contains the interrupt enable status) and any other registers that may be corrupted by a function call (such as any link register) to locations in the interrupt stack.
  - Enable interrupts.
  - Call `cyg_interrupt_call_pending_DSRs()`. This is a kernel functions that actually calls any pending DSRs.
  - Retrieve saved registers from the interrupt stack and switch back to the current thread stack.
  - Merge the interrupt enable state recorded in the save CPU status register with the current value of the status register to restore the previous enable state. If the status register does not contain any other persistent state then this can be a simple restore of the register. However if the register contains other state bits that might have been changed by a DSR, then care must be taken not to disturb these.
- Define any data items needed. Typically `vectors.S` may contain definitions for the VSR table, the interrupt tables and the interrupt stack. Sometimes these are only default definitions that may be overridden by the variant or platform HALs.

7. Write `context.S`. This file contains the context switch code. See the section called “Thread Context Switching” for details of how these functions operate. This file may also contain the implementation of `hal_setjmp()` and `hal_longjmp()`.
8. Write `hal_misc.c`. This file contains any C data and functions needed by the HAL. These might include:

   - `hal_interrupt_*[]`. In some HALs, if these arrays are not defined in `vectors.S` then they must be defined here.
   - `cyg_hal_exception_handler()`. This function is called from the exception VSR. It usually does extra decoding of the exception and invokes any special handlers for things like FPU traps, bus errors or memory exceptions. If there is nothing

special to be done for an exception, then it either calls into the GDB stubs, by calling `__handle_exception()`, or invokes the kernel by calling `cyg_hal_deliver_exception()`.

- `hal_arch_default_isr()`. The `hal_interrupt_handlers[]` array is usually initialized with pointers to `hal_default_isr()`, which is defined in the common HAL. This function handles things like Ctrl-C processing, but if that is not relevant, then it will call `hal_arch_default_isr()`. Normally this function should just return zero.

- `cyg_hal_invoke_constructors()`. This calls the constructors for all static objects before the program starts. eCos relies on these being called in the correct order for it to function correctly. The exact way in which constructors are handled may differ between architectures, although most use a simple table of function pointers between labels `__CTOR_LIST__` and `__CTOR_END__` which must called in order from the top down. Generally, this function can be copied directly from an existing architecture HAL.

- Bit indexing functions. If the macros `HAL_LSBIT_INDEX()` and `HAL_MSBIT_INDEX()` are defined as function calls, then the functions should appear here. The main reason for doing this is that the architecture does not have support for bit indexing and these functions must provide the functionality by conventional means. While the trivial implementation is a simple for loop, it is expensive and non-deterministic. Better, constant time, implementations can be found in several HALs (MIPS for example).

- `hal_delay_us()`. If the macro `HAL_DELAY_US()` is defined in `hal_intr.h` then it should be defined to call this function. While most of the time this function is called with very small values, occasionally (particularly in some ethernet drivers) it is called with values of several seconds. Hence the function should take care to avoid overflow in any calculations.

- `hal_idle_thread_action()`. This function is called from the idle thread via the `HAL_IDLE_THREAD_ACTION()` macro, if so defined. While normally this function does nothing, during development this is often a good place to report various important system parameters on LCDs, LED or other displays. This function can also monitor system state and report any anomalies. If the architecture supports a `halt` instruction then this is a good place to put an inline assembly fragment to execute it. It is also a good place to handle any power saving activity.

9. Create the `<architecture>.ld` file. While this file may need to be moved to the variant HAL in the future, it should initially be defined here, and only moved if necessary.

   This file defines a set of macros that are used by the platform `.ldi` files to generate linker scripts. Most GCC toolchains are very similar so the correct approach is to copy the file from an existing architecture and edit it. The main things that will need editing are the `OUTPUT_FORMAT()` directive and maybe the creation or allocation of extra sections to various macros. Running the target linker with just the `--verbose` argument will cause it to output its default linker script. This can be compared with the `.ld` file and appropriate edits made.

10. If GDB stubs are to be supported in RedBoot or eCos, then support must be included for these. The most important of these are `include/<architecture>-stub.h` and `src/<architecture>-stub.c`. In all existing architecture HALs these files, and any support files they need, have been derived from files supplied in `libgloss`, as part of the GDB toolchain package. If this is a totally new architecture, this may not have been done, and they must be created from scratch.

    `include/<architecture>-stub.h` contains definitions that are used by the GDB stubs to describe the size, type, number and names of CPU registers. This information is usually found in the GDB support files for the architecture. It also contains prototypes for the functions exported by `src/<architecture>-stub.c`; however, since this is common to all architectures, it can be copied from some other HAL.

    `src/<architecture>-stub.c` implements the functions exported by the header. Most of this is fairly straight forward: the implementation in existing HALs should show exactly what needs to be done. The only complex part is the support for single-stepping. This is used a lot by GDB, so it cannot be avoided. If the architecture has support for a trace or single-step trap then that can be used for this purpose. If it does not then this must be simulated by planting a breakpoint in the next instruction. This can be quite involved since it requires some analysis of the current instruction plus the state of the CPU to determine where execution is going to go next.

CDL Requirements

The CDL needed for any particular architecture HAL depends to a large extent on the needs of that architecture. This includes issues such as support for different variants, use of FPUs, MMUs and caches. The exact split between the architecture, variant and platform HALs for various features is also somewhat fluid.

To give a rough idea about how the CDL for an architecture is structured, we will take as an example the I386 CDL.

This first section introduces the CDL package and placed it under the main HAL package. Include files from this package will be put in the `include/cyg/hal` directory, and definitions from this file will be placed in `include/pkgconf/hal_i386.h`. The `compile` line specifies the files in the `src` directory that are to be compiled as part of this package.

```
cdl_package CYGPKG_HAL_I386 {
    display       "i386 architecture"
    parent        CYGPKG_HAL
    hardware
    include_dir   cyg/hal
    define_header hal_i386.h
    description   "
        The i386 architecture HAL package provides generic
        support for this processor architecture. It is also
        necessary to select a specific target platform HAL
        package."

    compile       hal_misc.c context.S i386_stub.c hal_syscall.c
```

Next we need to generate some files using non-standard make rules. The first is `vectors.S`, which is not put into the library, but linked explicitly with all applications. The second is the generation of the `target.ld` file from `i386.ld` and the startup-selected `.ldi` file. Both of these are essentially boilerplate code that can be copied and edited.

```
make {
    <PREFIX>/lib/vectors.o : <PACKAGE>/src/vectors.S
    $(CC) -Wp,-MD,vectors.tmp $(INCLUDE_PATH) $(CFLAGS) -c -o $@ $<
    @echo $@ ": \\" > $(notdir $@).deps
    @tail +2 vectors.tmp >> $(notdir $@).deps
    @echo >> $(notdir $@).deps
    @rm vectors.tmp
}

make {
    <PREFIX>/lib/target.ld: <PACKAGE>/src/i386.ld
    $(CC) -E -P -Wp,-MD,target.tmp -DEXTRAS=1 -xc $(INCLUDE_PATH) $(CFLAGS) -o $@ $<
    @echo $@ ": \\" > $(notdir $@).deps
    @tail +2 target.tmp >> $(notdir $@).deps
    @echo >> $(notdir $@).deps
    @rm target.tmp
}
```

The i386 is currently the only architecture that supports SMP. The following CDL simply enabled the HAL SMP support if required. Generally this will get enabled as a result of a `requires` statement in the kernel. The `requires` statement here turns off lazy FPU switching in the FPU support code, since it is inconsistent with SMP operation.

```
cdl_component CYGPKG_HAL_SMP_SUPPORT {
    display       "SMP support"
    default_value 0
    requires { CYGHWR_HAL_I386_FPU_SWITCH_LAZY == 0 }

    cdl_option CYGPKG_HAL_SMP_CPU_MAX {
        display       "Max number of CPUs supported"
        flavor        data
        default_value 2
    }
}
```

The i386 HAL has optional FPU support, which is enabled by default. It can be disabled to improve system performance. There are two FPU support options: either to save and restore the FPU state on every context switch, or to only switch the FPU state when necessary.

```
cdl_component CYGHWR_HAL_I386_FPU {
    display       "Enable I386 FPU support"
    default_value 1
    description   "This component enables support for the
        I386 floating point unit."

    cdl_option CYGHWR_HAL_I386_FPU_SWITCH_LAZY {
        display       "Use lazy FPU state switching"
        flavor        bool
        default_value 1

        description "
            This option enables lazy FPU state switching.
            The default behaviour for eCos is to save and
            restore FPU state on every thread switch, interrupt
            and exception. While simple and deterministic, this
            approach can be expensive if the FPU is not used by
            all threads. The alternative, enabled by this option,
            is to use hardware features that allow the FPU state
            of a thread to be left in the FPU after it has been
            descheduled, and to allow the state to be switched to
            a new thread only if it actually uses the FPU. Where
            only one or two threads use the FPU this can avoid a
            lot of unnecessary state switching."
    }
}
```

The i386 HAL also has support for different classes of CPU. In particular, Pentium class CPUs have extra functional units, and some variants of GDB expect more registers to be reported. These options enable these features. Generally these are enabled by `requires` statements in variant or platform packages, or in `.ecm` files.

```
cdl_component CYGHWR_HAL_I386_PENTIUM {
    display       "Enable Pentium class CPU features"
    default_value 0
    description   "This component enables support for various
        features of Pentium class CPUs."

    cdl_option CYGHWR_HAL_I386_PENTIUM_SSE {
        display       "Save/Restore SSE registers on context switch"
        flavor        bool
        default_value 0

        description "
            This option enables SSE state switching. The default
            behaviour for eCos is to ignore the SSE registers.
            Enabling this option adds SSE state information to
            every thread context."
    }

    cdl_option CYGHWR_HAL_I386_PENTIUM_GDB_REGS {
        display       "Support extra Pentium registers in GDB stub"
        flavor        bool
        default_value 0

        description "
        This option enables support for extra Pentium registers
        in the GDB stub. These are registers such as CR0-CR4, and
        all MSRs. Not all GDBs support these registers, so the
        default behaviour for eCos is to not include them in the
        GDB stub support code."
    }
}
```

In the i386 HALs, the linker script is provided by the architecture HAL. In other HALs, for example MIPS, it is provided in the variant HAL. The following option provides the name of the linker script to other elements in the configuration system.

```
    cdl_option CYGBLD_LINKER_SCRIPT {
        display "Linker script"
        flavor data
        no_define
        calculated  { "src/i386.ld" }
    }
```

Finally, this interface indicates whether the platform supplied an implementation of the `hal_i386_mem_real_region_top()` function. If it does then it will contain a line of the form: `implements CYGINT_HAL_I386_MEM_REAL_REGION_TOP`. This allows packages such as RedBoot to detect the presence of this function so that they may call it.

```
    cdl_interface CYGINT_HAL_I386_MEM_REAL_REGION_TOP {
        display  "Implementations of hal_i386_mem_real_region_top()"
    }

}
```

Chapter 8. Future developments

The HAL is not complete, and will evolve and increase over time. Among the intended developments are:

- Common macros for interpreting the contents of a saved machine context. These would allow portable code, such as debug stubs, to extract such values as the program counter and stack pointer from a state without having to interpret a HAL_SavedRegisters structure directly.

- Debugging support. Macros to set and clear hardware and software breakpoints. Access to other areas of machine state may also be supported.

- Static initialization support. The current HAL provides a dynamic interface to things like thread context initialization and ISR attachment. We also need to be able to define the system entirely statically so that it is ready to go on restart, without needing to run code. This will require extra macros to define these initializations. Such support may have a consequential effect on the current HAL specification.

- CPU state control. Many CPUs have both kernel and user states. Although it is not intended to run any code in user state for the foreseeable future, it is possible that this may happen eventually. If this is the case, then some minor changes may be needed to the current HAL API to accommodate this. These should mostly be extensions, but minor changes in semantics may also be required.

- Physical memory management. Many embedded systems have multiple memory areas with varying properties such as base address, size, speed, bus width, cacheability and persistence. An API is needed to support the discovery of this information about the machine's physical memory map.

- Memory management control. Some embedded processors have a memory management unit. In some cases this must be enabled to allow the cache to be controlled, particularly if different regions of memory must have different caching properties. For some purposes, in some systems, it will be useful to manipulate the MMU settings dynamically.

- Power management. Macros to access and control any power management mechanisms available on the CPU implementation. These would provide a substrate for a more general power management system that also involved device drivers and other hardware components.

- Generic serial line macros. Most serial line devices operate in the same way, the only real differences being exactly which bits in which registers perform the standard functions. It should be possible to develop a set of HAL macros that provide basic serial line services such as baud rate setting, enabling interrupts, polling for transmit or receive ready, transmitting and receiving data etc. Given these it should be possible to create a generic serial line device driver that will allow rapid bootstrapping on any new platform. It may be possible to extend this mechanism to other device types.

Part III. The ISO Standard C and Math Libraries

Table of Contents

Chapter 9. C and math library overview

eCos provides compatibility with the ISO 9899:1990 specification for the standard C library, which is essentially the same as the better-known ANSI C3.159-1989 specification (C-89).

There are three aspects of this compatibility supplied by *eCos*. First there is a *C library* which implements the functions defined by the ISO standard, except for the mathematical functions. This is provided by the eCos C library packages.

Then *eCos* provides a math library, which implements the mathematical functions from the ISO C library. This distinction between C and math libraries is frequently drawn — most standard C library implementations provide separate linkable files for the two, and the math library contains all the functions from the `math.h` header file.

There is a third element to the ISO C library, which is the environment in which applications run when they use the standard C library. This environment is set up by the C library startup procedure (the section called “C library startup”>) and it provides (among other things) a `main()` entry point function, an `exit()` function that does the cleanup required by the standard (including handlers registered using the `atexit()` function), and an environment that can be read with `getenv()`.

The description in this manual focuses on the *eCos*-specific aspects of the C library (mostly related to *eCos*'s configurability) as well as mentioning the omissions from the standard in this release. We do not attempt to define the semantics of each function, since that information can be found in the ISO, ANSI, POSIX and IEEE standards, and the many good books that have been written about the standard C library, that cover usage of these functions in a more general and useful way.

Included non-ISO functions

The following functions from the POSIX specification are included for convenience:

`extern char **`*environ* variable (for setting up the environment for use with `getenv()`)
`_exit()`
`strtok_r()`
`rand_r()`
`asctime_r()`
`ctime_r()`
`localtime_r()`
`gmtime_r()`

eCos provides the following additional implementation-specific functions within the standard C library to adjust the date and time settings:

```
void cyg_libc_time_setdst(
    cyg_libc_time_dst state
);
```

This function sets the state of Daylight Savings Time. The values for state are:

```
CYG_LIBC_TIME_DSTNA   unknown
CYG_LIBC_TIME_DSTOFF  off
CYG_LIBC_TIME_DSTON   on
```

These values will be reflected in the `tm_isdst` member of a struct tm. No other meaning is given to `CYG_LIBC_TIME_DSTNA`, and in particular it is not interpreted as any sort of "auto-detect" value, as eCos does not have the extensive timezone information that would be required in order to provide this. A call to `mktime()` with `tm_isdst` set to `-1` (which corresponds to `CYG_LIBC_TIME_DSTNA`) will be treated as if the supplied time is in UTC, i.e. with neither standard time nor Daylight Savings Time offsets applied.

```
void cyg_libc_time_setzoneoffsets(
    time_t stdoffset, time_t dstoffset
);
```

This function sets the offsets from UTC used when Daylight Savings Time is enabled or disabled. The offsets are in time_t,s, which are seconds in the current inplementation.

```
Cyg_libc_time_dst cyg_libc_time_getzoneoffsets(
    time_t *stdoffset, time_t *dstoffset
);
```

This function retrieves the current setting for Daylight Savings Time along with the offsets used for both STD and DST. The offsets are both in time_t,s, which are seconds in the current implementation.

```
cyg_bool cyg_libc_time_settime(
      time_t utctime
);
```

This function sets the current time for the system The time is specified as a time_t in UTC. It returns non-zero on error.

Math library compatibility modes

This math library is capable of being operated in several different compatibility modes. These options deal solely with how errors are handled.

There are 4 compatibility modes: ANSI/POSIX 1003.1; IEEE-754; X/Open Portability Guide issue 3 (XPG3); and System V Interface Definition Edition 3.

In IEEE mode, the `matherr()` function (see below) is never called, no warning messages are printed on the stderr output stream, and errno is never set.

In ANSI/POSIX mode, errno is set correctly, but `matherr()` is never called and no warning messages are printed on the stderr output stream.

In X/Open mode, errno is set correctly, `matherr()` is called, but no warning messages are printed on the stderr output stream.

In SVID mode, functions which overflow return a value HUGE (defined in `math.h`), which is the maximum single precision floating point value (as opposed to HUGE_VAL which is meant to stand for infinity). errno is set correctly and `matherr()` is called. If `matherr()` returns 0, warning messages are printed on the stderr output stream for some errors.

The mode can be compiled-in as IEEE-only, or any one of the above methods settable at run-time.

Note

This math library assumes that the hardware (or software floating point emulation) supports IEEE-754 style arithmetic, 32-bit 2's complement integer arithmetic, doubles are in 64-bit IEEE-754 format.

matherr()

As mentioned above, in X/Open or SVID modes, the user can supply a function `matherr()` of the form:

```
int matherr( struct exception *e )
```

where struct exception is defined as:

```
struct exception {
  int type;
  char *name;
  double arg1, arg2, retval;
};
```

type is the exception type and is one of:

DOMAIN

argument domain exception

SING

argument singularity

OVERFLOW

overflow range exception

UNDERFLOW

underflow range exception

TLOSS

total loss of significance

PLOSS

partial loss of significance

`name` is a string containing the name of the function

`arg1` and `arg2` are the arguments passed to the function

`retval` is the default value that will be returned by the function, and can be changed by `matherr()`

Note

matherr must have “C” linkage, not “C++” linkage.

If matherr returns zero, or the user doesn't supply their own matherr, then the following *usually* happens in SVID mode:

Table 9.1. Behavior of math exception handling

Type	Behavior
DOMAIN	0.0 returned, errno=EDOM, and a message printed on stderr
SING	HUGE of appropriate sign is returned, errno=EDOM, and a message is printed on stderr
OVERFLOW	HUGE of appropriate sign is returned, and errno=ERANGE
UNDERFLOW	0.0 is returned and errno=ERANGE
TLOSS	0.0 is returned, errno=ERANGE, and a message is printed on stderr
PLOSS	The current implementation doesn't return this type

X/Open mode is similar except that the message is not printed on stderr and HUGE_VAL is used in place of HUGE

Thread-safety and re-entrancy

With the appropriate configuration options set below, the math library is fully thread-safe if:

- Depending on the compatibility mode, the setting of the errno variable from the C library is thread-safe

- Depending on the compatibility mode, sending error messages to the stderr output stream using the C library `fputs()` function is thread-safe
- Depending on the compatibility mode, the user-supplied `matherr()` function and anything it depends on are thread-safe

In addition, with the exception of the `gamma*()` and `lgamma*()` functions, the math library is reentrant (and thus safe to use from interrupt handlers) if the Math library is always in IEEE mode.

Some implementation details

Here are some details about the implementation which might be interesting, although they do not affect the ISO-defined semantics of the library.

- It is possible to configure *eCos* to have the standard C library without the kernel. You might want to do this to use less memory.
- The opaque type returned by `clock()` is called clock_t, and is implemented as a 64 bit integer. The value returned by `clock()` is only correct if the kernel is configured with real-time clock support, as determined by the CYGVAR_KERNEL_COUNTERS_CLOCK configuration option in `kernel.h` .
- The FILE type is not implemented as a structure, but rather as a CYG_ADDRESS.
- The GNU C compiler will replace its own *built-in* implementations instead of calls to some C library functions. This can be turned off with the *-fno-builtin* option. But it is recommended for normal use to leave compiler builtins enabled. The functions affected by this are described in the documentation associated with the particular GNU compiler version you are using, but include at least:

  ```
  abs()
  cos()
  fabs()
  labs()
  memcmp()
  memcpy()
  sin()
  sqrt()
  strcmp()
  strcpy()
  strlen()
  ```

- `memcpy()` and `memset()` are located in the infrastructure package, not in the C library package. This is because the compiler calls these functions, and the kernel needs to resolve them even if the C library is not configured.
- Error codes such as EDOM and ERANGE, as well as `strerror()` , are implemented in the *error* package. The error package is separate from the rest of the C and math libraries so that the rest of *eCos* can use these error handling facilities even if the C library is not configured.
- The memory allocation package `CYGPKG_MEMALLOC` is responsible for providing the various heap management functions such as `malloc()`, `free()`, etc.
- Signals, as implemented by `<signal.h>`, are guaranteed to work correctly if raised using the `raise()` function from a normal working program context. Using signals from within an ISR or DSR context is not expected to work. Also, it is not guaranteed that if CYGSEM_LIBC_SIGNALS_HWEXCEPTIONS is set, that handling a signal using `signal()` will necessarily catch that form of exception. For example, it may be expected that a divide-by-zero error would be caught by handling `SIGFPE`. However it depends on the underlying HAL implementation to implement the required hardware exception. And indeed the hardware itself may not be capable of detecting these exceptions so it may not be possible for the HAL implementer to do this in any case. Despite this lack of guarantees in this respect, the signals implementation is still ISO C compliant since ISO C does not offer any such guarantees either.
- If you include the POSIX compatibility layer in your configuration, by default it will present a conflict if the C library signals implementation is also present. Only one signals implementation may be present.
- The `getenv()` function is implemented (as long as the CYGPKG_LIBC_STDLIB package is present in your configuration), but there is no shell or `putenv()` function to set the environment dynamically. The environment is set in a global variable environ, declared as:

```
extern char **environ; // Standard environment definition
```

If the "ISO environment startup/termination" (CYGPKG_LIBC_STARTUP) package is included in your configuration, the environment can be statically initialized at startup time using the CYGDAT_LIBC_DEFAULT_ENVIRONMENT option. If so, remember that the final entry of the array initializer must be NULL.

Here is a minimal *eCos* program which demonstrates the use of environments (see also the test case in `language/c/libc/`*`VERSION`*`/tests/stdlib/getenv.c`):

```
#include <stdio.h>
#include <stdlib.h> // Main header for stdlib functions

extern char **environ; // Standard environment definition

int
main( int argc, char *argv[] )
{
    char *str;
    char *env[] = { "PATH=/usr/local/bin:/usr/bin",
    "HOME=/home/fred",
    "TEST=1234=5678",
    "home=hatstand",
    NULL };

    printf("Display the current PATH environment variable\n");

    environ = (char **)&env;

    str = getenv("PATH");

    if (str==NULL) {
        printf("The current PATH is unset\n");
    } else {
        printf("The current PATH is \"%s\"\n", str);
    }
    return 0;
}
```

Thread safety

The ISO C library has configuration options that control thread safety, i.e. working behavior if multiple threads call the same function at the same time.

The following functionality has to be configured correctly, or used carefully in a multi-threaded environment:

- **`mblen`** `();`
- **`mbtowc`** `();`
- **`wctomb`** `();`
- **`printf`** `();`

  and all standard I/O functions except for

  `sprintf()` `();`
  `sscanf` `();`
- **`strtok`** `();`
- **`rand`** `();`

  srand ();

- **signal** ();
  raise ();

- **asctime** ();
  ctime ();
  gmtime ();
  localtime ();

- the `errno` variable

- the `environ` variable

- date and time settings

In some cases, to make *eCos* development easier, functions are provided (as specified by POSIX 1003.1) that define re-entrant alternatives, i.e. `rand_r()`, `strtok_r()`, `asctime_r()`, `ctime_r()`, `gmtime_r()`, and `localtime_r()`. In other cases, configuration options are provided that control either locking of functions or their shared data, such as with standard I/O streams, or by using per-thread data, such as with the `errno` variable.

In some other cases, like the setting of date and time, no re-entrant or thread-safe alternative or configuration is provided as it is simply not a worthwhile addition (date and time should rarely need to be set.)

C library startup

The C library includes a function declared as:

```
void cyg_iso_c_start( void )
```

This function is used to start an environment in which an ISO C style program can run in the most compatible way.

What this function does is to create a thread which will invoke `main()` — normally considered a program's entry point. In particular, it can supply arguments to `main()` using the CYGDAT_LIBC_ARGUMENTS configuration option, and when returning from `main()`, or calling `exit()`, pending stdio file output is flushed and any functions registered with `atexit()` are invoked. This is all compliant with the ISO C standard in this respect.

This thread starts execution when the *eCos* scheduler is started. If the *eCos* kernel package is not available (and hence there is no scheduler), then `cyg_iso_c_start()` will invoke the `main()` function directly, i.e. it will not return until the `main()` function returns.

The `main()` function should be defined as the following, and if defined in a C++ file, should have “C” linkage:

```
extern int main(
    int argc,
    char *argv[] )
```

The thread that is started by `cyg_iso_c_start()` can be manipulated directly, if you wish. For example you can suspend it. The kernel C API needs a handle to do this, which is available by including the following in your source code.

```
extern cyg_handle_t cyg_libc_main_thread;
```

Then for example, you can suspend the thread with the line:

```
cyg_thread_suspend( cyg_libc_main_thread );
```

If you call `cyg_iso_c_start()` and do not provide your own `main()` function, the system will provide a `main()` for you which will simply return immediately.

In the default configuration, `cyg_iso_c_start()` is invoked automatically by the `cyg_package_start()` function in the infrastructure configuration. This means that in the simplest case, your program can indeed consist of simply:

```
int main( int argc, char *argv[] )
{
    printf("Hello eCos\n");
}
```

If you override `cyg_package_start()` or `cyg_start()`, or disable the infrastructure configuration option CYGSEM_START_ISO_C_COMPATIBILITY then you must ensure that you call `cyg_iso_c_start()` yourself if you want to be able to have your program start at the entry point of `main()` automatically.

Chapter 10. Overview of ISO Standards Compliance

This chapter has been prepared in order to provide an overview of the compliance of the eCos Standard C and Math libraries against ISO C and C++ Standards, as implemented in eCosPro.

It is intended to describe functionality required by the ISO/IEC 9899:2011 C and ISO/IEC 14882:2011 C++ standards which is missing, as well as behaviour which differs from the standards, or only meets the specification of the standards in part. This documentation is focused on these 2011 revisions of the standards, and earlier and later revisions of these standards are not covered.

Further details on compliance with the ISO C++ standard can be found in the documentation on eCosPro's Standard C++ support.

The general approach which will be taken is to describe standards compliance with regard to the APIs defined by each standard header file, in turn. By examining compliance on a header by header basis, rather than looking at each standard in turn, it is hoped to avoid confusing duplication due to the considerable overlap in headers between the C and C++ standards.

Definitions

Some terms are frequently used in this document, or only used in shorthand form, and are defined here to allow us to reference them later more conveniently.

C90
: The ISO/IEC 9899:1990 C standard, occasionally also known as C89, or informally as “ANSI C” (which differ only in formatting with the ISO standard).

C99
: The ISO/IEC 9899:1999 C standard

C11
: The ISO/IEC 9899:2011 C standard

C++11
: The ISO/IEC 14882:2011 C++ standard. Note that this standard does not implicitly require the C11 standard, but the C99 standard.

GCC
: The GNU Compiler Collection, including C and C++ compilers. Unless otherwise described, this should be taken as to refer to GCC 7 including patches supplied by eCosCentric in order to support use with eCosPro. Vanilla GCC sources from the main GCC download sites are not suitable for use without the eCosPro-specific patches.

libstdc++
: The Standard C++ Library provided by GCC. It provides much of the library functionality defined by the C++ standard.

Scope

Broadly, this documentation is only intended as an overview. Although it is intended to cover as much as possible, it is not guaranteed to be fully complete. While broad areas of non-compliance should be identified here, some elements of finer detail not in compliance with the standards may not have been addressed. As a result, this documentation does not warrant that, just because non-compliance has not been identified here, that that means that eCosPro and GCC are fully compliant in all remaining areas.

At time of writing, eCosPro is currently supplied with GCC 7, which is a version of the compiler intended to be compliant with the C11 and C++11 standards.

Although there are exceptions, in general, GCC provides language parts of standard, eCos provides runtime library parts of standard, but GCC must interact with eCos for various abstractions. GCC also provides the Standard C++ Library (libstdc++). In this document, we do not cover conformance of portions provided by GCC, including the majority of the very large Standard C++ library.

For more information on GCC's standards compliance, the webpage at https://gcc.gnu.org/onlinedocs/gcc-7.3.0/gcc/Standards.html may be informative (also note that the similarly named webpage https://gcc.gnu.org/onlinedocs/gcc/Standards.html refers to GCC standards compliance information for the very latest version of GCC, and not necessarily the version supplied by eCosCentric for use with eCosPro).

There is also detailed information on GCC's ongoing compliance efforts with the various C++ standards, described at https://www.gnu.org/software/gcc/projects/cxx-status.html.

This documentation only covers compliance against the C11 and C++11 standards. It does not cover the specifics of more recent revisions of the standards such as C18, C++14 or C++17, even though it is understood some of the differences are well understood, and for example in the case of C++11 versus C++14, the differences are relatively small.

General Overview

Much of the support for the runtime library portions of the standards comes from the C library suite of packages, with certain elements also coming from the Infrastructure package. In general, the eCos C library was written to be compliant with the C90 standard, with some elements of C99 compliance added subsequently.

Particularly notable and wide-ranging omissions are in the areas of internationalization and localization, such as wide character and multibyte character variations of functions, many of which are absent. This also affects the level of internationalization/localization functionality that libstdc++ is able to provide as well.

Standard eCos headers occasionally make use of identifiers which are not reserved by the standard, which means that these identifiers may not be able to be used by programs which would otherwise be C11/C++11 standards-compliant. This is typically known as "namespace pollution", and fortunately is usually easy to work around.

Building with the GCC compiler flag `-std=`... may not work as intended as eCos headers do not always adapt to different language standard requirements.

Function prototypes that are expected to use the "`restrict`" keyword for arguments widely do not do so.

Annex K of C11 describes bounds-checking variants of many standard functions. This is an optional part of the standard and eCos does not implement it.

Feature-test macros such as those defined in C11 sections 6.10.8.2 and 6.10.8.3 (all of which begin with the prefix "`__STDC_`") are not defined.

Common C/C++ headers

These headers are required by both the C11 and C++11 standards, although note that strictly C++11 itself only requires C99 compliance.

In all cases, where the C++11 standard describes a C++ wrapper for the listed header files, such as `<cassert>` for `<assert.h>` or `<cstdlib>` for `<stdlib.h>`, these comments apply equally to headers included via these wrappers.

`<assert.h>`

static_assert is not defined.

`<complex.h>`

This header is not provided if compiling for C11. Unusually, a compatibility header is provided by GCC if compiling for C++11 to provide an implementation of specific elements from the C++ standard.

<ctype.h>

No compliance issues noticed.

<errno.h>

No compliance issues noticed.

<fenv.h>

This header is not supported by eCos.

<float.h>

No compliance issues noticed.

<inttypes.h>

This header is provided by eCos. It includes integer type format string definitions to be used with the `printf()` and `scanf()` families of functions. It also includes the `imaxabs()`, `imaxdiv()`, `strtoimax()` and `strtoumax()` functions. However the wide character related functions `wcstoimax()` and `wcstoumax()` are not implemented.

<iso646.h>

No compliance issues noticed.

<limits.h>

No compliance issues noticed.

<locale.h>

“struct lconv” does not contain members: *`int_p_cs_precedes`*, *`int_n_cs_precedes`*, *`int_p_sep_by_space`*, *`int_n_sep_by_space`*, *`int_p_sign_posn`*, *`int_n_sign_posn`*.

<math.h>

The following identifiers are not defined: `HUGE_VALL`, `INFINITY`, `NAN`, `FP_FASTFMA*`, `FP_ILOG*`.

Error handling does not use `math_errhandling` to generate floating point exceptions. `MATH_ERRNO`, `MATH_ERREXCEPT`, `math_errhandling` are not defined.

No “long double” variants of any maths functions are provided. For example, `sin()` and `sinf()` are provided, but not `sinl()`.

None of `llround()`, `llroundf()` or `llroundl()` are provided. None of the `nexttoward()`, `nexttowardf()`, `nexttowardl()` functions are provided.

<setjmp.h>

While in general the `<setjmp.h>` functionality operates as defined, it is worth noting that in line with the comments about `<signal.h>`, in eCos configurations where the POSIX package is not used and where a hardware exception context is used to handle hardware exception related signals (SIGSEGV, SIGILL, etc.) the fact that the signal handler is being called in a non-standard CPU context means that the practice of longjmp()ing out of a signal handler must be avoided.

<signal.h>

The signal subsystem is broadly compliant. But it is worth noting that if the eCos configuration is set to use the C library signals package (CYGPKG_LIBC_SIGNALS), then while signals generated (with raise()) by software behave normally, hardware-related signals such as SIGILL, SIGFPE or SIGSEGV may result in the signal handler being invoked in an unusual CPU context such as an exception context, and the effect of returning from the signal handler or attempting to longjmp() out of it is undefined. In an exception context, calling certain functions such as those which may interact with synchronisation objects (mutexes etc.) or cause pre-emption are likely to misbehave.

For compliant behaviour, instead it is recommended to use the signals facility provide by the eCos POSIX compatibility package (CYGPKG_POSIX) in place of CYGPKG_LIBC_SIGNALS.

<stdarg.h>

No compliance issues noticed.

<stdbool.h>

No compliance issues noticed.

<stddef.h>

No compliance issues noticed.

<stdint.h>

No compliance issues noticed.

<stdio.h>

None of the wide character input or output functions, nor any other definitions and functionality related to wide character support, are provided. These include at least the functions: fgetwc(), fgetws(), getwc(), getwchar(), fwscanf(), wscanf(), vfwscanf(), vwscanf(), fputwc(), fputws(), putwc(), putwchar(), fwprintf(), wprintf(), vfwprintf(), vwprintf() and ungetwc(). It also includes the mbstate_t type.

freopen() will always report an error on return.

Not all format specifiers or length modifiers for the *printf() and *scanf() families of functions are supported. For example: the 'a' and 'A' floating-point format specifiers, the 'j' length modifier for intmax_t / uintmax_t, or the 't' length modifier for ptrdiff_t. Also the 'l' length modifier where its behaviour is intended to treat the associated characters/string as wide characters.

<stdlib.h>

The llabs() and lldiv() functions, and associated lldiv_t type are not provided.

The strtold() function is not supported.

The strtod() and strtof() functions (and implicitly atof()) do not accept input text of the form: “a 0x or 0X, then a nonempty sequence of hexadecimal digits optionally containing a decimal-point character, then an optional binary exponent part”. They also do not recognise the special strings INF, INFINITY, or any NAN strings.

The `aligned_alloc()` function is not provided.

The `_Exit()`, `quick_exit()` and `at_quick_exit()` functions are not provided.

As eCos is designed for embedded systems, it is not intended for its main process to exit, as there is no other process to return to. Therefore while functions like `abort()`, `exit()`, and `atexit()` exist and are implemented, for that reason they may not behave in the expected way that developers using Unix systems may expect. Depending on eCos configuration, exiting may cause the system to halt, or only cause the `main()` thread to exit, allowing other threads to continue running.

<string.h>

No compliance issues noticed.

<tgmath.h>

This header file is not supported.

<time.h>

Some conversion specifiers for `strftime()` are not supported, for example: `%C`, `%F`, `%g`, `%G`, `%h`, `%n`, `%r`, `%R`, `%t`, `%u`, `%V`, `%z`. Also, the ‘`E`’ and ‘`O`’ modifiers are not supported.

<wchar.h>

This header file is not supported.

<wctype.h>

This header file is not supported.

C11 specific headers

<stdalign.h>

No compliance issues noticed.

<stdatomic.h>

Although eCosCentric has not tested this functionality, no compliance issues have been noticed.

<threads.h>

This header file is not supported.

<uchar.h>

This header file is not supported.

Part IV. eCosPro Standard C++ library support package

Table of Contents

Chapter 11. Introduction

This documentation describes the eCos support for the GNU standard C++ library v3 which is a component of the GNU Compiler Collection (GCC). This library, also known as libstdc++, has been designed to fully implement the requirements of the ISO 14822 standard C++ specification, and also provides some of the underlying support for language features such as C++ exceptions and run-time type identification (RTTI).

As with normal GNU toolchains, the standard C++ library is prebuilt alongside the toolchain. The library itself is not contained in this eCos package. Instead the purpose of this package is to provide any ancillary support for the library, provide the CDL definitions required for the correct operation of the library, provide a rigorous and broad testsuite for the library, and of course provide this documentation.

Although the standard C++ library is part of the toolchain, some enhancements have been made to the GCC compiler specifically to support eCos, details of which are found in Chapter 14, *Toolchain*.

Overview of features

The GNU standard C++ library implements virtually all the library requirements of the C++ standard. Details of the status of the library including known issues may be found on the GNU C++ Standard Library documentation pages.

In summary, the library provides support for standard C++ functionality such as:

- C++ exceptions
- Run-time type identification (RTTI) and type information
- Memory allocation routines: new, delete, allocators, etc.
- I/O streams, string streams and I/O manipulators
- C++ friendly numeric limits
- Strings and character traits
- Containers: queues, deques, lists, stacks, vectors, maps, sets and bitsets
- Iterators
- Algorithms such as sort, find, compare, count, replace, etc. that usually operate on containers and iterators
- Complex numbers
- Numeric arrays
- Numeric algorithms such as accumulate, inner product, partial sum, adjacent difference
- etc…

This eCos package for libstdc++ also provides support for thread-safe exceptions when using the eCos kernel, as well as expressing with CDL the requirements that the C++ library has on the rest of the eCos system. This is in fact an option within the package named `CYGPKG_LIBSTDCXX_LIBRARY`, which may be overridden and disabled, although this must be done at the developer's own risk.

This package also contains a large number of tests, including some rigorous tests of core functionality such as C++ exceptions (and in particular their thread-safety and correct operation in a multi-threaded environment), RTTI, and the main library features.

These may be found in the `tests` subdirectory within this package. The GNU libstdc++ v3 testsuite has also been imported and is found in the `tscpp` subdirectory.

The GNU libstdc++ implementation configures itself on the basis of underlying OS support. In a few areas, where underlying eCos support does not exist, the library configures itself to avoid the requirement for that support. This is normally of little consequence, for example due to libstdc++ providing an alternative implementation with a minor performance impact, or some trivial divergence from strict C++ standard semantics. In some cases the affected functionality is optional in the first place, for example for aspects of C99 standard support. There is one notable area which is affected however, which is that eCos contains very little support for wide characters (wchar). As such, libstdc++ configures itself to omit its own wide character interface that would have been implemented using the underlying OS wide character support. For example, this removes provision of the various wstring and wstreams classes and functions.

Chapter 12. Usage

The easiest way to start using eCos for C++ development is to use the special configuration template included in this release for this purpose. With command line configuration it may be used as in the following example for the Atmel EB40A target:

```
$ ecosconfig new eb40a libstdc++
```

If using the graphical tool, the template may be selected with the `Build->Templates` menu item.

Once the eCos libraries are configured and built, you may link your application. Be sure to append *`-lstdC++`* to the end of the link line. If you wish to use C++ exceptions, be sure to either remove *`-fno-exceptions`* from your compilation line, or append *`-fexceptions`* at the end of the compilation line. Similarly, to use RTTI, either remove *`-fno-rtti`* from your compilation line, or append *`-frtti`* to the end of the compilation line. Finally, despite what some targets may have used for their default compiler flags it is important that the option *`-fvtable-gc`* is *not* used.

Requirements

As noted earlier, this package uses CDL to set constraints on the rest of the eCos system for correct and standards compliant operation of the C++ library. By selecting with the libstdc++ configuration template, you will be able to start with a configuration with all the necessary packages included and options set.

Building C++ programs, particularly those that use templates heavily (either directly, or using the templates from libstdc++), can take a lot of memory on the build machine - figures in excess of 220Mb have been observed building the testsuite included with this package. Be sure to have sufficient RAM to prevent extended build times.

This package requires a patched version of the GNU compiler in order to correctly support thread-safe C++ exceptions, and avoid problems with common infrastructure underlying most standard C++ library classes. Refer to the tools building notes below for further details.

Warning

If a compiler is used which was not supplied by eCosCentric, nor built with eCosCentric's patch, the facilities provided by the standard C++ library will *not* be thread-safe. This may result in corruption, unexpected behaviour or a crash. In particular, C++ exceptions will not be thread-safe. Even if an application does not use exceptions directly, they may be used internally within the standard C++ library. Furthermore, certain infrastructure used by multiple C++ library objects (such as allocators) will also not be thread-safe.

The multi-thread protection provided by the standard C++ library intentionally only extends to subsystems shared by multiple classes. It does not mean that multiple threads can safely access individual object instances. If an individual object may be accessed by multiple threads simultaneously, access to that object will still need to be protected separately by the user at the application level, for example with a mutex. The thread-safety protection provided by the eCosCentric-enhanced version of the standard C++ library only ensures that threads can safely access different instances of C++ library objects without special protection by the user, despite many of these objects using functionality which requires shared global state. This approach is an intentional design decision in order to avoid unnecessary locking overhead.

Due partly to this specialised toolchain support it is not possible to use this package with the synthetic target, as native toolchains are built with specialised knowledge of the C++ runtime installed on the native OS. This is not solely due to the aforementioned patches, but also because of direct assumptions made as part of the GNU toolchain build procedure. As such, use under the synthetic target is unlikely ever to be possible.

It has been observed that GDB releases prior to GDB 6.1 can have difficulty debugging complex C++ applications, particularly those that extensively include template classes containing virtual functions. GDB 6.1 or above is recommended.

Issues to consider

Using C++ exceptions

There are a number of considerations when using C++ exceptions:

1. Care should be taken when compiling C++ code with `-fexceptions` (the default). GNU C++ has been designed so that exceptions do not necessarily add overhead to functions. However they may add overhead in the following circumstances. Namely, when:

    a. Exceptions are actually used; OR

    b. i. The function (funcA) calls another function (funcB) with a non-null exception specifier or no exception specifier; AND

        ii. that function (funcA) contains an automatic object; AND

        iii. that object is defined to use a destructor, and would therefore need to have the destructor called if the called function (funcB) threw an exception through this stack frame.

2. Not all eCos API functions have null exception specifiers yet, which may lead to very small unnecessary overhead when used from C functions in certain circumstances described earlier. Annotating all functions with null exception specifiers throughout the entirety of eCos is a massive job beyond the scope of the work done to provide C++ support. However many of the key APIs can be updated. In the current eCos sources, some obvious APIs have been updated such as all of the kernel C API and much of the ISO C/POSIX APIs.

3. Exceptions thrown from signal handlers are not supported.

4. Exceptions are not supported in ISRs, DSRs, nor ASRs. It is neither feasible nor sensible to support exceptions in ISRs or DSRs. Support for ASRs may be added at a future date, although there are no plans at present.

5. Use of C++ support from this package can increase the thread stack requirements markedly. For example, if you use C++ exceptions, you should expect to add around 4Kbytes to your stack requirements for each thread which can throw exceptions. Developers may find it useful to enable kernel thread stack overflow checking (`CYGFUN_KERNEL_THREADS_STACK_CHECKING` and possibly also `CYGFUN_KERNEL_ALL_THREADS_STACK_CHECKING`), especially if erratic behaviour is observed in threads using C++ features.

Application size

It is widely acknowledged that when using C++ libraries, memory can quickly be consumed, particularly code (ROM/Flash) space. Small targets may have difficulty running even short C++ programs. For example, it is recommended to use the MEC01 memory extension card on the Atmel AT91 evaluation board platforms (EB40, EB40A, EB42, EB55) to provide extra space for applications. The eCosPro AT91EB40A port can take advantage of the MEC01 if eCos is configured with CYGHWR_HAL_ARM_AT91_EB40A_MEC01_RAM enabled.

Similarly, because of the generally larger code size, download times can be lengthy if using a slow transfer mechanism such as 38400 bps serial. Alternative download options such as ethernet or fast JTAG emulator should be considered for an efficient development/debug cycle.

It may be possible in future to reduce the code size overhead using linker garbage collection more fully. Currently linker garbage collection is not performed on the GNU standard C++ library itself. However it is anticipated the savings will turn out to be small.

C++ exceptions in callbacks

eCos itself is not built with exception support (`-fexceptions`) for the reasons given earlier concerning the overhead of exception support. As a consequence throwing exceptions from callbacks will not work, e.g. when using `qsort()`, `bsearch()`, etc. eCos does not yet support a means of building individual files with differing flags - flags are manipulated only for complete packages or by using custom build rules which would be unacceptable due to maintenance overheads.

Licensing

The GNU standard C++ library has its own license distinct from that of eCos. As a basis it uses, like eCos, the GNU General Public License. Also like eCos it includes an exception that permits the use of the library in proprietary applications. The exception is as follows:

> As a special exception, you may use this file as part of a free software library without restriction. Specifically, if other files instantiate templates or use macros or inline functions from this file, or you compile this file and link it with other files to produce an executable, this file does not by itself cause the resulting executable to be covered by the GNU General Public License. This exception does not however invalidate any other reasons why the executable file might be covered by the GNU General Public License.

This exception is very similar to the eCos GPL exception, and is compatible with it. Further information on this license can be found in the C++Introduction of the libstdc++ online documentation set.

Most of the test files imported from the libstdc++ testsuite are covered by the full GPL without any exception. This means that distributing binaries of the test executables themselves gives a requirement to make available the full source code of that binary under the terms of the GPL. This is not considered an onerous obligation as distributing test binaries for this testsuite publically is unlikely to be a common requirement.

Standards Compliance

Versions of eCosPro which are shipped with GCC 7.3 or later are broadly compliant with the ISO/IEC 14882:2011 C++ standard (also known as C++11), albeit with some notable exceptions.

Information on GCC 7.3's standards compliance can be found at https://gcc.gnu.org/onlinedocs/gcc-7.3.0/gcc/Standards.html, or elsewhere on the same website for other GCC release versions.

In particular, there is more detail on the compliance efforts with the various C++ standards described at https://www.gnu.org/software/gcc/projects/cxx-status.html.

The C++11 standard in turn assumes that the library features of the system are compliant with the ISO/IEC 9899:1999 C standard (also known as C99). Information on eCosPro compliance with ISO C standards can be found in the relevant section of the eCos C and Math library documentation.

Futher notes on compliance with the C++11 standard are:

- Any programs requiring C++11 library features must include the eCos package "ISO standard C++" (`CYGPKG_LIBSTDCXX`) in the eCos configuration, with the "C++ library support" (`CYGPKG_LIBSTDCXX_LIBRARY`) option enabled.
- As noted earlier, a particularly notable divergence of eCos from the standards is the absence of most wide character support and functions.
- The level of functionality and standards compliance of the clocks provided for use with the `<chrono>` header are highly dependent on the eCos configuration in use, such as the presence of the common clock package (`CYGPKG_CLOCK_COMMON`), POSIX clocks (`CYGPKG_POSIX_CLOCKS`) and/or the eCos wallclock subsystem (`CYGPKG_IO_WALLCLOCK`).

  Additionally the “`steady_clock`” is not provided in the eCosPro implementation, and any use of it will result in the normal system clock being used instead.
- Although the C11 `<complex.h>` is not supported, the `<complex>` header defined by C++11 is.

- At the time of writing, the thread support library portion of C++11 (section 30) provided by the headers `<thread>`, `<mutex>`, `<shared_mutex>`, `<condition_variable>` and `<future>` are not yet of production quality, and despite appearing to be usable in eCosPro, their use cannot yet be considered supportable.

- C++ destructors for global C++ objects are not run on program exit.

Open issues

GCC 3.3.x issues

At this time there are only two significant known open issues that developers should be aware of which may impact development:

- GCC 3.3.x misoptimizes code in functions with complex number parameters. The workaround is to compile without -O2 (or append -O0 to the end of the compile line). This issue has been filed with the GCC project as bug #15061. As a consequence of this compiler bug, the complex2 test in this package is likely to fail.

- GCC 3.3.x fails to return NULL when using the `std::nothrow` variant of the `new` operator when an amount of memory is requested beyond what the system has available. Instead of NULL, 4 is returned. This is listed with the GCC project as bug #13215 and the problem is not going to be addressed in the GCC 3.3.x series. The problem is fixed in GCC 3.4. As a consequence of this compiler bug, the new1 test in this package will fail with GCC 3.3.x.

GCC 3.4.x issues

The only known issue affecting the use of GCC 3.4.x is specific to M68K/Coldfire, where the software floating point emulation is too imprecise, and causes a small number of tests within the libstdc++ package to fail, primarily those that test long double support.

Generic issues

It is worth mentioning that, as previously mentioned above, wide character support is not included. Support for wide characters may be developed in due course, but it would require significant development in the underlying eCos C library.

Chapter 13. Testing

As noted earlier, tests have been written to verify the operation of certain specific areas of interest in the C++ library support, particularly the use of exceptions from multiple threads which is addressed in the `throw*` tests.

The GNU libstdc++ v3 testsuite has been imported into this package and may be found under the `tscpp` subdirectory of this package. There are a large number of tests within the libstdc++ testsuite of varying rigour. An analysis of the coverage has been made, and any notable gaps in the test coverage have been addressed in the custom tests in the `tests` subdirectory of this package.

The testsuite is quite large and takes some time to build, and so although built by default it may disabled with the `CYGPKG_LIBSTDCXX_OFFICIAL_TESTSUITE` CDL option. Some tests contain aspects which only operate if the RAM filesystem package is enabled, therefore to test the library more thoroughly developers may wish to consider enabling the RAM filesystem.

Notes on how the libstdc++ testsuite was imported, including what types of changes were made and what the results were, are available within this package in the `tscpp/NOTES` text file.

Chapter 14. Toolchain

To build GCC for use with this package, it is necessary to follow some additional steps compared with what would ordinarily be required for building the compiler. These steps are required to provide the eCos header files which are used by the GCC build, to determine properties of the run-time system and to apply a set of changes (a “patch”) to allow eCos to provide C++ exception support in a flexible and future-proof way. This patch takes particular care to ensure that the compiler and libstdc++ continue to behave correctly when no eCos kernel is present.

1. With eCos installed, the `ECOS_REPOSITORY` environment variable set and **ecosconfig** in your `PATH` variable, run the following commands at a **bash** shell prompt in an empty directory, choosing a TARGET of the appropriate architecture:

   ```
   $ ecosconfig new TARGET libstdc++
     $ ecosconfig tree
     $ make headers
   ```

2. Take the header tree generated under `install/include` and install it in the `TARGET/sys-include` subdirectory where you intend to install your tools. For example if you wish to install the new tools to `/opt/newtools`, then place the headers in a new directory `/opt/newtools/TARGET/sys-include`. (Note you must ensure you have write-access to `/opt` in this example, or you can choose an alternate path).

3. A C++ exception support patch is supplied on the eCosPro Developer's Kit CD-ROM. Once the patch has been applied, it is necessary to run the following command:

   ```
    $ contrib/gcc_update --touch
   ```

4. Configure GCC from within an empty build directory as follows, ensuring that the GNU binary utilities are at the head of the `PATH`:

   ```
    $ /src/gcc-3.x.x/configure --target=TARGET \
             --prefix=/opt/newtools --enable-languages=c,C++ \
             --with-gnu-as --with-gnu-ld --with-newlib \
             --enable-threads
   ```

Part V. eCos Support for Dynamic Memory Allocation

Name

malloc, calloc, realloc, free, mallinfo, operator new, operator new[], operator delete and operator delete[] — Access the System Heap

Synopsis

#include <stdlib.h>

#include <new>

`void* malloc (size);`

`void* calloc (nmemb, size);`

`void* realloc (ptr, size);`

`void free (ptr);`

`struct mallinfo mallinfo ();`

`void* operator new (size);`

`void* operator new[] (size);`

`void* operator new (size, );`

`void* operator new(nothrow)[] (size, );`

`void operator delete (ptr);`

`void operator delete[] (ptr);`

`void operator delete (ptr, );`

`void operator delete[] (ptr, );`

Description

The dynamic memory allocation package `CYGPKG_MEMALLOC` provides support for the ISO standard C functions `malloc`, `calloc`, `realloc` and `free`. Optionally it can provide the C++ `new` and `delete` operators. There is extensive support for debugging various problems associated with dynamic memory allocation.

Some of the available target RAM will be needed for application code and static data. If the target uses RedBoot or another ROM monitor for bootstrap then that may also reserve some of the available RAM. On most targets the system heap occupies all remaining RAM, and this is used to satisfy the memory allocation requests. By default the Doug Lea memory allocator (dlmalloc) code is used to manage the heap. This provides a good trade off between efficient use of the memory, fast operation, and resistance to fragmentation. If the eCos configuration includes the kernel then by default the various memory allocation routines will be thread-safe.

To complement the standard APIs the memory allocation package provides support for custom memory pools.

C library functions

The main dynamic memory allocation routines defined by standard C is `malloc()`: this allocates a chunk of memory from the heap at least as large as the amount requested, satisfying any alignment restrictions imposed by the architecture. The initial contents of the allocated chunk is undefined. If the heap cannot satisfy the allocation request then a null pointer will be returned.

The standard does not define what happens when `malloc()` is passed a size of 0. In eCos this is controlled by a configuration option `CYGSEM_MEMALLOC_MALLOC_ZERO_RETURNS_NULL`. By default the option is disabled and an argument of *0* will still result in an allocation of the smallest size supported by the memory allocator. If the option is enabled then a null pointer will be returned instead.

`calloc()` tries to allocate a memory chunk of at least `nmemb*size` bytes. If the allocation succeeds then the memory will be filled with zeroes. Otherwise a null pointer is returned.

`realloc()` tries to change the size of an existing allocation while leaving the contents unchanged. This may involve resizing the chunk in situ, or it may involve a `malloc()`/`memcpy()`/`free()` sequence. If the operation succeeds a valid pointer will be returned, which may or may not be the same as the original pointer. If the operation fails then a null pointer will be returned and the original data remains intact. There are two special cases: if the *ptr* argument is a null pointer then `realloc()` will act like `malloc()`; otherwise if the *size* argument is 0 then `realloc()` will act like `free()`.

`free()` takes a pointer previously returned by `malloc()`, `calloc()` or `realloc()` and returns the memory to the heap.

`mallinfo()` is not defined by the C standard but is provided for compatibility with other systems. It returns information about the current state of the heap in the form of a mallinfo structure:

```
struct mallinfo {
    int arena;    /* total size of memory arena */
    int ordblks;  /* number of ordinary memory blocks */
    int uordblks; /* space used by ordinary memory blocks */
    int fordblks; /* space free for ordinary blocks */
    int maxfree;  /* size of largest free block */
};
```

arena gives the total heap size. *ordblks* and *uordblks* give some information on current allocations, and *fordblks* indicates how much is left. The remaining memory may be fragmented so *maxfree* indicates the largest allocation that is currently possible. A mallinfo structure contains a number of other fields but those are not used by eCos and exist only for compatibility reasons.

C++ operators

C++ applications can use the standard C library routines for dynamic memory allocation, but it is more common to use the C++ `new` and `delete` operators. There are a number of different implementations of these:

1. The infrastructure package contains empty versions of the delete operators which do not interact with the system heap in any way. This is necessary because of the way the **g++** compiler handles certain language constructs. Whenever there is a class with a virtual destructor the generated code always contains a reference to the `delete` operator. The linker is unable to delete this. Therefore if the application uses such a class, directly or indirectly, the final executable will contain a `delete` implementation - even if there is no dynamic allocation. The usual `delete` operator would pull in the system heap and hence the memory allocation package, significantly increasing code size for no good reason. Providing an empty `delete` avoids this.

   Unfortunately this solution is imperfect. If instead the application does want to create and destroy objects on the heap, by default the empty `delete` operators will still get linked in and the memory never gets freed. It is not possible to handle both scenarios cleanly with current tools, so instead the application developer has to configure eCos appropriately. To suppress the empty delete operators the configuration option `CYGFUN_INFRA_EMPTY_DELETE_FUNCTIONS` should be disabled. If an eCos package performs dynamic memory allocation using C++ `new` and `delete` then it should automatically disable this option via a CDL **requires** property.

2. The next implementation of `new` and `delete` comes in the C++ support library `libsupc++.a`, which is normally available as part of the GNU toolchain. These versions are straightforward, simply calling `malloc()` and `free()` to access the system heap. When linking an application with an eCos linker script the C++ support library is searched automatically, so application developers only need to worry about disabling `CYGFUN_INFRA_EMPTY_DELETE_FUNCTIONS`.

3. Finally the memory allocation package can also provide implementations of the C++ operators. These access the system heap directly rather than going via `malloc()` and `free()` so can be marginally faster, but at the cost of some increased code size.

There is a significant difference of the system is configured for collecting memory debug data. These implementations of `new` and `delete` integrate directly with the debug data code, so more information will be collected. This is especially useful on architectures where the compiler only provides limited backtrace support.

The configuration option `CYGFUN_MEMALLOC_MALLOC_CXX_OPERATORS` controls whether or not the memory allocation package's versions of the C++ operators get built. By default this option is disabled, unless `CYGDBG_MEMALLOC_DEBUG_DEBUGDATA` is enabled. When linking an application with an eCos linker script these operators will automatically be used in preference to the ones in `libsupc++.a`.

Debug Support

An application that uses dynamic memory allocation is often more difficult to debug than one that relies entirely on static allocation. To assist developers the memory allocation package provides a number of debugging facilities. The main one involves the collection of additional debug data for every memory allocation. This debug data can be transferred to the host and analyzed using a custom tool **ecosmdd**. Full documentation on this is provided elsewhere.

This package also provides support for some simple debugging techniques which can help detect certain problems. The first is memory guards: every allocated chunk is surrounded by a number of guard bytes. When the chunk is freed, using `free()` or the appropriate C++ `delete` operator, the guards are checked and any discrepancy is treated as an assertion failure. The head guard can detect certain buffer overflows in the previously allocated chunk. If the chunk contains a thread stack and the architecture involves a descending stack then the head guard can also detect stack overflows. The tail guard can detect certain overflows in the chunk being freed and underflows in the next chunk. The guards are reset during a free operation, which can help to catch attempts to free the same chunk twice. Guard checks only happen during a free operation so a corruption may go undetected for a long time, possibly too long, but are still better to never detecting corruption.

Memory guards are controlled by the configuration option `CYGDBG_MEMALLOC_MALLOC_DEBUG_GUARDS`. By default they are enabled if system-wide debugging (`CYGPKG_INFRA_DEBUG`) is enabled, otherwise disabled.

The second debugging technique is to fill memory chunks when they are freed. This helps to catch some attempts to use a pointer which is no longer valid. Such problems are particularly common in multi-threaded applications where thread A frees a chunk that thread B is still using. When freed chunks are filled thread B will suddenly see spurious data, often resulting in bus errors or other exceptions. The relevant configuration option is `CYGDBG_MEMALLOC_MALLOC_DEBUG_FILL_FREE`, which by default is also enabled if `CYGPKG_INFRA_DEBUG` is enabled. The option's value determines what the freed chunk gets filled with, usually 0xff.

Name

cyg_mempool_fix_*() and cyg_mempool_var_*() — Additional Memory Pools

Synopsis

#include <cyg/kernel/kapi.h>

void **cyg_mempool_fix_create** (*base*, *size*, *blocksize*, *handle*, *fix*);

void **cyg_mempool_fix_delete** (*fixpool*);

void* **cyg_mempool_fix_alloc** (*fixpool*);

void* **cyg_mempool_fix_timed_alloc** (*fixpool*, *abstime*);

void* **cyg_mempool_fix_try_alloc** (*fixpool*);

void **cyg_mempool_fix_free** (*fixpool*, *ptr*);

cyg_bool_t **cyg_mempool_fix_waiting** (*fixpool*);

void **cyg_mempool_fix_get_info** (*fixpool*, *info*);

void **cyg_mempool_var_create** (*base*, *size*, *handle*, *var*);

void **cyg_mempool_var_delete** (*varpool*);

void* **cyg_mempool_var_alloc** (*varpool*, *size*);

void* **cyg_mempool_var_timed_alloc** (*varpool*, *size*, *abstime*);

void* **cyg_mempool_var_try_alloc** (*varpool*, *size*);

void **cyg_mempool_var_free** (*varpool*, *ptr*);

cyg_bool_t **cyg_mempool_var_waiting** (*varpool*);

void **cyg_mempool_var_get_info** (*varpool*, *info*);

Description

The memory allocation package provides support for additional memory pools, to complement the system heap. These pools are not created automatically by the system, they have to be created by application code or by other packages. There are exported APIs for two types of pool: fixed and variable.

Allocating memory from a fixed memory pool is very fast and, more importantly, deterministic. However the size of each allocation is fixed at the time the pool is created. This is not a problem if the required allocations are all the same size, or nearly so, but otherwise the memory will be used inefficiently. The pool cannot become fragmented.

Variable memory pools provide essentially the same functionality as the system heap, so are rarely used. However on some targets not all free memory is assigned automatically to the system heap. For example there may be a small area of fast on-chip memory as well as the slower external memory. The system heap will only use the latter. A variable memory pool can be created for the former, allowing application code to dynamically allocate fast memory where appropriate.

If the eCos configuration includes the kernel then by default the memory pool functions will be thread-safe. The pool functions do not implement the `malloc()` guard and free-fill debug facilities, nor the debug data support.

Fixed Memory Pools

A fixed memory pool must be created explicitly, for example:

```
#define BLOCK_SIZE    1024
#define BLOCK_COUNT   64

static cyg_uint32      pool_memory[((BLOCK_COUNT * BLOCK_SIZE)+3) / 4];
static cyg_handle_t    pool_handle;
static cyg_mempool_fix pool_data;

…

cyg_mempool_fix_create( (void*) pool_memory,
                        BLOCK_COUNT * BLOCK_SIZE,
                        BLOCK_SIZE,
                        &pool_handle, &pool_data );
```

This creates a pool of 63 1K blocks. *pool_memory* is normally allocated statically, but could also be a pointer to a special area of memory such as on-chip RAM, or it could even be dynamically allocated using `malloc()`. The pointer should be suitably aligned for the target architecture, usually to either a 32 or a 64 bit boundary. *pool_handle* can be used for subsequent pool operations. *pool_data* is a small data structure providing the space needed to administer the pool.

The above pool only provides 63 blocks, not 64. The administration overhead depends on the number of blocks so cannot all be allowed for in the *pool_data* structure. A small amount of the pool memory is consumed as well, effectively using up all of the first block. To eliminate this inefficiency:

```
#define BLOCK_SIZE    1024
#define BLOCK_COUNT   64
#define OVERHEAD      (((BLOCK_COUNT + 31) / 32) * sizeof(cyg_uint32))
#define ACTUAL_SIZE   ((BLOCK_SIZE * BLOCK_COUNT) + OVERHEAD)

static cyg_uint32      pool_memory[(ACTUAL_SIZE + 3) / 4];
static cyg_handle_t    pool_handle;
static cyg_mempool_fix pool_data;

…

cyg_mempool_fix_create( pool_memory,
                        ACTUAL_SIZE,
                        BLOCK_SIZE,
                        &pool_handle, &pool_data );
```

There are three functions for allocating memory. `cyg_mempool_fix_try_alloc()` is analogous to `malloc()`: it attempts to allocate a block from the pool, returning a null pointer if all blocks are currently in use. There is no need to specify the allocation size because all blocks are the same size. The other two functions are only available in configurations containing the eCos kernel. `cyg_mempool_fix_alloc()` will allocate a free block if there is one available, otherwise the current thread will be suspended until a block becomes available. A null pointer will only be returned if the thread is woken up again via `cyg_thread_release()`. `cyg_mempool_fix_timed_alloc()` may also suspend the current thread, but only for a number of clock ticks. If no block becomes free before the specified time is reached then a null pointer will be returned. The *abstime* argument is an absolute time, typically calculated by adding a cyg_tick_count_t timeout to the result of `cyg_current_time()`. In other words the pool API works in exactly the same way as kernel functions such as `cyg_semaphore_timed_wait()`. `cyg_mempool_fix_waiting()` can be used to check whether any threads are currently suspended waiting for a free block.

A block can be released using `cyg_mempool_fix_free()`. If a pool is no longer required it can be destroyed by a call to `cyg_mempool_fix_delete()`. Information about the current state of a pool can be obtained with `cyg_mempool_fix_get_info()`, in the form of a cyg_mempool_info structure:

```
typedef struct {
    cyg_int32 totalmem;
    cyg_int32 freemem;
    void*     base;
    cyg_int32 size;
    cyg_int32 blocksize;
    cyg_int32 maxfree;                  // The largest free block
} cyg_mempool_info;
```

Variable Memory Pools

The variable memory pool API is very similar to the fixed pool API. The key differences are:

1. `cyg_mempool_var_create()` does not take a block size parameter since the pool supports allocations of any size.

2. There is no special need to worry about overheads when creating the pool. The overheads will be shared between the allocations so spread throughout the pool

3. The block size is no longer implicit, so the three allocation routines need an explicit *`size`* argument.

4. Allocation operations are not deterministic and may take significantly longer than a fixed pool allocation. A variable pool is also vulnerable to memory fragmentation.

Name

mdd_dump and ecosmdd — Analyze Memory Usage

Synopsis

`mdd_dump`

`mdd_dumpnow`

`mdd_reset`

`ecosmdd stats` *`mddout.0`*

`ecosmdd dump` [options] [*`exe`*] *`mddout.0`*

`ecosmdd history` [options] [*`exe`*] *`mddout.0`* [*`mddout.1`* ...]

`ecosmdd diff` [options] [*`exe`*] *`mddout.0`* *`mddout.1`*

Description

Generally it is more difficult to debug an application that allocates memory dynamically than one that relies entirely on static allocation. Some problems such as buffer overflows can affect both. However the locations of static variables are readily determined from the linker map and debug information, so it is much easier to figure out which static buffer overflowed and then find the offending code. With dynamic allocation buffer overflows can still be detected, but it is much harder to figure out what each buffer is used for.

Another problem is excessive memory usage. A typical embedded system is designed with the smallest amount of memory that should suffice for the application. Often the application uses more memory than expected, and it is necessary to find out exactly where it is all going and where savings could be made. The alternative is a hardware redesign, associated delays, and increased manufacturing costs. A linker map gives details of the static data but not of dynamic allocations.

A third problem is memory leaks. If an application allocates memory that does not get freed then the heap will eventually run out. Usually this causes a system failure and means a reboot. It may take hours, days or even weeks, but any system failure is at best undesirable and at worst totally unacceptable.

The memory allocation package provides a debug data facility to assist developers faced with these problems. This involves storing additional metadata on the target for each allocated memory chunk, for example the function where the allocation occurred and the time that it happened. Configuration options control exactly what metadata gets collected. The debug data can be transferred from the target to the host in a gdb session, and then analyzed using the **ecosmdd** program. This provides a number of sub-commands: **stats**, **dump**, **history** and **diff**. It also provides various options for filtering, sorting and formatting the debug data.

Configuration Options

Memory debug data is not free. Collecting the debug data on the target requires extra memory and cpu cycles. To be useful the debug data has to be transferred to the host, and this can be time-consuming. If the application developer is tracking down problems with running out of memory then the debug data exacerbates the situation. Hence by default memory debug data is disabled, and there are configuration options to control exactly what gets enabled.

The first option to consider is `CYGDBG_MEMALLOC_DEBUG_DEBUGDATA`. This has to be explicitly enabled by the developer. If it is left disabled then no debug data functionality is available.

Once the main debug data option has been enabled the memory allocation code will collect information about all current allocations. The minimum information needed is a pointer to the allocated data, the number of bytes involved, a 32-bit sequence number to allow the host-side to identify and sort the allocations, plus another pointer for linked list management. This gives a minimum overhead of 16 bytes per allocated chunk (assuming a typical 32-bit processor). However this allows for only limited analysis. Additional fields are controlled by separate configuration options:

`CYGDBG_MEMALLOC_DEBUG_DEBUGDATA_ACTUAL_SIZE`

When the application requests say 12 bytes of data the memory allocation code will actually allocate more than this. There is some unavoidable overhead to keep track of the various allocations. There may be alignment restrictions. Optional Debug guards add to the overhead. If `CYGDBG_MEMALLOC_DEBUG_DEBUGDATA_ACTUAL_SIZE` is enabled then the debug data will include the actual size of each allocation, not just the requested size. By default this option is enabled. The cost is an extra size_t, usually four bytes, in each allocation record.

`CYGDBG_MEMALLOC_DEBUG_DEBUGDATA_TIMESTAMP`

Every allocation record in the debug data contains a unique sequence number, a simple 32-bit counter. Amongst other uses this allows host-side tools to sort allocation events in time-order. However a sequence number does not give any information about the time elapsed between allocations. More detailed time information can be very useful, for example to associate allocations with external events. This takes the form of a cyg_tick_count_t as returned by the kernel function `cyg_current_time()`. The typical cost is an extra eight bytes in each allocation record.

`CYGDBG_MEMALLOC_DEBUG_DEBUGDATA_TIMESTAMP` is enabled by default if the eCos kernel `CYGPKG_KERNEL` is present. It cannot be enabled if the configuration does not include the kernel.

`CYGDBG_MEMALLOC_DEBUG_DEBUGDATA_THREAD`

In multi-threaded applications it can be useful to know which thread allocated which chunk of memory. For example if the application is structured as a set of mostly independent subsystems operating in a separate threads then each subsystem's memory usage can be analyzed separately. Optionally the debug data can include thread information, consisting of a unique numerical thread id, the cyg_handle_t identifying the thread, and the thread name as passed to `cyg_thread_create()`. The overhead is a 32-bit integer in each allocation record, plus a small amount of extra memory to keep track of the threads that have performed memory allocations.

`CYGDBG_MEMALLOC_DEBUG_DEBUGDATA_THREAD` is enabled by default if the eCos kernel `CYGPKG_KERNEL` is present. It cannot be enabled if the configuration does not include the kernel.

`CYGDBG_MEMALLOC_DEBUG_DEBUGDATA_BACKTRACE`

Arguably the most useful information about each memory allocation is a partial backtrace, identifying the code responsible for each allocation. On the target side this is implemented using the support function `__builtin_return_address()` provided by the **gcc** compiler. On the host-side the executable can be disassembled to map a return address onto the calling function. If the executable contains *`-g`* debug information then it may also be possible to work out the corresponding source file name and line number, and hence the exact line of code that performed the allocation.

`CYGDBG_MEMALLOC_DEBUG_DEBUGDATA_BACKTRACE` is enabled by default, with a value of 1. This means the debug data will contain a single level of backtrace, e.g. the function that called `malloc()`. The backtrace level can be increased up to a maximum of 8, giving more detailed information about each allocation. This is especially useful when allocations occur inside library code since it gives a closer association between application actions and memory allocations. Higher levels do involve extra memory overhead, a 32-bit integer per level per allocation record, and extra cpu cycles.

Important

On many architectures the GNU tools only provided limited backtrace functionality. Often only a single level of backtrace is available. If `CYGDBG_MEMALLOC_DEBUG_DEBUGDATA_BACKTRACE` is set to a value greater than 1 the compiler will issue warnings when building the memory allocation package, and the extra debug data backtrace slots will just be filled with zeroes.

Even if backtrace information is available it is not always as useful as might be thought. Because of compiler optimizations the relation between the generated code and the original source is not always obvious, so when the host-side tools convert a return address to a source file and line number the results may not be exactly correct. For backtrace levels greater than 1 the results may even be completely wrong. The details will vary from architecture to architecture. When the code involves C++ template instantiation the compiler may not provide enough debug information to allow the backtrace pointers to be analyzed fully.

Depending on which options and how many backtrace levels are enabled, each allocation record will take up between 16 and 64 bytes of data on a 32-bit processor, and somewhat more on a 64-bit processor.

By default memory debug data is collected only for current allocations. This is sufficient for many debug purposes. For example if the problem is a buffer overflow then looking at the current allocations usually allows the developer to determine what the buffer and the surrounding allocations are used for. A complete dump of all current allocations can be used to figure out what all the memory is being used for. Examining two dumps separated in time can be used to track down memory leaks. However sometimes it is necessary to know about free operations as well as current allocations. A good example is identifying which thread freed a chunk that other threads still believe to be usable. To support this it is possible to collect historical debug data as well as the details of all current allocations.

There is a major problem with historical debug data. The number of current allocations is limited by the memory available on the target, so typically will be somewhere between 100 and 10000. The corresponding debug data will occupy between 2K and 640K of the available target-side memory, and there is an implicit upper bound. Historical data does not have an upper bound: an application may make millions of `malloc()` and `free()` calls yet never have more than a 100 allocations at any one time. Those millions of history records would occupy many megabytes of target-side memory. Typical targets do not have such amounts of spare memory, and even if they do transferring the history to the host for analysis would be very time-consuming. Therefore it is not practical to keep a full history. Instead the history debug data goes into a circular buffer, so only the last `n` records are kept. Overflows are detected and the application developer can take action, if desired.

By default history is disabled, controlled by the configuration option `CYGDBG_MEMALLOC_DEBUG_DEBUGDATA_HISTORY`. If enabled the number of entries in the history circular buffer is controlled by `CYGDBG_MEMALLOC_DEBUG_DEBUGDATA_HISTORY_RECORDS`, with a default value of 2048. Each history record stores both allocation and free debug data, so is approximately twice the size of an allocation record. With default settings the history circular buffer will occupy approximately 100K of target-side memory.

Enabling memory debug data does not affect the memory allocation APIs: applications just call `malloc()` and `free()` as usual. Similarly C++ applications can use the `new` and `delete` operators, but to get the maximum benefits of the backtrace info it is desirable to enable `CYGFUN_MEMALLOC_MALLOC_CXX_OPERATORS`.

Dumping the Debug Data with GDB

When an application is linked with a suitable eCos configuration, the memory debug data will be collected automatically on the target-side. This debug data needs to be transferred to the host, and a number of gdb macros are provided for this purpose. The application is debugged in a gdb session as usual. At an appropriate time the target is halted and the appropriate gdb macro is invoked. This will transfer the current debug data to the host, generating a file `mddout.0` which can then be fed into the **ecosmdd** analysis program.

The gdb macros can be found in the file `mdd.gdb` in the memory allocation packages' `host` subdirectory. Typically this gdb script will be **source**'d by the user's own `.gdbinit` gdb initialization script, so that the macros are always available. Alternatively the macros can be copied directly into that file, albeit at the risk of complications if the macros get updated in a future version of this package. The `host` subdirectory also contains a program **ecosmdd** (actually a portable Tcl script). This must be installed in an appropriate location that is on the user's `PATH`. The gdb macros rely on being able to execute this program.

The main macro is **mdd_dump**. It does not take any arguments. Usually it will just transfer the memory debug data to the host. However there is a problem if the target-side code was in the middle of updating the debug data: that data may not be in an entirely consistent state. To avoid problems the **mdd_dump** will check a target-side busy flag. If appropriate it will report that a dump may currently be unsafe, instead of proceeding with the dump anyway. The function `cyg_memalloc_dd_done` will be called once the debug data has been updated, so an application developer can set a temporary breakpoint on that function and let the application continue briefly. Alternatively there is a separate macro **mdd_dumpnow**. This will ignore the busy flag and proceed with the dump, irrespective of what the target happened to be doing when it was halted. There is a very small possibility that the resulting dump file will have problems.

Note

The memory allocation code treats the actual allocation and the updating of the debug info as separate steps. Hence it is possible that a chunk of memory has just been allocated or freed, but the `mddout.0` dump file will not yet show this. Usually this temporary discrepancy is not important: it can only matter if the application developer is analysing the debug data and the target-side state concurrently. However application developers should be aware of the possibility. An alternative implementation involving more locking would be possible, but at the cost of potentially significant changes in the application's behaviour.

The time taken to generate a dump file will depend both on how much debug data is collected and on the debug communication channel. It can take anywhere from several seconds to many minutes. Enabling the history circular support can significantly increase the time needed.

Sometimes it is desirable to generate more than one `mddout` dump file in a single debug session. For example the user may want to halt the application at two specific points in the run and find out what allocations have occurred between these points. The first invocation of **mdd_dump** or **mdd_dumpnow** will produce a dump file `mddout.0`. Subsequent invocations will produce dump files `mddout.1`, `mddout.2`, and so on. If desired the numbering can be reset using the **mdd_reset** macro. The next debug session will again produce files `mddout.0`, `mddout.1` and so on, overwriting the previous run's results. The macro scripting facilities in gdb are rather limited, so the file naming is actually handled by invoking the **ecosmdd** program.

If the debug data includes the history circular buffer there is special support for handling overflows. This makes it possible to collect complete history information, spread over a number of `mddout` dump files, which can then be analyzed together. When an overflow occurs the target-side will call the function `cyg_memalloc_dd_history_overflow()`. Application developers can set a breakpoint on this function, and use **mdd_dump** whenever the breakpoint is hit to generate another dump file with a whole buffer's worth of history records. **mdd_dump** will automatically reset the circular buffer. `CYGDBG_MEMALLOC_DEBUG_DEBUGDATA_HISTORY_RECORDS` can be increased to reduce the number of dump files that are needed, at the cost of target-side memory.

A similar technique can be used for other purposes. For example the application developer may want to know the state of the heap once it has reached approximately 80% full. One way of achieving this is to have a separate high-priority thread which calls `mallinfo` at regular intervals. When it detects the desired condition it calls a special function. The developer sets a breakpoint on that function and can then take appropriate action when the condition is satisfied.

Extracting Statistics

The **ecosmdd stats** command is the simplest of the available analysis tools. It just takes a single argument, an `mddout` dump file:

```
$ ecosmdd stats mddout.0
mddout.0: statistics
Heap      : 0x00097d68 to 0x01ffffff, size 32160K (32932504 bytes)
History   : 132773 memory allocations, 130257 frees
Current   : 1268K (1298850 bytes) used in 2516 allocations
Actual    : approximately 1508K (1544784 bytes)
Overhead  : approximately 240K (245934 bytes), 15%
Debugdata : approximately 107K (110280 bytes) static, 92K (94628 bytes) dynamic
          : (debug data is in addition to other overheads)
Allocators:
malloc() 1009
new(nothrow) 788
new(nothrow)[] 451
calloc() 251
realloc() 17
Threads   :
        1 : handle 0x00075670, Idle Thread
        2 : handle 0x00093af8, main
        3 : handle 0x000739b0, thread_0
        4 : handle 0x00073a50, thread_1
        5 : handle 0x00073af0, thread_2
        6 : handle 0x00073b90, thread_3
Options   : actual_size enabled, time stamps enabled, thread info enabled
          : backtrace enabled, 1 levels
          : history enabled, 2048 records max
```

The fields in the output are as follows:

1. The start and end address of the heap and its size. This example is for a development board with a generous 32MB. Approximately 600K is used for application code and static data and for RedBoot, leaving most of the memory available for dynamic memory allocation.

2. Total numbers of past allocations and frees, with the difference corresponding to the number of current allocations. Note that the total size of past allocations is not recorded because of the likelihood of an overflow and hence misleading data.

3. Totals for the current allocations, giving the size as requested by application code.

4. The actual amount of memory used for these allocations, allowing for overhead. This information is only available if `CYGDBG_MEMALLOC_DEBUG_DEBUGDATA_ACTUAL_SIZE` is enabled. Note that the numbers are approximate: they only count per-allocation overhead; there may be additional costs for pool data structures and the like which are not included; usually these are sufficiently small that they can be ignored.

5. The difference between the above two fields. For this example the overhead is comparatively high. The configuration included support for debug guards which adds an extra 12 bytes to each allocation plus whatever was needed by the allocation code itself. Most of the allocations were small, so the guards have a disproportionate effect.

6. Additional memory needed for the debug data, both static and dynamic. The configuration included a history circular buffer with default settings, accounting for most of the static cost. The debug data for 2516 current allocations account for most of the dynamic costs, and is not included in the earlier figures. The results of `mallinfo()` will include the dynamic debug data.

7. Counts for the various types of dynamic memory allocation.

8. A list of the various threads: unique id, a cyg_handle_t handle, and the name passed to `cyg_thread_create`. This information is only available if `CYGDBG_MEMALLOC_DEBUG_DEBUGDATA_THREAD` is enabled. The ids can be used in a filter to show only allocations performed by the specified thread. The code only keeps track of threads involved with dynamic memory allocation, not every thread in the system. It is actually unlikely that the idle thread allocated any memory. Instead allocations during system initialization, before the scheduler was started, will usually be ascribed to the idle thread.

9. Details of the relevant configuration options. This can be useful when figuring out what filters, sort keys, or format specifiers are permitted, as an alternative to checking the configuration options.

Dumping Current Allocations

The **ecosmdd dump** can be used to analyze an `mddout` dump file and report on all current allocations.

```
$ ecosmdd dump consume mddout.0
0x00097d78 : malloc() 256 bytes, actual size 272 (+16), seqno 0, time 0
  By thread 1, 0x00075670 Idle Thread
  1) backtrace 0x0004da74 function Cyg_StdioStreamBuffer::set_buffer(unsigned, unsigned char*)
    /opt/ecos/packages/language/c/libc/stdio/current/src/common/streambuf.cxx:96
    "        malloced_buf = (cyg_uint8 * )malloc( size );"
0x000c0f50 : malloc() 13 bytes, actual size 32 (+19), seqno 229605, time 3960
  By thread 3, 0x000739b0 thread_0
  1) backtrace 0x00040ed8 function worker2()
    /tmp/mdd/consume.cxx:393
    "            allocs[index].data.c    = malloc(size);"
0x000c0f70 : new(nothrow) 1024 bytes, actual size 1040 (+16), seqno 251083, time 4329
```

```
  By thread 5, 0x00073af0 thread_2
  1) backtrace 0x00040c48 function worker1(int)
    /tmp/mdd/consume.cxx:315
    "            allocs[index].data.large    = new(std::nothrow) Large;"
…
```

`consume` is the executable. This output shows the first three allocation records, sorted in address order. The fields are as follows:

1. The address of the allocated chunk. This is the pointer that would be returned by e.g. `malloc()`. The memory allocation code may store some header information before this address, but that is transparent to the application. There is a big gap between the first and second records because the application freed a large buffer just before the dump file was generated.

2. The memory allocation function that was called to get this chunk. This may be a standard C library function or a C++ operator.

3. The allocation size requested by the application.

4. The actual allocation size and, in brackets, the overhead. This is provided only if `CYGDBG_MEMALLOC_DEBUG_DEBUGDATA_ACTUAL_SIZE` is enabled.

5. A sequence number. The first record shows the very first dynamic memory allocation in this test run, performed by the standard I/O initialization code. Sequence numbers are generated using a simple incrementing counter and are unique within a test run. The counter can overflow, but that is only likely to happen if an application makes very intensive use of `malloc()` and runs for several days.

6. A timestamp. This is a kernel cyg_tick_count_t as returned by the kernel function `cyg_current_time()`. Usually it corresponds to a counter running at 100Hz, so the second record is for a `malloc()` that occurred about 40 seconds into the run. Timestamps are only listed if `CYGDBG_MEMALLOC_DEBUG_DEBUGDATA_TIMESTAMP` is enabled.

7. A line of thread information showing the thread id, handle and name. This requires `CYGDBG_MEMALLOC_DEBUG_DEBUGDATA_THREAD`.

8. The level 1 backtrace. The first line gives the return address and the calling function. The second line gives a source code file name and line number. The third line shows the actual source line. In the third record the source code shows a C++ Large object being created. If enabled, additional levels of backtrace will follow.

   The function name is only available if the executable is specified on the command line. The file name and line number are only available if the executable contains *`-g`* debug information for the specified function. Usually this will be true for the application code itself and for eCos code, but not for other libraries supplied in binary format. The source line is only available if the file name and line number are known and the relevant file can be found on the current system. Again this may not be true for libraries supplied in binary format.

The executable does not have to be specified on the command line. Disassembling it can take considerable time, and serves only to provide more detailed backtrace information. Typical output without an executable would look like:

```
$ ecosmdd dump mddout.0 | more
0x00097d78 : malloc() 256 bytes, actual size 272 (+16), seqno 0, time 0
  By thread 1, 0x00075670 Idle Thread
  1) backtrace return address 0x0004da74
…
```

The **dump** subcommand accepts the standard options for architecture, ignoring certain files, sorting the output, applying filters, and formatting each record. For example to show only partial information for the allocations performed by thread 4 between approximately 40 and 42 seconds into the run, sorted by size with largest first, then by allocation time earliest first, the following can be used:

```
$ ecosmdd dump -Fthread=4 -Ftime_min=4000 -Ftime_max=4200 -SNs \
        -f '%p %a %n @ %T' mddout.0
0x002a43e8 malloc() 1553 @ 4079
0x000f9308 malloc() 1139 @ 4149
0x000c2a80 new(nothrow) 1024 @ 4104
```

```
0x00292428 new(nothrow)[] 388 @ 4194
0x000e0998 malloc() 240 @ 4147
0x00275678 new(nothrow) 128 @ 4013
0x00238c40 new(nothrow) 128 @ 4104
0x0023f7a8 new(nothrow) 128 @ 4194
0x000e1048 malloc() 18 @ 4014
0x0032cfe8 new(nothrow) 16 @ 4106
0x00131fa0 malloc() 8 @ 4107
0x0015d1a8 malloc() 8 @ 4125
0x002162d8 malloc() 7 @ 4129
0x001217c0 malloc() 7 @ 4190
```

The options should immediately follow the **dump** subcommand, before the executable or `mddout` file.

Showing the History

```
$ ecosmdd history consume mddout.0
Caution: history is incomplete.

malloc() 256 bytes: 0x00097d78 , actual size 272 (+16), seqno 0, time 0
  By thread 1, 0x00075670 Idle Thread
  1) backtrace 0x0004da74 function Cyg_StdioStreamBuffer::set_buffer(unsigned, unsigned char*)
    /opt/ecos/packages/language/c/libc/stdio/current/src/common/streambuf.cxx:96
    "        malloced_buf = (cyg_uint8 * )malloc( size );"
malloc() 131072 bytes: 0x00097e88 (freed) , actual size 131088 (+16), seqno 1, time 0
  By thread 2, 0x00093af8 main
  1) backtrace 0x000415b4 function main
    /tmp/mdd/consume.cxx:575
    "    spare   = malloc(128 * 1024);"
new(nothrow) 16 bytes: 0x00319270 (freed) , actual size 32 (+16), seqno 223425, time 3851
  By thread 3, 0x000739b0 thread_0
  1) backtrace 0x00040f4c function worker2()
    /tmp//consume.cxx:409
    "            allocs[index].data.small    = new(std::nothrow) Small;"
…
delete 16 bytes: 0x00156218 , actual size 40 (+24), seqno 258950, time 4461
  By thread 5, 0x000739b0 thread_0
  1) backtrace 0x00040cb8 function worker1(int)
    /tmp/mdd/consume.cxx:251
    "        break;"
free() 347 bytes: 0x001e6b08 , actual size 368 (+21), seqno 258951, time 4461
  By thread 6, 0x000739b0 thread_0
  1) backtrace 0x000409a4 function worker1(int)
    /tmp/mdd/consume.cxx:216
    "            free(allocs[index].data.c);"
…
```

Here **ecosmdd** has processed the executable and read in both the history data and the current allocation records from `mddout.0`. The file does not contain complete history information: there have been at least 258951 allocation and free operations, and the history buffer only stores the last 2048 frees. Each record is output in a similar format to **ecosmdd dump**. However history analysis is based around the order of events rather than the current state of the heap so the allocation function is shown before the heap.

The first record shows the first allocation in the system, and it is still allocated. Next comes the second allocation, which has been freed. This information will have come from the history circular buffer, implying that the buffer was freed in one of the last 2048 free operations. The third record shows another buffer that has been freed recently. There are no records between sequence numbers 1 and 223425, so all memory that has been allocated in the interval has already been freed and the relevant records are no longer in the history buffer.

The next two records show `delete` and `free()` operations. The format is essentially the same. The sequence number, timestamp, thread and backtrace information correspond to the free operation, not the allocation. Note that for the delete operation **ecosmdd** failed to get the source line number right: the `delete` invocation actually occurred a couple of lines earlier. Unfortunately the debug information in the executable was not sufficiently precise.

By default the history records will be shown earliest first. This order can be reversed with a `-r` option. **ecosmdd history** also accepts the standard options for architecture, ignoring certain files, applying filters, and formatting each record. The standard sort option is not supported because history implies sorting in time order. For example:

```
$ ecosmdd history -r -f '%a %p, %n bytes, seqno %s' consume mddout.0
Caution: history is incomplete.

free() 0x00097e88, 131072 bytes, seqno 263029
free() 0x00302490, 11 bytes, seqno 263028
new(nothrow) 0x00182cc0, 128 bytes, seqno 263027
…
```

If the desired history information is spread over more than one `mddout` file then they can all be passed to **ecosmdd history**. For example:

```
$ ecosmdd history -r consume mddout.0 mddout.1 mddout.2 mddout3
…
```

Options and the executable are handled as before. The `mddout` files should be listed in order of creation, and should correspond to a single test run. **ecosmdd** will extract both the history circular buffer and the current allocation data for the last file, but only the history buffers for the earlier ones - details of their current allocations can be found in later files. Obviously if eCos has been configured with `CYGDBG_MEMALLOC_DEBUG_DEBUGDATA_HISTORY` disabled then only the last file will contain useful information.

Comparing Two mddout Files

Sometimes, especially when tracking down a memory leak, it is useful to compare two dump files taken at different times and see what has changed. This functionality is provided by **ecosmdd diff**:

```
$ ecosmdd diff consume mddout.1 mddout.2
File mddout.1 : 1496K (1532048 bytes) used in 2543 allocations.
File mddout.2 : 1488K (1523769 bytes) used in 2483 allocations.
1331 new allocations in mddout.2 but not in mddout.1
1391 allocations in mddout.1 but freed in mddout.2

New allocations in mddout.2 but not in mddout.1
0x000ba8a8 : new(nothrow)[] 228 bytes, actual size 256 (+28), seqno 11001, time 214
  By thread 3, 0x000739b0 thread_0
  1) backtrace 0x00040fc0 function worker2()
    /tmp/mdd/consume.cxx:417
    "            allocs[index].data.smallv   = new(std::nothrow) Small[count];"
0x000ba9a8 : new(nothrow) 128 bytes, actual size 144 (+16), seqno 11528, time 222
  By thread 5, 0x00073af0 thread_2
  1) backtrace 0x00040b78 function worker1(int)
    /tmp/mdd/consume.cxx:296
    "            allocs[index].data.medium    = new(std::nothrow) Medium;"
…
Allocations in mddout.1 but freed in mddout.2
0x000ba9a8 : new(nothrow) 128 bytes, actual size 144 (+16), seqno 8213, time 162
  By thread 5, 0x00073af0 thread_2
  1) backtrace 0x00040b78 function worker1(int)
    /tmp/mdd/consume.cxx:296
    "            allocs[index].data.medium    = new(std::nothrow) Medium;"
0x000baa68 : new(nothrow) 128 bytes, actual size 144 (+16), seqno 9539, time 185
  By thread 6, 0x00073b90 thread_3
  1) backtrace 0x00041028 function worker2()
    /tmp/mdd/consume.cxx:424
    "            allocs[index].data.medium    = new(std::nothrow) Medium;"
…
```

The output begins with some statistics about the two dump files. Next comes a list of all memory chunks allocated in the second file but not in the first, and of all chunks allocated in the first but not the second. The diff uses the unique sequence number so will not be fooled if a chunk is freed and then allocated again.

ecosmdd diff accepts the standard options for architecture, ignoring certain files, sorting the output, applying filters, and formatting each record. Optionally these options can be followed by the executable, to get extended backtrace information. Finally there should be two `mddout` files:

```
$ ecosmdd diff -Fsize_min=10240 -f '%n bytes at %p by %f1' -SN \
        consume mddout.1 mddout.2
File mddout.1 : 1496K (1532048 bytes) used in 2543 allocations.
File mddout.2 : 1488K (1523769 bytes) used in 2483 allocations.
1331 new allocations in mddout.2 but not in mddout.1
1391 allocations in mddout.1 but freed in mddout.2

New allocations in mddout.2 but not in mddout.1
19691 bytes at 0x0025d498 by worker1(int)
19233 bytes at 0x00273758 by worker2()
…

Allocations in mddout.1 but freed in mddout.2
19858 bytes at 0x0025d498 by worker2()
18085 bytes at 0x001a2bf8 by worker2()
…
```

Standard Options

The various **ecosmdd** subcommands accept a number of standard options for specifying the architecture, ignoring certain source files, sorting and filtering the output, and formatting each record.

Specifying the Architecture

To provide extended backtrace information **ecosmdd** needs to disassemble the supplied executable. This involves running the appropriate **objdump** command, for example **arm-elf-objdump** or **m68k-elf-objdump**. **ecosmdd** reads in the executable's ELF header and uses this to work out the architecture. If it fails the architecture must instead be specified on the command line, for example:

```
$ ecosmdd dump -Adeepthought-elf …
```

ecosmdd will now try to run **deepthought-elf-objdump** to disassemble the executable.

Ignoring Selected Source Files

When the application involves extended use of header files with inline functions, the backtrace information can get even more confused than usual. Consider a function `tom()` which invokes an inline function `dick()` in a header file `<harry.h`, and `dick()` makes a memory allocation call. At run-time, because of the inlining the return address will be inside function `tom()`. However the debug information for the return address will usually specify the header file, not the source file containing `tom()`. This can make it much more difficult to interpret the backtrace.

There is no perfect solution to this problem, but **ecosmdd** contains an attempt at a partial solution. When disassembling an executable by default it will ignore any debug info where the file name matches the glob pattern `*/include/*`, if more accurate information for the current function is already available. This should catch inline functions in eCos, gcc and libstdc++ headers, and hence the backtrace output should more closely match what is actually happening in the application.

The default behaviour can be suppressed using the *-n* option, for example:

```
$ ecosmdd dump -n consume mddout.0
…
```

Alternatively a different glob pattern can be specified with the *-I* option (taking care to stop the shell from expanding the glob pattern prematurely):

```
$ ecosmdd dump -I\*.h consume mddout.0
```

Sorting the Output

By default the **dump** and **diff** will output their results sorted by increasing address. A different sort can be specified using the *-S* option, for example:

```
$ ecosmdd dump -SNs consume mddout.0
```

The *-S* should be followed by one or more sort keys. In the above example the primary sort key is `N`, specifying sort by decreasing allocation size so the largest allocations come first. When two allocations are the same size the secondary sort key (if specified) comes into play. Here the secondary key is `s`, meaning by increasing sequence number, so two allocations of the same size will be shown in history order. Any number of sort keys can be specified but it does not make sense to repeat a sort key or its inverse. Sequence numbers are unique so it also does not make sense to specify another sort key after `s` or `S`. If two allocations remain unsorted after all the specified sort keys have been processed then the output order is undefined. The available sort keys are:

p	Sort by increasing address, so the lowest address comes first.
P	Sort by decreasing address, so the highest address comes first.
n	Sort by increasing allocation size, so the smallest allocations come first.
N	Sort by decreasing allocation size, so the largest allocations come first.
s	Sort by increasing sequence numbers, so oldest allocations come first.
S	Sort by decreasing sequence number, so newest allocations come first.
a	Sort by memory allocation function, so for example all `realloc()` allocations will be grouped together.
t	Sort by increasing thread id. **ecosmdd stats** can be used to get details of the various threads. This sort key is only available if `CYGDBG_MEMALLOC_DEBUG_DEBUGDATA_THREAD` is enabled.
T	Sort by decreasing thread id. **ecosmdd stats** can be used to get details of the various threads. This sort key is only available if `CYGDBG_MEMALLOC_DEBUG_DEBUGDATA_THREAD` is enabled.

Filtering out Unwanted Data

Non-trivial applications can result in very large amounts of memory debug data. **ecosmdd** provides a number of filters to eliminate unwanted data. For example, to show only allocations of 1K or larger:

```
$ ecosmdd dump -Fsize_min=1024 consume mddout.0
…
```

A filter takes the form *-F<key>=<value>*. The supported keys are:

thread=<id>	Only show allocations performed by the specified thread. This filter can only be used if `CYGDBG_MEMALLOC_DEBUG_DEBUGDATA_THREAD` is enabled.
size_min=<size>	Ignore any allocations smaller than the specified size.
size_max=<size>	Ignore any allocations larger than the specified size.
seqno_min=<start>	Only show the event identified by the sequence number and subsequent ones.
seqno_max=<end>	Only show events up to and including the one identified by the sequence number.
time_min=<start>	Discard any records prior to the specified time.
time_max=<end>	Discard any records after the specified time.
ptr_min=<base>	Filter out allocations before the specified address.
ptr_max=<limit>	Filter out allocations after the specified address.

Multiple filters can be specified. For example to show only allocations performed by thread 6 which are larger than 4K and which occurred in a certain time interval:

```
$ ecosmdd dump -Fthread=6 -Fsize_min=4096 -Ftime_min=4000 -Ftime_max=5000 \
    consume mddout.0
```

Formatting the Output

By default **ecosmdd** outputs all available information for each record. Sometimes it is better to see only some of the fields. At other times a different format may be preferred, for example to feed the **ecosmdd** output into some other tool. Hence it is possible to specify a custom format string, along similar lines to the C `strftime` and `printf` functions:

```
$ ecosmdd dump -f '%a for %n bytes -> %p'
malloc() for 256 bytes -> 0x00097d78
malloc() for 131072 bytes -> 0x00097e88
malloc() for 4 bytes -> 0x000b7e98
calloc() for 3724 bytes -> 0x000b7eb0
…
```

A % character introduces a conversion sequence. Other characters are just passed straight through. The supported conversion sequences are:

%%	A single % character.
%p	The address of the allocated chunk.
%n	The requested allocation size.
%m	The actual allocation size. This requires `CYGDBG_MEMALLOC_DEBUG_DEBUGDATA_ACTUAL_SIZE`.
%o	The allocation overhead for this chunk. This requires `CYGDBG_MEMALLOC_DEBUG_DEBUGDATA_ACTUAL_SIZE`.
%s	The sequence number.
%a	The allocating function, for example `malloc()`
%T	A timestamp for the event. This requires `CYGDBG_MEMALLOC_DEBUG_DEBUGDATA_TIMESTAMP`
%t	The thread identifier. This requires `CYGDBG_MEMALLOC_DEBUG_DEBUGDATA_THREAD`
%h	The thread handle. This requires `CYGDBG_MEMALLOC_DEBUG_DEBUGDATA_THREAD`
%N	The thread name. This requires `CYGDBG_MEMALLOC_DEBUG_DEBUGDATA_THREAD`
%b1 to %b8	The backtrace return address for the appropriate level. It is an error to specify a level greater than what is actually present in the `mddout` file.
%f1 to %f8	The backtrace function name for the appropriate level. This can only be used if the executable has been specified on the command line.
%w1 to %w8	The backtrace location for the appropriate level, in the form filename:linenumber. This can only be used if the executable has been specified on the command line, and even then the information is not always available.
%l1 to %l8	The backtrace source line for the appropriate level. This can only be used if the executable has been specified on the command line, and even then the information is not always available.

The usual format string for a dump operation, assuming default configuration settings, is: '%p : %a %n bytes, %m (+%o), seqno %s, time %T\n By thread %t, %h %N\n 1) backtrace %b1 function %f1\n %w1\n \"%l1\"'

Part VI. I/O Package (Device Drivers)

Table of Contents

Chapter 15. Introduction

The I/O package is designed as a general purpose framework for supporting device drivers. This includes all classes of drivers from simple serial to networking stacks and beyond.

Components of the I/O package, such as device drivers, are configured into the system just like all other components. Additionally, end users may add their own drivers to this set.

While the set of drivers (and the devices they represent) may be considered static, they must be accessed via an opaque “handle”. Each device in the system has a unique name and the `cyg_io_lookup()` function is used to map that name onto the handle for the device. This “hiding” of the device implementation allows for generic, named devices, as well as more flexibility. Also, the `cyg_io_lookup()` function provides drivers the opportunity to initialize the device when usage actually starts.

All devices have a name. The standard provided devices use names such as `“/dev/console”` and `“/dev/serial0”`, where the `“/dev/”` prefix indicates that this is the name of a device.

The entire I/O package API, as well as the standard set of provided drivers, is written in C.

Basic functions are provided to send data to and receive data from a device. The details of how this is done is left to the device [class] itself. For example, writing data to a block device like a disk drive may have different semantics than writing to a serial port.

Additional functions are provided to manipulate the state of the driver and/or the actual device. These functions are, by design, quite specific to the actual driver.

This driver model supports layering; in other words, a device may actually be created “on top of” another device. For example, the “tty” (terminal-like) devices are built on top of simple serial devices. The upper layer then has the flexibility to add features and functions not found at the lower layers. In this case the “tty” device provides for line buffering and editing not available from the simple serial drivers.

Some drivers will support visibility of the layers they depend upon. The “tty” driver allows information about the actual serial device to be manipulated by passing get/set config calls that use a serial driver “key” down to the serial driver itself.

Chapter 16. User API

All functions, except `cyg_io_lookup()` require an I/O "handle".

All functions return a value of the type Cyg_ErrNo. If an error condition is detected, this value will be negative and the absolute value indicates the actual error, as specified in `cyg/error/codes.h`. The only other legal return value will be `ENOERR`. All other function arguments are pointers (references). This allows the drivers to pass information efficiently, both into and out of the driver. The most striking example of this is the "length" value passed to the read and write functions. This parameter contains the desired length of data on input to the function and the actual transferred length on return.

```
// Lookup a device and return its handle
Cyg_ErrNo cyg_io_lookup(
    const char      *name,
    cyg_io_handle_t *handle )
```

This function maps a device name onto an appropriate handle. If the named device is not in the system, then the error `-ENOENT` is returned. If the device is found, then the handle for the device is returned by way of the handle pointer *`*handle`*.

```
// Write data to a device
Cyg_ErrNo cyg_io_write(
    cyg_io_handle_t handle,
    const void      *buf,
    cyg_uint32      *len )
```

This function sends data to a device. The size of data to send is contained in *`*len`* and the actual size sent will be returned in the same place.

```
// Read data from a device
Cyg_ErrNo  cyg_io_read(
    cyg_io_handle_t handle,
    void            *buf,
    cyg_uint32      *len )
```

This function receives data from a device. The desired size of data to receive is contained in *`*len`* and the actual size obtained will be returned in the same place.

```
// Get the configuration of a device
Cyg_ErrNo cyg_io_get_config(
    cyg_io_handle_t handle,
    cyg_uint32      key,
    void *          buf,
    cyg_uint32 *    len )
```

This function is used to obtain run-time configuration about a device. The type of information retrieved is specified by the *`key`*. The data will be returned in the given buffer. The value of *`*len`* should contain the amount of data requested, which must be at least as large as the size appropriate to the selected key. The actual size of data retrieved is placed in *`*len`*. The appropriate key values differ for each driver and are all listed in the file `<cyg/io/config_keys.h>`.

```
// Change the configuration of a device
Cyg_ErrNo  cyg_io_set_config(
    cyg_io_handle_t handle,
    cyg_uint32      key,
    const void      *buf,
    cyg_uint32      *len )
```

This function is used to manipulate or change the run-time configuration of a device. The type of information is specified by the *`key`*. The data will be obtained from the given buffer. The value of *`*len`* should contain the amount of data provided, which must match the size appropriate to the selected key. The appropriate key values differ for each driver and are all listed in the file `<cyg/io/config_keys.h>`.

Chapter 17. Serial driver details

Two different classes of serial drivers are provided as a standard part of the eCos system. These are described as “raw serial” (serial) and “tty-like” (tty).

Raw Serial Driver

Use the include file `<cyg/io/serialio.h>` for this driver.

The raw serial driver is capable of sending and receiving blocks of raw data to a serial device. Controls are provided to configure the actual hardware, but there is no manipulation of the data by this driver.

There may be many instances of this driver in a given system, one for each serial channel. Each channel corresponds to a physical device and there will typically be a device module created for this purpose. The device modules themselves are configurable, allowing specification of the actual hardware details, as well as such details as whether the channel should be buffered by the serial driver, etc.

Runtime Configuration

Runtime configuration is achieved by exchanging data structures with the driver via the `cyg_io_set_config()` and `cyg_io_get_config()` functions.

```
typedef struct {
    cyg_serial_baud_rate_t baud;
    cyg_serial_stop_bits_t stop;
    cyg_serial_parity_t parity;
    cyg_serial_word_length_t word_length;
    cyg_uint32 flags;
} cyg_serial_info_t;
```

The field *`word_length`* contains the number of data bits per word (character). This must be one of the values:

```
CYGNUM_SERIAL_WORD_LENGTH_5
CYGNUM_SERIAL_WORD_LENGTH_6
CYGNUM_SERIAL_WORD_LENGTH_7
CYGNUM_SERIAL_WORD_LENGTH_8
```

The field *`baud`* contains a baud rate selection. If the configuration does not implement the `CYGINT_IO_SERIAL_BAUD_ARBITRARY` interface support for arbitrary baud rate values then this field must be one of the values:

```
CYGNUM_SERIAL_BAUD_50
CYGNUM_SERIAL_BAUD_75
CYGNUM_SERIAL_BAUD_110
CYGNUM_SERIAL_BAUD_134_5
CYGNUM_SERIAL_BAUD_150
CYGNUM_SERIAL_BAUD_200
CYGNUM_SERIAL_BAUD_300
CYGNUM_SERIAL_BAUD_600
CYGNUM_SERIAL_BAUD_1200
CYGNUM_SERIAL_BAUD_1800
CYGNUM_SERIAL_BAUD_2400
CYGNUM_SERIAL_BAUD_3600
CYGNUM_SERIAL_BAUD_4800
CYGNUM_SERIAL_BAUD_7200
CYGNUM_SERIAL_BAUD_9600
CYGNUM_SERIAL_BAUD_14400
CYGNUM_SERIAL_BAUD_19200
CYGNUM_SERIAL_BAUD_38400
CYGNUM_SERIAL_BAUD_57600
```

```
CYGNUM_SERIAL_BAUD_115200
CYGNUM_SERIAL_BAUD_234000
```

For configurations where `CYGINT_IO_SERIAL_BAUD_ARBITRARY` is enabled then the manifests above define the respective baud rate, but the underlying device driver is capable of accepting arbitrary baud rate values as required. e.g. `76800`.

The field *`stop`* contains the number of stop bits. This must be one of the values:

```
CYGNUM_SERIAL_STOP_1
CYGNUM_SERIAL_STOP_1_5
CYGNUM_SERIAL_STOP_2
```

Note

On most hardware, a selection of 1.5 stop bits is only valid if the word (character) length is 5.

The field *`parity`* contains the parity mode. This must be one of the values:

```
CYGNUM_SERIAL_PARITY_NONE
CYGNUM_SERIAL_PARITY_EVEN
CYGNUM_SERIAL_PARITY_ODD
CYGNUM_SERIAL_PARITY_MARK
CYGNUM_SERIAL_PARITY_SPACE
```

The field *`flags`* is a bitmask which controls the behavior of the serial device driver. It should be built from the values `CYG_SERIAL_FLAGS_xxx` defined below:

```
#define CYG_SERIAL_FLAGS_RTSCTS 0x0001
```

If this bit is set then the port is placed in “hardware handshake” mode. In this mode, the CTS and RTS pins control when data is allowed to be sent/received at the port. This bit is ignored if the hardware does not support this level of handshake.

```
typedef struct {
    cyg_int32 rx_bufsize;
    cyg_int32 rx_count;
    cyg_int32 tx_bufsize;
    cyg_int32 tx_count;
} cyg_serial_buf_info_t;
```

The field *`rx_bufsize`* contains the total size of the incoming data buffer. This is set to zero on devices that do not support buffering (i.e. polled devices).

The field *`rx_count`* contains the number of bytes currently occupied in the incoming data buffer. This is set to zero on devices that do not support buffering (i.e. polled devices).

The field *`tx_bufsize`* contains the total size of the transmit data buffer. This is set to zero on devices that do not support buffering (i.e. polled devices).

The field *`tx_count`* contains the number of bytes currently occupied in the transmit data buffer. This is set to zero on devices that do not support buffering (i.e. polled devices).

API Details

cyg_io_write

```
cyg_io_write(handle, buf, len)
```

Send the data from *`buf`* to the device. The driver maintains a buffer to hold the data. The size of the intermediate buffer is configurable within the interface module. The data is not modified at all while it is being buffered. On return, *`*len`* contains the amount of characters actually consumed .

It is possible to configure the write call to be blocking (default) or non-blocking. Non-blocking mode requires both the configuration option `CYGOPT_IO_SERIAL_SUPPORT_NONBLOCKING` to be enabled, and the specific device to be set to non-blocking mode for writes (see `cyg_io_set_config()`).

In blocking mode, the call will not return until there is space in the buffer and the entire contents of *`buf`* have been consumed.

In non-blocking mode, as much as possible gets consumed from *`buf`*. If everything was consumed, the call returns `ENOERR`. If only part of the *`buf`* contents was consumed, `-EAGAIN` is returned and the caller must try again. On return, *`*len`* contains the number of characters actually consumed .

The call can also return `-EINTR` if interrupted via the `cyg_io_get_config()`/`ABORT` key.

cyg_io_read

```
cyg_io_read(handle, buf, len)
```

Receive data into the buffer, *`buf`*, from the device. No manipulation of the data is performed before being transferred. An interrupt driven interface module will support data arriving when no read is pending by buffering the data in the serial driver. Again, this buffering is completely configurable. On return, *`*len`* contains the number of characters actually received.

It is possible to configure the read call to be blocking (default) or non-blocking. Non-blocking mode requires both the configuration option `CYGOPT_IO_SERIAL_SUPPORT_NONBLOCKING` to be enabled, and the specific device to be set to non-blocking mode for reads (see `cyg_io_set_config()`).

In blocking mode, the call will not return until the requested amount of data has been read.

In non-blocking mode, data waiting in the device buffer is copied to *`buf`*, and the call returns immediately. If there was enough data in the buffer to fulfill the request, `ENOERR` is returned. If only part of the request could be fulfilled, `-EAGAIN` is returned and the caller must try again. On return, *`*len`* contains the number of characters actually received.

The call can also return `-EINTR` if interrupted via the `cyg_io_get_config()`/`ABORT` key.

cyg_io_get_config

```
cyg_io_get_config(handle, key, buf, len)
```

This function returns current [runtime] information about the device and/or driver.

`CYG_IO_GET_CONFIG_SERIAL_INFO`

Buf type: cyg_serial_info_t

Function: This function retrieves the current state of the driver and hardware. This information contains fields for hardware baud rate, number of stop bits, and parity mode. It also includes a set of flags that control the port, such as hardware flow control.

`CYG_IO_GET_CONFIG_SERIAL_BUFFER_INFO`

Buf type: cyg_serial_buf_info_t

Function: This function retrieves the current state of the software buffers in the serial drivers. For both receive and transmit buffers it returns the total buffer size and the current number of bytes occupied in the buffer. It does not take into account any buffering such as FIFOs or holding registers that the serial device itself may have.

`CYG_IO_GET_CONFIG_SERIAL_OUTPUT_DRAIN`

Buf type: void *

Function: This function waits for any buffered output to complete. This function only completes when there is no more data remaining to be sent to the device.

CYG_IO_GET_CONFIG_SERIAL_OUTPUT_FLUSH

Buf type: void *

Function: This function discards any buffered output for the device.

CYG_IO_GET_CONFIG_SERIAL_INPUT_DRAIN

Buf type: void *

Function: This function discards any buffered input for the device.

CYG_IO_GET_CONFIG_SERIAL_ABORT

Buf type: void*

Function: This function will cause any pending read or write calls on this device to return with -EABORT.

CYG_IO_GET_CONFIG_SERIAL_READ_BLOCKING

Buf type: cyg_uint32 (values 0 or 1)

Function: This function will read back the blocking-mode setting for read calls on this device. This call is only available if the configuration option CYGOPT_IO_SERIAL_SUPPORT_NONBLOCKING is enabled.

CYG_IO_GET_CONFIG_SERIAL_WRITE_BLOCKING

Buf type: cyg_uint32 (values 0 or 1)

Function: This function will read back the blocking-mode setting for write calls on this device. This call is only available if the configuration option CYGOPT_IO_SERIAL_SUPPORT_NONBLOCKING is enabled.

cyg_io_set_config

```
cyg_io_set_config(handle, key, buf,len)
```

This function is used to update or change runtime configuration of a port.

CYG_IO_SET_CONFIG_SERIAL_INFO

Buf type: cyg_serial_info_t

Function: This function updates the information for the driver and hardware. The information contains fields for hardware baud rate, number of stop bits, and parity mode. It also includes a set of flags that control the port, such as hardware flow control.

CYG_IO_SET_CONFIG_SERIAL_READ_BLOCKING

Buf type: cyg_uint32 (values 0 or 1)

Function: This function will set the blocking-mode for read calls on this device. This call is only available if the configuration option CYGOPT_IO_SERIAL_SUPPORT_NONBLOCKING is enabled.

CYG_IO_SET_CONFIG_SERIAL_WRITE_BLOCKING

Buf type: cyg_uint32 (values 0 or 1)

Function: This function will set the blocking-mode for write calls on this device. This call is only available if the configuration option `CYGOPT_IO_SERIAL_SUPPORT_NONBLOCKING` is enabled.

TTY driver

Use the include file `<cyg/io/ttyio.h>` for this driver.

This driver is built on top of the simple serial driver and is typically used for a device that interfaces with humans such as a terminal. It provides some minimal formatting of data on output and allows for line-oriented editing on input.

Runtime configuration

Runtime configuration is achieved by exchanging data structures with the driver via the `cyg_io_set_config()` and `cyg_io_get_config()` functions.

```
typedef struct {
    cyg_uint32 tty_out_flags;
    cyg_uint32 tty_in_flags;
} cyg_tty_info_t;
```

The field *`tty_out_flags`* is used to control what happens to data as it is send to the serial port. It contains a bitmap comprised of the bits as defined by the `CYG_TTY_OUT_FLAGS_xxx` values below.

```
#define CYG_TTY_OUT_FLAGS_CRLF 0x0001 // Map '\n' => '\r\n' on output
```

If this bit is set in *`tty_out_flags`*, any occurrence of the character "\n" will be replaced by the sequence "\r\n" before being sent to the device.

The field *`tty_in_flags`* is used to control how data is handled as it comes from the serial port. It contains a bitmap comprised of the bits as defined by the `CYG_TTY_IN_FLAGS_xxx` values below.

```
#define CYG_TTY_IN_FLAGS_CR 0x0001 // Map '\r' => '\n' on input
```

If this bit is set in *`tty_in_flags`*, the character "\r" (“return” or “enter” on most keyboards) will be mapped to "\n".

```
#define CYG_TTY_IN_FLAGS_CRLF 0x0002 // Map '\r\n' => '\n' on input
```

If this bit is set in *`tty_in_flags`*, the character sequence "\r\n" (often sent by DOS/Windows based terminals) will be mapped to "\n".

```
#define CYG_TTY_IN_FLAGS_ECHO 0x0004 // Echo characters as processed
```

If this bit is set in *`tty_in_flags`*, characters will be echoed back to the serial port as they are processed.

```
#define CYG_TTY_IN_FLAGS_BINARY 0x0008 // No input processing
```

If this bit is set in *`tty_in_flags`*, the input will not be manipulated in any way before being placed in the user,s buffer.

API details

```
cyg_io_read(handle, buf, len)
```

This function is used to read data from the device. In the default case, data is read until an end-of-line character ("\n" or "\r") is read. Additionally, the characters are echoed back to the [terminal] device. Minimal editing of the input is also supported.

Note

When connecting to a remote target via GDB it is not possible to provide console input while GDB is connected. The GDB remote protocol does not support input. Users must disconnect from GDB if this functionality is required.

```
cyg_io_write(handle, buf, len)
```

This function is used to send data to the device. In the default case, the end-of-line character "\n" is replaced by the sequence "\r\n".

```
cyg_io_get_config(handle, key, buf, len)
```

This function is used to get information about the channel,s configuration at runtime.

`CYG_IO_GET_CONFIG_TTY_INFO`

Buf type:	cyg_tty_info_t
Function:	This function retrieves the current state of the driver.

Serial driver keys (see above) may also be specified in which case the call is passed directly to the serial driver.

```
cyg_io_set_config(handle, key, buf, len)
```

This function is used to modify the channel,s configuration at runtime.

`CYG_IO_SET_CONFIG_TTY_INFO`

Buf type:	cyg_tty_info_t
Function:	This function changes the current state of the driver.

Serial driver keys (see above) may also be specified in which case the call is passed directly to the serial driver.

Chapter 18. How to Write a Driver

A device driver is nothing more than a named entity that supports the basic I/O functions - read, write, get config, and set config. Typically a device driver also uses and manages interrupts from the device. While the interface is generic and device driver independent, the actual driver implementation is completely up to the device driver designer.

That said, the reason for using a device driver is to provide access to a device from application code in as general purpose a fashion as reasonable. Most driver writers are also concerned with making this access as simple as possible while being as efficient as possible.

Most device drivers are concerned with the movement of information, for example data bytes along a serial interface, or packets in a network. In order to make the most efficient use of system resources, interrupts are used. This will allow other application processing to take place while the data transfers are under way, with interrupts used to indicate when various events have occurred. For example, a serial port typically generates an interrupt after a character has been sent “down the wire” and the interface is ready for another. It makes sense to allow further application processing while the data is being sent since this can take quite a long time. The interrupt can be used to allow the driver to send a character as soon as the current one is complete, without any active participation by the application code.

The main building blocks for device drivers are found in the include file: `<cyg/io/devtab.h>`

All device drivers in *eCos* are described by a device table entry, using the cyg_devtab_entry_t type. The entry should be created using the `DEVTAB_ENTRY()` macro, like this:

```
DEVTAB_ENTRY(l, name, dep_name, handlers, init, lookup, priv)
```

Arguments

`l` The "C" label for this device table entry.

`name` The "C" string name for the device.

`dep_name` For a layered device, the "C" string name of the device this device is built upon.

`handlers` A pointer to the I/O function "handlers" (see below).

`init` A function called when eCos is initialized. This function can query the device, setup hardware, etc.

`lookup` A function called when `cyg_io_lookup()` is called for this device.

`priv` A placeholder for any device specific data required by the driver.

The interface to the driver is through the *`handlers`* field. This is a pointer to a set of functions which implement the various `cyg_io_XXX()` routines. This table is defined by the macro:

```
DEVIO_TABLE(l, write, read, get_config, set_config)
```

Arguments

`l` The "C" label for this table of handlers.

write The function called as a result of `cyg_io_write()`.

read The function called as a result of `cyg_io_read()`.

get_config The function called as a result of `cyg_io_get_config()`.

set_config The function called as a result of `cyg_io_set_config()`.

When *eCos* is initialized (sometimes called “boot” time), the `init()` function is called for all devices in the system. The `init()` function is allowed to return an error in which case the device will be placed “off line” and all I/O requests to that device will be considered in error.

The `lookup()` function is called whenever the `cyg_io_lookup()` function is called with this device name. The lookup function may cause the device to come “on line” which would then allow I/O operations to proceed. Future versions of the I/O system will allow for other states, including power saving modes, etc.

How to Write a Serial Hardware Interface Driver

The standard serial driver supplied with *eCos* is structured as a hardware independent portion and a hardware dependent interface module. To add support for a new serial port, the user should be able to use the existing hardware independent portion and just add their own interface driver which handles the details of the actual device. The user should have no need to change the hardware independent portion.

The interfaces used by the serial driver and serial implementation modules are contained in the file `<cyg/io/serial.h>`

Note

In the sections below we use the notation <<xx>> to mean a module specific value, referred to as “xx” below.

DevTab Entry

The interface module contains the devtab entry (or entries if a single module supports more than one interface). This entry should have the form:

```
DEVTAB_ENTRY(<<module_name>>,
    <<device_name>>,
    0,
    &serial_devio,
    <<module_init>>,
    <<module_lookup>>,
    &<<serial_channel>>
);
```

Arguments

module_name
The "C" label for this devtab entry

device_name
The "C" string for the device. E.g. `/dev/serial0`.

serial_devio
The table of I/O functions. This set is defined in the hardware independent serial driver and should be used.

module_init
The module initialization function.

module_lookup
The device lookup function. This function typically sets up the device for actual use, turning on interrupts, configuring the port, etc.

serial_channel
This table (defined below) contains the interface between the interface module and the serial driver proper.

Serial Channel Structure

Each serial device must have a “serial channel”. This is a set of data which describes all operations on the device. It also contains buffers, etc., if the device is to be buffered. The serial channel is created by the macro:

```
SERIAL_CHANNEL_USING_INTERRUPTS(l, funs, dev_priv, baud,stop, parity, word_length,
    flags, out_buf, out_buflen, in_buf, in_buflen)
```

Arguments

l
The "C" label for this structure.

funs
The set of interface functions (see below).

dev_priv
A placeholder for any device specific data for this channel.

baud
The initial baud rate value (cyg_serial_baud_t).

stop
The initial stop bits value (cyg_serial_stop_bits_t).

parity
The initial parity mode value (cyg_serial_parity_t).

word_length
The initial word length value (cyg_serial_word_length_t).

flags
The initial driver flags value.

out_buf
Pointer to the output buffer. `NULL` if none required.

out_buflen
The length of the output buffer.

in_buf
pointer to the input buffer. `NULL` if none required.

in_buflen
The length of the input buffer.

If either buffer length is zero, no buffering will take place in that direction and only polled mode functions will be used.

The interface from the hardware independent driver into the hardware interface module is contained in the *funs* table. This is defined by the macro:

Serial Functions Structure

```
SERIAL_FUNS(l, putc, getc, set_config, start_xmit, stop_xmit)
```

Arguments

l
The "C" label for this structure.

putc
`bool (*putc)(serial_channel *priv, unsigned char c)`

This function sends one character to the interface. It should return `true` if the character is actually consumed. It should return `false` if there is no space in the interface

getc
`unsigned char (*getc)(serial_channel *priv)`

This function fetches one character from the interface. It will be only called in a non-interrupt driven mode, thus it should wait for a character by polling the device until ready.

set_config
`bool (*set_config)(serial_channel *priv,cyg_serial_info_t *config)`

This function is used to configure the port. It should return `true` if the hardware is updated to match the desired configuration. It should return `false` if the port cannot support some parameter specified by the given configuration. E.g. selecting 1.5 stop bits and 8 data bits is invalid for most serial devices and should not be allowed.

start_xmit

`void (*start_xmit)(serial_channel *priv)`

In interrupt mode, turn on the transmitter and allow for transmit interrupts.

stop_xmit

`void (*stop_xmit)(serial_channel *priv)`

In interrupt mode, turn off the transmitter.

Callbacks

The device interface module can execute functions in the hardware independent driver via `chan->callbacks`. These functions are available:

```
void (*serial_init)( serial_channel *chan )
```

This function is used to initialize the serial channel. It is only required if the channel is being used in interrupt mode.

```
void (*xmt_char)( serial_channel *chan )
```

This function would be called from an interrupt handler after a transmit interrupt indicating that additional characters may be sent. The upper driver will call the `putc` function as appropriate to send more data to the device.

```
void (*rcv_char)( serial_channel *chan, unsigned char c )
```

This function is used to tell the driver that a character has arrived at the interface. This function is typically called from the interrupt handler.

Furthermore, if the device has a FIFO it should require the hardware independent driver to provide block transfer functionality (driver CDL should include "implements CYGINT_IO_SERIAL_BLOCK_TRANSFER"). In that case, the following functions are available as well:

```
bool (*data_xmt_req)( serial_channel  *chan,
                      int             space,
                      int*            chars_avail,
                      unsigned char** chars)
void (*data_xmt_done)(serial_channel *chan)
```

Instead of calling `xmt_char()` to get a single character for transmission at a time, the driver should call `data_xmt_req()` in a loop, requesting character blocks for transfer. Call with a *`space`* argument of how much space there is available in the FIFO.

If the call returns `true`, the driver can read *`chars_avail`* characters from *`chars`* and copy them into the FIFO.

If the call returns `false`, there are no more buffered characters and the driver should continue without filling up the FIFO.

When all data has been unloaded, the driver must call `data_xmt_done()`.

```
bool (*data_rcv_req)( serial_channel  *chan,
                      int             avail,
                      int             *space_avail,
                      unsigned char** space)
void (*data_rcv_done)(serial_channel *chan)
```

Instead of calling `rcv_char()` with a single character at a time, the driver should call `data_rcv_req()` in a loop, requesting space to unload the FIFO to. *`avail`* is the number of characters the driver wishes to unload.

If the call returns `true`, the driver can copy *`space_avail`* characters to *`space`*.

If the call returns `false`, the input buffer is full. It is up to the driver to decide what to do in that case (callback functions for registering overflow are being planned for later versions of the serial driver).

When all data has been unloaded, the driver must call `data_rcv_done()`.

Serial testing with ser_filter

Rationale

Since some targets only have one serial connection, a serial testing harness needs to be able to share the connection with GDB (however, the test and GDB can also run on separate lines).

The *serial filter* (ser_filter) sits between the serial port and GDB and monitors the exchange of data between GDB and the target. Normally, no changes are made to the data.

When a test request packet is sent from the test on the target, it is intercepted by the filter.

The filter and target then enter a loop, exchanging protocol data between them which GDB never sees.

In the event of a timeout, or a crash on the target, the filter falls back into its pass-through mode. If this happens due to a crash it should be possible to start regular debugging with GDB. The filter will stay in the pass-though mode until GDB disconnects.

The Protocol

The protocol commands are prefixed with an `"@"` character which the serial filter is looking for. The protocol commands include:

`PING`

> Allows the test on the target to probe for the filter. The filter responds with `OK`, while GDB would just ignore the command. This allows the tests to do nothing if they require the filter and it is not present.

`CONFIG`

> Requests a change of serial line configuration. Arguments to the command specify baud rate, data bits, stop bits, and parity. [This command is not fully implemented yet - there is no attempt made to recover if the new configuration turns out to cause loss of data.]

`BINARY`

> Requests data to be sent from the filter to the target. The data is checksummed, allowing errors in the transfer to be detected. Sub-options of this command control how the data transfer is made:
>
> `NO_ECHO`
>
> > (serial driver receive test) Just send data from the filter to the target. The test verifies the checksum and PASS/FAIL depending on the result.
>
> `EOP_ECHO`
>
> > (serial driver half-duplex receive and send test) As `NO_ECHO` but the test echoes back the data to the filter. The filter does a checksum on the received data and sends the result to the target. The test PASS/FAIL depending on the result of both checksum verifications.
>
> `DUPLEX_ECHO`
>
> > (serial driver duplex receive and send test) Smaller packets of data are sent back and forth in a pattern that ensures that the serial driver will be both sending and receiving at the same time. Again, checksums are computed and verified resulting in PASS/FAIL.

`TEXT`

This is a test of the text translations in the TTY layer. Requests a transfer of text data from the target to the filter and possibly back again. The filter treats this as a binary transfer, while the target ma be doing translations on the data. The target provides the filter with checksums for what it should expect to see. This test is not implemented yet.

The above commands may be extended, and new commands added, as required to test (new) parts of the serial drivers in eCos.

The Serial Tests

The serial tests are built as any other eCos test. After running the **make tests** command, the tests can be found in `install/tests/io_serial/`

`serial1`

A simple API test.

`serial2`

A simple serial send test. It writes out two strings, one raw and one encoded as a GDB O-packet

`serial3` [requires the serial filter]

This tests the half-duplex send and receive capabilities of the serial driver.

`serial4` [requires the serial filter]

This test attempts to use a few different serial configurations, testing the driver's configuration/setup functionality.

`serial5` [requires the serial filter]

This tests the duplex send and receive capabilities of the serial driver.

All tests should complete in less than 30 seconds.

Serial Filter Usage

Running the ser_filter program with no (or wrong) arguments results in the following output:

```
Usage: ser_filter [-t -S] TcpIPport SerialPort BaudRate
    or: ser_filter -n [-t -S] SerialPort BaudRate

  -t: Enable tracing.
  -S: Output data read from serial line.
  -c: Output data on console instead of via GDB.
  -n: No GDB.
```

The normal way to use it with GDB is to start the filter:

```
$ ser_filter -t 9000 com1 38400
```

In this case, the filter will be listening on port 9000 and connect to the target via the serial port `COM1` at 38400 baud. On a UNIX host, replace "`COM1`" with a device such as "`/dev/ttyS0`".

The `-t` option enables tracing which will cause the filter to describe its actions on the console.

Now start GDB with one of the tests as an argument:

```
$ mips-tx39-elf-gdb -nw install/tests/io_serial/serial3
```

Then connect to the filter:

```
(gdb) target remote localhost:9000
```

This should result in a connection in exactly the same way as if you had connected directly to the target on the serial line.

```
(gdb) c
```

Which should result in output similar to the below:

```
Continuing.
INFO:<BINARY:16:1!>
PASS:<Binary test completed>
INFO:<BINARY:128:1!>
PASS:<Binary test completed>
INFO:<BINARY:256:1!>
PASS:<Binary test completed>
INFO:<BINARY:1024:1!>
PASS:<Binary test completed>
INFO:<BINARY:512:0!>
PASS:<Binary test completed>
…
PASS:<Binary test completed>
INFO:<BINARY:16384:0!>
PASS:<Binary test completed>
PASS:<serial13 test OK>
EXIT:<done>
```

If any of the individual tests fail the testing will terminate with a `FAIL`.

With tracing enabled, you would also see the filter's status output:

The `PING` command sent from the target to determine the presence of the filter:

```
[400 11:35:16] Dispatching command PING
[400 11:35:16] Responding with status OK
```

Each of the binary commands result in output similar to:

```
[400 11:35:16] Dispatching command BINARY
[400 11:35:16] Binary data (Size:16, Flags:1).
[400 11:35:16] Sending CRC: '170231!', len: 7.
[400 11:35:16] Reading 16 bytes from target.
[400 11:35:16] Done. in_crc 170231, out_crc 170231.
[400 11:35:16] Responding with status OK
[400 11:35:16] Received DONE from target.
```

This tracing output is normally sent as O-packets to GDB which will display the tracing text. By using the `-c` option, the tracing text can be redirected to the console from which ser_filter was started.

A Note on Failures

A serial connection (especially when driven at a high baud rate) can garble the transmitted data because of noise from the environment. It is not the job of the serial driver to ensure data integrity - that is the job of protocols layering on top of the serial driver.

In the current implementation the serial tests and the serial filter are not resilient to such data errors. This means that the test may crash or hang (possibly without reporting a `FAIL`). It also means that you should be aware of random errors - a `FAIL` is not necessarily caused by a bug in the serial driver.

Ideally, the serial testing infrastructure should be able to distinguish random errors from consistent errors - the former are most likely due to noise in the transfer medium, while the latter are more likely to be caused by faulty drivers. The current implementation of the infrastructure does not have this capability.

Debugging

If a test fails, the serial filter's output may provide some hints about what the problem is. If the option `-S` is used when starting the filter, data received from the target is printed out:

```
[400 11:35:16] 0000 50 41 53 53 3a 3c 42 69 'PASS:<Bi'
[400 11:35:16] 0008 6e 61 72 79 20 74 65 73 'nary.tes'
[400 11:35:16] 0010 74 20 63 6f 6d 70 6c 65 't.comple'
[400 11:35:16] 0018 74 65 64 3e 0d 0a 49 4e 'ted>..IN'
[400 11:35:16] 0020 46 4f 3a 3c 42 49 4e 41 'FO:<BINA'
[400 11:35:16] 0028 52 59 3a 31 32 38 3a 31 'RY:128:1'
[400 11:35:16] 0030 21 3e 0d 0a 40 42 49 4e '!..@BIN'
[400 11:35:16] 0038 41 52 59 3a 31 32 38 3a 'ARY:128:'
[400 11:35:16] 0040 31 21 .. .. .. .. .. .. '1!'
```

In the case of an error during a testing command the data received by the filter will be printed out, as will the data that was expected. This allows the two data sets to be compared which may give some idea of what the problem is.

Chapter 19. Device Driver Interface to the Kernel

This chapter describes the API that device drivers may use to interact with the kernel and HAL. It is primarily concerned with the control and management of interrupts and the synchronization of ISRs, DSRs and threads.

The same API will be present in configurations where the kernel is not present. In this case the functions will be supplied by code acting directly on the HAL.

Interrupt Model

eCos presents a three level interrupt model to device drivers. This consists of Interrupt Service Routines (ISRs) that are invoked in response to a hardware interrupt; Deferred Service Routines (DSRs) that are invoked in response to a request by an ISR; and threads that are the clients of the driver.

Hardware interrupts are delivered with minimal intervention to an ISR. The HAL decodes the hardware source of the interrupt and calls the ISR of the attached interrupt object. This ISR may manipulate the hardware but is only allowed to make a restricted set of calls on the driver API. When it returns, an ISR may request that its DSR should be scheduled to run.

A DSR will be run when it is safe to do so without interfering with the scheduler. Most of the time the DSR will run immediately after the ISR, but if the current thread is in the scheduler, it will be delayed until the thread is finished. A DSR is allowed to make a larger set of driver API calls, including, in particular, being able to call `cyg_drv_cond_signal()` to wake up waiting threads.

Finally, threads are able to make all API calls and in particular are allowed to wait on mutexes and condition variables.

For a device driver to receive interrupts it must first define ISR and DSR routines as shown below, and then call `cyg_drv_interrupt_create()`. Using the handle returned, the driver must then call `cyg_drv_interrupt_attach()` to actually attach the interrupt to the hardware vector.

Synchronization

There are three levels of synchronization supported:

1. Synchronization with ISRs. This normally means disabling interrupts to prevent the ISR running during a critical section. In an SMP environment, this will also require the use of a spinlock to synchronize with ISRs, DSRs or threads running on other CPUs. This is implemented by the `cyg_drv_isr_lock()` and `cyg_drv_isr_unlock()` functions. This mechanism should be used sparingly and for short periods only. For finer grained synchronization, individual spinlocks are also supplied.

2. Synchronization with DSRs. This will be implemented in the kernel by taking the scheduler lock to prevent DSRs running during critical sections. In non-kernel configurations it will be implemented by non-kernel code. This is implemented by the `cyg_drv_dsr_lock()` and `cyg_drv_dsr_unlock()` functions. As with ISR synchronization, this mechanism should be used sparingly. Only DSRs and threads may use this synchronization mechanism, ISRs are not allowed to do this.

3. Synchronization with threads. This is implemented with mutexes and condition variables. Only threads may lock the mutexes and wait on the condition variables, although DSRs may signal condition variables.

Any data that is accessed from more than one level must be protected against concurrent access. Data that is accessed by ISRs must be protected with the ISR lock, or a spinlock at all times, *even in ISRs*. Data that is shared between DSRs and threads should be protected with the DSR lock. Data that is only accessed by threads must be protected with mutexes.

SMP Support

Some eCos targets contain support for Symmetric Multi-Processing (SMP) configurations, where more than one CPU may be present. This option has a number of ramifications for the way in which device drivers must be written if they are to be SMP-compatible.

Since it is possible for the ISR, DSR and thread components of a device driver to execute on different CPUs, it is important that SMP-compatible device drivers use the driver API routines correctly.

Synchronization between threads and DSRs continues to require that the thread-side code use `cyg_drv_dsr_lock()` and `cyg_drv_dsr_unlock()` to protect access to shared data. While it is not strictly necessary for DSR code to claim the DSR lock, since DSRs are run with it claimed already, it is good practice to do so.

Synchronization between ISRs and DSRs or threads requires that access to sensitive data be protected, in all places, by calls to `cyg_drv_isr_lock()` and `cyg_drv_isr_unlock()`. Disabling or masking interrupts is not adequate, since the thread or DSR may be running on a different CPU and interrupt enable/disable only work on the current CPU.

The ISR lock, for SMP systems, not only disables local interrupts, but also acquires a spinlock to protect against concurrent access from other CPUs. This is necessary because ISRs are not run with the scheduler lock claimed. Hence they can run in parallel with the other components of the device driver.

The ISR lock provided by the driver API is just a shared spinlock that is available for use by all drivers. If a driver needs to implement a finer grain of locking, it can use private spinlocks, accessed via the `cyg_drv_spinlock_*()` functions.

Device Driver Models

There are several ways in which device drivers may be built. The exact model chosen will depend on the properties of the device and the behavior desired. There are three basic models that may be adopted.

The first model is to do all device processing in the ISR. When it is invoked the ISR programs the device hardware directly and accesses data to be transferred directly in memory. The ISR should also call `cyg_drv_interrupt_acknowledge()`. When it is finished it may optionally request that its DSR be invoked. The DSR does nothing but call `cyg_drv_cond_signal()` to cause a thread to be woken up. Thread level code must call `cyg_drv_isr_lock()`, or `cyg_drv_interrupt_mask()` to prevent ISRs running while it manipulates shared memory.

The second model is to defer device processing to the DSR. The ISR simply prevents further delivery of interrupts by either programming the device, or by calling `cyg_drv_interrupt_mask()`. It must then call `cyg_drv_interrupt_acknowledge()` to allow other interrupts to be delivered and then request that its DSR be called. When the DSR runs it does the majority of the device handling, optionally signals a condition variable to wake a thread, and finishes by calling `cyg_drv_interrupt_unmask()` to re-allow device interrupts. Thread level code uses `cyg_drv_dsr_lock()` to prevent DSRs running while it manipulates shared memory. The eCos serial device drivers use this approach.

The third model is to defer device processing even further to a thread. The ISR behaves exactly as in the previous model and simply blocks and acknowledges the interrupt before request that the DSR run. The DSR itself only calls `cyg_drv_cond_signal()` to wake the thread. When the thread awakens it performs all device processing, and has full access to all kernel facilities while it does so. It should finish by calling `cyg_drv_interrupt_unmask()` to re-allow device interrupts. The eCos ethernet device drivers are written to this model.

The first model is good for devices that need immediate processing and interact infrequently with thread level. The second model trades a little latency in dealing with the device for a less intrusive synchronization mechanism. The last model allows device processing to be scheduled with other threads and permits more complex device handling.

Synchronization Levels

Since it would be dangerous for an ISR or DSR to make a call that might reschedule the current thread (by trying to lock a mutex for example) all functions in this API have an associated synchronization level. These levels are:

Thread
: This function may only be called from within threads. This is usually the client code that makes calls into the device driver. In a non-kernel configuration, this will be code running at the default non-interrupt level.

DSR
: This function may be called by either DSR or thread code.

ISR
: This function may be called from ISR, DSR or thread code.

The following table shows, for each API function, the levels at which is may be called:

	Callable from:		
Function	ISR	DSR	Thread
cyg_drv_isr_lock	X	X	X
cyg_drv_isr_unlock	X	X	X
cyg_drv_spinlock_init			X
cyg_drv_spinlock_destroy			X
cyg_drv_spinlock_spin	X	X	X
cyg_drv_spinlock_clear	X	X	X
cyg_drv_spinlock_try	X	X	X
cyg_drv_spinlock_test	X	X	X
cyg_drv_spinlock_spin_intsave	X	X	X
cyg_drv_spinlock_clear_intsave	X	X	X
cyg_drv_dsr_lock		X	X
cyg_drv_dsr_unlock		X	X
cyg_drv_mutex_init			X
cyg_drv_mutex_destroy			X
cyg_drv_mutex_lock			X
cyg_drv_mutex_trylock			X
cyg_drv_mutex_unlock			X
cyg_drv_mutex_release			X
cyg_drv_cond_init			X
cyg_drv_cond_destroy			X
cyg_drv_cond_wait			X
cyg_drv_cond_signal		X	X
cyg_drv_cond_broadcast		X	X
cyg_drv_interrupt_create			X
cyg_drv_interrupt_delete			X
cyg_drv_interrupt_attach	X	X	X
cyg_drv_interrupt_detach	X	X	X
cyg_drv_interrupt_mask	X	X	X
cyg_drv_interrupt_unmask	X	X	X
cyg_drv_interrupt_acknowledge	X	X	X
cyg_drv_interrupt_configure	X	X	X
cyg_drv_interrupt_level	X	X	X
cyg_drv_interrupt_set_cpu	X	X	X
cyg_drv_interrupt_get_cpu	X	X	X

The API

This section details the Driver Kernel Interface. Note that most of these functions are identical to Kernel C API calls, and will in most configurations be wrappers for them. In non-kernel configurations they will be supported directly by the HAL, or by code to emulate the required behavior.

This API is defined in the header file `<cyg/hal/drv_api.h>`.

cyg_drv_isr_lock

Function: `void cyg_drv_isr_lock()`

Arguments: None

Result: None

Level: ISR

Description: Disables delivery of interrupts, preventing all ISRs running. This function maintains a counter of the number of times it is called.

cyg_drv_isr_unlock

Function: `void cyg_drv_isr_unlock()`

Arguments: None

Result: None

Level: ISR

Description: Re-enables delivery of interrupts, allowing ISRs to run. This function decrements the counter maintained by `cyg_drv_isr_lock()`, and only re-allows interrupts when it goes to zero.

cyg_drv_spinlock_init

Function: `void cyg_drv_spinlock_init(cyg_spinlock_t *lock, cyg_bool_t locked )`

Arguments: *lock* - pointer to spinlock to initialize

locked - initial state of lock

Result: None

Level: Thread

Description: Initialize a spinlock. The *locked* argument indicates how the spinlock should be initialized: `TRUE` for locked or `FALSE` for unlocked state.

cyg_drv_spinlock_destroy

Function: `void cyg_drv_spinlock_destroy(cyg_spinlock_t *lock )`

Arguments: *lock* - pointer to spinlock destroy

Result: None

Level: Thread

Description: Destroy a spinlock that is no longer of use. There should be no CPUs attempting to claim the lock at the time this function is called, otherwise the behavior is undefined.

cyg_drv_spinlock_spin

Function: `void cyg_drv_spinlock_spin(cyg_spinlock_t *lock )`

Arguments: *lock* - pointer to spinlock to claim

Result: None

Level: ISR

Description: Claim a spinlock, waiting in a busy loop until it is available. Wherever this is called from, this operation effectively pauses the CPU until it succeeds. This operations should therefore be used sparingly, and in situations where deadlocks/livelocks cannot occur. Also see `cyg_drv_spinlock_spin_intsave()`.

cyg_drv_spinlock_clear

Function: `void cyg_drv_spinlock_clear(cyg_spinlock_t *lock )`

Arguments: *lock* - pointer to spinlock to clear

Result: None

Level: ISR

Description: Clear a spinlock. This clears the spinlock and allows another CPU to claim it. If there is more than one CPU waiting in `cyg_drv_spinlock_spin()` then just one of them will be allowed to proceed.

cyg_drv_spinlock_try

Function: `cyg_bool_t cyg_drv_spinlock_try(cyg_spinlock_t *lock )`

Arguments: *lock* - pointer to spinlock to try

Result: `TRUE` if the spinlock was claimed, `FALSE` otherwise.

Level: ISR

Description: Try to claim the spinlock without waiting. If the spinlock could be claimed immediately then `TRUE` is returned. If the spinlock is already claimed then the result is `FALSE`.

cyg_drv_spinlock_test

Function: `cyg_bool_t cyg_drv_spinlock_test(cyg_spinlock_t *lock )`

Arguments: *lock* - pointer to spinlock to test

Result: `TRUE` if the spinlock is available, `FALSE` otherwise.

Level: ISR

Description: Inspect the state of the spinlock. If the spinlock is not locked then the result is `TRUE`. If it is locked then the result will be `FALSE`.

cyg_drv_spinlock_spin_intsave

Function:

```
void cyg_drv_spinlock_spin_intsave( cyg_spinlock_t *lock,
                                    cyg_addrword_t *istate )
```

Arguments: *lock* - pointer to spinlock to claim

istate - pointer to interrupt state save location

Result: None

Level: ISR

Description: This function behaves exactly like `cyg_drv_spinlock_spin()` except that it also disables interrupts before attempting to claim the lock. The current interrupt enable state is saved in **istate*. Interrupts remain disabled once the spinlock had been claimed and must be restored by calling `cyg_drv_spinlock_clear_intsave()`.

In general, device drivers should use this function to claim and release spinlocks rather than the non-`_intsave()` variants, to ensure proper exclusion with code running on both other CPUs and this CPU.

cyg_drv_spinlock_clear_intsave

Function:

```
void cyg_drv_spinlock_clear_intsave( cyg_spinlock_t *lock,
                                     cyg_addrword_t istate )
```

Arguments: *lock* - pointer to spinlock to clear

istate - interrupt state to restore

Result: None

Level: ISR

Description: This function behaves exactly like `cyg_drv_spinlock_clear()` except that it also restores an interrupt state saved by `cyg_drv_spinlock_spin_intsave()`. The *istate* argument must have been initialized by a previous call to `cyg_drv_spinlock_spin_intsave()`.

cyg_drv_dsr_lock

Function:

```
void cyg_drv_dsr_lock()
```

Arguments: None

Result: None

Level: DSR

Description: Disables scheduling of DSRs. This function maintains a counter of the number of times it has been called.

cyg_drv_dsr_unlock

Function:

```
void cyg_drv_dsr_unlock()
```

Arguments: None

Result:

None

Level: DSR

Description: Re-enables scheduling of DSRs. This function decrements the counter incremented by `cyg_drv_dsr_lock()`. DSRs are only allowed to be delivered when the counter goes to zero.

cyg_drv_mutex_init

Function: `void cyg_drv_mutex_init(cyg_drv_mutex_t *mutex)`

Arguments: *`mutex`* - pointer to mutex to initialize

Result: None

Level: Thread

Description: Initialize the mutex pointed to by the `mutex` argument.

cyg_drv_mutex_destroy

Function: `void cyg_drv_mutex_destroy( cyg_drv_mutex_t *mutex )`

Arguments: *`mutex`* - pointer to mutex to destroy

Result: None

Level: Thread

Description: Destroy the mutex pointed to by the *`mutex`* argument. The mutex should be unlocked and there should be no threads waiting to lock it when this call in made.

cyg_drv_mutex_lock

Function: `cyg_bool cyg_drv_mutex_lock( cyg_drv_mutex_t *mutex )`

Arguments: *`mutex`* - pointer to mutex to lock

Result: `TRUE` it the thread has claimed the lock, `FALSE` otherwise.

Level: Thread

Description: Attempt to lock the mutex pointed to by the *`mutex`* argument. If the mutex is already locked by another thread then this thread will wait until that thread is finished. If the result from this function is `FALSE` then the thread was broken out of its wait by some other thread. In this case the mutex will not have been locked.

cyg_drv_mutex_trylock

Function: `cyg_bool cyg_drv_mutex_trylock( cyg_drv_mutex_t *mutex )`

Arguments: *`mutex`* - pointer to mutex to lock

Result: `TRUE` if the mutex has been locked, `FALSE` otherwise.

Level: Thread

Description: Attempt to lock the mutex pointed to by the *`mutex`* argument without waiting. If the mutex is already locked by some other thread then this function returns `FALSE`. If the function can lock the mutex without waiting, then `TRUE` is returned.

cyg_drv_mutex_unlock

Function: `void cyg_drv_mutex_unlock( cyg_drv_mutex_t *mutex )`

Arguments: *mutex* - pointer to mutex to unlock

Result: None

Level: Thread

Description: Unlock the mutex pointed to by the *mutex* argument. If there are any threads waiting to claim the lock, one of them is woken up to try and claim it.

cyg_drv_mutex_release

Function: `void cyg_drv_mutex_release( cyg_drv_mutex_t *mutex )`

Arguments: `mutex` - pointer to mutex to release

Result: None

Level: Thread

Description: Release all threads waiting on the mutex pointed to by the *mutex* argument. These threads will return from `cyg_drv_mutex_lock()` with a `FALSE` result and will not have claimed the mutex. This function has no effect on any thread that may have the mutex claimed.

cyg_drv_cond_init

Function: `void cyg_drv_cond_init( cyg_drv_cond_t *cond, cyg_drv_mutex_t *mutex )`

Arguments: *cond* - condition variable to initialize

mutex - mutex to associate with this condition variable

Result: None

Level: Thread

Description: Initialize the condition variable pointed to by the *cond* argument. The *mutex* argument must point to a mutex with which this condition variable is associated. A thread may only wait on this condition variable when it has already locked the associated mutex. Waiting will cause the mutex to be unlocked, and when the thread is reawakened, it will automatically claim the mutex before continuing.

cyg_drv_cond_destroy

Function: `void cyg_drv_cond_destroy( cyg_drv_cond_t *cond )`

Arguments: *cond* - condition variable to destroy

Result: None

Level: Thread

Description: Destroy the condition variable pointed to by the *cond* argument.

cyg_drv_cond_wait

Function:

```
void cyg_drv_cond_wait( cyg_drv_cond_t *cond )
```

Arguments: *cond* - condition variable to wait on

Result: None

Level: Thread

Description: Wait for a signal on the condition variable pointed to by the *cond* argument. The thread must have locked the associated mutex, supplied in `cyg_drv_cond_init()`, before waiting on this condition variable. While the thread waits, the mutex will be unlocked, and will be re-locked before this function returns. It is possible for threads waiting on a condition variable to occasionally wake up spuriously. For this reason it is necessary to use this function in a loop that re-tests the condition each time it returns. Note that this function performs an implicit scheduler unlock/relock sequence, so that it may be used within an explicit `cyg_drv_dsr_lock()`... `cyg_drv_dsr_unlock()` structure.

cyg_drv_cond_signal

Function:

```
void cyg_drv_cond_signal( cyg_drv_cond_t *cond )
```

Arguments: *cond* - condition variable to signal

Result: None

Level: DSR

Description: Signal the condition variable pointed to by the *cond* argument. If there are any threads waiting on this variable at least one of them will be awakened. Note that in some configurations there may not be any difference between this function and `cyg_drv_cond_broadcast()`.

cyg_drv_cond_broadcast

Function:

```
void cyg_drv_cond_broadcast( cyg_drv_cond_t *cond )
```

Arguments: *cond* - condition variable to broadcast to

Result: None

Level: DSR

Description: Signal the condition variable pointed to by the *cond* argument. If there are any threads waiting on this variable they will all be awakened.

cyg_drv_interrupt_create

Function:

```
void cyg_drv_interrupt_create(
    cyg_vector_t    vector,
    cyg_priority_t  priority,
    cyg_addrword_t  data,
    cyg_ISR_t       *isr,
    cyg_DSR_t       *dsr,
    cyg_handle_t    *handle,
    cyg_interrupt   *intr)
```

Arguments: *vector* - vector to attach to

priority - queuing priority

data - data pointer

isr - interrupt service routine

dsr - deferred service routine

handle - returned handle

intr - put interrupt object here

Result: None

Level: Thread

Description: Create an interrupt object and returns a handle to it. The object contains information about which interrupt vector to use and the ISR and DSR that will be called after the interrupt object is attached to the vector. The interrupt object will be allocated in the memory passed in the *intr* parameter. The interrupt object is not immediately attached; it must be attached with the `cyg_interrupt_attach()` call.

The *data* argument will be passed to both the registered ISR and DSR. Typically it will be a pointer to some data structure.

cyg_drv_interrupt_delete

Function:

```
void cyg_drv_interrupt_delete( cyg_handle_t interrupt )
```

Arguments: *interrupt* - interrupt to delete

Result: None

Level: Thread

Description: Detach the interrupt from the vector and free the memory passed in the *intr* argument to `cyg_drv_interrupt_create()` for reuse.

cyg_drv_interrupt_attach

Function:

```
void cyg_drv_interrupt_attach( cyg_handle_t interrupt )
```

Arguments: *interrupt* - interrupt to attach

Result: None

Level: ISR

Description: Attach the interrupt to the vector so that interrupts will be delivered to the ISR when the interrupt occurs.

cyg_drv_interrupt_detach

Function:

```
void cyg_drv_interrupt_detach( cyg_handle_t interrupt )
```

Arguments: *interrupt* - interrupt to detach

Result: None

Level: ISR

Description: Detach the interrupt from the vector so that interrupts will no longer be delivered to the ISR.

cyg_drv_interrupt_mask

Function: `void cyg_drv_interrupt_mask(cyg_vector_t vector )`

Arguments: *`vector`* - vector to mask

Result: None

Level: ISR

Description: Program the interrupt controller to stop delivery of interrupts on the given vector. On architectures which implement interrupt priority levels this may also disable all lower priority interrupts.

cyg_drv_interrupt_mask_intunsafe

Function: `void cyg_drv_interrupt_mask_intunsafe(cyg_vector_t vector )`

Arguments: *`vector`* - vector to mask

Result: None

Level: ISR

Description: Program the interrupt controller to stop delivery of interrupts on the given vector. On architectures which implement interrupt priority levels this may also disable all lower priority interrupts. This version differs from `cyg_drv_interrupt_mask()` in not being interrupt safe. So in situations where, for example, interrupts are already known to be disabled, this may be called to avoid the extra overhead.

cyg_drv_interrupt_unmask

Function: `void cyg_drv_interrupt_unmask(cyg_vector_t vector )`

Arguments: *`vector`* - vector to unmask

Result: None

Level: ISR

Description: Program the interrupt controller to re-allow delivery of interrupts on the given *`vector`*.

cyg_drv_interrupt_unmask_intunsafe

Function: `void cyg_drv_interrupt_unmask_intunsafe(cyg_vector_t vector )`

Arguments: *`vector`* - vector to unmask

Result: None

Level: ISR

Description: Program the interrupt controller to re-allow delivery of interrupts on the given *vector*. This version differs from `cyg_drv_interrupt_unmask()` in not being interrupt safe.

cyg_drv_interrupt_acknowledge

Function:

```
void cyg_drv_interrupt_acknowledge( cyg_vector_t vector )
```

Arguments: *vector* - vector to acknowledge

Result: None

Level: ISR

Description: Perform any processing required at the interrupt controller and in the CPU to cancel the current interrupt request on the *vector*. An ISR may also need to program the hardware of the device to prevent an immediate re-triggering of the interrupt.

cyg_drv_interrupt_configure

Function:

```
void cyg_drv_interrupt_configure( cyg_vector_t vector,
                                  cyg_bool_t level,
                                  cyg_bool_t up)
```

Arguments: *vector* - vector to configure

level - level or edge triggered

up - rising/falling edge, high/low level

Result: None

Level: ISR

Description: Program the interrupt controller with the characteristics of the interrupt source. The *level* argument chooses between level- or edge-triggered interrupts. The *up* argument chooses between high and low level for level triggered interrupts or rising and falling edges for edge triggered interrupts. This function only works with interrupt controllers that can control these parameters.

cyg_drv_interrupt_level

Function:

```
void cyg_drv_interrupt_level( cyg_vector_t vector,
                              cyg_priority_t level)
```

Arguments: *vector* - vector to configure

level - level to set

Result: None

Level: ISR

Description: Program the interrupt controller to deliver the given interrupt at the supplied priority level. This function only works with interrupt controllers that can control this parameter.

cyg_drv_interrupt_set_cpu

Function:

```
void cyg_drv_interrupt_set_cpu( cyg_vector_t vector,
                                        cyg_cpu_t cpu)
```

Arguments: *vector* - interrupt vector to route

cpu - destination CPU

Result: None

Level: ISR

Description: This function causes all interrupts on the given vector to be routed to the specified CPU. Subsequently, all such interrupts will be handled by that CPU. This only works if the underlying hardware is capable of performing this kind of routing. This function does nothing on a single CPU system.

cyg_drv_interrupt_get_cpu

Function:

```
cyg_cpu_t cyg_drv_interrupt_set_cpu( cyg_vector_t vector )
```

Arguments: *vector* - interrupt vector to query

Result: The CPU to which this vector is routed

Level: ISR

Description: In multi-processor systems this function returns the id of the CPU to which interrupts on the given vector are current being delivered. In single CPU systems this function returns zero.

cyg_ISR_t

Type:

```
typedef cyg_uint32 cyg_ISR_t(
    cyg_vector_t    vector,
    cyg_addrword_t  data
)
```

Fields: *vector* - vector being delivered

data - data value supplied by client

Result: Bit mask indicating whether interrupt was handled and whether the DSR should be called.

Description: Interrupt Service Routine definition. A pointer to a function with this prototype is passed to `cyg_interrupt_create()` when an interrupt object is created. When an interrupt is delivered the function will be called with the vector number and the data value that was passed to `cyg_interrupt_create()`.

The return value is a bit mask containing one or both of the following bits:

CYG_ISR_HANDLED indicates that the interrupt was handled by this ISR. It is a configuration option whether this will prevent further ISR being run.

CYG_ISR_CALL_DSR causes the DSR that was passed to `cyg_interrupt_create()` to be scheduled to be called.

cyg_DSR_t

Type:

```
typedef void cyg_DSR_t(
        cyg_vector_t    vector,
        cyg_ucount32    count,
        cyg_addrword_t  data
)
```

Fields: *`vector`* - vector being delivered

`count` - number of times DSR has been scheduled

`data` - data value supplied by client

Result: None

Description: Deferred Service Routine prototype. A pointer to a function with this prototype is passed to `cyg_interrupt_create()` when an interrupt object is created. When the ISR requests the scheduling of its DSR, this function will be called at some later point. In addition to the *`vector`* and *`data`* arguments, which will be the same as those passed to the ISR, this routine is also passed a *`count`* of the number of times the ISR has requested that this DSR be scheduled. This counter is zeroed each time the DSR actually runs, so it indicates how many interrupts have occurred since it last ran.

Instrumentation

If the system instrumentation support is enabled then the I/O package provides support for generating instrumentation records for various events within the general purpose I/O framework.

Instrumentation records will only be generated if the `CYGIMP_IO_INSTRUMENTATION` option is enabled, and then only if the relevant individual event code sub-options are also enabled. The default state is for all the instrumentation to be disabled. Some options will generate a lot of instrumentation records in a heavily loaded system and so care may need to be taken regarding the instrumentation that is enabled vs the instrumentation recording mechanism being used to avoid missing events. Depending on why the I/O framework instrumentation is being enabled (debugging, timing validation, etc.) the user can choose which events they wish to record by enabling the specific CDL options.

Part VII. File System Support Infrastructure

Table of Contents

Chapter 20. Introduction

This document describes the filesystem infrastructure provided in eCos. This is implemented by the FILEIO package and provides POSIX compliant file and IO operations together with the BSD socket API. These APIs are described in the relevant standards and original documentation and will not be described here. See the Posix Standard Support documentation for details of which parts of the POSIX standard are supported.

This document is concerned with the interfaces presented to client filesystems and network protocol stacks.

The FILEIO infrastructure consist mainly of a set of tables containing pointers to the primary interface functions of a file system. This approach avoids problems of namespace pollution (for example several filesystems can have a function called `read()`, so long as they are static). The system is also structured to eliminate the need for dynamic memory allocation.

New filesystems can be written directly to the interfaces described here. Existing filesystems can be ported very easily by the introduction of a thin veneer porting layer that translates FILEIO calls into native filesystem calls.

The term filesystem should be read fairly loosely in this document. Object accessed through these interfaces could equally be network protocol sockets, device drivers, fifos, message queues or any other object that can present a file-like interface.

Chapter 21. File System Table

The filesystem table is an array of entries that describe each filesystem implementation that is part of the system image. Each resident filesystem should export an entry to this table using the `FSTAB_ENTRY()` macro.

Note

At present we do not support dynamic addition or removal of table entries. However, an API similar to `mount()` would allow new entries to be added to the table.

The table entries are described by the following structure:

```
struct cyg_fstab_entry
{
    const char          *name;          // filesystem name
    CYG_ADDRWORD        data;           // private data value
    cyg_uint32          syncmode;       // synchronization mode

    int     (*mount)    ( cyg_fstab_entry *fste,
                          cyg_mtab_entry  *mte );
    int     (*umount)   ( cyg_mtab_entry  *mte,
                          cyg_bool        force );
    int     (*open)     ( cyg_mtab_entry  *mte,
                          cyg_dir         dir,
                          const char      *name,
                          int             mode,
                          cyg_file        *fte );
    int     (*unlink)   ( cyg_mtab_entry  *mte,
                          cyg_dir         dir,
                          const char      *name );
    int     (*mkdir)    ( cyg_mtab_entry  *mte,
                          cyg_dir         dir,
                          const char      *name );
    int     (*rmdir)    ( cyg_mtab_entry  *mte,
                          cyg_dir         dir,
                          const char      *name );
    int     (*rename)   ( cyg_mtab_entry  *mte,
                          cyg_dir         dir1,
                          const char      *name1,
                          cyg_dir         dir2,
                          const char      *name2 );
    int     (*link)     ( cyg_mtab_entry  *mte,
                          cyg_dir         dir1,
                          const char      *name1,
                          cyg_dir         dir2,
                          const char      *name2,
                          int             type );
    int     (*opendir)  ( cyg_mtab_entry  *mte,
                          cyg_dir         dir,
                          const char      *name,
                          cyg_file        *fte );
    int     (*chdir)    ( cyg_mtab_entry  *mte,
                          cyg_dir         dir,
                          const char      *name,
                          cyg_dir         *dir_out );
    int     (*stat)     ( cyg_mtab_entry  *mte,
                          cyg_dir         dir,
                          const char      *name,
                          struct stat     *buf);
    int     (*getinfo)  ( cyg_mtab_entry  *mte,
                          cyg_dir         dir,
                          const char      *name,
                          int             key,
                          char             *buf,
                          int              len );
    int     (*setinfo)  ( cyg_mtab_entry   *mte,
                          cyg_dir          dir,
                          const char       *name,
                          int              key,
                          char             *buf,
                          int              len );
};
```

The *name* field points to a string that identifies this filesystem implementation. Typical values might be "romfs", "fatfs", "ext2" etc.

The *data* field contains any private data that the filesystem needs, perhaps the root of its data structures.

The *syncmode* field contains a description of the locking protocol to be used when accessing this filesystem. It will be described in more detail in Chapter 25, *Synchronization*.

The remaining fields are pointers to functions that implement filesystem operations that apply to files and directories as whole objects. The operation implemented by each function should be obvious from the names, with a few exceptions:

The `opendir()` function pointer opens a directory for reading. See Chapter 24, *Directories* for details.

The `getinfo()` and `setinfo()` function pointers provide support for various minor control and information functions such as `pathconf()` and `access()`.

With the exception of the `mount()` and `umount()` functions, all of these functions take three standard arguments, a pointer to a mount table entry (see later) a directory pointer (also see later) and a file name relative to the directory. These should be used by the filesystem to locate the object of interest.

Chapter 22. Mount Table

The mount table records the filesystems that are actually active. These can be seen as being analogous to mount points in Unix systems.

There are two sources of mount table entries. Filesystems (or other components) may export static entries to the table using the `MTAB_ENTRY()` macro. Alternatively, new entries may be installed at run time using the `mount()` function. Both types of entry may be unmounted with the `umount()` function.

A mount table entry has the following structure:

```
struct cyg_mtab_entry
{
    const char          *name;          // name of mount point
    const char          *fsname;        // name of implementing filesystem
    const char          *devname;       // name of hardware device
    const char          *options;       // mount option string
    CYG_ADDRWORD        data;           // private data value
    cyg_bool            valid;          // Valid entry?
    cyg_fstab_entry     *fs;            // pointer to fstab entry
    cyg_dir             root;           // root directory pointer
};
```

The *name* field identifies the mount point. This is used to direct rooted filenames (filenames that begin with "/") to the correct filesystem. When a file name that begins with "/" is submitted, it is matched against the *name* fields of all valid mount table entries. The entry that yields the longest match terminating before a "/", or end of string, wins and the appropriate function from the filesystem table entry is then passed the remainder of the file name together with a pointer to the table entry and the value of the *root* field as the directory pointer.

For example, consider a mount table that contains the following entries:

```
{ "/",    "fatfs", "/dev/hd0", … }
{ "/fd",  "fatfs", "/dev/fd0", … }
{ "/rom", "romfs", "", … }
{ "/tmp", "ramfs", "", … }
{ "/dev", "devfs", "", … }
```

An attempt to open "/tmp/foo" would be directed to the RAM filesystem while an open of "/bar/bundy" would be directed to the hard disc FATFS filesystem. Opening "/dev/tty0" would be directed to the device management filesystem for lookup in the device table.

Unrooted file names (those that do not begin with a '/') are passed straight to the filesystem that contains the current directory. The current directory is represented by a pair consisting of a mount table entry and a directory pointer.

The *fsname* field points to a string that should match the *name* field of the implementing filesystem. During initialization the mount table is scanned and the *fsname* entries looked up in the filesystem table. For each match, the filesystem's _mount_ function is called and if successful the mount table entry is marked as valid and the *fs* pointer installed.

The *devname* field contains the name of the device that this filesystem is to use. This may match an entry in the device table (see later) or may be a string that is specific to the filesystem if it has its own internal device drivers.

The *data* field is a private data value. This may be installed either statically when the table entry is defined, or may be installed during the `mount()` operation.

The *valid* field indicates whether this mount point has actually been mounted successfully. Entries with a false *valid* field are ignored when searching for a name match.

The *fs* field is installed after a successful `mount()` operation to point to the implementing filesystem.

The *`root`* field contains a directory pointer value that the filesystem can interpret as the root of its directory tree. This is passed as the *`dir`* argument of filesystem functions that operate on rooted filenames. This field must be initialized by the filesystem's `mount()` function.

Chapter 23. File Table

Once a file has been opened it is represented by an open file object. These are allocated from an array of available file objects. User code accesses these open file objects via a second array of pointers which is indexed by small integer offsets. This gives the usual Unix file descriptor functionality, complete with the various duplication mechanisms.

A file table entry has the following structure:

```
struct CYG_FILE_TAG
{
    cyg_uint32                  f_flag;         /* file state                   */
    cyg_uint16                  f_ucount;       /* use count                    */
    cyg_uint16                  f_type;         /* descriptor type              */
    cyg_uint32                  f_syncmode;     /* synchronization protocol     */
    struct CYG_FILEOPS_TAG      *f_ops;         /* file operations              */
    off_t                       f_offset;       /* current offset               */
    CYG_ADDRWORD                f_data;         /* file or socket               */
    CYG_ADDRWORD                f_xops;         /* extra type specific ops      */
    cyg_mtab_entry              *f_mte;         /* mount table entry            */
};
```

The *f_flag* field contains some FILEIO control bits and some bits propagated from the *flags* argument of the `open()` call (defined by `CYG_FILE_MODE_MASK`).

The *f_ucount* field contains a use count that controls when a file will be closed. Each duplicate in the file descriptor array counts for one reference here. It is also incremented around each I/O operation to ensure that the file cannot be closed while it has current I/O operations.

The *f_type* field indicates the type of the underlying file object. Some of the possible values here are `CYG_FILE_TYPE_FILE`, `CYG_FILE_TYPE_SOCKET` or `CYG_FILE_TYPE_DEVICE`.

The *f_syncmode* field is copied from the *syncmode* field of the implementing filesystem. Its use is described in Chapter 25, *Synchronization*.

The *f_offset* field records the current file position. It is the responsibility of the file operation functions to keep this field up to date.

The *f_data* field contains private data placed here by the underlying filesystem. Normally this will be a pointer to, or handle on, the filesystem object that implements this file.

The *f_xops* field contains a pointer to any extra type specific operation functions. For example, the socket I/O system installs a pointer to a table of functions that implement the standard socket operations.

The *f_mte* field contains a pointer to the parent mount table entry for this file. It is used mainly to implement the synchronization protocol. This may contain a pointer to some other data structure in file objects not derived from a filesystem.

The *f_ops* field contains a pointer to a table of file I/O operations. This has the following structure:

```
struct CYG_FILEOPS_TAG
{
    int (*fo_read)      (struct CYG_FILE_TAG *fp, struct CYG_UIO_TAG *uio);
    int (*fo_write)     (struct CYG_FILE_TAG *fp, struct CYG_UIO_TAG *uio);
    int (*fo_lseek)     (struct CYG_FILE_TAG *fp, off_t *pos, int whence );
    int (*fo_ioctl)     (struct CYG_FILE_TAG *fp, CYG_ADDRWORD com, CYG_ADDRWORD data);
    int (*fo_select)    (struct CYG_FILE_TAG *fp, int which, CYG_ADDRWORD info);
    int (*fo_fsync)     (struct CYG_FILE_TAG *fp, int mode );
    int (*fo_close)     (struct CYG_FILE_TAG *fp);
    int (*fo_fstat)     (struct CYG_FILE_TAG *fp, struct stat *buf );
    int (*fo_getinfo)   (struct CYG_FILE_TAG *fp, int key, char *buf, int len );
    int (*fo_setinfo)   (struct CYG_FILE_TAG *fp, int key, char *buf, int len );
};
```

It should be obvious from the names of most of these functions what their responsibilities are. The `fo_getinfo()` and `fo_setinfo()` function pointers, like their counterparts in the filesystem structure, implement minor control and info functions such as `fpathconf()`.

The second argument to the `fo_read()` and `fo_write()` function pointers is a pointer to a UIO structure:

```
struct CYG_UIO_TAG
{
    struct CYG_IOVEC_TAG  *uio_iov;     /* pointer to array of iovecs */
    int                   uio_iovcnt;   /* number of iovecs in array */
    off_t                 uio_offset;   /* offset into file this uio corresponds to */
    ssize_t               uio_resid;    /* residual i/o count */
    enum cyg_uio_seg      uio_segflg;   /* see above */
    enum cyg_uio_rw       uio_rw;       /* see above */
};

struct CYG_IOVEC_TAG
{
    void                  *iov_base;    /* Base address. */
    ssize_t               iov_len;      /* Length. */
};
```

This structure encapsulates the parameters of any data transfer operation. It provides support for scatter/gather operations and records the progress of any data transfer. It is also compatible with the I/O operations of any BSD-derived network stacks and filesystems.

When a file is opened (or a file object created by some other means, such as `socket()` or `accept()`) it is the responsibility of the filesystem open operation to initialize all the fields of the object except the *f_ucount*, *f_syncmode* and *f_mte* fields. Since the *f_flag* field will already contain bits belonging to the FILEIO infrastructure, any changes to it must be made with the appropriate logical operations.

Chapter 24. Directories

Filesystem operations all take a directory pointer as one of their arguments. A directory pointer is an opaque handle managed by the filesystem. It should encapsulate a reference to a specific directory within the filesystem. For example, it may be a pointer to the data structure that represents that directory (such as an inode), or a pointer to a pathname for the directory.

The `chdir()` filesystem function pointer has two modes of use. When passed a pointer in the *`dir_out`* argument, it should locate the named directory and place a directory pointer there. If the *`dir_out`* argument is NULL then the *`dir`* argument is a previously generated directory pointer that can now be disposed of. When the infrastructure is implementing the `chdir()` function it makes two calls to filesystem `chdir()` functions. The first is to get a directory pointer for the new current directory. If this succeeds the second is to dispose of the old current directory pointer.

The `opendir()` function is used to open a directory for reading. This results in an open file object that can be read to return a sequence of struct dirent objects. The only operations that are allowed on this file are `read`, `lseek` and `close`. Each read operation on this file should return a single struct dirent object. When the end of the directory is reached, zero should be returned. The only seek operation allowed is a rewind to the start of the directory, by supplying an offset of zero and a *`whence`* specifier of `SEEK_SET`.

Most of these considerations are invisible to clients of a filesystem since they will access directories via the POSIX `opendir()`, `readdir()` and `closedir()` functions. The struct dirent object returned by `readdir()` will always contain d_name as required by POSIX. When `CYGPKG_FILEIO_DIRENT_DTYPE` is enabled it will also contain d_type, which is not part of POSIX, but often implemented by OSes. Currently only the FATFS, RAMFS, ROMFS and JFFS2 filesystem sets this value. For other filesystems a value of 0 will be returned in the member.

Support for the `getcwd()` function is provided by three mechanisms. The first is to use the `FS_INFO_GETCWD` getinfo key on the filesystem to use any internal support that it has for this. If that fails it falls back on one of the two other mechanisms. If `CYGPKG_IO_FILEIO_TRACK_CWD` is set then the current directory is tracked textually in `chdir()` and the result of that is reported in getcwd(). Otherwise an attempt is made to traverse the directory tree to its root using ".." entries.

This last option is complicated and expensive, and relies on the filesystem supporting "." and ".." entries. This is not always the case, particularly if the filesystem has been ported from a non-UNIX-compatible source. Tracking the pathname textually will usually work, but might not produce optimum results when symbolic links are being used.

Chapter 25. Synchronization

The FILEIO infrastructure provides a synchronization mechanism for controlling concurrent access to filesystems. This allows existing filesystems to be ported to eCos, even if they do not have their own synchronization mechanisms. It also allows new filesystems to be implemented easily without having to consider the synchronization issues.

The infrastructure maintains a mutex for each entry in each of the main tables: filesystem table, mount table and file table. For each class of operation each of these mutexes may be locked before the corresponding filesystem operation is invoked.

The synchronization protocol required by a filesystem is described by the *syncmode* field of the filesystem table entry. This is a combination of the following flags:

`CYG_SYNCMODE_FILE_FILESYSTEM`

Lock the filesystem table entry mutex during all filesystem level operations.

`CYG_SYNCMODE_FILE_MOUNTPOINT`

Lock the mount table entry mutex during all filesystem level operations.

`CYG_SYNCMODE_IO_FILE`

Lock the file table entry mutex during all I/O operations.

`CYG_SYNCMODE_IO_FILESYSTEM`

Lock the filesystem table entry mutex during all I/O operations.

`CYG_SYNCMODE_IO_MOUNTPOINT`

Lock the mount table entry mutex during all I/O operations.

`CYG_SYNCMODE_SOCK_FILE`

Lock the file table entry mutex during all socket operations.

`CYG_SYNCMODE_SOCK_NETSTACK`

Lock the network stack table entry mutex during all socket operations.

`CYG_SYNCMODE_NONE`

Perform no locking at all during any operations.

The value of the *syncmode* field in the filesystem table entry will be copied by the infrastructure to the open file object after a successful `open()` operation.

Chapter 26. Initialization and Mounting

As mentioned previously, mount table entries can be sourced from two places. Static entries may be defined by using the `MTAB_ENTRY()` macro. Such entries will be automatically mounted on system startup. For each entry in the mount table that has a non-null *`name`* field the filesystem table is searched for a match with the *`fsname`* field. If a match is found the filesystem's *`mount`* entry is called and if successful the mount table entry marked valid and the *`fs`* field initialized. The `mount()` function is responsible for initializing the *`root`* field.

The size of the mount table is defined by the configuration value `CYGNUM_FILEIO_MTAB_MAX`. Any entries that have not been statically defined are available for use by dynamic mounts.

A filesystem may be mounted dynamically by calling `mount()`. This function has the following prototype:

```
int mount(const char *devname,
          const char *dir,
          const char *fsname);
```

The *`devname`* argument identifies a device that will be used by this filesystem and will be assigned to the *`devname`* field of the mount table entry.

The *`dir`* argument is the mount point name, it will be assigned to the *`name`* field of the mount table entry.

The *`fsname`* argument is the name of the implementing filesystem, it will be assigned to the *`fsname`* entry of the mount table entry. This argument may also contain options that control the mode in which the filesystem is mounted.

Since these three arguments are assigned directly to the mount table entry, the memory pointed to by these arguments must not change for the duration of the mount. This means they must be allocated from memory that will persist unchanged until unmounting, such as constant strings, dynamically allocated memory, or static or automatic variables that do not pass out of scope or get their values changed before unmounting.

The options attached to the *`fsname`* argument consist of a comma separated list of single keywords or keyword=value pairs separated from the filesystem name by a colon. For example, to mount the FAT filesystem with write-through cache synchronization the string would be: `"fatfs:sync=write"` and to mount it read-only: `"fatfs:readonly"`.

The process of mounting a filesystem dynamically is as follows. First a search is made of the mount table for an entry with a NULL *`name`* field to be used for the new mount point. The filesystem table is then searched for an entry whose name matches *`fsname`*. If this is successful then the mount table entry is initialized and the filesystem's `mount()` operation called. If this is successful, the mount table entry is marked valid and the *`fs`* field initialized.

Mounting a filesystem dynamically at the current working directory name, does not in fact change the current directory to one on the newly mounted filesystem. Instead the current working directory remains on the previous filesystem (or no filesystem in the case of '/' with no filesystems previously mounted). This is in line with usual POSIX/UNIX behaviour. To change to the new filesystem, a `chdir()` call must be made, even if it is to the current directory name as given by `getcwd()`. This is especially relevant when mounting a filesystem on '/' as the current working directory is usually also '/' .

Normally you can access files and directories with both absolute paths (for example '`/fs/dir1/file1.txt`') or paths relative to the current working directory (for example '`./dir1/file1.txt`' or just '`dir1/file1.txt`'). As a special exception, you cannot use relative paths if your current working directory is the root directory '/', unless there is a filesystem mounted directly on '/'. This is a deliberate simplification due to the fact that the current working directory is not really a valid directory and there is no true filesystem at '/' to navigate within. Instead it is recommended to either change directory into a filesystem after mounting using `chdir()`, use absolute paths, or mount a filesystem at '/'.

It should also be noted that there is no requirement for there to be a directory entry for a filesystem mount point if mounted within another filesystem. So for example, there need not be a directory named “`/dev`” in the directory list of “`/`” even though there is a filesystem mounted on “`/dev`”.

Unmounting a filesystem is done by the `umount()` function. This can unmount filesystems whether they were mounted statically or dynamically.

The `umount()` function has the following prototype:

```
int umount( const char *name );
```

The mount table is searched for a match between the *`name`* argument and the entry *`name`* field. When a match is found the filesystem's `umount()` operation, with the *`force`* argument set to `false` is called and if successful, the mount table entry is invalidated by setting its *`valid`* field false and the *`name`* field to NULL.

There is also an `umount_force()` function with the following prototype:

```
int umount_force( const char *name );
```

The main difference between this and the standard `umount()` function is that it forces the filesystem to be unmounted. In the FILEIO package this means that all open files will be forced to close, the current directory will be moved away from the filesystem if it points to it and any threads waiting for access to the filesystem will be forced to return. In general, any buffered data not yet written to the medium will be lost; such buffering may take place in libraries like C standard I/O, C++ streams or the filesystem itself. If the programmer wishes for buffered data to be committed beforehand, they must use whatever mechanism has been provided by the layers performing the buffering. This is not always the case however, such as if the reason to force an unmounting is because the medium has been removed. When the filesystem's `umount()` function is called, the *`force`* argument will be set `true`, and the filesystem should take steps to free all resources and detach from the underlying device.

Care must be taken if mounting a filesystem on “`/`” as it will not be possible to unmount the filesystem later if it is in use as the current working directory. Instead it will be necessary to change directory to a different filesystem before unmounting.

Chapter 27. Automounter

Where removable media is supported by the filesystem and the hardware device driver (currently only the FAT filesystem and the eCosPro USB mass storage device driver have this support) it is possible to configure an automounter which will automatically mount any filesystems found on any device that is inserted. It will also automatically unmount the filesystem when the device is removed.

The automounter is controlled by a number of configuration options:

`CYGPKG_IO_FILEIO_AUTOMOUNT`

> This option enables the eCos automounter. It is only active if there are device drivers present that are capable of dealing with removable media.
>
> Default value: 0

`CYGDAT_IO_FILEIO_AUTOMOUNT_ROOT`

> Any automounted filesystems will be mounted under this root directory.
>
> Default value: "/auto"

`CYGDAT_IO_FILEIO_AUTOMOUNT_DEVICES`

> This option is a list of device names of the devices that will be monitored by the automounter. Each entry is two strings within braces, with separate entries separated by commas. The first string gives the device name, the second the stub for making the mount point name under the automount root. The name of the filesystem root will be manufactured by appending the disk number and partition number to the name stub, separated by underscores. For example with the default values typical filesystem root names might be: "/auto/usb_0_1" or "/auto/usb_1_2".
>
> Default value: { "/dev/usbmass/", "usb" }

The Automounter also defines a callback that may be used by applications to receive notifications that new filesystems have been mounted or unmounted. The `fileio.h` header contains the following definitions if the automounter is enabled:

```
typedef void cyg_automount_handler( int event, char *mountpoint, CYG_ADDRWORD data );
#define CYG_AUTOMOUNT_MOUNT     1
#define CYG_AUTOMOUNT_UMOUNT    2

__externC int cyg_automount_register_handler( char *devname,
                                              cyg_automount_handler *handler,
                                              CYG_ADDRWORD data );
```

The function `cyg_automount_register_handler()` causes the callback handler to be registered. The *`devname`* identifies the device to which the callback will be attached, it should match one of the device names defined in `CYGDAT_IO_FILEIO_AUTOMOUNT_DEVICES`. The *`handler`* argument is the callback function and *`data`* is a user defined data value.

When the handler is called, the *`event`* argument indicates the event being notified, `CYG_AUTOMOUNT_MOUNT` or `CYG_AUTOMOUNT_UMOUNT`. The *`mountpoint`* argument is the name of the root of the filesystem being notified, it will be composed as described above from `CYGDAT_IO_FILEIO_AUTOMOUNT_ROOT` and the stub part of the relevant `CYGDAT_IO_FILEIO_AUTOMOUNT_DEVICES` entry. The *`data`* argument is the data value passed in from `cyg_automount_register_handler()`.

The handler will be called by the automounter just after the filesystem has been mounted, or just before it is unmounted. Application code should avoid running too much code in the handler and offload long running tasks to another thread. This is because the handler is called directly from the automounter thread and while it is executing, no other automount operations can be run.

Chapter 28. Sockets

If a network stack is present, then the FILEIO infrastructure also provides access to the standard BSD socket calls.

The netstack table contains entries which describe the network protocol stacks that are in the system image. Each resident stack should export an entry to this table using the `NSTAB_ENTRY()` macro.

Each table entry has the following structure:

```
struct cyg_nstab_entry
{
    cyg_bool            valid;          // true if stack initialized
    cyg_uint32          syncmode;       // synchronization protocol
    char                *name;          // stack name
    char                *devname;       // hardware device name
    CYG_ADDRWORD        data;           // private data value

    int     (*init)( cyg_nstab_entry *nste );
    int     (*socket)( cyg_nstab_entry *nste, int domain, int type,
    int protocol, cyg_file *file );
};
```

This table is analogous to a combination of the filesystem and mount tables.

The *`valid`* field is set `true` if the stack's `init()` function returned successfully and the *`syncmode`* field contains the `CYG_SYNCMODE_SOCK_*` bits described above.

The *`name`* field contains the name of the protocol stack.

The *`devname`* field names the device that the stack is using. This may reference a device under "/dev", or may be a name that is only meaningful to the stack itself.

The `init()` function pointer is called during system initialization to start the protocol stack running. If it returns non-zero the *`valid`* field is set false and the stack will be ignored subsequently.

The `socket()` function is called to attempt to create a socket in the stack. When the `socket()` API function is called the netstack table is scanned and for each valid entry the `socket()` function pointer is called. If this returns non-zero then the scan continues to the next valid stack, or terminates with an error if the end of the table is reached.

The result of a successful socket call is an initialized file object with the *`f_xops`* field pointing to the following structure:

```
struct cyg_sock_ops
{
    int (*bind)      ( cyg_file *fp, const sockaddr *sa, socklen_t len );
    int (*connect)   ( cyg_file *fp, const sockaddr *sa, socklen_t len );
    int (*accept)    ( cyg_file *fp, cyg_file *new_fp,
                       struct sockaddr *name, socklen_t *anamelen );
    int (*listen)    ( cyg_file *fp, int len );
    int (*getname)   ( cyg_file *fp, sockaddr *sa, socklen_t *len, int peer );
    int (*shutdown)  ( cyg_file *fp, int flags );
    int (*getsockopt)( cyg_file *fp, int level, int optname,
                       void *optval, socklen_t *optlen);
    int (*setsockopt)( cyg_file *fp, int level, int optname,
                       const void *optval, socklen_t optlen);
    int (*sendmsg)   ( cyg_file *fp, const struct msghdr *m,
                       int flags, ssize_t *retsize );
    int (*recvmsg)   ( cyg_file *fp, struct msghdr *m,
                       socklen_t *namelen, ssize_t *retsize );
};
```

It should be obvious from the names of these functions which API calls they provide support for. The `getname()` function pointer provides support for both `getsockname()` and `getpeername()` while the `sendmsg()` and `recvmsg()` function pointers provide support for `send()`, `sendto()`, `sendmsg()`, `recv()`, `recvfrom()` and `recvmsg()` as appropriate.

Chapter 29. Select

The infrastructure provides support for implementing a select mechanism. This is modeled on the mechanism in the BSD kernel, but has been modified to make it implementation independent.

The main part of the mechanism is the `select()` API call. This processes its arguments and calls the `fo_select()` function pointer on all file objects referenced by the file descriptor sets passed to it. If the same descriptor appears in more than one descriptor set, the `fo_select()` function will be called separately for each appearance.

The *`which`* argument of the `fo_select()` function will either be `CYG_FREAD` to test for read conditions, `CYG_FWRITE` to test for write conditions or zero to test for exceptions. For each of these options the function should test whether the condition is satisfied and if so return true. If it is not satisfied then it should call `cyg_selrecord()` with the *`info`* argument that was passed to the function and a pointer to a cyg_selinfo structure.

The cyg_selinfo structure is used to record information about current select operations. Any object that needs to support select must contain an instance of this structure. Separate cyg_selinfo structures should be kept for each of the options that the object can select on - read, write or exception.

If none of the file objects report that the select condition is satisfied, then the `select()` API function puts the calling thread to sleep waiting either for a condition to become satisfied, or for the optional timeout to expire.

A selectable object must have some asynchronous activity that may cause a select condition to become true - either via interrupts or the activities of other threads. Whenever a selectable condition is satisfied, the object should call `cyg_selwakeup()` with a pointer to the appropriate cyg_selinfo structure. If the thread is still waiting, this will cause it to wake up and repeat its poll of the file descriptors. This time around, the object that caused the wakeup should indicate that the select condition is satisfied, and the `select()` API call will return.

Note that `select()` does not exhibit real time behaviour: the iterative poll of the descriptors, and the wakeup mechanism mitigate against this. If real time response to device or socket I/O is required then separate threads should be devoted to each device of interest and should use blocking calls to wait for a condition to become ready.

Chapter 30. Devices

Devices are accessed by means of a pseudo-filesystem, "devfs", that is mounted on "/dev". Open operations are translated into calls to `cyg_io_lookup()` and if successful result in a file object whose *`f_ops`* functions translate filesystem API functions into calls into the device API.

Chapter 31. Writing a New Filesystem

To create a new filesystem it is necessary to define the fstab entry and the file IO operations. The easiest way to do this is to copy an existing filesystem: either the test filesystem in the FILEIO package, or the RAM or ROM filesystem packages.

To make this clearer, the following is a brief tour of the FILEIO relevant parts of the RAM filesystem.

First, it is necessary to provide forward definitions of the functions that constitute the filesystem interface:

```
//==========================================================================
// Forward definitions

// Filesystem operations
static int ramfs_mount    ( cyg_fstab_entry *fste, cyg_mtab_entry *mte );
static int ramfs_umount   ( cyg_mtab_entry *mte, cyg_bool force );
static int ramfs_open     ( cyg_mtab_entry *mte, cyg_dir dir, const char *name,
                            int mode,  cyg_file *fte );
static int ramfs_unlink   ( cyg_mtab_entry *mte, cyg_dir dir, const char *name );
static int ramfs_mkdir    ( cyg_mtab_entry *mte, cyg_dir dir, const char *name );
static int ramfs_rmdir    ( cyg_mtab_entry *mte, cyg_dir dir, const char *name );
static int ramfs_rename   ( cyg_mtab_entry *mte, cyg_dir dir1, const char *name1,
                            cyg_dir dir2, const char *name2 );
static int ramfs_link     ( cyg_mtab_entry *mte, cyg_dir dir1, const char *name1,
                            cyg_dir dir2, const char *name2, int type );
static int ramfs_opendir  ( cyg_mtab_entry *mte, cyg_dir dir, const char *name,
                            cyg_file *fte );
static int ramfs_chdir    ( cyg_mtab_entry *mte, cyg_dir dir, const char *name,
                            cyg_dir *dir_out );
static int ramfs_stat     ( cyg_mtab_entry *mte, cyg_dir dir, const char *name,
                            struct stat *buf);
static int ramfs_getinfo  ( cyg_mtab_entry *mte, cyg_dir dir, const char *name,
                            int key, void *buf, int len );
static int ramfs_setinfo  ( cyg_mtab_entry *mte, cyg_dir dir, const char *name,
                            int key, void *buf, int len );

// File operations
static int ramfs_fo_read      (struct CYG_FILE_TAG *fp, struct CYG_UIO_TAG *uio);
static int ramfs_fo_write     (struct CYG_FILE_TAG *fp, struct CYG_UIO_TAG *uio);
static int ramfs_fo_lseek     (struct CYG_FILE_TAG *fp, off_t *pos, int whence );
static int ramfs_fo_ioctl     (struct CYG_FILE_TAG *fp,
                               CYG_ADDRWORD com,
                               CYG_ADDRWORD data);
static int ramfs_fo_fsync     (struct CYG_FILE_TAG *fp, int mode );
static int ramfs_fo_close     (struct CYG_FILE_TAG *fp);
static int ramfs_fo_fstat     (struct CYG_FILE_TAG *fp, struct stat *buf );
static int ramfs_fo_getinfo   (struct CYG_FILE_TAG *fp, int key, void *buf, int len );
static int ramfs_fo_setinfo   (struct CYG_FILE_TAG *fp, int key, void *buf, int len );

// Directory operations
static int ramfs_fo_dirread   (struct CYG_FILE_TAG *fp, struct CYG_UIO_TAG *uio);
static int ramfs_fo_dirlseek  (struct CYG_FILE_TAG *fp, off_t *pos, int whence );
```

We define all of the fstab entries and all of the file IO operations. We also define alternatives for the *fo_read* and *fo_lseek* file IO operations.

We can now define the filesystem table entry. There is a macro, FSTAB_ENTRY to do this:

```
//==========================================================================
// Filesystem table entries

// -------------------------------------------------------------------------
// Fstab entry.
// This defines the entry in the filesystem table.
// For simplicity we use _FILESYSTEM synchronization for all accesses since
// we should never block in any filesystem operations.

FSTAB_ENTRY( ramfs_fste, "ramfs", 0,
    CYG_SYNCMODE_FILE_FILESYSTEM|CYG_SYNCMODE_IO_FILESYSTEM,
    ramfs_mount,
    ramfs_umount,
    ramfs_open,
    ramfs_unlink,
    ramfs_mkdir,
    ramfs_rmdir,
    ramfs_rename,
    ramfs_link,
    ramfs_opendir,
    ramfs_chdir,
    ramfs_stat,
    ramfs_getinfo,
    ramfs_setinfo);
```

The first argument to this macro gives the fstab entry a name, the remainder are initializers for the field of the structure.

We must also define the file operations table that is installed in all open file table entries:

```
// -------------------------------------------------------------------------
// File operations.
// This set of file operations are used for normal open files.

static cyg_fileops ramfs_fileops =
{
    ramfs_fo_read,
    ramfs_fo_write,
    ramfs_fo_lseek,
    ramfs_fo_ioctl,
    cyg_fileio_seltrue,
    ramfs_fo_fsync,
    ramfs_fo_close,
    ramfs_fo_fstat,
    ramfs_fo_getinfo,
    ramfs_fo_setinfo
};
```

These all point to functions supplied by the filesystem except the *fo_select* field which is filled with a pointer to `cyg_fileio_seltrue()`. This is provided by the FILEIO package and is a select function that always returns true to all operations.

Finally, we need to define a set of file operations for use when reading directories. This table only defines the *fo_read* and *fo_lseek* operations. The rest are filled with stub functions supplied by the FILEIO package that just return an error code.

```
// -------------------------------------------------------------------------
// Directory file operations.
// This set of operations are used for open directories. Most entries
// point to error-returning stub functions. Only the read, lseek and
// close entries are functional.

static cyg_fileops ramfs_dirops =
{
    ramfs_fo_dirread,
    (cyg_fileop_write *)cyg_fileio_enosys,
    ramfs_fo_dirlseek,
    (cyg_fileop_ioctl *)cyg_fileio_enosys,
    cyg_fileio_seltrue,
    (cyg_fileop_fsync *)cyg_fileio_enosys,
    ramfs_fo_close,
    (cyg_fileop_fstat *)cyg_fileio_enosys,
    (cyg_fileop_getinfo *)cyg_fileio_enosys,
    (cyg_fileop_setinfo *)cyg_fileio_enosys
};
```

If the filesystem wants to have an instance automatically mounted on system startup, it must also define a mount table entry. This is done with the `MTAB_ENTRY` macro. This is an example from the test filesystem of how this is used:

```
MTAB_ENTRY( testfs_mte1,
      "/",
      "testfs",
      "",
      "",
      0);
```

The first argument provides a name for the table entry. The following arguments provide initialization for the *`name`*, *`fsname`*, *`devname`* *`options`* and *`data`* fields respectively.

These definitions are adequate to let the new filesystem interact with the FILEIO package. The new filesystem now needs to be fleshed out with implementations of the functions defined above. Obviously, the exact form this takes will depend on what the filesystem is intended to do. Take a look at the RAM and ROM filesystems for examples of how this has been done.

Part VIII. FAT File System Support

Table of Contents

Chapter 32. Introduction

This document describes the FAT filesystem provided in eCos. This is implemented by the FATFS package which uses the facilities of the FILEIO package to present its functionality to the user.

The FAT filesystem supports FAT12, FAT16 and FAT32 disk formats.

The FAT filesystem includes optional support for long file names. This functionality is covered by patents belonging to Microsoft in several territories, including Europe and the the USA. By default the long file name support in eCos (CYGCFG_FS_FAT_LONG_FILE_NAMES) is disabled. If you wish to enable this feature on products that are distributed within these territories then you may need to aquire a license from Microsoft.

Chapter 33. Configuring the FAT Filesystem

This chapter shows how to include the FAT filesystem into an eCos configuration and how to configure it once installed.

Including FAT Filesystem in a Configuration

The FAT filesystem is contained in a single eCos package, CYGPKG_FS_FAT. However, it depends on the services of a collection of other packages for complete functionality:

CYGPKG_IO_FILEIO

> The File IO package. This provides the POSIX compatible API by which the FAT filesystem is accessed.

CYGPKG_IO

> Device IO package. This provides all the infrastructure for the disk devices.

CYGPKG_IO_DISK

> Disk device IO support. This provides the top level generic disk driver functions. It also interprets partition tables and provides a separate access channel for each partition. This package is described in detail elsewhere.

CYGPKG_LINUX_COMPAT

> Linux compatibility library. The FAT filesystem only used the list and RBtree features of this library.

CYGPKG_LIBC_STRING

> Strings library. This provides the string and memory move and compare routines used by the filesystem.

CYGPKG_MEMALLOC

> The FAT filesystem currently uses malloc() to allocate its memory resources, such as the node and block caches, so this package is required.

To add the FAT filesystem to a configuration, it is necessary to add all of these packages. This is best done by using an import file. The following file will add the FAT filesystem and all the necessary packages to any configuration:

```
cdl_savefile_version 1;
cdl_savefile_command cdl_savefile_version {};
cdl_savefile_command cdl_savefile_command {};
cdl_savefile_command cdl_configuration { description hardware template package };
cdl_savefile_command cdl_package { value_source user_value wizard_value inferred_value };
cdl_savefile_command cdl_component { value_source user_value wizard_value inferred_value };
cdl_savefile_command cdl_option { value_source user_value wizard_value inferred_value };
cdl_savefile_command cdl_interface { value_source user_value wizard_value inferred_value };

cdl_configuration eCos {
    package CYGPKG_FS_FAT current ;
    package CYGPKG_IO_DISK current ;
    package CYGPKG_LINUX_COMPAT current ;
    package CYGPKG_IO_FILEIO current ;
    package CYGPKG_IO current ;
    package CYGPKG_LIBC_STRING current ;
    package CYGPKG_MEMALLOC current ;
};
```

In addition to these packages, hardware-specific device driver packages will be needed for the disk devices to be used. These device drivers are usually part of the target description in the eCos database and will be enabled if the CYGPKG_IO_DISK package is included.

Configuring the FAT Filesystem

Once added to the configuration, the FAT filesystem has a number of configuration options:

CYGNUM_FS_FAT_NODE_HASH_TABLE_SIZE

> This option controls the number of slots in the hash table used to store file nodes using filenames as keys.
>
> Default value: 32

CYGNUM_FS_FAT_NODE_POOL_SIZE

> This option controls the size of the node pool used for storing file nodes. This value should be set to the maximum required number of simultaneously open files plus the desired size of unused node cache.
>
> Default value: CYGNUM_FILEIO_NFILE + 2

CYGNUM_FS_FAT_BLOCK_CACHE_BLOCKSIZE

> This option controls the size of blocks in the block cache. This value should be a power-of-2 multiple of 512. It must be at least as great as the underlying disk sector size (usually 512) but can be greater, allowing multiple underlying blocks to be cached within a single cache block.
>
> With some underlying disk devices, performance can be greatly improved by increasing the size of this option, as it may allow multiple disk blocks to be transferred in one transaction. This is known to be particularly true with MMC or SD card media and it is recommended to increase the size of this option with such media.
>
> Default value: 512

CYGNUM_FS_FAT_BLOCK_CACHE_MEMSIZE

> This option controls the amount of memory used for the block cache.
>
> Default value: 20 * CYGNUM_FS_FAT_BLOCK_CACHE_BLOCKSIZE

CYGDBG_FS_FAT_NODE_CACHE_EXTRA_CHECKS

> This option controls the inclusion of extra sanity checks in node cache code.
>
> Default value: 1

CYGCFG_FS_FAT_USE_ATTRIBUTES

> This option controls whether the FAT filesystem supports or honors the FAT filesystem file attributes.
>
> Default value: 0

CYGCFG_FS_FAT_LONG_FILE_NAMES

> This option controls the FAT filesystem support for long file names.
>
> Default value: 0

CYGNUM_FS_FAT_LONG_FILE_NAME_MAX

> This option defines the maximum size of long file names supported by the filesystem. The default value of 64 corresponds to NAME_MAX, which defines the size of d_name[] in a struct dirent.
>
> Default value: 64

CYGSEM_FS_FAT_ASYNC_IO

> Normally every operation on a mounted FAT filesystem must complete before another operation can start. However sometimes a thread must block while waiting for a read or write to the underlying disk medium to complete. As a performance improvement, this option allows other threads to perform operations on the FAT filesystem while other threads are blocked. This can be especially beneficial when the operations can be performed entirely using the in-built file system block and node cache.
>
> However, a known limitation is that there is not currently sufficient protection in place to handle situations where two threads update a directory at the same time, for example, both creating, renaming or removing files or directories (Issue #1001183). Other operations should be safe, but for the time being care is advised if enabling this option, and eCosCentric cannot provide support for resulting problems if it is enabled.
>
> Default value: 0

CYGSEM_FS_FAT_FORMAT

> This option enables support for formatting a FAT filesystem on the device before mounting it. Only FAT16 and FAT32 formats are supported, FAT12 is not. The variety of options supported is also limited to choosing the format, block size, and volume label, where appropriate.
>
> Default value: 1

Normally these options should be left as they are unless you have a specific need to change them. Once the configuration had been created, it should be possible to compile eCos and link it with the application without any errors.

Chapter 34. Using the FAT Filesystem

The FAT filesystem is accessed through the FILEIO package and responds to all the standard filesystem functions such as `open()`, `close()`, `read()` and `write()`. To use these operations the filesystem must first be mounted.

A FAT filesystem may be mounted using the `mount()` function. The following is an example of how to mount a FAT filesystem:

```
    err = mount( "/dev/hd0/1", "/disk0",  "fatfs:sync=write" );
```

This function call will mount the first partition of hard disk 0 (see the documentation on the DISKIO package for a full description of the device name format). The root of this disk can then be accessed with the name "/disk0". The `mount()` function will return zero if the mount succeeded, or -1 if it failed for any reason, for example if the partition does not exist, or the filesystem is not in FAT format.

The options after the colon in the filesystem name are passed to the filesystem to control various aspects of the filesystem. The options currently supported are:

sync
: This option controls the synchronization behaviour of the block cache. If omitted then the cache is run on an entirely write-back basis and blocks are only written back to disk when they need to be replaced with new data, when `sync()` is called, or the filesystem is unmounted. This is generally the most efficient mode, but is prone to losing data or corrupting the filesystem if power is lost while the filesystem is mounted.

  If this option is set to "write" then the cache is operated on a write-back basis and every block update is written immediately back to disk. This is the least efficient mode since any extension to a file may result in several blocks being written back to disk. It does, however, keep the filesystem up to date on disk.

  If this option is set to "close" then the cache is only written back to disk whenever a file is closed. Note that this causes the entire cache to be written, not just those blocks associated with the file being closed. In terms of efficiency, this is a good compromise between performance and safety.

readonly
: This is a stand-alone option which causes the filesystem to be mounted read-only. The effect of this is to prevent the filesystem writing anything back to the disk. Under normal circumstances this cannot be guaranteed for a normal mount, even when files are only read, since the filesystem may need to update the access time for files that have been read.

When finished with, a filesystem may be unmounted using the `umount()` function. The following would unmount the filesystem mounted above:

```
    err = umount( "/disk0" );
```

Warning!

It is important to unmount any removable devices before removing them, otherwise there is no guarantee that all cached data blocks will have been written to disk. The same is true of resetting the system before unmounting non-removable devices.

Chapter 35. Removable Media Support

The FAT filesystem has support for Removable Media, which is implemented in conjunction with support in the FILEIO package, the generic DISKIO layer and the hardware disk driver. At present, the drivers which support this include the MMC/SD card driver in bus mode (not SPI mode), subject to the presence of underlying hardware support; and also the USB mass storage device driver.

To support Removable media, application code, or the automounter, should register a disk event callback to capture device insertion and removal. When an insert event is detected, an attempt to mount the filesystem should be made. If this is successful, then the files on the device can be accessed.

When the device is removed the hardware driver will reject all IO operations with an EIO response. The filesystem will propagate these errors back to any current file IO operations which will result in `read()` or `write()` returning an EIO error. Application code should be ready for this to happen.

A disk event callback will also be delivered and the application should arrange to call `umount_force()` to force the filesystem to be unmounted. The FAT filesystem handles this by releasing all resources and detaching from the disk device.

To help applications indicate to the user whether the medium may be removed (by means of a LED or an on-screen icon) the FAT filesystem supports the filesystem callback mechanism defined in the FILEIO package. This relies on the following definitions in `fileio.h`:

```
typedef void cyg_fs_callback( cyg_int32 event, CYG_ADDRWORD data );

struct cyg_fs_callback_info
{
    cyg_fs_callback     *callback;      // Callback function
    CYG_ADDRWORD        data;           // User data value
};

// Callback events
#define CYG_FS_CALLBACK_SAFE    1       // The filesystem is up to date on disk
#define CYG_FS_CALLBACK_UNSAFE  2       // The filesystem is not up to date
```

A callback function may be registered after a filesystem has been mounted by using `cyg_fs_setinfo()` as follows:

```
struct cyg_fs_callback_info callback;

…

callback.callback = fs_callback;
callback.data = my_data;
err = cyg_fs_setinfo("/disk0", FS_INFO_CALLBACK, &callback, sizeof(callback));
```

Following this, whenever the filesystem has dirty cache blocks that are not up to date on the disk, `fs_callback()` will be called with *`event`* set to `CYG_FS_CALLBACK_UNSAFE`. When all blocks become up to date on disk it will be called with `CYG_FS_CALLBACK_SAFE`.

Chapter 36. Non-ASCII Character Set Support

If long filename support is enabled (by setting `CYGCFG_FS_FAT_LONG_FILE_NAMES`) then all strings passed to and from the filesystem may be encoded using UTF-8. This allows files to be named using characters beyond the basic ASCII set.

If long filename support is disabled then file names are limited to the standard 8.3 format. However, these file names are preserved in an 8-bit clean format, if they contain non-ASCII characters, so that any multi-byte encodings are preserved.

Filesystems created by devices that do not support long filenames may have 8.3 names that are encoded using non-ASCII and non-Unicode character sets. Typically these will be encoded according to Microsoft code page character sets. To permit these names to pass through the rest of the filesystem, and compare correctly during file searches, when long filename support is enabled, these names need to be translated into Unicode. Since the filesystem has no built-in internationalization support, beyond Unicode, it is the responsibility of the application or middleware layers to supply the translation of these values to and from Unicode. The FILEIO package defines callbacks that may be used to do this:

```
typedef int cyg_fs_mbcs_to_utf16le( CYG_ADDRWORD data,
                                    const cyg_uint8 *mbcs,
                                    int size,
                                    cyg_uint16 *utf16le);

typedef int cyg_fs_utf16le_to_mbcs( CYG_ADDRWORD data,
                                    const cyg_uint16 *utf16le,
                                    int size,
                                    cyg_uint8 *mbcs);

struct cyg_fs_mbcs_translate
{
    cyg_fs_mbcs_to_utf16le      *mbcs_to_utf16le;
    cyg_fs_utf16le_to_mbcs      *utf16le_to_mbcs;
    CYG_ADDRWORD                data;
};
```

These callback functions may be registered after a filesystem has been mounted by using `cyg_fs_setinfo()` as follows:

```
struct cyg_fs_mbcs_translate translate;

…

translate.mbcs_to_utf16le = my_mbcs_to_utf16le;
translate.utf16le_to_mbcs = my_utf16le_to_mbcs;
translate.data = (CYG_ADDRWORD)my_data;
err = cyg_fs_setinfo("/disk0", FS_INFO_MBCS_TRANSLATE, &translate, sizeof(translate));
```

Following this, whenever the filesystem encounters a short file name that contains non-ASCII characters the registered `mbcs_to_utf16le()` function will be called to translate it. In the call, the *`data`* argument will be a copy of the *`data`* field of the cyg_fs_mbcs_translate structure. The *`mbcs`* argument points to the sequence of *`size`* bytes to be translated. The resulting translation should be stored in *`utf16le`* and the number of 16-bit values stored returned from the function.

When the filesystem needs to encode a string into the multibyte character set, it will call the `utf16le_to_mbcs()` function. In the call, the *`data`* argument will be a copy of the *`data`* field of the cyg_fs_mbcs_translate structure. The *`utf16le`* argument points to the sequence of *`size`* 16-bit values to be translated. The resulting translation should be stored in *`mbcs`* and the number of bytes stored returned from the function.

It is important to note that translation is to and from UTF-16LE. All 16 bit values are stored in little endian byte order and Unicode code points outside the Basic Multilingual Plane are encoded as surrogate pairs. This is the format mandated by Microsoft for long file names in the FAT filesystem. See IETF RFC2781 for details of the encoding.

In the current implementation the `utf16le_to_mbcs()` will not be called. If long filename support is disabled, then the filesystem will store multibyte characters as they are supplied. If long filename support is enabled then new files will be created with long names if any non-ASCII characters are present. Renamed files will be converted to the long name form automatically. This function is present in case future enhancements require it. For now applications should install a function that simply returns zero.

Chapter 37. Formatting Support

If the option CYGSEM_FS_FAT_FORMAT is enabled, the filesystem is able to format the chosen volume with a FAT filesystem while mounting it. This is controlled by a number of options passed to the filesystem through the mount function.

format
: This option enables formatting. If this is present without an argument, then the volume will only be formatted if the file system cannot be mounted. With an argument of force the volume will be formatted regardless of its current contents. Without this option present, the following options will be ignored.

fat16
: This option forces the file system to be formatted using FAT16.

fat32
: This option forces the file system to be formatted using FAT32. If neither the fat16 or fat32 options are provided, the formatter will choose the format based on the size of the volume. At present, any volume less than 512MiB in size will be formatted as FAT16, all others will be FAT32.

clsize
: This option takes an argument giving the size of each cluster in KiB. It should be a power of 2. If not supplied, the formatter will choose a cluster size based on the size of the volume.

label
: This option takes a name which should be up to 11 characters in length. This is used to set the volume label in the boot block and to create a volume label entry in the root directory. If not supplied a default volume label will be set (currently "eCos DISK").

A somewhat contrived example of a mount call with the format option might be as follows:

```
err = mount("/dev/hd0/1", "/disk0", "fatfs:format=force,fat16,clsize=8,label=My Volume");
```

Under normal circumstances, just giving the format option should be sufficient for most purposes:

```
err = mount("/dev/hd0/1", "/disk0", "fatfs:format");
```

This will cause the volume to be formatted according to its size, but only if it is not already mountable.

Chapter 38. Testing

There are a number of tests available for the FAT filesystem. These test various aspects of the FAT file system from basic operation to support for long file names, Unicode and code page encodings, performance, synchronization and formatting.

An important feature of these tests is that they use the `format` mount option to auto-format any volume that cannot be mounted. Therefore it is advisable not to run these tests on any device that contains data that should be retained.

Testing the FAT filesystem depends on the availability of a suitable device to perform the tests on. The following configuration options are defined in the target specific disk device driver to configure the tests for the available hardware:

CYGDAT_DEVS_DISK_TEST_DEVICE

> Device name of test disk or partition. This device will be mounted on the mountpoint given in `CYGDAT_DEVS_DISK_TEST_MOUNTPOINT` and tests carried out in the directory given by `CYGDAT_DEVS_DISK_TEST_DIRECTORY`.

CYGDAT_DEVS_DISK_TEST_MOUNTPOINT

> Mountpoint for test disk.

CYGDAT_DEVS_DISK_TEST_DIRECTORY

> Subdirectory in test device where tests can create files and directories.

CYGDAT_DEVS_DISK_TEST_DEVICE2

> Device name of optional second test disk or partition. If this is not defined then the tests will carry out any operations that would have been executed on the second disk in the test directory on the main test disk.

CYGDAT_DEVS_DISK_TEST_MOUNTPOINT2

> Mountpoint for optional second test disk. If `CYGDAT_DEVS_DISK_TEST_DEVICE2` is not defined then this option is not needed.

CYGDAT_DEVS_DISK_TEST_DIRECTORY2

> Subdirectory in optional second test device where tests can create files and directories. If `CYGDAT_DEVS_DISK_TEST_DEVICE2` is not defined then this option is not needed.

Part IX. Multimedia File System

Important

This eCosPro-MMFS Middleware package is STRICTLY LICENSED FOR NON-COMMERCIAL PURPOSES ONLY. It may not be used for Commercial purposes in full or in part in any format, including source code, binary code and object code format.

A Commercial eCosPro License version 3 (or above) which explicity includes this Middleware Package is required for Commercial use.

Table of Contents

Chapter 39. Introduction

This document describes the eCosPro Multimedia Filesystem (eCosPro-MMFS) provided with eCosPro. This is implemented by the MMFS package which uses the facilities of the FILEIO package to present its functionality to the user.

MMFS is intended to support continuous media intensive applications such as Personal Video Recorders, Video JukeBoxes and Video-on-Demand. It is designed to support the recording and playback of data streams at constant and variable rates.

MMFS was designed with the following goals:

- Provide PVR functionality allowing several data streams to be recorded simultaneously while also replaying a stream, which may be one of the streams being recorded. Provide the ability to fast-forward and rewind data streams.
- Make efficient use of disk storage, access times and bandwidth.
- Automatic recovery of disk data structures on restart after a power failure or other interruption. Automatic formatting of a new disk or one that is irretrievably corrupt.

These goals are achieved by simplifying the filesystem as much as possible. So, for example, there is only one directory that contains all files; file metadata is permanently allocated and of fixed size; caches are small and special-purpose. Many aspects are configurable so that the filesystem may be tuned to the application.

Chapter 40. Disk Data Structure

A disk consists of a linear array of 512 byte sectors addressed by a sector number. MMFS aggregates these sectors into blocks which are typically 128, 256 or 512 KiB in size. These blocks are the basic unit of addressing and allocation for MMFS. File data is transferred between the disk and memory in whole blocks, but metadata is accessed in smaller segments.

This disk is divided into four areas, the directory, the freelist, the block allocation tables (BATs) and the data area. The following sections describe these in detail.

Directory

The directory occupies the first one or two blocks of the disk. It consists of an array of directory entries. Each directory entry contains the following fields:

type
: Entry type:

  `MMFS_TYPE_EMPTY` Unused entry. Available for allocation.

  `MMFS_TYPE_VOLUME` Volume label. The `data` field contains a volume label that describes the format of the filesystem.

  `MMFS_TYPE_FILE` File. A standard data file.

  `MMFS_TYPE_RESERVED` Reserved entry. An entry that exists only to occupy a directory slot. Used to protect the volume label against overwriting while updating adjacent entries.

bat
: The block number of the Block Allocation Table for this file.

size
: File size in bytes. For streamed files this reflects the number of data blocks in the BAT. For random access files this is the offset of the last byte of the last block in the BAT.

created
: Creation time. A timestamp in seconds since the epoch implemented by the system wallclock. If no wallclock is present then this merely records the time since the last system restart.

state
: File state. This records the state of the file and aids in system recovery. The possible states are:

  `MMFS_STATE_CREATING`: The file is open for creation and is being actively extended.

  `MMFS_STATE_CREATED`: The file has been closed and will no longer be extendable. However, random access files may be extended in this state.

  `MMFS_STATE_DELETING`: The file is being deleted.

checksum
: Checksum over directory entry. This ensures that the directory entry is correct and consistent.

data
: Per-entry data. The contents of this field depend on the entry type. For the volume label it contains the filesystem format parameters. For files it may contain user-specified data. It is unused in other entry types.

name
: File name. A zero terminated string naming the file. This field is 64 bytes long so, with zero termination, filenames may be a maximum of 63 bytes. There are no limits on the characters allowed.

The data field of a volume label contains the disk format parameters. This consists of the following sub-fields:

signature1
: Volume label signature. This is used, together with `signature2`, to ensure that this is a valid volume label. If these two fields do no contain the expected values then the disk is presumed to be new or corrupt and the filesystem will reformat it.

version
Filesystem version number. Together with the revision number this is used to determine which version of the filesystem formatted this disk.

revision
Filesystem revision number.

sector_size
The size of sectors on this disk. This should match the sector size reported by the disk device itself. At present only sectors of 512 bytes are supported.

phys_block_size
Physical block size. This is the size of the physical blocks supported by this disk. This may differ from the sector size in some cases.

block_size
Size of MMFS blocks in sectors.

disk_size
Total number of blocks on the disk. If the number of sectors on the disk is not an exact multiple of the block_size then the last partial block will be unused.

rootdir_size
Size of the directory in blocks.

freelist_start
Block address of the first block of the freelist. This will be just after the directory.

freelist_size
Size of the freelist in blocks. This is calculated from `disk_size` so that the freelist is large enough to contain all the blocks on the disk.

bat_size
Size of each Block Allocation Table in blocks. The number of blocks per BAT is set during the formatting process.

bat_count
Number of BATs. The number of BATs is set during the formatting process and defaults to 200.

direntry_size
Size of a directory entry in bytes. This is currently fixed at 256 bytes. It is present to permit changes to the directory entry size in the future.

name_size
Size of the name field in a directory entry. This is currently fixed at 64 bytes. It is present to permit changes to the directory entry size in the future.

data_size
Size of the data field in a directory entry. This is currently fixed at 160 bytes. It is present to permit changes to the directory entry size in the future.

signature2
Second signature word.

Free List

The free list occupies a whole number of blocks following the directory. It is viewed as an array of block numbers and is large enough to contain the number of every block on the disk, plus enough spare to make it up to a whole number of blocks.

The free list is organized as a circular list with a head and a tail. Blocks are allocated from the head and are freed to the tail. When they reach the top of the free list, the head and tail pointers wrap back around to zero. These pointers are not stored on the disk but are discovered each time the filesystem is mounted by scanning the free list.

The free list is organized in this way for several reasons. First, it separates block allocation from freeing. Allocations need to proceed at a rate determined by the streaming of data onto the disk. Blocks are only freed when a file is deleted, and can be handled as a background task. Second, the separation makes recovery of filesystem integrity simpler, since blocks will not get reused immediately they are freed. Third, blocks that are allocated together in a particular file will be returned to the free list together, preserving locality.

Block Allocation Tables

The BAT area follows the free list. The size and number of BATs is defined when the filesystem is formatted. BATs are arrays of block addresses for the blocks that contain the data of the file. The number of BATs gives a hard upper limit to the number of files permitted. Usually this is set to equal the number of directory entries. There is little point in making it larger, but it may be useful to set it smaller if the minimal size of the directory exceeds the desired maximum number of files.

The size of each BAT represents a hard upper limit on the size of a file. BAT size should be set to cover the expected range of file sizes. Larger data sets can be handled at application level by splitting the data across several files.

Data Area

The last, and largest, area is the file data area. This comprises the rest of the disk following the last BAT. During formatting each block in this area is added to the free list.

Chapter 41. Runtime Filesystem Organization

This section covers the organization of the file system at runtime. MMFS is divided into a number of modules, each of which covers a specific area of functionality. The following sections cover these in detail.

FILEIO Interface

This module provides the interface to the FILEIO package to present a standard file system interface. This is achieved by exporting a filesystem table entry for the "mmfs" filesystem type. In fact two filesystems are exported, "mmfs" and "mmfs.format". These behave identically except that "mmfs.format" causes the filesystem to be reformatted as part of the mount operation.

File and Directory Handling

The directory module supports operations on the directory. It provides support for searching the directory for a given file, creation and deletion of entries and renaming entries.

A small cache of directory entries, called `dirnodes`, is maintained. This allows separate opens of the same file to share the directory entry and other information.

To allow easy location of unused directory entries, and to avoid searching free entries, the module maintains a bitmap of which directory entries are allocated. This map is constructed during the initial scan and maintained as entries are added and removed.

The file module supports the creation, deletion, reading and writing of the contents of a file. The standard file IO operations are supported together with streaming access. Each open file is accessed through a `file` object, which is also maintained by this module.

The block freelist is also managed by the file module, as is a bitmap recording the allocation state of all the BATs.

Caches

The filesystem has two caches. The metadata cache is used to cache portions of the directory, freelist and BATs. The data cache is used to contain blocks of file data. The two caches are identical other than that the metadata cache uses small (typically 4KiB) segments, while the data cache operates in terms of whole filesystem blocks. The caches also cause disk transfers originated from different caches to have different priorities.

The cache module exports a variety of functions for reading and writing directory entries in the directory, block numbers in the freelist and BATs, and for accessing file data. These functions perform the necessary translations into sector addresses and access the appropriate cache.

Disk Interface

The disk interface module provides support for handling transfers to and from the disk. It consists of a priority ordered queue of block descriptors plus a thread that picks the first descriptor off the queue and submits it to the disk device driver. The block descriptors used by the disk module are the same as those used by the caches.

Scan and Format

When a filesystem is mounted it performs a startup scan to determine the format of the disk and fix up any problems caused by any unexpected failures. The scan goes through the following steps:

- Scan the freelist looking for the head and tail offsets. Each block seen in the freelist is also recorded as having been seen and as being free.
- Scan the directory. For each entry, check that its checksum is correct. If not, mark the entry empty and correct the checksum. For each file, if it is in CREATING state, complete the operation by ensuring that each block in the BAT is not also in the freelist and changing its state to CREATED. If it is in DELETING state, complete the operation by returning all the blocks in its BAT to the freelist and deleting the directory entry.
- Scan the BATs of all files, recording that they have been seen and checking that they are not also in the freelist. Any block that is both in the freelist and a BAT is removed from the freelist.
- If any blocks have not yet been seen, then these orphaned blocks are inserted into the freelist.
- If any of the previous steps have updated the freelist, then the on-disk data structure is rebuilt. This has the side effect of sorting the freelist into block order, improving performance in future.

If the scan finds that the disk is corrupt or unformatted, or the filesystem has been mounted using the "mmfs.format" filesystem, then the disk is reformatted. Formatting consists of zeroing the directory and all the BATs, and building the freelist with all the blocks in the data area. Finally a volume label is written to the first entry in the directory.

Chapter 42. Configuration

Configuration Options

The format and footprint of the filesystem are controlled by a number of configuration options, described in the following sections.

General Options

The following options define the version and revision of the filesystem.

CYGNUM_FS_MMFS_VERSION

> This is the version of the filesystem supported.
>
> Default value: 1

CYGNUM_FS_MMFS_REVISION

> This is the revision of the filesystem supported.
>
> Default value: 0

Formatting Options

These options control the formatting of an MMFS disk. They are only used when a filesystem is formatted. Under normal circumstances the filesystem will fetch these values from the disk volume label.

CYGNUM_FS_MMFS_BLOCK_SIZE

> This option defines the size of filesystem blocks. The value is defined in KiB and must be a power of 2.
>
> Default value: 256

CYGNUM_FS_MMFS_ROOTDIR_SIZE

> This option defines the size of the root directory in blocks. Since all files are contained in this directory, its size gives a hard limit to the number of files that the filesystem may contain.
>
> Default value: 1

CYGNUM_FS_MMFS_BAT_SIZE

> This option defines the size of the Block Allocation Tables used to store the addresses of file data blocks. This gives a hard upper limit on the size of a file.
>
> Default value: 2

CYGNUM_FS_MMFS_BAT_COUNT

> This option defines the number of BATs allocated in the filesystem. The default is to define 200 BATs.
>
> Default value: 200

Footprint Options

These options control the memory footprint and other parameters for an active filesystem.

CYGNUM_FS_MMFS_FILE_COUNT

This option defines the maximum number of open files supported by the filesystem. This depends on the expected number of data streams, plus any random access files, that may be open simultaneously.

Default value: 4

CYGNUM_FS_MMFS_DIRNODE_COUNT

This option defines the maximum number of cached directory entries. At least one is required for each open file, plus a few for handling other filesystem operations such as renaming or deleting.

Default value: CYGNUM_FS_MMFS_FILE_COUNT+4

CYGNUM_FS_MMFS_MULTI_BUFFER

This defines the level of per-file multi-buffering. During streaming the filesystem will read ahead and write behind by this number of data blocks.

Default value: 2

CYGNUM_FS_MMFS_DATA_CACHE_SIZE

This defines the amount of memory occupied by the data cache. The default value is calculated from the multi-buffering level, and the number of files.

Default value: (CYGNUM_FS_MMFS_MULTI_BUFFER+2) * CYGNUM_FS_MMFS_FILE_COUNT * CYGNUM_FS_MMFS_BLOCK_SIZE

CYGNUM_FS_MMFS_META_CACHE_SIZE

This defines the amount of memory occupied by the metadata cache. The default value is calculated from the number of files plus an overhead to support the freelist and directory scanning.

Default value: (CYGNUM_FS_MMFS_FILE_COUNT+6) * CYGNUM_FS_MMFS_META_BLOCK_SIZE

CYGNUM_FS_MMFS_META_BLOCK_SIZE

This defines the size of a metadata cache block. These are used to contain portions of the the directory, freelist and BATs.

Default value: 4

CYGNUM_FS_MMFS_DISKIO_PRIORITY

This defines the priority of the disk IO thread. This thread should generally run at a high priority since it does very little but it is vital to the performance of the filesystem.

Default value: 4

CYGNUM_FS_MMFS_FLUSH_INTERVAL

This defines the interval at which metadata cache blocks are flushed. Each time this interval expires the oldest dirty block in the cache is written to disk. This allows dirty data to be trickled out to disk without severely impacting streaming transfers.

Default value: 10

CYGNUM_FS_MMFS_FLUSH_PERIOD

This defines the cache flush period in multiples of the cache flush interval. Each time this period expires, the entire metadata cache will be flushed to disk. The action of the flush interval will generally cause this operation to do nothing.

Default value: 6

Configuration Guidelines

This section attempts to give some guidelines about how to configure MMFS and the various tradeoffs that can be made.

Block Size

The choice of block size is the most important configuration option. The filesystem uses the large block size to amortize access time across large data transfers. The blocks also provide high locality for the data they contain, avoiding the need to implement complex localizing allocation and access mechanisms in the filesystem. The choice of block size depends on several factors: the access time and data transfer rate of the disk, the number and rate of the streams to be sustained.

The important disk performance factors to consider are the worst-case access time and the minimum sustained transfer rate. Disk manufacturers generally quote the average access time for disks and keep the worst case figures under wraps since they are often considerably higher than the average. Access time generally consists of seek time plus settling time plus rotational delay plus command submission overhead. The worst case seek time is generally a move from one edge of the disk to the other. Worst case rotational delay occurs when the target sector has just passed the head when it reached the destination cylinder; for a 7200RPM disk, this is 8.3ms. Settling and command time tend to be constant, although if the access includes a head switch then there may be a small contribution from that. As a rule of thumb, worst case access time can be taken to be about 3 times the manufacturers quoted average access time.

The sustained transfer rate for a disk varies across the disk with the differences in recording density due to zoning. Most current disks have 10 or more zones. The best data rate comes from the outer zones, and the worst from the inner zones and may differ by several MiB/s. Transfer of data to or from the disk will also incur head change and single cylinder seek delays for large multi-sector transfers. Another factor that contributes to the transfer rate is the speed with which data can be transferred across the disk interface. This will depend on things like the DMA modes supported by the disk and the host interface, cable design, cache and MMU factors. Embedded systems often do not have the kind of high performance interfaces that are common on data-centre servers.

A standard definition TV stream uses a data rate of 4-10Mb/s. An HDTV stream can run up to 27Mb/s, although current systems only run at 14 to 17 Mb/s. These are encoded using MPEG-2, which provides a highly variable data rate depending on source and contents between 2 and 14Mb/s.

To see what effect different block sizes have on throughput, let us consider an 8.2Mb/s stream, which conveniently approximates to 1MiB/s. The disk is assumed to spin at 7200 RPM, have a worst case access time of 30ms and a worst case sustained transfer rate of 20MiB/s. If this disk is formatted with 256KiB blocks, then the time to fetch one block is 42.5 ms (30ms worst case access time plus 12.5ms worst case transfer time). One second's worth of data is four blocks, taking 170ms. If the disk is formatted with 64KiB blocks, then the time to fetch one block is 33.125ms (30ms worst case access time plus 3.125ms worst case transfer time). One second's worth of data is sixteen blocks, taking 530ms.

From this we can see that using 256KiB blocks, we have enough throughput on this disk to run five or six 1MiB/s streams, but with 64KiB blocks there is barely the capacity for running two streams. The figures used here are worst case times, and on average the disk will be able to sustain more streams and higher data rates. However, if guarantees are to be met for glitch-free recording and playback, it is necessary to calculate for the most demanding scenario where seek distance, rotational delay and stream data rate conspire to make things difficult, even if such situations are rare and transient in real life.

BAT Size

The size of the Block Allocation Tables determines the amount of data that can be recorded in a single file. If the disk is formatted with a 256KiB block size a single block will contain 64Ki block addresses, which, at 1MiB/s, will record 16Ki seconds of data, or about 4.5 hours. This is sufficient for most PVR applications where most recordings are 30 minutes or an hour. It even accommodates most movies and sporting events. Increasing the BAT size to two or more blocks will allow longer recordings to be made in a single file, but at the expense of wasting space in the common case. An alternative approach would be to record a single stream in multiple files at the application level.

The number of BATs is also an important factor to consider, and is linked to the directory size. This relationship will be described in the next section. However, an important factor in choosing the size and number of BATs is the time taken to format the disk and perform filesystem startup. During formatting all the BATs must be zeroed, something that can take a long time if they are large an numerous. During filesystem startup, all BATs allocated to current files are scanned to detect orphaned blocks. The time taken to do this is proportional to the size of the BATs and the number of files.

Directory Size

The size of the directory provides one of the limits on how many files may be stored in the filesystem. The directory occupies a whole number of blocks, and with 256KiB blocks and 256 byte directory entries, each directory block can contain 1024 entries. This may be more than enough for most purposes: on a 160GB disk this averages to about 160MB per file, or 2m40s at 1MiB/s. Another way of looking at this is that a 160GB disk can contain about 40 hours of recorded TV, or about 80 30 minute programs. In this context, 1024 entries is more than adequate.

The other limit on the number of files is the number of BATs. These are allocated dynamically to files as they are created. Running out of BATs will cause file creation to fail, even if there are directory entries free. Having more BATs that directory entries is wasteful. Even having the same number, given the calculation above, can be seen as excessive. For a 160GB disk, about 200 BATs would be a more suitable figure.

Cache Sizes

The filesystem contains two caches: a metadata cache for the directory, freelist and BATs; and a data cache for file contents. The number of blocks in each cache is important to the correct functioning of the filesystem. Too many blocks and the filesystem occupies too much RAM. Too few blocks and data may be evicted from the cache too soon and result in performance problems.

The size of the metadata cache depends on the free list, the number of open files and any directory searches that are being made. The free list requires two cache blocks, one for the head and one for the tail. Each open file needs a block to contain the current read or write position in the BAT and, occasionally, an extra block to handle the prefetch of the next block in the BAT. Concurrent directory searches also consume metadata cache blocks. The default size of the metadata cache is therefore set to use two blocks for the free list, plus one for each possible open file, plus four to take up the prefetches and searches.

The size of the data cache depends only on the maximum number of open files. For each file we need a buffer for each level of multi-buffering, plus two to support the read-ahead or write-behind.

Chapter 43. Usage

MMFS is accessed through the FILEIO package which presents a standard POSIX compatible IO interface through which applications use standard `open()`, `read()`, `write()` and `close()` calls. Streaming support is provided through a small library, mmfslib, that presents a more application-friendly interface.

FILEIO Interface

MMFS supplies most of the standard file IO functionality. However, since it is optimized for supporting streamed data, it has a number of restrictions that mean that it does not always behave like a general-purpose filesystem.

- Files may not be resized after creation and are essentially write-once/read-many. Between the initial `open()` and `close()` that creates a file it will be extended as requires. On subsequent opens, even those that specify `O_WRITE`, data may only be written to the existing file extent.
- If an attempt is made to create a file that already exists, the `open()` will fail. Instead the file must be deleted first and may then be created anew.
- Creating a file with `O_EXCL` will always fail.
- If an attempt is made to rename a file to a filename that already exists, the `rename()` will fail, rather than overwriting the destination file. This includes attempting to rename the file to its own name. Instead the destination file must be deleted first.

MMFSLib

MMFS provides a simple library for handling streamed data. This comprises a small set of API function. The following sections describe the API, followed by a simple example.

MMFSLib API

The following functions are supported.

`int mmfs_stream_open( const char *path, int flags);`

Open an MMFS file for streaming. The *`flags`* argument is either `MMFS_FLAGS_READ` to open an existing file for reading or `MMFS_FLAGS_WRITE` to create a new file for writing. If the file doesn't exist (when opening for reading) or it does exist (when opening for writing) -1 will be returned and `errno` will be set to a suitable error code. On success a file descriptor is returned which may be used in other mmfslib calls, or in normal FILEIO calls.

The stream starts in `RANDOM` mode and must be set to streaming mode with a call to `mmfs_stream_set_mode()`.

`int mmfs_stream_info( int fd, mmfs_file_info *info );`

This function returns information about the file. The mmfs_file_info structure contains the following fields:

`buffer_size`

The size of data buffers that will be exchanged using `mmfs_stream_next_buffer()`.

`multi_buffers`

The number of buffers that the application may have in hand between calls to `mmfs_stream_next_buffer()`.

`file_size`

The current size of the file.

`max_size`

The maximum size the file may grow to.

`int mmfs_stream_set_mode( int fd, int mode, int stride );`

Set the file mode. The mode may be one of the following:

`MMFS_MODE_RANDOM`

This is the default mode. A file in this mode should be accessed using the standard filesystem API.

`MMFS_MODE_READ_FORWARD`

In this mode the file is being read forward. The *`stride`* argument defines the number of buffers skipped between each call to mmfs_stream_next_buffer(). A stride of 1 will read the whole file sequentially. A stride of 2 will read every other buffer; a stride of 3 will read every third buffer, and so on.

`MMFS_MODE_READ_BACKWARD`

This is similar to `MMFS_MODE_READ_FORWARD` except that the file is read backwards. The stride applies in exactly the same way except, obviously, the buffers supplied move progressively backwards through the file.

`MMFS_MODE_WRITE`

This sets the file up for streamed writing. The *`stride`* argument is not used and is forced to 1.

Mode changes take effect immediately and apply from the file's current location. A file only becomes capable of streaming after `mmfs_stream_set_mode()` has been called for the first time. Changing file mode during streaming may incur a performance penalty as new data blocks are fetched. A file may not be switched from a read mode to a write mode or vice versa.

`int mmfs_stream_get_mode( int fd, int *mode, int *speed );`

This function returns the mode and speed previously set by a call to `mmfs_stream_set_mode()`.

`int mmfs_stream_next_buffer( int fd, void **buffer );`

This function fetches the next stream buffer. The exact semantics of this function depend on the mode and the level of multi-buffering.

If the file has been set to one of the read modes, then each call returns the next buffer full of data from the file according to the direction and stride. The level of multi-buffering determines how many buffers the application may have in hand at any one time. For example, with a multi-buffering level of 2, the first two calls to this routine will return the first two buffers from the stream. The third call will return the third buffer, but will also cause the first buffer to become invalid and be returned to the filesystem for reuse. The fourth call will return the fourth buffer but will also invalidate the second buffer, and so on through the stream.

If the file has been set to the write mode, then each call returns an empty buffer for the application to fill with data. The multi-buffering level determines when the buffers will be written to the file. For example, with a multi-buffering level of 2, the first two calls will return an empty buffer each. The third call will cause the first buffer to be written to the file and will return a new empty buffer to replace it. The fourth call will cause the second buffer to be written to the file and a new buffer to be returned, and so on.

`int mmfs_stream_set_data( int fd, void *buffer );`

This function sets the per-directory entry data on the file. The `buffer` argument must point to `MMFS_DATASIZE` bytes of data that will be written into the directory entry.

`int mmfs_stream_get_data( int fd, void *buffer );`

This function reads the per-directory entry data on the file. The `buffer` argument must point to `MMFS_DATASIZE` bytes of memory that will be set to the data read from the directory entry.

`int mmfs_stream_close( int fd );`

This function closes the file. Any buffers still in possession of the application will be invalidated. If the file was open for writing the contents of these buffers will be written to the file.

Example

The following code provides a very simple example of how MMFSLib should be used. The code presented here is somewhat simplified and for clarity does not contain any error checking and recovery. It is assumed that the IO devices are accessed via a simple DMA interface; clearly real devices might be more complex than this.

First, a simple routine to stream data from a device to a file for a given duration:

```
static void write_stream( char *name, int duration )
{
    int i;
    int fd;
    int result;
    void *buffer;
    int bufno = 0;
    int buffer_size;
    mmfs_file_info info;
    cyg_tick_count end;

    // Open the stream for writing.
    fd = mmfs_stream_open( name, MMFS_OPEN_WRITE );

    // Get stream information, we are only interested in the buffer
    // size.
    result = mmfs_stream_info( fd, &info );

    buffer_size = info.buffer_size;

    // Set the stream into streamed write mode. The filesystem in now
    // ready to stream data to this file.
    result = mmfs_stream_set_mode( fd, MMFS_MODE_WRITE, 1 );

    // Convert duration from seconds to an absolute end time in system
    // ticks.
    end = cyg_current_time() + duration*ticks_per_second;

    // Prime the device with the first set of buffers, this will start
    // the DMA transfers going.
    for( i = 0; i < CYGNUM_FS_MMFS_MULTI_BUFFER; i++ )
    {
        result = mmfs_stream_next_buffer( fd, &buffer );

        dma_start( &input, bufno, DMA_READ, buffer, buffer_size );

        bufno++;
        if( bufno >= CYGNUM_FS_MMFS_MULTI_BUFFER ) bufno = 0;
    }
    // Wait for the first buffer to fill.
    dma_wait( &input, bufno );

    // Now stream to the file for the given duration.
    while( cyg_current_time() < end )
    {
        // Fetch a new buffer from MMFS. As a side effect this also
        // invalidates the oldest buffer, which will be the one that
        // has just finished its DMA transfer.
        result = mmfs_stream_next_buffer( fd, &buffer );

        // Set up a DMA transfer from the device
        dma_start( &input, bufno, DMA_READ, buffer, buffer_size );

        bufno++;
        if( bufno >= CYGNUM_FS_MMFS_MULTI_BUFFER ) bufno = 0;

        // Wait for the next device buffer to complete.
        dma_wait( &input, bufno );
    }

    // Wait for remaining buffers to finish
    for( i = 0; i < CYGNUM_FS_MMFS_MULTI_BUFFER; i++ )
    {
        bufno++;
        if( bufno >= CYGNUM_FS_MMFS_MULTI_BUFFER ) bufno = 0;

        dma_wait( &input, bufno );
    }

    // Finally close the stream.
    result = mmfs_stream_close( fd );
}
```

The code to read a stream is very similar, although this time it is parameterized by the required stride rather than the duration:

```
static void read_stream( char *name, int stride )
{
    int i;
    int fd;
    int result;
    void *buffer;
    int bufno = 0;
    int buffer_size;
    mmfs_file_info info;

    // Open the stream for reading.
    fd = mmfs_stream_open( name, MMFS_OPEN_READ );

    // Get stream information, we are only interested in the buffer
    // size.
    result = mmfs_stream_info( fd, &info );

    buffer_size = info.buffer_size;

    // Set the stream into streamed read mode using the given
    // stride. The filesystem will start preparation for streaming by
    // prefetching the first data blocks up to the multi-buffer limit.
    result = mmfs_stream_set_mode( fd, MMFS_MODE_READ_FORWARD, stride );

    // Prime the device with the first set of buffers, this will start
    // the DMA transfers going.
    for( i = 0; i < CYGNUM_FS_MMFS_MULTI_BUFFER; i++ )
    {
        result = mmfs_stream_next_buffer( fd, &buffer );

        dma_start( &output, bufno, DMA_WRITE, buffer, buffer_size );
        bufno++;
        if( bufno >= CYGNUM_FS_MMFS_MULTI_BUFFER ) bufno = 0;
    }

    // Wait for the first buffer to empty.
    dma_wait( &input, bufno );

    // Now stream from the file to the device until we reach the end
    // of the file.
    for(;;)
    {
        // Fetch a new buffer full of data from MMFS. As a side effect
        // this also invalidates the oldest buffer, which will be
        // the one that has just finished its DMA transfer.
        result = mmfs_stream_next_buffer( fd, &buffer );
        if( result < 0 && errno == EEOF )
            break;

        // Set up a DMA transfer to the device
        dma_start( &output, bufno, DMA_WRITE, buffer, buffer_size );

        bufno++;
        if( bufno >= CYGNUM_FS_MMFS_MULTI_BUFFER ) bufno = 0;

        // Wait for the next device buffer to complete.
        dma_wait( &output, bufno );
    }

    // Wait for remaining buffers to finish
    for( i = 0; i < CYGNUM_FS_MMFS_MULTI_BUFFER; i++ )
    {
        bufno++;
        if( bufno >= CYGNUM_FS_MMFS_MULTI_BUFFER ) bufno = 0;

        dma_wait( &output, bufno );
    }

    // Finally close the stream.
    result = mmfs_stream_close( fd );
}
```

The `example.c` test program contains versions of both of these routines.

Chapter 44. Testing

The MMFS package contains a number of test programs:

mmfs1
This program just tests the standard FILEIO interface of the filesystem. It is a simplified version of the filesystem functionality tests used by FATFS and RAMFS.

stream1
This program is a simple test of the streaming support. It writes and reads streams at defined data rates and checks that the rate is maintained and that data integrity is preserved.

pvr1
A basic emulation of a Personal Video Recorder. Two streams are written and one of them read back, after a delay. This simulates a PVR recording one channel while using a pause-live feature on a second channel.

pvr2
This program records a large number of short streams on the disk. This checks directory handling.

pvr3
This is a variant on pvr1, which records 3 streams while replaying one.

format
A simple test that uses the "mmfs.format" filesystem instead of "mmfs", thus reformatting the disk.

example
This contains versions of the `write_stream()` and `read_stream()` example functions described earlier, together with sufficient infrastructure to allow them to be run.

Part X. Disk IO Package

Table of Contents

Chapter 45. Introduction

This document describes the Disk I/O subsystem provided in eCos. This is implemented by the DISKIO package. This package presents a disk driver API to clients, interprets partition tables and provides the infrastructure in which hardware-specific disk device drivers operate.

Chapter 46. Configuring the DISK I/O Package

This chapter shows how to include the DISK I/O package an eCos configuration and how to configure it once installed.

Including DISK I/O in a Configuration

The DISK I/O subsystem is contained in a single eCos package, CYGPKG_IO_DISK. However, it depends on the services of the following other packages for complete functionality:

CYGPKG_IO

Device IO package. Disk devices operate within the generic device infrastructure.

CYGPKG_ERROR

The error package. Disk devices need to return standard error codes when operations fail.

The DISK I/O package can be added to any configuration just by adding its package during configuration. However, it is usually added as part of a group which also includes a filesystem implementation and hardware device drivers.

Configuring the DISK I/O Package

The option "Detect FAT boot sector" (`CYGSEM_IO_DISK_DETECT_FAT_BOOT`) can be used with certain types of removable media which are treated undeterministically by Windows PCs, such as USB keys. These usually contain a partition table, but if completely wiped, Windows will not recreate the partition table, and instead place a FAT filesystem directly on the device with no partition table. Enabling this option detects this and fabricates a partition table in memory instead. This option is usually only needed on systems which use removable media likely to be shared with Windows PCs, as indicated from low level drivers with the `CYGINT_IO_DISK_DETECT_FAT_BOOT_DEFAULT CDL` interface.

The CDL option `CYGSEM_IO_DISK_EFI_GPT` is used to support media formatted with an Extensible Firmware Interface (EFI) GUID Partition Table (GPT). This is limited to media presenting a single \"protective\" MBR partition entry covering the GPT managed space. Currently the feature is limited to providing the number of partitions supported by the parent filesystem package, and only those partitions accessible using 32-bit LBA values.

The CDL interface `CYGINT_IO_DISK_ALIGN_BUFS_TO_CACHELINE` indicates to the disk I/O package, and to its users, that data transfer buffers need to be aligned to data cache line boundaries.

The CDL interface `CYGINT_IO_DISK_REMOVABLE_MEDIA_SUPPORT` indicates that the hardware disk driver implements removable media support. This will cause the disk I/O package to implement the `CYG_IO_SET_CONFIG_DISK_EVENT` and `CYG_IO_GET_CONFIG_DISK_EVENT` configuration option keys.

Chapter 47. Usage

The user API for disk devices is the same as that for serial devices except that `cyg_io_bread()` and `cyg_io_bwrite()` are used for data transfers instead of `cyg_io_read()` and `cyg_io_write()`.

```
// Write data to a block device
Cyg_ErrNo cyg_io_bwrite(
    cyg_io_handle_t handle,
    const void *buf,
    cyg_uint32 *len,
    cyg_uint32 pos )
```

This function sends data to a device. The size of data to send is contained in `*len` and the actual size sent will be returned in the same place. This value must be a count of 512 byte sectors. The `pos` specifies the position on the disk to which the data will be written. It is a linear sector number.

```
// Read data from a block device
Cyg_ErrNo cyg_io_bread(
    cyg_io_handle_t handle,
    void *buf,
    cyg_uint32 *len,
    cyg_uint32 pos )
```

This function receives data from a device. The desired size of data to receive is contained in `*len` and the actual size obtained will be returned in the same place. This value must be a count of 512 byte sectors. The `pos` specifies the position on the disk from which the data will be read. It is a linear sector number.

Disk devices are named in the same way as other devices, thus "/dev/hd0" would name the first hard disk. The exact names used for any disk are usually part of the configuration for the target-specific device driver.

Disk devices may be partitioned using a standard DOS partition table. If this is the case then an additional element to the device name specifies the partition to be used. Thus "/dev/hd0/1" specifies partition 1, "/dev/hd0/2" partition 2, "/dev/hd0/3" partition 3 and "/dev/hd0/4" partition 4. The special name "/dev/hd0/0" specifies the entire disk without honoring the partition table. This is only really useful for accessing the master boot record in sector zero, to change the partitioning. Any other kind of access may corrupt the disk.

Application code may also install a disk event callback function. It does this using the `CYG_IO_SET_CONFIG_DISK_EVENT` configuration option using the `cyg_io_set_config()`. The buffer should point to an instance of the following structure:

```
typedef void disk_channel_event_t( cyg_uint32 event, cyg_uint32 devno, CYG_ADDRWORD data );

struct  cyg_disk_event_t
{
    disk_channel_event_t        *event;         // Event callback function
    CYG_ADDRWORD                event_data;     // Event callback user data
};

// Codes for disk_channel_event_t event argument.
#define CYG_DISK_EVENT_CONNECT          0x01
#define CYG_DISK_EVENT_DISCONNECT       0x02
```

The function should be installed on the base disk device driver (i.e. "/dev/hd0" rather than "/dev/hd0/0") and will be called when certain events occur. At present these are restricted to CONNECT and DISCONNECT events when a new disk is attached to, or detached from, the disk controller. Support for this depends on the ability of the underlying hardware device driver being able to detect these events.

A successful lookup of the base disk device driver does not necessarily imply that removeable media is present at that time. Hardware drivers supporting removeable media will typically indicate a successful lookup, irrespective of the presence of a physical disk.

The reason for this is that the call to `cyg_io_set_config()` for the `CYG_IO_SET_CONFIG_DISK_EVENT` operation may quite reasonably occur when no physical media is present, so the lookup must succeed in order to obtain a valid I/O handle. That handle is, however, unusable for operations other than `CYG_IO_SET_CONFIG_DISK_EVENT`.

The context in which the callback is made depends on the implementation of the disk driver. It cannot be assumed that it will occur in a context in which it is safe to make complex calls. It should be assumed that the call is being made from DSR context, where blocking kernel calls may not be made. In general the function should record the details of the call and pass off control to a worker thread. See the automounter in the FILEIO package for an example of how this can be done.

When hardware drivers notice the `CYG_IO_SET_CONFIG_DISK_EVENT` call, they should immediately check for the presence of a disk, in case one is already connected at the time of event registration. The event callback should be made for any connected disk devices (and not for unconnected disks).

Chapter 48. Hardware Driver Interface

While the DISK I/O package provides the top level, hardware independent, part of each disk driver, the actual hardware interface is handled by a hardware dependent interface module. To add support for a new disk device, the user should be able to use the existing hardware independent portion and just add their own interface driver which handles the details of the actual device. The user should have no need to change the hardware independent portion.

The interfaces used by the disk driver and disk implementation modules are contained in the file `<cyg/io/disk.h>`.

Note

In the sections below we use the notation <<xx>> to mean a module specific value, referred to as “xx” below.

DevTab Entry

The interface module contains the devtab entry (or entries if a single module supports more than one interface). This entry should have the form:

```
BLOCK_DEVTAB_ENTRY(<<module_name>>,
                   <<device_name>>,
                   0,
                   &cyg_io_disk_devio,
                   <<module_init>>,
                   <<module_lookup>>,
                   &<<disk_channel>>
                  );
```

Arguments

`module_name` The "C" label for this devtab entry

`device_name` The "C" string for the device. E.g. `/dev/serial0`.

`cyg_io_disk_devio` The table of I/O functions. This set is defined in the hardware independent disk driver and should be used exactly as shown here.

`module_init` The module initialization function.

`module_lookup` The device lookup function. This function typically sets up the device for actual use, turning on interrupts, configuring the controller, etc.

`disk_channel` This table (defined below) contains the interface between the interface module and the disk driver proper.

Disk Controller Structure

The arrangement of disk hardware usually has a number of physical disks connected to a common controller. For example, each IDE interface connects to just two disk devices, a SCSI controller may be connected to several disks. The important feature to consider here is that any current data transfer for any one disk on a controller prevents transfers being started on any other disks on that controller until it is finished. Disk controllers are therefore the level at which concurrency and interrupt controls must be implemented.

Each disk controller is created by the macro:

```
DISK_CONTROLLER(l, dev_priv)
```

Arguments

l
The "C" label for this structure.

dev_priv
A placeholder for any device specific data for this controller.

Disk Channel Structure

Each physical disk connected to a controller is represented by a disk channel. Each channel is defined with the following macro:

```
DISK_CHANNEL(l, funs, dev_priv, controller, mbr_supp, max_part_num )
```

Arguments

l
The "C" label for this structure.

funs
The set of interface functions (see below).

dev_priv
A placeholder for any device specific data for this channel.

controller
Pointer to controller to which this disk channel is attached.

mbr_supp
Does this disk support partitioning.

max_part_num
The maximum number of partitions to be supported.

The interface from the hardware independent driver into the hardware interface module is contained in the *funs* table. This is defined by the DISK_FUNS macro.

If the space for the channel has been allocated elsewhere, the following macro may be used to initialise it:

```
DISK_CHANNEL_INIT(dc, funs, dev_priv, controller, disk_info, part_dev_tab,
                  part_chan_tab, part_tab, mbr_supp, max_part_num )
```

The arguments are as for `DISK_CHANNEL()` except for the following:

Arguments

dc
The name of an object of type disk_channel. This object will be initialised by the macro.

disk_info
The name of an object of type disk_info.

part_dev_tab
The name of an array of objects of type struct cyg_devtab_entry. The number of array members must equal max_part_num, plus one.

part_chan_tab
The name of an array of objects of type disk_channel. The number of array members must equal max_part_num.

part_tab
The name of an array of objects of type cyg_disk_partition_t. The number of array members must equal max_part_num.

Disk Functions Structure

```
DISK_FUNS(l, read, write, get_config, set_config)
```

Arguments

l

The "C" label for this structure.

read

```
Cyg_ErrNo (*read)(disk_channel *priv,
                  void         *buf,
                  cyg_uint32    len,
                  cyg_uint32    block_num)
```

This function reads *len* sectors of data from the disk at the sector number given by *block_num*. The actual quantity of data transferred depends on the disk's sector size, which can be obtained using the `CYG_IO_GET_CONFIG_DISK_INFO` key.

If the read completes immediately, or the low level driver is configured to do all IO synchronously, this function will return `ENOERR`, and if it fails will return a negative error code, for example `-EIO`. If the function returns `-EWOULDBLOCK` then it has only started the transfer and will indicate its completion by calling the `transfer_done` callback.

write

```
Cyg_ErrNo (*write)(disk_channel *priv,
                   void         *buf,
                   cyg_uint32    len,
                   cyg_uint32    block_num)
```

This function writes *len* sectors of data to the disk at the block given by *block_num*. The actual quantity of data transferred depends on the disk's sector size, which can be obtained using the `CYG_IO_GET_CONFIG_DISK_INFO` key.

If the write completes immediately, or the low level driver is configured to do all IO synchronously, this function will return `ENOERR`, and if it fails will return a negative error code, for example `-EIO`. If the function returns `-EWOULDBLOCK` then it has only started the transfer and will indicate its completion by calling the `transfer_done` callback.

get_config

```
bool (*get_config)(serial_channel *priv,
                   cyg_uint32 key, const void *xbuf, cyg_uint32 *len);
      )
```

This function is used to get configuration data from the device. The *key* argument defines the configuration data to be fetched. The *xbuf* and `*len` arguments describe a buffer into which the data will be put. The function should return `true` if the key type is supported and the buffer of sufficient length to contain the data. The value of **len* should be updated to actual length of the data returned. The function should return `false` if the driver cannot support the key value or the buffer is of insufficient length.

The following keys may be used to get information from a disk device.

`CYG_IO_GET_CONFIG_DISK_INFO`

This key causes a cyg_disk_info_t structure, as defined in `diskio.h` to be returned.

`CYG_IO_GET_CONFIG_DISK_EVENT`

This key returns a copy of the cyg_disk_event_t previously set by `CYG_IO_SET_CONFIG_DISK_EVENT`.

set_config

```
bool (*set_config)(serial_channel *priv,
                   cyg_uint32 key, const void *xbuf, cyg_uint32 *len);
```

```
      )
```

This function is used to change the configuration of the device. The *key* argument defines the kind of configuration data to be set. The *xbuf* and `*len` arguments describe a buffer in which the data is supplied. The function should return `true` if the key type is supported and the buffer of the correct length and the data appears valid. The function should return `false` if the driver cannot support the key value or the buffer is the wrong length, or the data is invalid in some other way.

The following keys can be sent to a driver:

`CYG_IO_SET_CONFIG_DISK_MOUNT`

This is invoked from the filesystem after locating the device driver to record that the device has been mounted. The generic device layer records the mount against both the partition and physical disk and passes the call on down to the driver. The *xbuf* and **len* arguments are unused.

`CYG_IO_SET_CONFIG_DISK_UMOUNT`

This is invoked from the filesystem to record that the device has been unmounted. The generic device layer records the unmount against both the partition and physical disk and passes the call on down to the driver. If the `chan->info->mounts` counter is zero, the driver should call the `disk_disconnected()` callback to prepare the generic layer for a potential media change. The *xbuf* and **len* arguments are unused.

`CYG_IO_SET_CONFIG_DISK_EVENT`

This may be invoked by the application to set a disk event callback function. The generic disk layer is mostly responsible for handling this by recording the event function in the disk channel structure. The call is additionally passed down to the hardware driver so that it may prepare the hardware, if necessary. The *xbuf* should point to a cyg_disk_event_t structure.

Callbacks

The interface from the hardware specific driver to the hardware independent driver is contained in a disk_callbacks_t structure. A pointer to this is automatically included into the disk channel structure *callbacks* field by the `DISK_CHANNEL()` macro. The disk_callbacks_t structure contains the following function pointers:

disk_init

```
cyg_bool (*disk_init)(struct cyg_devtab_entry *tab);
```

Initialize the disk. This must be called from the disk driver's init routine to initialize the device independent driver's data structures for this disk.

disk_connected

```
Cyg_ErrNo (*disk_connected)(struct cyg_devtab_entry *tab,
                            cyg_disk_identify_t     *ident);
```

This is called when a valid disk device has been recognised on the given disk channel. At this point, if the disk supports partitioning the disk's partition table will be read and the partitions determined. This may be called either from the driver's init routine, for fixed disks, or alternatively from the driver's lookup routine. It may also be called from other places when, for example, disk insertion is detected. All the fields of the `ident` structure must be filled in by the driver before this call is made.

disk_disconnected

```
Cyg_ErrNo (*disk_disconnected)(struct disk_channel *chan);
```

This is called when, for example, disk removal is detected. It invalidates all the existing partition and driver information and renders the channel ready for a new disk device to be inserted.

disk_lookup

```
Cyg_ErrNo (*disk_lookup)(struct cyg_devtab_entry **tab,
                         struct cyg_devtab_entry  *sub_tab,
                         const char               *name);
```

This must be called from the driver's lookup function to complete the lookup process. It is here that the interpretation of the partition number element of the device name is done and a new devtab entry created for the partition if necessary.

disk_transfer_done

```
void (*disk_transfer_done)(struct disk_channel  *chan,
                           Cyg_ErrNo            res);
```

When the call to the `read()` or `write()` disk function returns `-EWOULDBLOCK` then the driver must indicate completion of the actual transfer by calling this function. This function should not be called from an ISR, but it may be called from the DSR.

In addition to these functions in disk_callbacks_t, the hardware driver is also responsible for calling the disk event callback. The calls should be made as follows:

```
    disk_channel *chan = <get pointer to disk channel>;

    …

    chan->event( CYG_DISK_EVENT_CONNECT, devno, chan->event_data );
```

The first argument should be the event being notified: `CYG_DISK_EVENT_CONNECT` as shown here, or `CYG_DISK_EVENT_DISCONNECT`. The second argument is a device number; this is needed for devices that dynamically instantiate disk devices, such as USB. If the driver does not do this, then this argument should be -1. The third argument is the user data value passed in when the callback was registered.

The driver may call this function at any time and from any context other than an ISR. Normally it will be called either from a DSR or from a thread context. By default, the generic disk layer will install a dummy function in the disk channel structure, so the driver can always make the call without needing to test for a NULL pointer. A CONNECT event call should be made when the driver detects that a new device has been inserted into the drive, and an DISCONNECT event call should be made when the device is removed.

A CONNECT event call should also be made if a disk device is already connected when the driver observes the application registering for notification of disk events by use of the `CYG_IO_SET_CONFIG_DISK_EVENT` `cyg_io_set_config()` operation. However, this only applies to connected disks - the driver does not indicate DISCONNECT events for unconnected disks.

Putting It All Together

The above descriptions, while strictly useful as documentation, do not really show how it all gets put together to make a device driver. The following example of how to create the data structures for a device driver, for a standard PC target, are derived from the eCosPro IDE disk driver.

The first thing to do is to define the disk controllers:

```
static ide_controller_info_t ide_controller_info_0 = {
    ctlr:       0,
    vector:     HAL_IDE_INTERRUPT_PRI
};

DISK_CONTROLLER( ide_disk_controller_0, ide_controller_info_0 );

static ide_controller_info_t ide_controller_info_1 = {
    ctlr:       1,
    vector:     HAL_IDE_INTERRUPT_SEC
};
```

```
DISK_CONTROLLER( ide_disk_controller_1, ide_controller_info_1 );
```

A typical PC target has two IDE controllers, so we define two controllers. The ide_controller_info_t structure is defined by the driver and contains information needed to access the controller. In this case this is the controller number, zero or one, and the interrupt vector it uses. The `DISK_CONTROLLER()` macro generates a system defined controller structure and populates it with a pointer to the matching controller info structure.

The next step is to define the disk functions that will be called to perform data transfers on this driver. These functions the main part of the driver, together with the init and lookup functions and any ISR and DSR functions.

```
DISK_FUNS(ide_disk_funs,
          ide_disk_read,
          ide_disk_write,
          ide_disk_get_config,
          ide_disk_set_config
);
```

We can now start generating per-disk-channel data structures. To make this easier we define a macro, `IDE_DISK_INSTANCE()` to make this easier.

```
#define IDE_DISK_INSTANCE(_number_,_ctlr_,_dev_,_mbr_supp_)     \
static ide_disk_info_t ide_disk_info##_number_ = {              \
    num:           _number_,                                    \
    ctlr:          &ide_controller_info_##_ctlr_,               \
    dev:           _dev_,                                       \
};                                                              \
DISK_CHANNEL(ide_disk_channel##_number_,                        \
             ide_disk_funs,                                     \
             ide_disk_info##_number_,                           \
             ide_disk_controller_##_ctlr_,                      \
             _mbr_supp_,                                        \
             4                                                  \
);                                                              \
BLOCK_DEVTAB_ENTRY(ide_disk_io##_number_,                       \
             CYGDAT_IO_DISK_IDE_DISK##_number_##_NAME,          \
             0,                                                 \
             &cyg_io_disk_devio,                                \
             ide_disk_init,                                     \
             ide_disk_lookup,                                   \
             &ide_disk_channel##_number_                        \
);
```

The first thing this macro does is generate an instance of the ide_disk_info_t. This is a driver-defined structure to contain any info that does not fit in the system defined structures. In this case the important things are the number of the device on the controller, zero or one mapping to master or slave, and a pointer to the driver-defined controller structure. The `DISK_CHANNEL()` macro creates a disk channel object and populates it with the function list defined earlier, a pointer to the matching local info structure just defined, and a pointer to the controller it is attached to. Finally, a device table entry is created. This uses linker features to install an entry into the device table that allows the IO subsystem to locate this device.

Finally we need to instantiate all the channels that this driver will support.

```
IDE_DISK_INSTANCE(0, 0, 0, true);
IDE_DISK_INSTANCE(1, 0, 1, true);
IDE_DISK_INSTANCE(2, 1, 0, true);
IDE_DISK_INSTANCE(3, 1, 1, true);
```

Each invocation of `IDE_DISK_INSTANCE()` generates all the data structures needed to access each possible physical disk that may be present.

Part XI. USB Mass Storage Support

Name

CYGPKG_DEVS_DISK_USBMS — eCosPro Support for USB Mass Storage

Description

This document describes the eCosPro USB Mass Storage class driver. This driver provides support for USB memory sticks, Hard disks and any other devices that supports the same protocol.

Protocol support is limited to Bulk transport only, and only the SCSI transparent command set is supported. This covers most modern devices, however very old devices may support other protocols.

Configuration Options

The `CYGPKG_DEVS_DISK_USBMS` package needs to be loaded in order to use the driver. In addition the `CYGPKG_IO_USB` USB package must be loaded to provide generic USB functionality and the board target entry in `ecos.db` must contain a Host Controller Driver package and a platform configuration package in order to access the USB hardware. This package also depends on support from the generic disk package `CYGPKG_IO_DISK`. If file system support is needed, then packages like `CYGPKG_FS_FAT`, `CYGPKG_IO_FILEIO` and `CYGPKG_LINUX_COMPAT` will need to be loaded along with any packages that they may depend upon. Depending on the template used to create the initial configuration some of these may be loaded already.

For more information on the USB subsystem and supported classes, consult the general USB documentation.

cdl_component CYGPKG_IO_USB_HOST

> Ensure that the host USB package has been loaded and enabled for your target so that USB mass storage can function.

cdl_option CYGDAT_DEVS_DISK_USBMS_DISK_NAME

> This is the base device name used to access the raw disk devices in eCos, for example for mount operations. Individual mass storage devices will be named using a trailing number and partitions with a yet further trailing partition number. For example disk 0, partition 1 would be named /dev/usbms/0/1. This is the default mount point for a standard USB memory stick.

cdl_option CYGNUM_DEVS_DISK_USBMS_MAX_TRANSFER

> This defines the maximum transfer size submitted to the device. Larger disk transfer requests will be split into smaller transfers of this size. If the USB stack or host device drivers have reduced resources, this value should be reduced to consume fewer resources for each transfer.

cdl_option CYGNUM_DEVS_DISK_USBMS_CONTROLLER_COUNT

> This defines the number of disk controllers the USBMS driver can handle simultaneously. There is a one-to-one correspondence between controllers and USB devices, so this defines the number of USB mass storage devices that can be accessed at one time.

cdl_option CYGNUM_DEVS_DISK_USBMS_CHANNEL_COUNT

> This defines the number of disk channels the USBMS driver can handle simultaneously. Each channel corresponds to a Logical Unit Number within a controller or device. Most USB mass storage devices contain a single LUN, so the default is to set this to the same value as the controller count. If devices with more LUNs are expected to be used regularly, then this value should be increased.

Part XII. MMC, SD, SDHC and SDIO Media Card Disk Driver

Name

CYGPKG_DEVS_DISK_MMC — eCos Support for MMC, SD, SDHC and SDIO media Cards

Description

This package provides a disk device driver for two commercial flash memory card standards: MultiMedia Cards (MMC), and Secure Digital (SD) cards, including the high-capacity SDHC variant. The package also provides some (non-disk) basic SD I/O (SDIO) card support.

The MMC card implementation is intended to allow operation with memory cards compliant with the MultiMediaCard Standard version 2, as published by the MultiMediaCard Association. The SD implementation is intended to allow operation with cards compliant with the SD Physical Layer Specification version 2, as published by the SD Card Association.

This package evolved from an MMC-only implementation and as such the naming of certain aspects such as the CDL package name reflects that heritage. Any identifiers which reference MMC usually refer to either MMC or SD cards unless otherwise noted. Similarly, the package provides (limited) support for SDIO cards which do *NOT* require the presence of the CYGPKG_IO_DISK infrastructure and do not present as disk (memory) devices.

An MMC/SD card provides non-volatile storage in a small footprint (24mm * 32mm * 1.4mm), and weighing less than 2 grams. Typical card sizes are 128MB to 2GB, with an upper limit of 4GB for MMC and SDv1; and 32GB for SDHC cards in SDv2. It should be noted that these sizes are measured in millions of bytes, not 2^20. This driver provides support for 4GB MMC and SDv1 cards, although in practice, the FAT16 filesystem layout on such cards is unusual and may not be supported by a filesystem implementation using this driver. This problem should not occur with cards of size 2GB and less, or with SDHC cards.

At the hardware level there are two ways of accessing an MMC card. The first it to use a custom interface frequently known as either an an MCI (Multimedia Card Interface, although this allows supports for SD as well) or an MMC/SD bus. The second interface is via connection to an SPI bus. A card will detect the interface in use at run-time. The custom MCI interface allows for better performance but requires additional hardware. SPI peripheral support is more readily available on many existing CPUs. At this time, the SPI bus mode of interface does not support SD or SDIO cards in this driver.

Theoretically an MMC/SD memory card can be used with any file system. In practice all cards are formatted for PC compatibility, with a partition table in the first block and a single FAT file system on the rest of the card. The SPI mode driver always checks the format of the MMC card and will only allow access to a card if it is formatted this way. The MCI card bus driver can adapt to a card with no partition table as long as it contains a FAT filesystem starting from the first block. This non-standard format can sometimes be created by Windows when reformatting a corrupted card. This ability is controlled by the CYGSEM_IO_DISK_DETECT_FAT_BOOT CDL configuration option in the generic disk device driver package CYGPKG_IO_DISK.

Card Insertion and Removal

An MMC or SD socket allows cards to be removed and inserted at any time. It is a common feature for such sockets to contain a contact allowing the presence of cards to be detected. On some hardware that signal is routed to the processor allowing it to be sampled, usually connected as a GPIO signal or to an interrupt line (or to a GPIO interrupt if available).

In such cases, the MMC/SD bus driver layer in this package is able to be informed by the hardware MMC/SD bus driver of whether cards are present or not, and if possible, can be informed by an event callback that a card has just been inserted or removed. The SPI mode driver in this package does not yet support this feature.

If using the MMC/SD bus driver with appropriate hardware and driver support, the MMC/SD bus driver layer in this package can plug into the removeable media support offered by the generic disk driver layer (CYGPKG_IO_DISK) if the configuration option CYGFUN_DEVS_DISK_MMCSD_BUS_REMOVABLE_MEDIA_SUPPORT is enabled. This option may only be enabled if a hardware driver indicates that support is available. This facility allows for event notification when a card is inserted or removed from the socket. This information can be used directly by the application using the disk package APIs (see that package's documentation), or to allow use of, for example, the automounter support provided in the File I/O package (CYGPKG_IO_FILEIO).

If card detection by an interrupt is not possible, or if using the SPI bus driver, then the only time the device driver will detect removal events is when the next I/O operation happens. At that point, the operation will fail, typically with an error code such as ENODEV, ETIMEDOUT or possible EIO. It is left to higher-level code to recover from this error - the MMC/SD driver is unable to do anything since the card has gone. In the case of the eCosPro implementation of the FAT filesystem, it has been made robust to such events such that it will always be able to force an unmount using the `umount_force` function instead of the standard `umount` function.

Without card detection by interrupt, use of the automounter is not possible, therefore expected usage is that application code will explicitly `mount` the card before attempting any file I/O.

Irrespective of card detection abilities, it is expected that the application will `umount` the card before it is removed. Until unmounted, the system is likely to keep some disk blocks cached, for performance reasons. If the card is removed before the `umount` then it may end up with a corrupted file system. Application design to inform users of when it is safe to remove card media, and regular uses of the standard `sync` function will reduce the risk of file system corruption.

If card detection support is available, but is only pollable, rather than being connected to an interrupt, then this has limited benefits other than to accelerate the process of determining whether a card has been removed, which otherwise necessitate attempting operations and waiting for potential timeouts. In a future revision of this driver it may become possible to use a polling thread to check periodically for whether cards have been inserted or removed.

Write Protection and Security

The MMC and SD specifications allow cards to be write-protected in software. The current device driver does not yet make it possible to mark a card as write-protected, however it does respect the setting, and on mounting such a card will mark it internally as read-only. Any attempt to write to the card will fail with the error EROFS.

SD cards additionally feature a write-protect or 'lock' switch to indicate that cards must not be written to. This is not a physical protection however - instead it is expected that the lock switch position is detected by a contact in the socket, and it is for software to sample the state of that contact to determine whether the card is write-protected. Therefore the lock switch may not be respected if either the hardware or hardware driver does not support sampling the lock switch position from the socket. If sampling is supported however, the MMC/SD bus driver will respect that and mark the card internally as read-only.

SD (and to a lesser extent MMC) support other security features such as password protection and encryption. This driver does not yet support these features.

Configuration Options

`CYGPKG_DEVS_DISK_MMC` is a hardware package which should get loaded automatically when you configure for a suitable eCos target platform. In this case suitable means that the hardware either:

a. has an MMC/SD socket connected to an SPI bus, that an SPI bus driver package exists and is also automatically loaded, and that the platform HAL provides information on how the card is connected to the SPI bus; or

b. has an MMC/SD socket connected to a custom MCI interface's card bus and a driver package for the MCI exists and is also automatically loaded, or exists in the HAL.

For memory card support the package depends on support from the generic disk package `CYGPKG_IO_DISK`. That will not be loaded automatically: the presence of an MMC/SD socket on the board does not mean that the application has any need for a file system. Hence by default `CYGPKG_DEVS_DISK_MMC` will be inactive and will not contribute any code or data to the application's memory footprint. To activate the driver it will be necessary to add one or more packages to the configuration using **ecosconfig add** or the graphical configuration tool: the generic disk support `CYGPKG_IO_DISK`; usually a file system, `CYGPKG_FS_FAT`; support for the file I/O API `CYGPKG_IO_FILEIO`; and possibly additional support packages that may be needed by the file system, for example `CYGPKG_LINUX_COMPAT` for FAT. Depending on the template used to create the initial configuration some of these may be loaded already.

For non-memory SDIO cards it is possible for the package to be used without the disk I/O infrastructure. This is controlled by the `CYGFUN_DEVS_DISK_MMCSD_SDIO` option, which is available when the target platform indicates that it implements the relevant SDIO support. This allows for embedded (non-removable) SDIO device support on platforms without incurring the cost of including the unnecessary disk I/O code.

SPI mode operation configuration

The package provides two main configuration options when using the SPI mode of operation. CYGDAT_DEVS_DISK_MMC_SPI_DISK0_NAME specifies the name of the raw disk device, for example /dev/mmcdisk0. Allowing for partition tables that makes /dev/mmcdisk0/1 the first argument that shoul be passed to a mount call. If the hardware has multiple disk devices then each one will need a unique name. CYGIMP_DEVS_DISK_MMC_SPI_POLLED controls whether the SPI bus will be accessed in interrupt-driven or polled mode. It will default to interrupt-driven if the application is multi-threaded, which is assumed to be the case if the kernel is present. If the kernel is absent, for example in a RedBoot configuration, then the driver will default to polled mode. With some hardware polled mode may significantly increase disk throughput even in a multi-threaded application, but will consume cpu cycles that could be used by other threads.

MMC/SD card bus mode operation configuration

When using an MMC/SD card bus, there a number of CDL configuration settings to be aware of within this driver.

Number of sockets on the MMC/SD bus (CYGINT_DEVS_DISK_MMCSD_BUS_CONNECTORS)

> This CDL interface indicates the number of sockets capable of being supported by the MMC/SD card bus driver. It is usually implemented by either a hardware device driver or the platform HAL. At the present time there can only be 1 socket supported. This limitation is intended to be lifted in the future.

SD card support (CYGFUN_DEVS_DISK_MMCSD_SD)

> This option is present to allow SD card support to be disabled. SD card support is considered a superset of MMC support, and therefore it is not possible to disable MMC card support. If SD cards are not to be used, this option can be disabled to reduce code and memory footprints, along with slightly faster execution.

SDIO card support (CYGFUN_DEVS_DISK_MMCSD_SDIO)

> This option is present to allow SDIO card support to be enabled for targets that do not require memory MMC/SD card disk support. It is enabled by default when the target platform/variant indicates the requirement, and is not normally an option the user should need to manually configure.

Device name for the MMC/SD disk 0 device (CYGDAT_DEVS_DISK_MMCSD_BUS_DISK0_NAME)

> This is the name of the raw disk or SDIO device. For disks it provides the prefix used for the separate disk device strings which are passed to the mount call. For example, a setting of /dev/mmcsd0/ would allow the first partition on the card to be accessed as /dev/mmcsd0/1, the second as /dev/mmcsd0/2, etc. /dev/mmcsd0/0 is a special device name used to access the entire device (including the partition table if present. Furthermore, the /dev/mmcsd0 device can be used for registering disk insertion/removal events with the disk layer. Consult the disk package documentation for details. The setting of this configuration option must end with a slash character ('/').

Hardware drivers support card detection (CYGINT_DEVS_DISK_MMCSD_BUS_CARD_DETECTION)

> This CDL interface is implemented by a hardware device driver or platform HAL to indicate that it is able to report the presence or absence of cards.

Removable MMC/SD media support (CYGFUN_DEVS_DISK_MMCSD_BUS_REMOVABLE_MEDIA_SUPPORT)

> This option is used to determine whether the MMC/SD bus layer will plug into the generic disk package's removeable media support, i.e. allowing notification of insertion or removal of cards. There is no point enabling this option without hardware and

driver support, so it is not possible to enable it if CYGINT_DEVS_DISK_MMCSD_BUS_CARD_DETECTION has not been implemented. Some code can be saved if this option is disabled.

MMC/SD debug output (CYGDBG_DEVS_DISK_MMCSD_BUS_DEBUG)

Detailed debugging output is possible via the diagnostic console. By default there is no debugging output, but setting this option to 1 or 2 will provide increased verbosity of debugging output.

Certain MMC/SD bus device drivers may provide support for multi-sector I/O. But if you are using the FAT filesystem, it will not take advantage of this facility unless you make a configuration change within the FAT filesystem package (CYGPKG_FS_FAT). You may increase the value of the "FAT block cache block size" (CYGNUM_FS_FAT_BLOCK_CACHE_BLOCKSIZE) to a higher power of two, in order to increase the number of sectors read or written in a chunk by the filesystem. This will cause multi-sector I/O to be employed within this driver. It has been noticed that certain models of SD cards (including some made by brand-name manufacturers like Sandisk and Kingston) perform disproportionately poorly if only using single block I/O; therefore we recommend that where possible you do adjust this option to a higher value (e.g. 16384). Note that memory usage will go up proportionately unless you also reduce the "FAT block cache memory size" (CYGNUM_FS_FAT_BLOCK_CACHE_MEMSIZE), which you may wish to do depending on your memory requirements.

Additional SPI Mode Functionality

When using the SPI mode to access MMC cards, the disk driver package exports a variable cyg_mmc_spi_polled. This defaults to true or false depending on the configuration option CYGIMP_DEVS_DISK_MMC_SPI_POLLED. If the default mode is interrupt-driven then file I/O, including mount operations, are only allowed when the scheduler has started and interrupts have been enabled. Any attempts at file I/O earlier during system initialization, for example inside a C++ static constructor, will lock up. If it is necessary to perform file I/O at this time then the driver can be temporarily switched to polling mode before the I/O operation by setting cyg_mmc_spi_polled, and clearing it again after the I/O. Alternatively the default mode can be changed to polling by editing the configuration, and then the main() thread can change the mode to interrupt-driven once the scheduler has started.

Porting to New Hardware

SPI mode

Assuming that the MMC connector is hooked up to a standard SPI bus and that there is already an eCos SPI bus driver, porting the MMC disk driver package should be straightforward. Some other package, usually the platform HAL, should provide a cyg_spi_device structure cyg_spi_mmc_dev0. That structure contains the information needed by this package to interact with the MMC card via the usual SPI interface, for example how to activate the appropriate chip select. The platform HAL should also implement the CDL interface CYGINT_DEVS_DISK_MMC_SPI_CONNECTORS.

When defining cyg_spi_mmc_dev0 special care must be taken with the chip select. The MMC protocol is transaction-oriented. For example a read operation involves an initial command sent to the card, then a reply, then the actual data, and finally a checksum. The card's chip select must be kept asserted for the entire operation, and there can be no interactions with other devices on the same SPI bus during this time.

Optionally the platform HAL may define a macro HAL_MMC_SPI_INIT which will be invoked during a mount operation. This can take any hardware-specific actions that may be necessary, for example manipulating GPIO pins. Usually no such macro is needed because the hardware is set up during platform initialization.

On some targets there may be additional hardware to detect events such as card insertion or removal, but there is no support for exploiting such hardware at present.

Only a single MMC socket is supported. Given the nature of SPI buses there is a problem if the MMC socket is hooked up via an expansion connector rather than being attached to the main board. The platform HAL would not know about the socket so would not implement the CDL interface CYGINT_DEVS_DISK_MMC_SPI_CONNECTORS, and the ecos.db target entry would not include

CYGPKG_DEVS_DISK_MMC. Because this is a hardware package it cannot easily be added by hand. Instead this scenario would require some editing of the existing platform HAL and target entry.

Card bus mode

Creating a hardware driver for accessing a card connected via a card bus requires a large amount of detailed description closely related to the specific code definitions. Therefore comprehensive descriptions of functionality has been provided in the mmcsd_bus.h header file in the include directory of this package. Drivers should include this file, although before doing so they must define the C preprocessor macro __MMCSD_DRIVER_PRIVATE in order to obtain definitions private to card bus drivers.

It is appropriate to provide a high-level overview of the porting process however. A driver package must implement the CDL interface CYGINT_DEVS_DISK_MMCSD_BUS_CONNECTORS to indicate the presence of a socket driven as a card bus. It may also implement CYGINT_DEVS_DISK_MMCSD_BUS_CARD_DETECTION if appropriate.

The driver in this package accesses the hardware driver through the abstraction of the card bus. This is done by instantiating a bus object using the CYG_MMCSD_BUS macro. This takes as arguments an opaque word of private data which may be useful to the hardware driver for identifying this bus or for any relevant bus state, and it also takes a function callback list. The CYG_MMCSD_BUS instantiation must exist in a module which is always included in the program image. This is usually performed when building the package by including it in the libextras.a library (which is converted to extras.o in the eCos build process and forcibly included in the program image that way).

This function callback list must be instantiated using the CYG_MMCSD_BUS_FUNS macro. This provides a table identifying driver functions to: initialise the bus at system startup time; (re-)initialise the socket when attempting to access a card in it for the first time; shutting down a socket to conserve power; doing specialised configuration options; preparing to select a card in a socket; sending a command to a card; and transferring data blocks to or from a card. At this point the byte and stream operations may be left as NULL and are only present for potential future expansion. Details on the purpose and arguments to these functions can be found in mmcsd_bus.h.

If the hardware and driver is capable of reporting card insertion/removal events, then notification of insertion or removal can be performed by calling the MMCSD_CARD_DETECT_EVENT() macro to register this with the MMC/SD layer, which will perform any further processing required. It must be called in DSR or thread context, not ISR context.

SDIO Support

Due to the undefined nature of SDIO card features, the package (currently) provides basic initialisation and device access support. Custom drivers will be needed to support specific SDIO cards or embedded devices. A simple API is exposed to allow the underlying SD commands to be passed to the SDIO card compliant with the SD Specifications Part E1 SDIO Simplified Specification Version 3.00 document as published by the SD Card Association.

A custom driver will reference the SDIO card via an I/O handle obtained via a call to the cyg_io_lookup() function. This handle can be used to perform MMC/SD bus driver “config” calls as well as perform SD operations via the SDIO specific functions exposed by this package. Currently two SDIO specific functions are available.

```
Cyg_ErrNo cyg_sdio_transaction_direct(cyg_io_handle_t handle,
                                      cyg_uint32      cmd,
                                      cyg_uint32      arg,
                                      cyg_uint32 *response);
```

The function above can be used to send card control commands (e.g. CMD0, CMD5, etc.) or the SDIO single register read/write CMD52 whereas the function below is an interface to the specific SDIO block data transfer CMD53 support.

```
Cyg_ErrNo cyg_sdio_transaction_extended(cyg_io_handle_t handle,
                                        cyg_uint32      arg,
                                        cyg_uint32      *response,
                                        cyg_bool        read,
                                        cyg_uint32      block_length,
                                        cyg_uint32      block_count,
                                                  cyg_uint8 *buf);
```

Notes

1. The current SDIO implementation is limited to platforms that define the `HAL_MMCSD_PLF_SDIO_INIT_EARLY_EXIT` macro since the detection of MMC/SD-vs-SDIO (and combo) cards during card specific initialisation has not yet been implemented.
2. It is currently the responsibility for the custom SDIO device driver to perform card initialisation (CMD0, CMD5, et-al) via the exposed `cyg_sdio_transaction_direct()` API.

Part XIII. MMC/SD Card Device Drivers

Table of Contents

Chapter 49. Atmel SAM series Multimedia Card Interface (MCI) driver

Name

Atmel SAM series Multimedia Card Interface (MCI) driver — Using MMC/SD cards with block drivers and filesystems

Overview

The MultiMedia Card Interface (MCI) driver in the SAM MCI device driver package allows use of MultiMedia Cards (MMC cards) and Secure Digital (SD) flash storage cards within eCos, exported as block devices. This makes them suitable for use as the underlying devices for filesystems such as FAT.

This driver can support boards based on either the SAM9 or SAMA5 processors, where underlying platform HAL support exists.

Configuration

This driver provides the necessary support for the generic MMC bus layer within the `CYGPKG_DEVS_DISK_MMC` package to export a disk block device. The disk block device is only available if the generic disk I/O layer found in the package `CYGPKG_IO_DISK` is included in the configuration.

The block device may then be used as the device layer for a filesystem such as FAT. Example devices are "`/dev/mmc0/1`" to refer to the first partition on the card, or "`/dev/mmc0/0`" to address the whole device including potentially the partition table at the start.

Typically, platform HALs provide an option to permit enabling or disabling MCI support.

If the driver is enabled, there are three CDL configuration options:

`CYGIMP_DEVS_MMCSD_ATMEL_SAM_MCI_INTMODE`

This indicates that the driver should operate in interrupt-driven mode if possible. This is enabled by default if the eCos kernel is enabled. Note though that if the driver finds that global interrupts are off when running, then it will fall back to polled mode even if this option is enabled. This allows for use of the MCI driver in an initialisation context.

`CYGNUM_DEVS_MMCSD_ATMEL_SAM_MCI_POWERSAVE_DIVIDER`

The SAM MCI peripheral allows the MCI clock to be divided down if told to enter power saving mode. This option specifies the divider to use. The driver itself does not implement any power saving - it is up to the application to enable power saving in the MCI control register if it is required.

`CYGHWR_DEVS_MMCSD_ATMEL_SAM_MCI_DEVICE`

Some SAM processors have two or even three MCI interfaces. This option selects which of these will be used by the MMC driver. The default for the SAM9263 is to select interface 1, since the PIO lines for interface 0 are shared with SPI0, which will usually be occupied by a dataflash device or card. On the SAM9G45 the pins are not shared, so we select MCI0 by default. The SAMA5D3 permits three MCI devices, and these can be selected here.

Usage notes

MMC/SD cards may only be used in a MMC/SD card slot, and not a dataflash slot. The driver will detect the appropriate card sizes. Hotswapping of cards is supported by the driver, and in the case of eCosPro, the FAT filesystem. Although any cards removed before explicit unmounting or a `sync()` call to flush filesystem buffers will likely result in a corrupted filesystem on the removed card.

The MMC/SD bus layer will parse partition tables, although it can be configured to allow handling of cards with no partition table.

This driver implements multi-sector I/O operations. If you are using the FAT filesystem, see the generic MMC/SD driver documentation which describes how to exploit this feature when using FAT.

Part XIV. The Yaffs filesystem

Important

The Yaffs filesystem package is distributed with eCosPro under the GNU Public License (GPL). The viral nature of the GPL license is incompatible with both the the eCosPro License, making this package suitable for internal evaluation and testing purposes only. NO PART OF ECOSPRO IN ANY FORMAT MAY BE REDISTRIBUTED WHEN LINKED WITH THIS PACKAGE OR ANY OTHER GPL CODE. Shipment of prototypes, hardware or products containing eCosPro licensed code in any format that has been linked with this package are therefore STRICTLY PROHIBITED.

A separate COMMERCIAL LICENSE for Yaffs from eCosCentric is therefore required to permit distribution of binary forms of eCosPro with this package.

Some releases of eCosPro may not include this package. In this case, please contact eCosCentric for licensing and availability.

Table of Contents

Chapter 50. What is Yaffs?

Yaffs is a filesystem for NAND flash chips.

Yaffs is accessed through the FILEIO package which presents a standard POSIX compatible IO interface through which applications use standard open(), read(), write() and close() calls.

Yaffs is a *journalling* filesystem with *wear-levelling*. It is particularly suited to NAND flash parts, having been designed with their unique properties in mind. The use of traditional filesystems (FAT, ext2, etc) which do not have these features is not recommended on such chips because their design requires the use of fixed address on the underlying hardware. Such behaviour causes flash sectors to wear out, the consequence of which would typically be to cause the whole device to become unbootable.

Yaffs also provides a high degree of robustness, which is usually a requirement of embedded devices. A power failure or other crash can leave a traditional filesystem in an inconsistent state which is often difficult to repair, especially in the field.

Yaffs can also be built into RedBoot, which allows you to store application images on NAND flash and boot them with RedBoot's usual flexible scripting system.

For more information about NAND flash chips, how they differ from NOR flash parts and other ways to access them, refer to the documentation for the eCos NAND Flash Library.

For more information about Yaffs itself, refer to yaffs.net.

Chapter 51. Getting started with Yaffs

Licensing considerations

Before you can use Yaffs, you must accept its license. *Yaffs is not covered by the standard eCos license.* You will be reminded of your license to use Yaffs when you install it.

Most users will only have access to Yaffs under the GNU GPL. This costs nothing to license. However this usually means that if you ever distribute your application, you must do so under the GPL. This requires you to publish or otherwise make all of your application, eCos and everything else you link with it available as source code. For full details refer to the text of the GPL (v2).

If you cannot accept the restrictions and obligations of the GPL, Yaffs for eCos is available under a proprietary license, for a fee. Details are available on request from eCosCentric or Aleph One.

Installation

Yaffs is included within the standard eCosPro distribution and no additional installation of the GPL-licensed version is required. The proprietary licensed version is supplied as an EPK (eCos Package) file which may be installed alongside the GPL-licensed version. The remainder of this section deals with the installation of the proprietary licensed version.

Installation via the eCos Configuration Tool

1. Open up the eCos Configuration Tool.
2. Open up the *Administration* dialog, from the *Tools* menu.
3. Press the *Add* button.
4. A file browser windows opens. Navigate to the Yaffs EPK file.
5. The License screen shows. You must accept the license in order to install the package. Press Yes if you do.

Installing from the command-line

Advanced users may alternatively use the `ecosadmin.tcl` tool from the command line. You will be prompted to accept the license at the appropriate time during the procedure.

```
tclsh $ECOS_REPOSITORY/ecosadmin.tcl add yaffs-v1_2_3.epk
```

Configuration and Building

After installing the EPK, Yaffs is added to your eCos repository and is configured and built in the normal way.

Package dependencies

To link Yaffs into your application, add `CYGPKG_FS_YAFFS` to your eCos configuration in the normal way, either using the eCos Configuration Tool (*Packages* dialog on the *Build* menu), or the `ecosconfig` command-line tool.

You will also need to add `CYGPKG_IO_NAND` and `CYGPKG_IO_FILEIO` to your configuration if they are not already present. Your platform HAL should supply packages for all NAND device(s) present.

If you started with a smaller template than *default*, you may also need to add some of the following:

- `CYGPKG_LIBC_STDLIB`
- `CYGPKG_LIBC_STRING`
- `CYGPKG_MEMALLOC` or something else which provides `CYGINT_ISO_MALLOC`
- `CYGPKG_LIBC_I18N` or something else which provides `CYGINT_ISO_CTYPE`

Configuration options

Yaffs provides a number of package options, including tuning parameters.

`CYGPKG_FS_YAFFS_CFLAGS_ADD`
`CYGPKG_FS_YAFFS_CFLAGS_REMOVE`

These settings allow specific build options to be added to or removed from the CFLAGS list when building Yaffs.

`CYGSEM_FS_YAFFS_CACHE_SHORT_NAMES`

If set, caches files' short names in RAM. This consumes more RAM but improves performance.

`CYGPKG_FS_YAFFS_RET_DIRENT_DTYPE`

Controls whether Yaffs supports setting the *`d_type`* field in a struct dirent. If you don't need this, leave it switched off to save a little code size.

`CYGNUM_FS_YAFFS_RESERVED_BLOCKS`

The number of blocks to keep in reserve to allow for garbage collection and block failures. The recommended value is 5, but you can tune it for performance. *This setting is a global default and may be changed by a mount-time option.*

`CYGNUM_FS_YAFFS_SHORTOP_CACHES`

The number of page cache entries to use. Values of 10 to 20 are recommended; increasing the number consumes more RAM, and 0 disables it altogether. *This setting is a global default and may be changed by a mount-time option.*

`CYGNUM_FS_YAFFS_TRACEMASK`

This is a 32-bit bitfield which controls diagnostic output. The bit definitions are found in `yportenv.h`; they are only useful if you are debugging Yaffs itself.

`CYGNUM_FS_YAFFS_TEMP_BUFFERS`

Yaffs requires temporary buffers in many places throughout the code. To avoid the overhead of a dynamic malloc every time, a number of buffers are preallocated at mount time. This setting controls how many; should it not prove enough, Yaffs will call malloc on demand as required. The default setting is 6; most users will not need to change it.

`CYGSEM_FS_YAFFS_SMALLPAGE_MODE`

This option only affects behaviour on so-called "small page" NAND devices (those whose pages are 512 bytes long). Such devices do not have enough space in their Out Of Band area to store a full set of Yaffs metadata tags. There are two ways to work around this:

- `YAFFS2` mode - the default - uses regular tags, but at a price: it steals 16 bytes from the available space per page to store them. *This reduces the apparent available size of your filesystem by 1/32!*

- YAFFS1 mode places a smaller tagset in the OOB area, but with a different side-effect: whenever a page is deleted, one byte of the tags area has to be rewritten. Some devices forbid rewrite-without-erase in this way, so it may not be safe for you to use this option. *You must refer to the spec sheet for the chip on your board before selecting this option!*

CYGSEM_FS_YAFFS_OMIT_YAFFS2_CODE

This causes all YAFFS2 code to be omitted from the build. This option only makes sense when all the devices on which Yaffs is to be used are small-page and operating in YAFFS1 mode.

Note

There is no corresponding option to omit YAFFS1 code, because that code is only compiled when CYGSEM_FS_YAFFS_SMALLPAGE_MODE is set to YAFFS1.

CYGMEM_FS_YAFFS_REDBOOT_HEAP_REQUIRED

RedBoot carefully controls the amount of memory available for its heap, allocating it from a fixed-size workspace. If Yaffs is being used with RedBoot, the heap space required is likely to go up substantially. The exact amount depends on properties of the filesystem being mounted. Consult the eCosPro Yaffs documentation for more details of Yaffs' memory requirements. This option ensures RedBoot's heap is increased to a more reasonable size, but it has been made an option in order to allow developers to decrease it, if they are sure the filesystem will not require as much memory as this.

Note

However, note that even this larger amount may not be adequate for some filesystems.

Using Yaffs

Yaffs appears in the eCos filesystem table. This means that you can mount a filesystem in the standard Unix-like way, then interact with it with calls to open, read, etc.

Mounting a filesystem

Before you can use a filesystem, it must be *mounted*.

A NAND device is logically organised as one or more *partitions*, which are usually set up by the relevant platform HAL. You need to tell the mount command which device and partition you wish to access. In the following example, we are mounting partition 0 of the *onboard* NAND device.

```
rv = mount("onboard/0", "/nand", "yaffs");
```

Note

The *device* argument to the mount call is a NAND-specific device name, not an entry in /dev. Refer to the documentation for your platform HAL for details of how the NAND device(s) are named, and to the NAND library documentation for details of how partitions are addressed.

You can, if you wish, make the filesystem mounting automatic at static constructor time with the MTAB_ENTRY macro. (Your platform HAL may already do this; check it carefully.)

```
MTAB_ENTRY(my_nand, "/nand", "yaffs", "onboard/0", "", 0);
```

Note

This example is for eCosPro. In eCos, the MTAB_ENTRY macro takes only four arguments.

Mount-time options

eCosPro allows various options to be passed to a filesystem at mount time, by combining them with the filesystem argument in a particular format. Yaffs understands the following options:

- `reserved`=<int> The number of physical NAND blocks to reserve for garbage collection and block failures (minimum 2). The default is set in CDL as `CYGNUM_FS_YAFFS_RESERVED_BLOCKS`.
- `caches`=<int> The number of page cache entries to use. Values of 10 to 20 are recommended. The default is set in CDL as `CYGNUM_FS_YAFFS_SHORTOP_CACHES`.
- `skip-checkpoint-read` Instructs Yaffs to not attempt to reload the filesystem from a checkpoint, if one exists. In other words, this option forces a full filesystem scan whether or not one is necessary.
- `format` Instructs Yaffs to format the filesystem before it mounts it. This deletes its entire contents.

Data flushing

Yaffs operates a cacheing layer in order to save undue wear on the NAND chip if many small writes are performed. Because of this, if you wish to ensure that any data written to a still-open file has been fully flushed, you must make a synchronisation request. This is done with the `fsync` function, which takes as its argument the file descriptor of the file you wish to synchronise.

Checkpointing

When mounting a filesystem, Yaffs has to scan the NAND chip to recreate its internal state. This can be a slow process, but is made much faster if there is a valid *checkpoint*.

A checkpoint is written out automatically when you unmount the filesystem. At any other time, you can manually force a checkpoint to be written with one of the following functions:

- `cyg_fs_fssync(mountpoint)` synchronises a filesystem (automatically called on umount)
- `sync` synchronises all mounted filesystems.

Note

A checkpoint becomes invalid as soon as there have been any other writes to the filesystem. Finding a good place to sync is necessarily dependent on your application logic.

Limitations

Although Yaffs is a Unix-compatible filesystem, the eCos port does not provide support for the full range of Unix attributes.

- eCos does not check file or directory permissions; everything it creates is given fixed user and group IDs of zero and standard permissions (files rw-r--r--, directories rwxr-xr-x).
- It is not currently possible to change file ownership or permissions.
- It is not currently possible to create symbolic links, FIFOs (named pipes), sockets or device nodes.
- Hard links to files work in the expected way. Hard links to directories are forbidden.
- It is not possible to unlink the '.', '..' or 'lost+found' special directories.

Note

If you will be sharing a Yaffs filesystem between eCos and some other operating system, you are advised to carefully check the other system's definitions of mode (permission) bits and whether any translation may be required.

Memory requirements

The amount of RAM required by Yaffs to hold its in-memory data structures grows with the number of objects (files and directories) in your filesystem. You are recommended to test your application thoroughly to ensure that sufficient memory exists for Yaffs to operate with the most complicated filesystem it is likely to encounter.

If you wish to estimate your RAM usage, the Yaffs author provides the following calculation:

- The partition itself requires a yaffs_DeviceStruct of 3608 bytes.
- If CYGNUM_FS_YAFFS_SHORTOP_CACHES is enabled, each is the size of a NAND page plus 28 bytes.
- Every object (file, directory or hardlink) in the filesystem takes a yaffs_Object struct, which is 124 bytes.
- Every page of every file requires a Tnode entry, but they are always allocated in groups of 16 at a time.
  - The size of a single Tnode entry is the number of *bits* required to number all the pages in the NAND partition Yaffs is using, numbering from *one*; this is rounded up to a multiple of 2, and has an absolute minimum of sixteen bits.
  - For example: on a partition with 65536 pages, seventeen bits are required for the numbering, which round up to 18. Therefore each file takes 288 bits (36 bytes) per group of Tnodes, and one group of Tnodes will cover up to sixteen pages of data.

Note

Actual memory consumption will be slightly higher than suggested by the above. This arises from the tree structure holding the Tnodes, overheads from the heap itself, and so on.

Yaffs calls the standard `malloc` function to allocate memory and `free` to release it. Normally, the eCos heap occupies all spare RAM not needed for the program, its static data or the stacks. Therefore, most applications will not need to do anything special beyond ensuring there is enough spare RAM available on the platform.

Tip

If you wish to experiment with restricted-size heaps to determine much memory your application actually uses under Yaffs, you may find the option `CYGSEM_MEMALLOC_INVOKE_OUT_OF_MEMORY` of use.

Worked example

Consider a Yaffs filesystem hosted by a NAND partition with 65536 pages, each of size 2k, using the default setting of ten short-op caches. On this filesystem we shall store 10000 files each of size 10240 bytes, hence each requires a single group of Tnodes.

Table 51.1. Yaffs RAM use worked example

Consumer	**RAM used (bytes)**
yaffs_DeviceStruct	3,608
Short-op caches @ 10 x (2048+28)	20,760
Total	**1,624,368**

Consumer	**RAM used (bytes)**
yaffs_Objects @ 10,000 x 124	1,240,000
Tnode groups @ 10,000 x 36 *(see above)*	360,000
Total	**1,624,368**

This example is a close match to the actual consumption measured by eCosCentric during testing. (The measured consumption as reported by `mallinfo` was 1,681,112 bytes, which includes the heap's own overheads.)

Testing

Yaffs is supplied with a number of test programs, some of which are have been adapted from tests for other filesystems in eCos.

fops

This was the first basic test created for the port of the filesystem. It is believed to exercise all of the code paths (filesystem operations, file operations and directory operations) within the eCos-Yaffs adaptation layer.

This test was originally intended to run on a synthetic NAND filesystem. On real NAND chips, it deliberately omits the more stressful routines to avoid undue wear on the hardware.

yaffs1

A number of filesystem edge-case semantic tests, including file and directory creation and deletion, invalid open and rename operations, and removing nonexistent files and directories.

yaffs2

Concurrent multi-threaded filesystem access and consistency checks.

yaffs4

Semantic and edge-case testing - like yaffs1 - but with long file names.

yaffs5

Tests that file reading and writing works over reasonably large files (up to 1Mbyte) with different I/O chunk sizes. Some operation timings are collected and reported, as is the data rate on large files.

yaffs6

Semantic and edge-case testing - like yaffs1 - but with Cyrillic filenames in order to test UTF-8 correctness.

mounttime

A simple benchmark which repeatedly mounts and unmounts the filesystem and measures how long this takes. You can optionally use the `mkfiles` routine - also present in the tests directory - to create many short files so you can test performance on a loaded filesystem.

hammer [*]

A stress test designed to shake out corner cases. Repeatedly creates many files of varying sizes from multiple threads until the filesystem fills up, then verifies their contents and removes them. From time to time, all threads pause and the filesystem is unmounted and remounted.

This test is particularly useful when combined with the bad block injection functionality provided by the synthetic NAND device. It has been used in this way by eCosCentric to thoroughly test this package's stability under error conditions.

[*] This test runs forever, until interrupted.

Chapter 52. Using Yaffs with RedBoot

It is possible to link Yaffs into RedBoot and use it to boot an executable image stored on a NAND array.

This is done by configuring and building RedBoot largely in the normal way. You will need to add `CYGPKG_FS_YAFFS` and `CYGPKG_IO_FILEIO` to your configuration, plus their attendant dependencies.

The presence of CYGPKG_IO_FILEIO activates the `fs` series of RedBoot commands. The following (edited) transcript illustrates how they might be used in concert with other RedBoot commands to store an ELF image on a NAND partition, load it back and execute:

```
RedBoot> fs mount -d onboard/0 -t yaffs /nand
yaffs: restored from checkpoint
RedBoot> load -r -h my.tftp.ip.address -b %{freememlo} my.image
Using default protocol (TFTP)
Raw file loaded 0xa013f000-0xa01554b3, assumed entry at 0xa013f000
RedBoot> fs write /nand/myimg.elf
RedBoot> fs list /nand
   1 drwxr-xr-x  0 size      0 .
   1 drwxr-xr-x  0 size      0 ..
 262 -rw-r--r--  0 size  91316 myimg.elf
   2 drwxr-xr-x  0 size      0 lost+found
RedBoot> load -m file /nand/myimg.elf
RedBoot> go
Hello, NAND world!
```

Note

When you command RedBoot to execute an image, it first synchronises all mounted filesystems. Therefore, provided these filesystems support the synchronisation operation (which Yaffs does), it is not always necessary to unmount them before invoking an image.

Memory considerations under RedBoot

RedBoot traditionally has very limited requirements for memory management; the main need is for there to be free space in RAM at the correct (physical) address to load an image before jumping to it.

Introducing Yaffs brings with it not just the need to have dynamically allocatable RAM (`CYGPKG_MEMALLOC`), but enough to replay the filesystem journal. This must be balanced against the need for RAM to load images into.

When the `CYGPKG_MEMALLOC` package is present in RedBoot, by default a small (64k) heap is set up so that the maximum RAM possible is available for loading images. This is not enough to support Yaffs in any circumstances, so the following definition has been included in the `yaffs.cdl` file:

```
requires { CYGPKG_REDBOOT implies (CYGMEM_REDBOOT_WORKSPACE_HEAP_SIZE >= 0x00014000) }
```

Caution

This declaration only provideds the bare minimum heap required to mount a trivial Yaffs filesystem. More will be required for all but the simplest of cases and it is recommended that you test for typical use in your environment. Refer also to the section called “Memory requirements”.

It is recommended that, should you wish to make a filesystem usable by RedBoot, your platform HAL should make a similar declaration in its CDL to establish an appropriate heap size. For example, the platform HAL for the EA LPC2468 OEM board - which has a 128MB NAND chip on-board - contains the following declaration:

```
requires { (CYGPKG_REDBOOT && CYGPKG_FS_YAFFS) implies (CYGMEM_REDBOOT_WORKSPACE_HEAP_SIZE >= 0x20000) }
```

Tip

If you wish to experiment with restricted heaps to determine much memory your application actually uses under Yaffs, you may find the option `CYGSEM_MEMALLOC_INVOKE_OUT_OF_MEMORY` of use.

Part XV. eCos NAND I/O

Table of Contents

Chapter 53. The eCos NAND Flash Library

Description

This is a library which allows NAND flash devices to be accessed by the eCos kernel and applications. It is analogous to the eCos FLASH library, but for NAND devices. It exists as a separate library because of the fundamental differences between the two types of flash memory.

This library provides the following functionality:

- Interrogation to confirm that the expected device is present
- Reading from and writing to flash pages
- Erasing flash blocks
- The ability to divide a single device into multiple partitions, like those of a hard drive
- Creation and maintenance of a Bad Block Table
- Use of an Error Correcting Code to detect and correct single-bit errors, and to detect multiple-bit errors
- Packing of the ECC and application out-of-band data into the spare area on the device

Note

The spare area, ECC and bad block table have been deliberately created with the intention of compatibility with current versions of the Linux MTD layer. For example, this would allow a single NAND device to be accessed by RedBoot to load a Linux kernel, which could then go on to use another partition as its root filesystem.

Tip

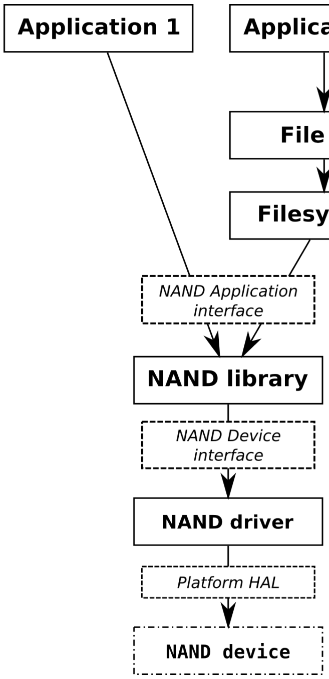
This library is also used as glue to allow appropriate filesystems to use NAND devices. This allows more useful higher-level access by applications and RedBoot via the File I/O and POSIX interfaces. In other words, your application may not need to invoke this library directly, though of course you may still have to write a driver for your chip and/or board.

Structure of the library

This library has two principal interfaces: one for *applications* to call into it, and another to call out to the chip-specific *drivers*. (The chip drivers themselves then require support from the relevant platform HAL to allow them to access the physical chip in an appropriate manner for the board - such as the memory-mapped I/O range to use.)

The following diagram illustrates the calls from two applications all the way to an underlying NAND device. Application 1 uses the NAND library directly, whereas application 2 is using a filesystem and the eCos File I/O layer.

Figure 53.1. Library layout diagram



Device support

Before this library can be used on a given board, an appropriate device driver must be created. Each driver is for a particular NAND part or family of parts; the HAL for each board then instantiates the relevant driver(s) appropriately with board-specific glue such as the memory-mapped I/O range to use. Full details on creating a driver are presented in Chapter 55, *Writing NAND device drivers*.

There is also a Synthetic Target NAND Flash Device for testing purposes, which is present on the *synth* target.

Danger, Will Robinson! Danger!

Unlike nearly every other aspect of embedded system programming, getting it wrong with FLASH devices can render your target system useless. Most targets have a boot loader in the FLASH. Without this boot loader the target will obviously not boot. So before starting to play with this library its worth investigating a few things. How do you recover your target if you delete the boot loader? Do you have the necessary JTAG cable? Or is specialist hardware needed? Is it even possible to recover the target boards or must it be thrown into the rubbish bin? How does killing the board affect your project schedule?

Differences between NAND and NOR flash

Most flash devices supported by the eCos Flash library are categorised as NOR flash. These are fundamentally different from NAND flash devices, both in terms of the storage cells deep within the chip, and how they are addressed and used by applications.

Attribute	NOR	NAND
Addressing of data	By byte address within the device. Usually expressed as memory-mapped addresses.	By row (page) number. Pages are a power of two; commonly 512 or 2048 bytes. Optimised for reading and writing a page at a time. Sometimes supports column (byte) addressing, but this library does not expose such functionality.

Attribute	NOR	NAND
Are direct reads and writes possible? [a]	Usually	Not in general, though a few special-case exceptions exist such as OneNAND devices.
Erase block size	May vary across the chip	A fixed number of pages, typically 64
Out-of-band data	Not supported	A small number of bytes per page - typically 16 "spare" per 512 "data" bytes - are usable by the application. They are read, written and erased at the same time as the "real" page data.
May factory-bad regions [b] exist on the chip?	No	Typically up to 20 eraseblocks are marked as factory-bad in their OOB area. The OS is expected to scan these to create a Bad Block Table. [c]
May data be rewritten without being erased first?	Usually (but only by resetting 1-bits to 0)	Usually, on SLC NAND chips; not on MLC chips.
Error detection and correction	Not present	Usually automatic. Typically this involves an Error Correcting Code, automatically calculated and stored in the OOB area, then checked on read.

[a] In other words, can the application read flash directly as if it was RAM, or does it have to invoke the driver to copy data in and out?

[b] Regions which were found during manufacture to be bad and marked in some way - usually by placing a special code in the Out Of Band area.

[c] Once a BBT exists it can then be used to keep track of any blocks which fail through wear during the lifetime of the device.

Since a NAND chip can in general only be read indirectly, its contents must be copied to RAM before they can be executed. This means that the caveats in the eCos FLASH library about disabling interrupts whilst programming do not apply here, except in special cases such as OneNAND devices.

Preparing for deployment

It is generally not recommended to hard-code physical on-NAND locations in case of factory bad blocks or block failures in the field. [1] Instead it is preferable to set up *partitions* on the chip with a generous safety margin and to store data in a location-independent way. This is commonly achieved by placing logical tags in the spare area of each page, or using a log-structured filesystem such as YAFFS. Such strategies remove the dependence on physical addressing, at the cost of increased complexity.

The upshot of this is that you cannot reliably create a simple binary image to bulk-program in the factory. A more complicated programming operation is required to take account of your chip partitions, logical addressing strategy and any bad blocks which may be encountered during write.

[1] Usually the first block is guaranteed to be defect free for a certain number of erase cycles. This tends to be necessary if bootstrapping the CPU off NAND, and is an obvious exception to this rule.

Chapter 54. Using the NAND library

The eCos NAND library exposes two principal APIs: one for applications to use and the other to communicate with device drivers.

Configuring the NAND library

The following configuration options are provided. They affect the library globally, i.e. across all drivers.

`CYGPKG_IO_NAND_CFLAGS_ADD`
`CYGPKG_IO_NAND_CFLAGS_REMOVE`

Allows specific build options to be added to or removed from the CFLAGS list when building this library.

`CYGSEM_IO_NAND_DEBUG`

This is the master switch for all debug reporting from the library.

`CYGSEM_IO_NAND_DEBUG_FN_DEFAULT`

This is the default function that the library will use when sending debugging output. It must behave like `printf`. The default - `cyg_nand_defaultprintf` - is a wrapper to `diag_printf`.

Note

Individual drivers may override this setting in their `devinit` routines by overwriting the pointer in the device struct.

`CYGSEM_IO_NAND_DEBUG_LEVEL`

Specifies the verbosity of the NAND library and device drivers. Ranges from 0 (off) to 9 (incredibly verbose); the default setting is 1. (Higher values are only likely to be of use during driver development, if ever.) When enabled, messages are printed using the per-device printf-like function (see above).

Note

Should a serious problem be encountered it will always be reported the printf-like function, regardless of this setting. Such messages may be suppressed altogether by turning off `CYGSEM_IO_NAND_DEBUG`.

`CYGSEM_IO_NAND_READONLY`

Globally disables all code which writes to NAND devices. This may be useful during driver development.

`CYGNUM_NAND_MAX_PARTITIONS`

Sets a compile-time limit on the number of partitions any NAND device may have. The default is 4, which should be enough for most purposes; unnecessarily setting this higher wastes RAM.

`CYGSEM_IO_NAND_USE_BBT`

Globally enables and disables the use of Bad Block Table.

Warning

This setting should not be disabled lightly! It is strongly recommended that you leave this setting enabled unless you have a very good reason to not use it. It is provided really as a convenience for allowing developers to recover their NAND from a confused state.

The NAND Application API

All of the functions described here are declared in the header file `<cyg/nand/nand.h>`, which should be included by all users of the NAND library.

Note

Most of the functions in the library are declared as returning int. *Unless otherwise stated, all functions return 0 for success, or a negative eCos error code if something went wrong.*

Device initialisation and lookup

NAND devices are identified to the library by name. In many cases there will be only one, commonly named *onboard*, but this flexibility allows for easy expansion later without cross-device confusion.

Note

The naming of NAND devices is set up by the code that instantiates their drivers. Normally this is done by the platform HAL port.

```
 __externC int cyg_nand_lookup(const char *devname, cyg_nand_device **dev_o);
```

On success, *dev_o will be set up to point to a cyg_nand_device struct. On failure, it will not; a return code of `-ENOENT` signifies that the requested device name was not found.

Applications will hardly, if ever, need to access the cyg_nand_device structs directly. The following members and convenience macros are most likely to be of relevance:

```
struct _cyg_nand_device_t {
…
cyg_nand_printf pf; // Diagnostic printf-like function for this device to use. May be changed at runtime.
…
size_t page_bits; // log2 of no of regular bytes per page
size_t spare_per_page; // OOB area size in bytes
size_t block_page_bits; // log2 of no of pages per eraseblock
size_t blockcount_bits; // log2 of number of blocks
size_t chipsize_log; // log2 of total chip size in BYTES.
…
};

#define CYG_NAND_BYTES_PER_PAGE(dev) (1<<(dev)->page_bits)
#define CYG_NAND_SPARE_PER_PAGE(dev) ((dev)->spare_per_page)
#define CYG_NAND_PAGES_PER_BLOCK(dev) (1<<(dev)->block_page_bits)
#define CYG_NAND_BLOCKCOUNT(dev) (1<<(dev)->blockcount_bits)
#define CYG_NAND_PAGECOUNT(dev) (NAND_BLOCKCOUNT(dev) * NAND_PAGES_PER_BLOCK(dev))
#define CYG_NAND_CHIPSIZE(dev) (1<<(dev)->chipsize_log)
#define CYG_NAND_APPSPARE_PER_PAGE(dev) ((dev)->oob->app_size)
#define CYG_NAND_BYTES_PER_BLOCK(dev)   (1<<( (dev)->block_page_bits + (dev)->page_bits ))
```

NAND device addressing

NAND devices are arranged as a series of *pages* and *eraseblocks*. The eCos NAND library numbers pages and eraseblocks sequentially, both starting at 0 and continuing until the end of the chip. For example, eraseblock 0 might contain pages 0 through 63; eraseblock 1, pages 64 through 127; and so on.

Caution

This numbering scheme is independent of the device's addressing scheme. Take care, particularly when erasing blocks; some devices and some applications effectively express the location to erase as a page number (or, in NAND-speak, as the *row address* to erase from).

Warning

Most NAND chip manufacturers document restrictions on the order in which pages may be written to their device. Typically, individual pages within an eraseblock must be written in sequential order starting from the first, and random-order writes are prohibited or unspecified. The eCos NAND library does not attempt to police such restrictions; if at all unsure, check the spec sheet for the part. You have been warned!

NAND devices are widely considered to be arranged as one or more *partitions*, and the eCos NAND library supports this. However, there is no universal scheme for partition sizes to be supplied to the driver, unlike hard drives which encode a partition table into their first sector. Partition arrangements are often implicitly hardcoded, such as by byte address within the device, though they could be encoded in a "partition table", user-set, or even variable under software control by some esoteric rules. Therefore, every device driver is responsible for configuring its partition information as appropriate for the device, and this might for example appear as CDL options.

Tip

Be sure to read the notes associated with the device driver to understand how partitions are set up; if no notes are provided, look in its `devinit` code.

NAND device partitions

After a NAND device has been initialised, its device struct contains a list of partitions. These are numbered from 0 and may go up to `CYGNUM_NAND_MAX_PARTITIONS`-1. Before an application can use the NAND device, it must obtain a partition context (pointer) with the following call:

```
__externC cyg_nand_partition* cyg_nand_get_partition(cyg_nand_device *dev, unsigned partno);
```

Note

This call returns a pointer to the partition struct, not an error code. If the given partition number is inactive or invalid, it returns NULL.

About the spare area

Every page on the NAND array has a small number of "spare" bytes associated with it. These are used by the NAND library to store the page's ECC; whatever is left over may be used by the application for whatever purposes may suit it.

Every page has `CYG_NAND_APPSPARE_PER_PAGE(dev)` bytes of spare area available to the application. (This amount is implicit from the driver configuration and cannot change during the lifetime of a device.)

Note

Application spare bytes are not subject to the ECC. When reading the spare area data, you must be prepared to cope with the consequences of the (unlikely) event of a bit drop-out or other failure.

Manipulating the NAND array

Now, finally, given a cyg_nand_partition*, your application can make use of the NAND array with the following functions:

Reading data

```
__externC int cyg_nand_read_page(cyg_nand_partition  *ctx,
                                 cyg_nand_page_addr  page,
                                 void *              dest,
                                 size_t              size,
                                 void *              spare,
                                 size_t              spare_size);
```

Reads a single page and its spare area. The data read from the chip will be automatically ECC-checked and repaired if necessary. Parameters are as follows:

ctx
: The partition that data is to be read from.

page [1]
: The page to be read, *numbered from the start of the partition*. As a double-check, the library will refuse the operation with `-ENOENT` if this address is not within partition *ctx*.

dest
: Where to put the data. May be NULL, in which case the page data is not read.

size
: The maximum amount of data to read. (In any event, no more than a single page will be read, but if your application knows it doesn't need the whole page, you can place a cap here.)

spare
: Where to store the application data read from the spare area. This may be NULL if spare data is not required.

spare_size
: The maximum number of bytes to read from the spare area. This will not be more than `CYG_NAND_APPSPARE_PER_PAGE(dev)` bytes.

An error response of `-EIO` means that a multiple-bit I/O error has occurred in the page data, which the ECC could not repair. The library stores the data read from the device in `*dest` and `*spare` on a best-effort basis; it should not be relied upon. The application should take steps to salvage what it can and erase the block as soon as possible.

Writing data

```
__externC int cyg_nand_write_page(cyg_nand_partition  *ctx,
                                  cyg_nand_page_addr  page,
                                  const void *        src,
                                  size_t              size,
                                  const void *        spare,
                                  size_t              spare_size);
```

Writes a single page and its spare area. The ECC will be computed and stored automatically. Parameters are as follows:

ctx
: The partition that data is to be written to.

page
: The page to be written, *numbered from the start of the partition*. As a double-check, the library will refuse the operation with `-ENOENT` if this address is not within partition *ctx*.

src
: Where to read the data from. May be NULL, in which case the page data is not written.

size
: The amount of data to write. (In any event, no more than a single page will be written.)

[1] This was changed in application interface v2; earlier page and block addresses were device-relative.

spare
: Where to read the data to go into the spare area; it will automatically be packed around the ECC as necessary. Again, this may be NULL if spare data is not required.

spare_size
: The number of bytes to write to the spare area. This should not be larger than `CYG_NAND_APPSPARE_PER_PAGE(dev)`; if it is, only that many bytes will be stored.

An error response of `-EIO` means that the page write failed. The application should copy out any data it wishes to keep from the rest of the eraseblock, then call `cyg_nand_bbt_markbad()` to put the block beyond use.

Erasing blocks

```
__externC int cyg_nand_erase_block(cyg_nand_partition *ctx, cyg_nand_block_addr blk);
```

ctx
: The partition that data is to be erased from.

blk
: The block to be erased, *numbered from the start of the partition*. As a double-check, the library will refuse the operation with `-ENOENT` if this address is not within partition *ctx*.

An error response of `-EIO` means that the block erase failed. In this case, the library automatically marks the block as bad, and the application need take no further action.

Common error returns

The following common error returns may be encountered when manipulating the NAND array using the above functions:

`-EIO`
: The operation could not be completed due to an I/O error. This may require the application to take further action; check the details provided above for the call you have just made.

`-ENOENT`
: The page or block address was not valid for the given partition.

`-EINVAL`
: The page (block) address was (within) a block that is marked bad.

Ancillary NAND functions

The following functions are provided to allow applications to interact with the Bad Block Table:

```
typedef enum {
    CYG_NAND_BBT_OK=0,
    CYG_NAND_BBT_WORNBAD=1,
    CYG_NAND_BBT_RESERVED=2,
    CYG_NAND_BBT_FACTORY_BAD=3
} cyg_nand_bbt_status_t;

__externC int cyg_nand_bbt_query(cyg_nand_partition *ctx, cyg_nand_block_addr blk);

__externC int cyg_nand_bbt_markbad(cyg_nand_partition *ctx, cyg_nand_block_addr blk);
```

To determine the status of an eraseblock, use `cyg_nand_bbt_query`; this returns an enum from cyg_nand_bbt_status_t or a negative eCos error code. All blocks which return a non-0 enum value are considered inaccessible by applications.

Occasionally, it is necessary for applications to mark a block as bad. This most commonly happens when a write operation fails (see the section called “Writing data” above). To do this, call `cyg_nand_bbt_markbad`; the return is 0 for success, or a negative eCos error code. *As with other calls, blocks are numbered from 0 at the start of the partition, and internally translated for the device as appropriate.*

Both of these calls may foreseeably return `-ENOENT` if the given block address was not valid, or `-EIO` if something awful happened with the on-chip bad block table.

Chapter 55. Writing NAND device drivers

Planning a port

Before you start, you will need to have sight of appropriate spec sheets for both the NAND chip and the board into which it is connected, and you need to know how the chip is to be partitioned.

Driver structure and layout

A typical NAND device driver falls into two parts:

- high-level operations specific to the NAND chip (page reads and writes); and
- board-specific plumbing (sending commands and data to the chip; reading data back from the chip).

This distinction is important in the interests of code reuse; the same part may appear on different boards, or indeed multiple times, but connected differently. It need not be maintained if there are good reasons not to.

The *NAND library device interface* consists of a C struct, cyg_nand_device, comprising a number of data fields and function pointers. Each NAND chip to be made available to the library requires exactly one instance of this struct.

Tip

The cyg_nand_device structure includes a *`void* priv`* member which is treated as opaque. The driver may use this member as it sees fit; it is intended to provide an easy means to identify the NAND array, MMIO addresses or function pointers to use and so on. Typically this is used by the chip driver for its own purposes, and includes a further opaque member for the use of the HAL port.

The function pointers in the struct form the driver's high-level functions; they make use of the low-level functions to talk to the chip. We present the high-level functions first, although there is no intrinsic reason to prefer either ordering during driver development.

The high-level chip-specific functions are traditionally laid out as an *inline file* in an appropriate package in `devs/nand/`*`CHIP`*. The board-specific functions should normally appear in the platform HAL and #include the inline.

Chip partitions

Before embarking on the port, you should determine how the NAND array will be partitioned. This is necessarily a board-specific question, and your layout must accommodate any other software users of the array. You will need to know either the fixed layout - converted to eraseblock addresses - or how to determine the layout at initialisation time.

Tip

It may be worthwhile to set up partitioning by way of some parameters in your platform's CDL, with sensible defaults, instead of outright hard-coding the partition layout.

Locking against concurrent access

The eCos NAND library provides per-device locking, to guard against concurrent access during high-level operations. This support is fully automatic; drivers need take no action to make use of it.

This strategy may not be sufficient on all target boards: sometimes, accessing a NAND chip requires mediation by CPLD or other device, which must be shared with other NAND chips or even other peripherals. *If this applies, it is the responsibility of the driver and platform port to provide further locking as appropriate!*

Tip

When using mutexes in a driver, one should use the *driver API* as defined in `<cyg/hal/drv_api.h>` instead of the full kernel API. This has the useful property that mutex operations are very cheaply implemented when the eCos kernel is not present, such as when operating in RedBoot.

Required CDL declarations

An individual NAND chip driver must declare the largest page size it supports by means of CDL. This is done with a statement like the following in its cdl_package stanza:

```
requires ( CYGNUM_NAND_PAGEBUFFER >= 2048 )
```

Note

This requirement is due to the internal workings of the eCos NAND library: a buffer is required for certain operations which manipulate up to a NAND page worth of data, internally to the library. This is declared once as a global buffer for safety under low-memory conditions; a page may be too big to use temporary storage on the C stack, and the NAND library deliberately avoids the use of `malloc`.

By convention, a driver package would declare `CYGPKG_IO_NAND` as its `parent` and use `cyg/devs/nand` as its `include_dir`, but there is no intrinsic reason why this should be so.

High-level (chip) functions

The high-level functions provided by the chip driver are typically created as an *inline file* providing a fully-populated cyg_nand_dev_fns_v1 struct, instantiated by the CYG_NAND_FUNS macro. The high-level driver should not directly read or write to the hardware itself, but instead call into functions in the low-level driver.

The form the low-level functions should take is not prescribed; typically functions will be required to write commands to the device, to read and write data, and to query any status line which may be present. The high-level driver should normally provide a header file containing prototypes for the functions it requires from the low-level driver. (The low-level source file would provide the low-level functions required, include the high-level include, then instantiate the combined driver using the CYG_NAND_DEVICE macro.)

This source code layout is not intended as a prescription. It would for example be entirely in order to store pointers to the low-level functions in a struct and set *`priv`* to point to that struct, which could be useful in some cases.

Note

The device driver must not call `malloc` or otherwise allocate memory; all data should be in the stack or set as globals. This is because the driver may be required to run within a minimal eCos configuration.

These functions should all return 0 on success, or a negative eCos error code. In the event of an error, do *not* call back into the NAND library; use the NAND_CHATTER macro to report, in case a human is watching, and return an error code. The library will take care of ensuring the correct response to the application and updating the BBT as necessary.

Device initialisation

```
static int my_devinit (cyg_nand_device *dev);
```

The `devinit` function is the most complex, and logically one to write first. It is responsible for:

- initialising the device, typically by sending a reset command;

- interrogating the device to confirm its presence and properties;
- setting up the partition table list (see "Planning a port" above);
- setting up mutexes as necessary (see "Locking against concurrent access" above);
- populating the other members of the cyg_nand_device struct (see below).

Interrogating the device is normally performed by sending a *Read ID* command and examining the result, which typically encodes some or all of the chip parameters.

Given the similarity between many NAND parts, it may be possible to write a generic driver to cover all of one or more manufacturer's parts, or indeed for all ONFI-compliant parts. At the time of writing, this has not yet been attempted.

The `devinit` function must set up the following struct members:

`page_bits`
The size of the regular (non-spare) part of a page, expressed as the logarithm in base 2 of the number of bytes. For example, if pages are 2048 bytes long, *`page_bits`* would be 11. Obviously, the size of a page must be an exact power of two.

`spare_per_page`
The number of bytes of spare area available in each page.

`block_page_bits`
The base-2 log of the number of pages per eraseblock.

`blockcount_bits`
The total number of erase blocks in the device, expressed as a base-2 log.

`chipsize_log`
The total size of the chip, not counting the spare areas. This is required so that the library can double-check that the given parameters make sense by comparing with the preceding fields. Again, this field is itself a base-2 logarithm.

`bbt.data`
Space for the in-memory Bad Block Table for this device.

`bbt.datasize`
This is the size of *`bbt.data`*, in bytes. At present, this should be two bits times the number of blocks in the device; in other words, `1<<(blockcount_bits-2)` bytes.

The cyg_nand_device struct has two further members *`ecc`* and *`oob`* which must be set up to point to the ECC and OOB descriptors to use for the device. This is normally done by the CYG_NAND_DEVICE low-level instantiation macro, so will be better described in that section, but at this level you should be aware that it is also safe to set up the descriptor block during `devinit`. for example if multiple semantics might be you had included logic to detect what semantics to use.

The Bad Block Table itself is implemented in a way which intends to be compatible with the Linux MTD layer. A full parameter struct is not currently provided, though one may be in future.

Reading, writing and erasing data

The read and write operations are divided into three phases, with the following flow:

- Begin. This is called once; the driver should lock any platform-level mutex and send the command and address.
- Stride. This is called one or more times to read the page data from the device.

Note

The reason for this is if the platform provides a NAND controller with hardware ECC: it is often necessary to read out the ECC registers every so often.

- Finish. This is called once; it should read or write the spare area, (on programming) send a "program confirm" command and check its status, and unlock any platform-level mutex.

Erasing is a single-shot call which should lock any platform-specific mutex, send the command, check its status and unlock the mutex.

```
static int my_read_begin(   cyg_nand_device *dev, cyg_nand_page_addr   page);
static int my_read_stride(  cyg_nand_device *dev, void * dest,  size_t size);
static int my_read_finish(  cyg_nand_device *dev, void * spare, size_t spare_size);

static int my_write_begin(  cyg_nand_device *dev, cyg_nand_page_addr  page);
static int my_write_stride( cyg_nand_device *dev, const void * src,   size_t size);
static int my_write_finish( cyg_nand_device *dev, const void * spare, size_t spare_size);

static int my_erase_block(  cyg_nand_device *dev, cyg_nand_block_addr blk);
```

Searching for factory-bad blocks

```
static int my_is_factory_bad(cyg_nand_device *dev, cyg_nand_block_addr blk);
```

The very first time a NAND chip is used, the library has to scan it to check for factory-bad eraseblocks and build up the Bad Block Table. This function is called repeatedly to do so, one block at a time; it should return 1 if the block is marked bad, or 0 if the block appears to be OK.

Typically this function will invoke `read_page`; blocks are usually marked factory-bad by the presence of a particular signature in the out-of-band area of the first or second page of that block.

Warning

It is extremely important that you get this function right; after an eraseblock has been written to, it is no longer possible to reliably determine whether the block was factory-bad. It is never safe to assume that the factory-bad signature for a chip is the same as that of a similarly-sized chip or another by the same manufacturer; *always* check the correct spec sheet for the actual part or part-family in use!

Tip

Because this function is critical and a subtle error could cripple your application some time later in the field when it runs across undetected factory-bad blocks, you might find it handy to have a double-check before proceeding. If you enable CYGSEM_IO_NAND_READONLY in your eCos configuration during early development, you can safely fire up a test application (which calls `cyg_nand_lookup`) whilst watching the chatter output: the scan will be performed, but no BBT will be written. You can then compare the number of bad blocks reported against the manufacturer's specification of the maximum. Double-check that your `is_factory_bad` function is correct before enabling read-write mode!

Declaring the function set

```
CYG_NAND_FUNS_V2(mydev_funs, my_devinit,
        my_read_begin, my_read_stride, my_read_finish,
        my_write_begin, my_write_stride, my_write_finish,
        my_erase_block, my_is_factory_bad);
```

This macro ties the above functions together into a struct whose name is given as its first argument. The name of the resulting struct must be quoted when the driver is formally instantiated, which is normally done by the low-level functions.

Note

Earlier versions of this library used a slightly different device interface, keyed off the macro CYG_NAND_FUNS. This interface has been retired.

Low-level (board) functions

The set and prototypes of the functions required here will necessarily depend on the board and to a lesser extent on the NAND part itself. The following functionality is typically required:

- Very low-level hardware initialisation - for example, GPIO pin direction and interrupt config - if this has not already been done by the platform HAL
- Set up the chip partition table (see below)
- Runtime hardware config as required, such as commanding an FPGA or CPLD to route lines to the NAND part
- Write a command (byte)
- Write an address (handful of bytes)
- Write data, usually at the chip's full bus width (typically 8 or 16 bits)
- Read data at full bus width
- Read data at 8-bit width (if the chip has a 16 bit data bus, some commands - commonly ReadID - may return 8-bit data)
- Poll any status lines required or - if supported - set them up as interrupts to allow sleeping-wait

Talking to the chip

It is impossible to prescribe how to achieve this, as it depends entirely on how the NAND part is wired up on the board.

The ideal situation is that the NAND part is wired in via the CPU's memory controller and that the controller is set up to do most of the hard work for you. In that case, reading and writing the device is as simple as accessing the correct memory-mapped I/O address; usually different address ranges connect to the device's command, address and data registers respectively.

Tip

The HAL provides a number of macros in `<cyg/hal/hal_io.h>` to read and write memory-mapped I/O.

Note

On platforms with an MMU, MMIO may be rerouted to different addresses to those on the board spec sheet. Check the MMU setup in the platform HAL.

On some platforms, you may have to invoke an FPGA or CPLD to be able to talk to the NAND chip. This might typically take the form of a handful of MMIO accesses, but should hopefully be fairly straightforward once you've figured out how the components interrelate.

The worst case is where you have no support from any sort of controller hardware and have to bit-bang GPIO lines to talk to the chip. This is a much more involved process; you have to take great care to get the timings right with carefully tuned delays. The result is usually quite CPU intensive, and could be clock speed sensitive too; you should check for and take account of any CDL settings in the architecture and variant HAL which allow the CPU clock frequency to be changed.

Tip

If your low-level functions take a cyg_nand_device pointer as an argument, you can use its *`priv`* member to hold or point to some relevant data like the MMIO addresses to use, which is preferable to hard-coding them. Indeed, if you wish your board port to support more than one chip, you should use the *`priv`* member to distinguish between them.

Setting up the chip partition table

It is the responsibility of the high-level `devinit` function to set up the device's partition table. (It may be appropriate for it to invoke a low-level function to do this.)

The partition definition is an array of cyg_nand_partition entries in the cyg_nand_device.

```
struct _cyg_nand_partition_t {
    cyg_nand_device *dev;
    cyg_nand_block_addr first;
    cyg_nand_block_addr last;
};
typedef struct _cyg_nand_partition_t cyg_nand_partition;

struct _cyg_nand_device_t {
    …
    cyg_nand_partition partition[CYGNUM_NAND_MAX_PARTITIONS];
    …
};
```

Application-visible partition numbers are simply indexes into this array.

- On a live partition, *`dev`* must point back to the cyg_nand_device containing it. If `NULL`, the partition is inactive.
- *`first`* is the number of the first block of the partition.
- *`last`* is the number of the last block of the partition (*not* the number of blocks, unless the partition starts at block 0).

Putting it all together…

Finally, with everything else in place, we turn to the CYG_NAND_DEVICE macro to instantiate it.

```
CYG_NAND_DEVICE(my_nand, "onboard", &mydev_funs, &my_priv_struct, &linux_mtd_ecc, &nand_mtd_oob_64);
```

In order, the arguments to this macro are:

- The name to give the resultant cyg_nand_device struct;
- the device identifier string, application-visible to be used in `cyg_nand_lookup()` ;
- a pointer to the device high-level function set to use, normally set up by the CYG_NAND_FUNS macro;
- the *`priv`* member to include in the struct;
- a pointer to the ECC semantics block to use. `linux_mtd_ecc` provides software ECC compatible with the Linux MTD layer, but it is strongly recommended to use onboard hardware ecc support if this is present as it gives a huge speed boost. See the section called “ECC implementation” for more details.
- a pointer to the OOB-area layout descriptor to use (see `nand_oob.h` : nand_mtd_oob_16 and nand_mtd_oob_64 are Linux-compatible layouts for devices with 16 and 64 bytes of spare area per page respectively).

The macro invokes the appropriate linker magic to pull all the compiled NAND device structs into one section so the NAND library can find them.

ECC implementation

The use of ECC is strongly recommended with NAND flash parts owing to their tendency to occasionally bit-flip. This is usually done with a variant of a Hamming code which calculates column and line parity. The computed ECC is stored in the spare area of the page to which it relates.

The NAND library automatically computes and stores the ECC of data as it is written to the chip. On read, the code is calculated for the data actually read; this is compared with the stored code and the data repaired if necessary.

The NAND library comes with a software ECC implementation named `linux_mtd_ecc`. This is compatible with the ECC used in the Linux MTD layer, hence its name. It calculates a 3-byte ECC on a 256-byte data block. This algorithm is adequate for most circumstances, but it is strongly recommended to use any hardware ECC support which may be available because of the performance gains it yields. (In testing, we observed that up to two thirds of the time taken by every page read and program call was used in computing ECC in software.)

The ECC interface

This library draws a semantic distinction between *hardware* and *software* ECC implementations.

- A software ECC implementation will typically not require an initialisation step. The calculation function will always be called with a pointer to the data bytes to compute.
- A hardware implementation is assumed to read and act upon the data *as it goes past*. Therefore, it will not be passed a pointer to the data when its `calculate` step is invoked.

An ECC is defined by the following parameters:

- The size of data block it handles, in bytes.
- The size of ECC it calculates on those blocks, in bytes.
- Whether the algorithm is hardware or software.

An ECC algorithm must provide the following functions:

```
/* Initialises an ECC computation. May be NULL if not required. */
void my_ecc_init(struct _cyg_nand_device_t *dev);


/* Returns the ECC for the given data block.
 * If IS_HARDWARE:
 *  - dat and nbytes are ignored
 * If ! IS_HARDWARE:
 *  - dat and nbytes are required
 *  - if nbytes is less than the chunk size, the remainder are
 *    assumed to be 0xff.
 */
void my_ecc_calc(struct _cyg_nand_device_t *dev,
                 const CYG_BYTE *dat, size_t nbytes, CYG_BYTE *ecc);


/* Repairs the ECC for the given data block, if needed.
 * Call this if your read-from-chip ECC doesn't match what you computed
 * over the data block. Both *dat and *ecc_read may be corrected.
 *
 * `nbytes' is the number of bytes we're interested in; if a correction
 * is indicated outside of that range, it will be ignored.
 *
 * Returns:
 *       0 for no errors
 *       1 for a corrected single bit error in the data
 *       2 for a corrected single bit error in the ECC
 *      -1 for an uncorrectable error (more than one bit)
 */
int my_ecc_repair(struct _cyg_nand_device_t *dev,
                  CYG_BYTE *dat, size_t nbytes,
                  CYG_BYTE *ecc_read, const CYG_BYTE *ecc_calc);
```

In some cases - particularly where hardware assistance is in use - it is necessary to specify different functions for calculating the ECC depending on whether the operation at hand is a page read or a page write. In that case, two *init* and *calc* functions may be supplied, each taking the same prototype.

The algorithm parameters and functions are then tied together with one of the following macros:

```
CYG_NAND_ECC_ALG_SW(my_ecc,  _datasize, _eccsize, my_ecc_init, my_ecc_calc, my_ecc_repair);

CYG_NAND_ECC_ALG_HW(my_ecc,  _datasize, _eccsize, my_ecc_init, my_ecc_calc, my_ecc_repair);

CYG_NAND_ECC_ALG_HW2(my_ecc, _datasize, _eccsize, my_ecc_init, my_ecc_calc_read,
                     my_ecc_calc_write, my_ecc_repair);

CYG_NAND_ECC_ALG_HW3(my_ecc, _datasize, _eccsize, my_ecc_init_read, my_ecc_init_write,
                     my_ecc_calc_read,  my_ecc_calc_write,    my_ecc_repair);
```

Tip

It's OK to use software ECC while getting things going, but if you do then switch to a hardware implementation, you probably need to erase your entire NAND chip including its Bad Block Table. The `nanderase` utility may come in handy for this.)

Warning

You must be sure that your ECC repair algorithm is correct. This can be quite tricky to test. However, it is often possible to hoodwink the controller into computing ECCs for you even if the data is not going to affect the data stored on the NAND chip, for example if you send it data but haven't told it to program a page. A variant of the `sweccwalk` test may come in handy for this purpose.

An example implementation, including an ECC calculation and repair test named `eccwalk`, may be found in the STM3210E evaluation board platform HAL, `packages/hal/cortexm/stm32/stm3210e_eval`. The chip NAND controller has on-board ECC calculation, but does not undertake to repair data; a repair function was written specially.

Chapter 56. Tests and utilities

Unit and functional tests

The NAND library includes a number of tests. The most useful to driver writers are `readwrite`, `rwbenchmark` and `sweccwalk`; the others are only likely to be of interest to library maintainers.

readwrite

Performs a read-write-erase cycle on the first NAND device it finds, checking that its operations have had the expected effect on the device contents. This is a potentially destructive test; do not run it on a device containing data you care about!

rwbenchmark

A more involved version of `readwrite`, this is a timing test which performs multiple reads, writes and erases and applies statistical techniques to the results in the same way that `tm_basic` instruments the speed of various eCos kernel functions. *This is a potentially destructive test; do not run it on a device containing data you care about!*

sweccwalk

Repeatedly makes single-bit changes to a data buffer and checks that the software ECC implementation correctly repairs them.

Tip

This test can be adapted to test out hardware ECC implementations. The test outputs the raw ECC codes as it goes, which is useful in confirming that the bits in the computed ECC are what you think they are.

nandunit

Some unit tests which do not require any NAND device: ECC known answer vectors, and OOB area packing/unpacking correctness.

readlimits

Attempts to read a block outside of a partition, confirming that it doesn't work.

There are some further tests of the library which require the synthetic NAND device.

Ancillary NAND utilities

The following utilities are included with the NAND library. They are standalone eCos applications; for convenience, you can set `CYGBLD_IO_NAND_BUILD_UTILS` in your eCos configuration and they will be built and placed into `install/tests/io/nand/current/utils`.

Warning

It is particularly dangerous to run the utility `erase_bbt_dangerous.c` on a production device, as it is generally not possible to later reconstruct the list of factory-bad blocks. It is intended only as an aid to driver authors.

erasenand.c

Loops over all the blocks of a partition, erasing all the blocks [1] which are not marked as bad. The device and partition to erase are set by #define.

erase_bbt_dangerous.c[2]

Erases the NAND blocks comprising the primary and mirror bad-block tables of a device. The device to erase is set by #define. (The tables are detected by the library in the usual way. If

[1] This will not normally erase the Bad Block Table. This is because the BBT reports its own blocks as "Reserved" when queried via `cyg_nand_bbt_query`, which makes them inaccessible to applications. However, if `CYGSEM_IO_NAND_USE_BBT` is turned off, then any BBT present will not be detected and hence will be erased.

[2] Note the warning regarding running this on a production device

none are present, the library will scan the device for factory-bad blocks to create such a table, then this code will immediately erase it.)

Chapter 57. The eCos configuration store

Overview

The eCos configuration store is a simple typed key/value store which uses NAND flash for its persistent storage.

The library is aimed at applications that wish to store simple configuration data without the overhead of a fully NAND-aware filesystem. It is also used by RedBoot to store persistent configuration data.

The following functionality is provided:

- Write data
- Read data
- Erase individual data items
- List and dump out store keys and contents (for debugging)

Design limitations

- The data which may be stored is limited to a total of one NAND block, including the store's internal metadata.
- The store is designed to be robust but simple. It is not expected to scale well; if there are a great many items in the store, read access will be slow.
- The entire store is rewritten on every write; this means that write access to a busy store will similarly be slow.
- The store is NAND-aware and incorporates simple wear-levelling logic but excessive numbers of writes will still risk burning out the NAND array. If the store is allocated only a small number of NAND blocks, this will exacerbate the effect. It is recommended to allow a reasonable number of blocks (5-10) to allow for blocks wearing out over the lifetime of the device.
- Only simple locking is used to prevent corruption; all config store operations block until they are able to secure the protecting mutex.

Using the config store

The main entry points to the config store logic are as follows:

```
/* From <cyg/configstore/write.h>.
 * These functions write out a key, overwriting it if it is already there.
 * They return 0 for success or a negative errno value; see the header
 * file for details.
 */
externC
int cyg_configstore_write_int(const char *key, cyg_uint32 i);
externC
int cyg_configstore_write_bool(const char *key, cyg_bool b);
externC
int cyg_configstore_write_bytes(const char *key, void *src, cyg_uint32 len);
externC
int cyg_configstore_write_string(const char *key, const char *data);

/* Erases a single key */
externC
int cyg_configstore_erase_keystr(const char *key);
```

```
/* From <cyg/configstore/read.h>.
 * These functions read out a key or header.
 * They return 0 for success or a negative errno value; see the header
 * file for details. */

externC
int cyg_configstore_read_int(const char *key, cyg_uint32 *i);
externC
int cyg_configstore_read_bool(const char *key, cyg_bool *b);

/* Note:
* For bytes and strings, check *len_io after read to see the actual number
* of bytes read, INCLUDING the trailing NUL. */
externC
int cyg_configstore_read_bytes(const char *key, CYG_BYTE *buf, unsigned *len_io);
externC
int cyg_configstore_read_string(const char *key, char *buf, unsigned *len_io);

/* Reading out only the header allows you to check a key's type and size. */
externC
int cyg_configstore_read_header(const char *key, cyg_configstore_header_t *hdr);
```

```
/* From <cyg/configstore/util.h>. */

/* Lists all keys in the store (to diag_printf).
 * Not really machine-readable; intended for human-read debugging. */
externC
void cyg_configstore_list(void);

/* Dumps out everything in the store (to diag_printf).
 * Intended for human-read debugging.
 * NOTE: This may emit large amounts of output, which may
 * take an excessive length of time over a slow debug channel. */
externC
void cyg_configstore_dump(void);
```

For more details of the types and structures used, refer to `<cyg/configstore/serialise.h>` and `<cyg/configstore/record.h>` .

Note

Both store keys, and strings in the store, should not contain the ASCII NUL (0x00) character. Behaviour in this case is undefined.

Locking

The config store uses mutexes in order to prevent corruption by concurrent access. If `CYGPKG_KERNEL` is loaded in your eCos configuration, the config store automatically inherits the configured mutex behaviour.

Configuration

If the NAND array reports an error when writing or erasing a block, the config store will automatically retry the operation, up to `CYGNUM_CONFIG_STORE_RETRIES` times. The default setting is 3 retries.

The config store is allocated a single NAND partition. The device and partition are configured by the CDL options `CYGDAT_CONFIG_STORE_DEVICE` and `CYGNUM_CONFIG_STORE_PARTITION.` Normally `CYGDAT_CONFIG_STORE_DEVICE` is set by the platform HAL; `CYGNUM_CONFIG_STORE_PARTITION` may also be hard-wired, if RedBoot or other boot loader needs it.

To configure partition sizes, refer to the eCos HAL documents for your platform. It is recommended to allow a reasonable number of blocks (5-10) for the config store, in order to allow for blocks wearing out over the lifetime of the device.

Caution

1. If the config store is to be shared between multiple clients - for example, RedBoot and an application - the partition geometry must be configured identically to both of them. Be aware that changing the geometry in CDL will not update RedBoot unless you also reconfigure, rebuild and reflash it!
2. If other applications write other data to the NAND array, care should be taken to not overwrite the config store. It is recommended to give them their own partition.

Storage details

The config store uses a single NAND block, conceptually contiguous from its component NAND pages but of course read and written a single page at a time.

The data block has the following contents:

- Magic number `CYG_CONFIGSTORE_MAGIC_HEADER` .
- Block serial number. These allow us to detect old versions and automatically clean them up. Serial Number Arithmetic (RFC1982) is used to compare.
- Zero or more records, each introduced by the magic number `CYG_CONFIGSTORE_MAGIC_RECORD.`
- Magic number `CYG_CONFIGSTORE_MAGIC_FOOTER` . This allows us to detect an incompletely-written block.

Every page written by the config store also contains magic numbers `CYG_CONFIGSTORE_ECOS_TAG` and `CYG_CONFIGSTORE_MAGIC_TAG` in the out-of-band (spare) area identifying it as belonging to the store. This allows us to attempt to be a tolerant neighbour and not erase data that appears to belong to another (possibly misconfigured) client of the NAND array.

Each record has the following contents:

- The key. This is a null-terminated string to the user, though the null is *not* stored on NAND. This is stored in the same way as a string (see below).
- The record type. This is one of Integer, Boolean, String or Bytes.
- The length of the data.
- The length of the data including padding.
- The data itself.
- Any padding required (see below).

Padding

Everything written is padded to the nearest 4-octet boundary.

- Integers are always stored unsigned as 4 octets in host byte order.
- Booleans are stored as integers with 1 mapping to True and 0 to False.
- Strings and Bytes are stored as a tuple (data length, data, padding). The difference is that strings are null-terminated in RAM; the trailing null is stripped on write and restored on read.

Scanning

On every access, both read and write, the config block is scanned for consistency. Any obsoleted or incompletely-written blocks are automatically erased. When writing, the old block is only erased once the new block has been completely written.

Part XVI. NAND Device Drivers

Table of Contents

Chapter 58. Samsung K9 family NAND chips

Overview

The `CYGPKG_DEVS_NAND_SAMSUNG_K9` driver package currently provides support for the Samsung K9F1G08, K9F2G08 and K9F1208 series of NAND flash chips, and is intended to be expanded to provide support for more of the K9 family.

Most users will not need to interact with this package; it should be included as a hardware dependency on all appropriate targets. This package provides only an inline code fragment which is intended to be instantiated by the target platform HAL and provided with appropriate board-specific low-level functions allowing it to access the hardware.

Notes

1. The large-page parts in this family are not quite ONFI-compliant, but this code could probably be extended to a much wider set of chips - or indeed to the ONFI specification - without too much trouble. Appropriate definitions will be required for the chip identifier, decoding of the Read ID response, and the chip's blockcount-bits and device-size fields.
2. At the present time, this driver has the limitation that it only supports 8-bit parts. This is an area of probable future expansion.

Using this driver in a board port

This driver's top-level chip support is currently provided as two files:

`cyg/devs/nand/k9_generic.h`
: Prototypes the low-level chip access functions required by the chip driver and declares a private struct for use by the driver.

`cyg/devs/nand/k9_generic.inl`
: Implements high-level chip functions and exposes them via the `CYG_NAND_FUNS` macro. This file is not intended to be compiled on its own, but to be included by the source file in a platform HAL which implements the low-level functions.

A platform HAL would typically make use of this driver in a single source file with the following steps:

- Declare a local private struct with contents as appropriate,
- `#include <cyg/devs/nand/k9_generic.h>`
- implement the required low-level functions,
- `#include <cyg/devs/nand/k9_generic.inl>`
- declare a list of the supported `k9_subtype` which may appear on the board, terminated by `K9_SUBTYPE_SENTINEL`
- declare a static instance of the `k9_priv` struct containing pointers to that list and to an instance of the local-private struct
- finally, instantiate the chip with the `CYG_NAND_DEVICE` macro, selecting the ECC and OOB semantics to use.

Note

If there is more than one chip on the board, each needs its own CYG_NAND_DEVICE with a distinct name and device name, its own instance of the k9_priv struct.

For more details about the infrastructure provided by the NAND layer and the semantics it expects of the chip driver, refer to Chapter 53, *eCos NAND Flash Library*. An example driver instantiation can be found in the NAND driver for the EA LPC2468 platform.

Memory usage

As discussed in the section called “High-level (chip) functions”, the chip initialisation function must set up the `bbt.data` pointer in the cyg_nand_device struct. This driver does so by including a large byte array in the k9_priv struct whose size is controlled by CDL depending on the enabled chip support. That struct is intended to be allocated as a static struct in the data or BSS segment (one per chip), which avoids adding a dependency on `malloc`.

Low-level functions required from the platform HAL

These functions are prototyped in `k9_generic.h`. They have no return value ("void"), except for `read_data_1` which returns the byte it has read.

`write_cmd(device, command)`
: Writes a single command byte to the chip's command latch.

`write_addrbytes(device, pointer to bytes, number of bytes)`
: Writes a number of address bytes in turn to the chip's address latch.

`CYG_BYTE read_data_1(device)`, `read_data_bulk(device, output pointer, number of bytes)`
: Reads data from the device, respectively a single byte and in bulk.

`write_data_1(device, byte)`, `write_data_bulk(device, data pointer, number of bytes)`
: Writes data to the device, respectively a single byte and in bulk.

`wait_ready_or_time(device, initial delay, fallback time)`
: Wait for the chip to signal READY line or, if this line is not available, fall back to a worst-case time delay (measured in microseconds).

`wait_ready_or_status(device, mask)`
: Wait for the chip to signal READY line or, if this line is not available, enter a loop waiting for its Status register (ANDed with the given mask) to be non-zero.

`k9_devlock(device)`, `k9_devunlock(device)`
: Hooks for any board-specific locking which may be required in addition to the NAND library's chip-level locking. (This would be useful if, for example, access to multiple chips was mediated by a single set of GPIO lines which ought not to be invoked concurrently.)

`k9_plf_init(device)`
: Board-level platform initialisation hook. This is called very early on in the chip initialisation routine; it should set up any locking required by the devlock and devunlock functions, interrupts for the driver and any further lines required to access the chip as approprate. *Once this has returned, the chip driver assumes that the platform is fully prepared for it to call the other chip access functions.*

`k9_plf_partition_setup(device)`
: Board-level partition initialisation hook. This should set up the `partition` array of the device struct in a way which is appropriate to the platform. For example, the partitions may be set as fixed ranges of blocks, or by CDL. This is called at the end of the chip initialisation routine and may, for example, call into the chip to read out a "partition table" if one is present on the board. *If you do not set up partitions, applications will not be able to use the high-level chip access functions provided the NAND library.*

Chapter 59. ST Microelectronics NANDxxxx3a chips

Overview

The CYGPKG_DEVS_NAND_ST_NANDXXXX3A driver package provides support for the NANDxxxx3A chip family by ST Microelectronics.

Most users will only need to add this package to their eCos configuration and not need to interact with it further. This package provides only an inline code fragment which is intended to be instantiated by the target platform HAL and provided with appropriate board-specific low-level functions allowing it to access the hardware.

Using this driver in a board port

This driver's chip support is currently provided as two files:

cyg/devs/nand/nandxxxx3a.h
: Prototypes the low-level chip access functions required by the chip driver, declares a private struct for use by the driver and provides a NANDXXXX3A_DEVICE macro for convenience.

cyg/devs/nand/nandxxxx3a.inl
: Implements high-level chip functions and exposes them via the CYG_NAND_FUNS macro. This file is not intended to be compiled on its own.

A platform HAL would typically make use of this driver in a single source file with the following steps:

- Declare a private struct and one or more static instances of it as appropriate,
- #include <cyg/devs/nand/nandxxxx3a.h>
- implement the required low-level functions,
- #include <cyg/devs/nand/nandxxxx3a.inl>
- finally, instantiate the chip with the NANDXXXX3A_DEVICE macro the appropriate number of times, giving each chip an appropriate name and its own private struct if need be, declaring its size, and selecting the ECC and OOB semantics to use.

For more details about the infrastructure provided by the NAND layer and the semantics it expects of the chip driver, refer to Chapter 53, *eCos NAND Flash Library*. An example driver instantiation can be found in the platform HAL for the STM3210E-EVAL board.

Memory usage note

As discussed in the section called “High-level (chip) functions”, the chip initialisation function must set up the bbt.data pointer in the cyg_nand_device struct. This driver does so by including pointer to a sufficiently large byte array in the nandxxx3a_priv struct. That struct is intended to be allocated as a static struct in the data or BSS segment (one per chip), which avoids adding a dependency on malloc.

Low-level functions required from the platform HAL

These functions are prototyped in nandxxxx3a.h. They have no return value ("void"), except where indicated.

`write_cmd(device,command)`
: Writes a single command byte to the chip's command latch.

`write_addrbytes(device, pointer to bytes, number of bytes)`
: Writes a number of address bytes in turn to the chip's address latch.

CYG_BYTE`read_data_1(device)`, `read_data_bulk(device, output pointer, number of bytes)`
: Reads data from the device, respectively a single byte and in bulk.

`write_data_1(device, byte)`, `write_data_bulk(device, data pointer, number of bytes)`
: Writes data to the device, respectively a single byte and in bulk.

`wait_ready_or_time(device, initial delay, fallback time)`
: Wait for the chip to signal READY or, if this line is not available, fall back to a worst-case time delay (measured in microseconds).

`wait_ready_or_status(device, mask)`
: Wait for the chip to signal READY or, if this line is not available, enter a loop waiting for its Status register (ANDed with the given mask) to be non-zero.

`nandxxxx3a_devlock(device)`, `nandxxxx3a_devunlock(device)`
: Hooks for any board-specific locking which may be required in addition to the NAND library's chip-level locking. (This would be useful if, for example, access to multiple chips was mediated by a single set of GPIO lines which ought not to be invoked concurrently.)

int `nandxxxx3a_plf_init(device)`
: Board-level platform initialisation hook. This is called very early on in the chip initialisation routine; it should set up any locking required by the devlock and devunlock functions, interrupts for the driver and any further lines required to access the chip as approprate. *Once this has returned, the chip driver assumes that the platform is fully prepared for it to call the other chip access functions.*

int `nandxxxx3a_plf_partition_setup(device)`
: Board-level partition initialisation hook. This should set up the `partition` array of the device struct in a way which is appropriate to the platform. For example, the partitions may be set as fixed ranges of blocks, or by CDL. This is called at the end of the chip initialisation routine and may, for example, call into the chip to read out a "partition table" if one is present on the board. *If you do not set up partitions, applications will not be able to use the high-level chip access functions provided the NAND library.*

Chapter 60. Micron MT29F family NAND chips

Overview

The CYGPKG_DEVS_NAND_MICRON_MT29F driver package currently provides support for the Micron MT29F2G08 NAND flash chip, and is intended to be expanded to provide support for more of the MT29F family.

Most users will not need to interact with this package; it should be included as a hardware dependency on all appropriate targets. This package provides only inline code fragments which are intended to be included and instantiated by the target platform HAL and provided with appropriate board-specific low-level functions allowing it to access the hardware.

Notes

1. The large-page parts in this family are not quite ONFI-compliant, but this code could probably be extended to a much wider set of chips - or indeed to the ONFI specification - without too much trouble. Appropriate definitions will be required for the chip identifier, decoding of the Read ID response, and the chip's blockcount-bits and device-size fields.

2. At the present time, this driver has the limitation that it only supports 8-bit parts. This is an area of probable future expansion.

Using this driver in a board port

This driver's top-level chip support is currently provided as three files:

`cyg/devs/nand/mt29f_generic.h`
: Prototypes the low-level chip access functions required by the chip driver and declares a private struct for use by the driver.

`cyg/devs/nand/mt29f_generic.inl`
: Implements high-level chip functions and includes mt29f_generic_lp.inl. This file is not intended to be compiled on its own, but to be included by the source file in a platform HAL which implements the low-level functions.

`cyg/devs/nand/mt29f_generic_lp.inl`
: Implements those high-level chip functions which are specific to large-page chips, completing the driver and exposing it via the CYG_NAND_FUNS macro. This file is not intended to be compiled or included directly by platform code.

A platform HAL would typically make use of this driver in a single source file with the following steps:

- Declare a local private struct with contents as appropriate,
- `#include <cyg/devs/nand/mt29f_generic.h>`
- implement the required low-level functions,
- `#include <cyg/devs/nand/mt29f_generic.inl>`
- declare a list of the supported `mt29f_subtype` which may appear on the board, terminated by `MT29F_SUBTYPE_SENTINEL`
- declare a static instance of the `mt29f_priv` struct containing pointers to that list and to an instance of the local-private struct
- finally, instantiate the chip with the `CYG_NAND_DEVICE` macro, selecting the ECC and OOB semantics to use.

Note

If there is more than one chip available on the board, each needs its own CYG_NAND_DEVICE with a distinct name and device name, its own instance of the mt29f_priv struct.

For more details about the infrastructure provided by the NAND layer and the semantics it expects of the chip driver, refer to Chapter 53, *eCos NAND Flash Library*.

Memory usage

As discussed in the section called “High-level (chip) functions”, the chip initialisation function must set up the `bbt.data` pointer in the cyg_nand_device struct. This driver does so by including a large byte array in the mt29f_priv struct whose size is controlled by CDL depending on the enabled chip support. That struct is intended to be allocated as a static struct in the data or BSS segment (one per chip), which avoids adding a dependency on `malloc`.

Low-level functions required from the platform HAL

These functions are prototyped in `mt29f_generic.h`. They have no return value ("void"), except for `read_data_1` which returns the byte it has read.

`write_cmd(device, command)`
: Writes a single command byte to the chip's command latch.

`write_addrbytes(device, pointer to bytes, number of bytes)`
: Writes a number of address bytes in turn to the chip's address latch.

`CYG_BYTE read_data_1(device)`, `read_data_bulk(device, output pointer, number of bytes)`
: Reads data from the device, respectively a single byte and in bulk.

`write_data_1(device, byte)`, `write_data_bulk(device, data pointer, number of bytes)`
: Writes data to the device, respectively a single byte and in bulk.

`wait_ready_or_time(device, initial delay, fallback time)`
: Wait for the chip to signal READY line or, if this line is not available, fall back to a worst-case time delay (measured in microseconds).

`wait_ready_or_status(device, mask)`
: Wait for the chip to signal READY line or, if this line is not available, enter a loop waiting for its Status register (ANDed with the given mask) to be non-zero.

`mt29f_devlock(device)`, `mt29f_devunlock(device)`
: Hooks for any board-specific locking which may be required in addition to the NAND library's chip-level locking. (This would be useful if, for example, access to multiple chips was mediated by a single set of GPIO lines which ought not to be invoked concurrently.)

`mt29f_plf_init(device)`
: Board-level platform initialisation hook. This is called very early on in the chip initialisation routine; it should set up any locking required by the devlock and devunlock functions, interrupts for the driver and any further lines required to access the chip as appropriate. *Once this has returned, the chip driver assumes that the platform is fully prepared for it to call the other chip access functions.*

`mt29f_plf_partition_setup(device)`
: Board-level partition initialisation hook. This should set up the `partition` array of the device struct in a way which is appropriate to the platform. For example, the partitions may be set as fixed ranges of blocks, or by CDL. This is called at the

end of the chip initialisation routine and may, for example, call into the chip to read out a "partition table" if one is present on the board. *If you do not set up partitions, applications will not be able to use the high-level chip access functions provided the NAND library*.

Name

Synthetic Target NAND Flash Device — Emulate NAND flash hardware in the synthetic target

Overview

The device driver `CYGPKG_DEVS_NAND_SYNTH` emulates NAND flash hardware inside the eCos synthetic target. In addition it provides a number of debug facilities which cannot readily be implemented on real embedded hardware, including:

1. The emulated NAND contents are held on a file in the Linux host. This makes it easy to archive and restore NAND images, allowing test runs to be repeated with the exact same state each time.
2. The device driver can log details of all NAND I/O to a separate logfile in the Linux host. This makes it easier to work out exactly what is happening in the application, and more importantly it can help with figuring out what went wrong when. For extended runs it is possible to limit the disk space used for logging. It is also possible to generate checkpoints, where the current NAND image is saved to a separate file.
3. It is possible to inject bad blocks at run-time, to check how the application would cope on real hardware if and when a NAND erase block developed a fault. These can be made to affect random blocks or specific blocks, for example ones holding filesystem metadata.

Some of the functionality is always available and uses compile-time configuration via CDL. This allows applications to be run stand-alone. The more advanced functionality such as logging and bad block injection is only available when running in conjunction with the synthetic target I/O auxiliary, when *`--io`* is used on the command line. The settings for logging and bad block injection usually come from the `default.tdf` target definition file. These can be changed on a per-run basis by adding *`--nanddebug`* to the command line, which will cause a suitable dialog box to pop up during NAND driver initialization.

Compile-time Configuration

This package `CYGPKG_DEVS_NAND_SYNTH` will automatically be loaded when creating a new eCos configuration for the Linux synthetic target. However the package will be inactive until the generic NAND support is added to the configuration.

The synthetic target NAND driver has been designed to be functional both when running stand-alone and when used with the I/O auxiliary. Hence some of the basic parameters of the emulated NAND device must be specified at compile-time, and this is handled via CDL configuration options.

`CYGDAT_NAND_SYNTH_FILENAME` specifies the host-side file that will be used to hold the NAND data. The default is `synth_nand.dat` in the current directory. If the file does not exist then the driver will create it during initialization. All data in a newly-created image file will be set to 0xFF, corresponding to an erased device. Hence deleting the current image file makes it possible to start a test run with a blank NAND device.

A NAND device consists of some number of erase blocks: erase operations affect all data in an erase block. Erase blocks are made up of some number of pages, and write operations typically affect a page at a time. Each page consists of a main data block plus some spare bytes, also known as out of band or OOB data. There are four CDL configuration options controlling the size and layout of the emulated flash device: `CYGNUM_NAND_SYNTH_BLOCK_COUNT`, `CYGNUM_NAND_SYNTH_PAGES_PER_BLOCK`, `CYGNUM_NAND_SYNTH_PAGESIZE`, and `CYGNUM_NAND_SYNTH_SPARE_PER_PAGE`. The default settings are 1024 erase blocks, 32 pages per block, 2K of data per page, and 64 bytes of OOB data per page. This gives an emulated device size of 64MB plus 2M OOB.

The size and layout parameters are encoded in each NAND image file. If these configuration options are changed then existing image files will be incompatible and the device driver will report a fatal error at run-time. This avoids compatibility problems with higher-level code: if a file system has formatted the NAND device for a 2K page size then it is likely to get very confused if the page size suddenly changes to 512 bytes.

The NAND device can be partitioned manually by enabling the component `CYGSEM_DEVS_NAND_SYNTH_PARTITION_MANUAL_CONFIG` and manipulating the options below this. The default is for a single partition occupying the entire NAND device.

Option CYGSEM_NAND_SYNTH_RANDOMLY_LOSE activates code in the driver which triggers frequent bit errors during read operations. These should be handled by error correcting codes within the generic NAND layer so should be transparent to higher-level code. The option exists mainly as an easy way of testing the automatic error correction support.

Run-time Customization

Logging and bad block injection are controlled by run-time customization via the synthetic target I/O auxiliary, not by compile-time CDL options. This allows the same test executable to be run with and without logging or with different sequences of injected bad blocks. If the executable is run without the I/O auxiliary, without the *--io* command line option, then both logging and bad block injection will be disabled.

The main way of customizing both logging and bad block injection is via the target definition file, usually default.tdf. The driver comes with a file nand.tdf holding the various options and explanatory text. This file should be incorporated into default.tdf and edited as appropriate. Note that all NAND-related settings should be inside a synth_device nand section.

Logging

The target definition file settings related to logging are as follows:

```
logfile             "/tmp/synthnand.log"
log                 read write erase error
max_logfile_size    16M
number_of_logfiles  4
generate_checkpoint_images
```

The logfile setting controls the location of the logfile. The default is to add a suffix .log to the CYGDAT_NAND_SYNTH_FILENAME setting, thus creating logfiles in the same directory as the NAND image file.

The log setting specifies which events should be logged. It is followed by a list of some or all of the following: read, READ, write, WRITE, erase, and error. read logs all calls to the driver's page read function, but not the data actually read. READ is like read but logs the actual data as well as the event. Similarly write and WRITE log calls to the driver's page write function, without or with the data being written. erase logs calls to the driver's block erase function. error logs any bad block injection events.

Logging can quickly generate very large files, especially when READ or WRITE debugging is enabled. This can have unfortunate side effects, for example an overnight stress test can fail because the logfile has filled all available disk space. To avoid this it is possible to limit the size of each logfile using the max_logfile_size setting. This is simply a number followed by a unit K, M or G.

When max_logfile_size is exceeded the NAND driver takes appropriate action. The default behaviour is just to delete the current logfile and create a new one, with the same name as before. This can be unfortunate if some particularly interesting event happened just before the maximum logfile size was exceeded because all logging information related to that event will have been lost. To avoid this it is possible to have multiple logfiles, limited by the number_of_logfiles setting. Assume a logfile name of /tmp/synthnand.log, a maximum logfile size of 16M and four logfiles. After the first 16MB of logging data has been written to /tmp/synthnand.log that file will be renamed to /tmp/synthnand.log.0 and a new current logfile /tmp/synthnand.log will be created. After another 16MB the current logfile will be renamed to /tmp/synthnand.log.1, and so on. After 64MB the maximum allowed number of logfiles has been created, so /tmp/synthnand.log.0 will be deleted and then /tmp/synthnand.log will be renamed to /tmp/synthnand.log.3. Hence the maximum disk space used will be between 48MB and 64MB, plus a small amount of overflow for each logfile. All logfiles are in plain text, one event per line, and a single event will not be spread over multiple logfiles.

In addition to the logfiles the NAND driver will generate image checkpoint files if generate_checkpoint_images is enabled. At the start of the test run the driver will copy the current NAND image to a new file /tmp/synthnand.log.checkpoint, using the same base filename as for logfiles. If support for multiple logfiles is enabled then the current checkpoint file will be renamed in the same way and at the same time as the current logfile, and a new checkpoint file will be created using the current image data. Hence for any logfile it is possible to examine both the starting image and the final image (which may be the current one).

Bad Block Injection

There are two settings related to bad block injection: `factory_bad` and `inject`. The former is straightforward:

```
factory_bad 17 42 256 1019
```

This setting is used only when the NAND image file does not yet exist and the driver has to create a new one. The specified erase blocks are marked as factory-bad and hence unusable. The setting consists simply of one or more numbers, each in the range 0 to `CYGNUM_NAND_SYNTH_BLOCK_COUNT-1`. A maximum of 32 blocks can be marked factory bad.

Bad block injection is rather more complicated, in an attempt to make it sufficiently flexible for a variety of uses. The target definition file can contain one or more `inject` settings. There is a limit of eight erase definitions and eight write definitions, giving a maximum of sixteen bad block injection definitions. However some of these can use `repeat`, so the number of bad blocks injected during a test run is limited only by the size of the emulated device. Example settings are:

```
inject erase current after rand% 1024 erases
inject write current after rand% 100000 calls repeat
inject erase block 1 after 3 block_erases
inject write page 9860 after 1000 writes
```

The `inject` keyword should be followed by either `erase` or `write`. If `erase` then the bad block injection happens during a call to the driver's erase function, otherwise it happens during a call to write. This is followed by a block or page definition, the keyword `after`, an event counter, and a couple of optional flags.

The simplest block or page definition is the keyword `current`. This simply means that whichever block is being erased, or whichever block contains the page being written, will be marked bad. Typically this will be used for injecting random faults. If a given block is the subject of an above-average number of erase or write operations then it is more likely to be the current block in one of these definitions, so heavily-used blocks are more likely to fail.

The alternative to `current` is to list a specific block for an erase definition, or a specific page for a write definition. Blocks go from 0 to `CYGNUM_NAND_SYNTH_BLOCK_COUNT` - 1. Pages go from 0 to (`CYGNUM_NAND_SYNTH_BLOCK_COUNT * CYGNUM_NAND_SYNTH_PAGES_PER_BLOCK`) - 1. This can be particularly useful when testing higher-level code that uses certain blocks specially, for example for storing filesystem metadata. It can also be useful when attempting to repeat a test run using information from a logfile.

The fields immediately following the `after` keyword specify when the bad block injection should trigger. They consist of the optional keyword `rand%`, a count, and an event identifier. If `rand%` is not specified then exactly the specified number of events must occur before the bad block injection triggers. Otherwise some number between 0 and count-1 events must occur. The event identifier can be one of `erases`, `writes`, `calls`, `block_erases`, or `page_writes`. `erases` means the number of calls to the driver's block erase function. `writes` means the number of calls to the page write function. `calls` means the total number of read, write or erase calls. `block_erases` means erase calls for a specific block. Similarly `page_writes` means write calls for a specific page. `block_erases` and `page_writes` cannot be used together with `current`, only with a specific block or page.

So, consider the first example again:

```
inject erase current after rand% 1024 erases
```

The driver will calculate a random number between 0 and 1023, say 427. Once there have been at least 427 erase calls the bad block injection will trigger. Since the definition is for erase current, the injection can happen immediately. Hence whichever block is specified during the 427th erase call will be marked bad and that erase call will fail with error code `EIO`.

Suppose that the definition used `write current` instead of `erase current`. The definition will still trigger during erase call 427, but it will not take effect immediately. Instead whichever page is the subject of the next write operation will be marked bad. Alternatively, suppose that the definition was for `erase block 42` but call 427 was for block 512 instead. If some time after call 427 there was another erase call for block 42, that later erase call will fail and cause the block to be marked bad.

Now consider the next definition:

```
inject write current after rand% 100000 calls repeat
```

This event will trigger after between 0 and 99999 calls into the driver. These calls can be reads, writes, or erases. If the triggering call is a write then the block affected will be the one containing the page being written. Otherwise whichever page is the subject of the next write operation will be the one affected.

```
inject erase block 1 after 3 block_erases
```

This definition will only ever affect block 1. The first two calls to erase block 1 will succeed. The third call will fail. Erase calls for any other block have no effect on this definition.

```
inject write page 9860 after 1000 writes
```

This definition will only ever affect page 9860. The definition will trigger after 1000 writes to any page. If the thousandth write happens to be for page 9860, it will fail. Otherwise the next write for page 9860 will fail, whenever that happens. If the definition specified event type `page_writes` instead of `writes` it would trigger only after 1000 writes to page 9860 instead of 1000 writes to any page.

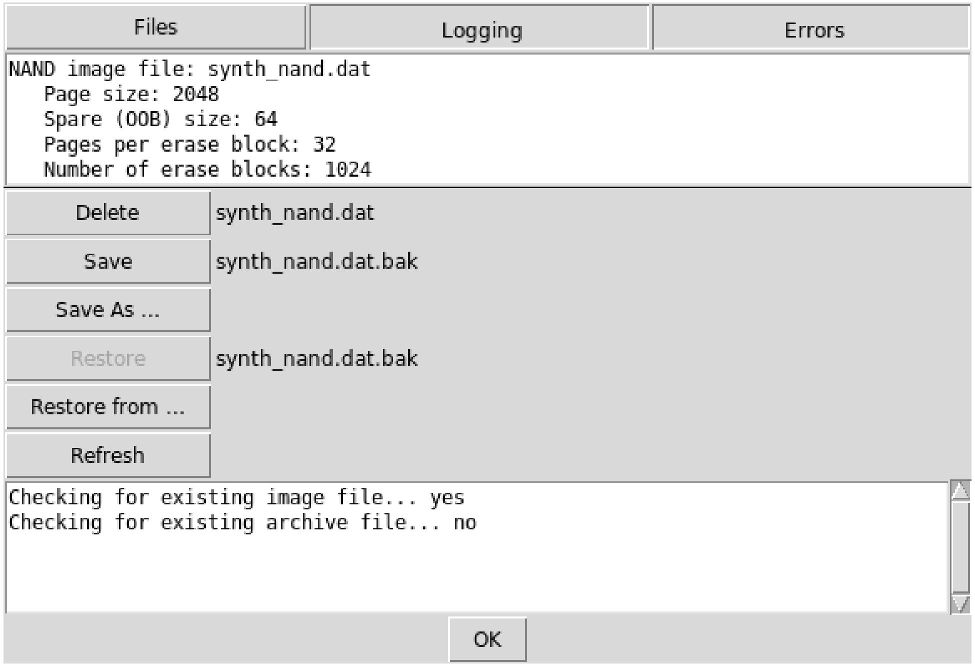
Trigger definitions can be followed by two optional keywords. The first is `repeat` and indicates that the definition should trigger multiple times, not just once. `repeat` can only be used for block or page `current`, not for a specific block or page, since any given block can only fail once. The second keyword is `disabled`. This can be used to create a bad block injection definition which is not active by default but which can be enabled in the GUI interface.

All bad block injection definitions operate in parallel, not sequentially. It is possible for multiple definitions to trigger during a single call but a given block can only fail once.

Interactive Dialog

Although it is possible to change the logging and other settings between test runs by editing the target definition file, the package also provides a way of changing these during driver initialization time. If the command line option *`--nanddebug`* is used in addition to *`--io`* then the I/O auxiliary will pop up a dialog box allowing the various settings to be edited.

Figure 2. I/O auxiliary Dialog, Files



```
NAND image file: synth_nand.dat
   Page size: 2048
   Spare (OOB) size: 64
   Pages per erase block: 32
   Number of erase blocks: 1024
```

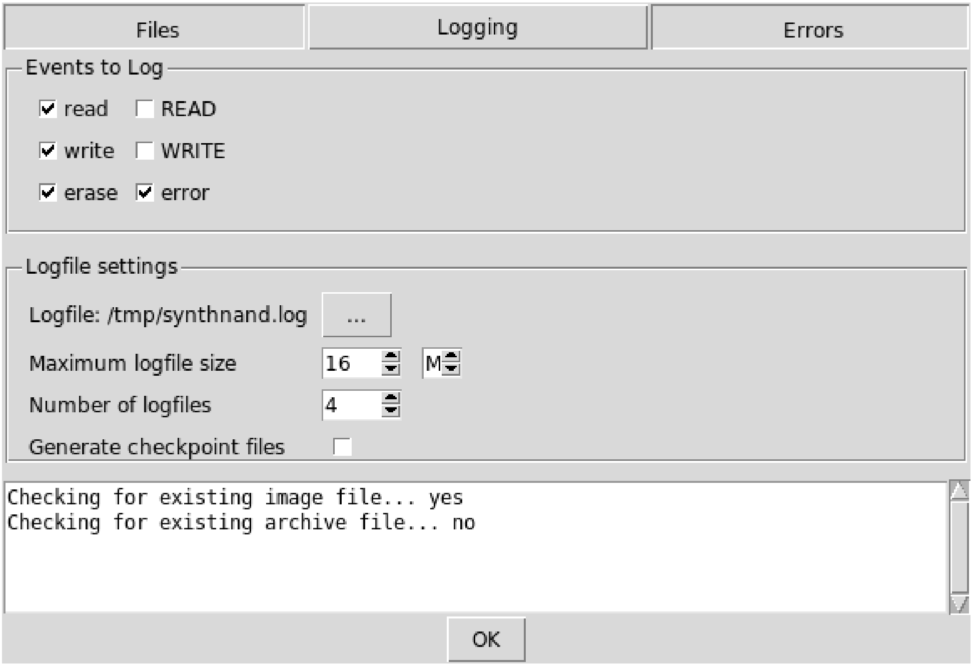
Delete: synth_nand.dat
Save: synth_nand.dat.bak
Restore: synth_nand.dat.bak

```
Checking for existing image file... yes
Checking for existing archive file... no
```

The dialog consists of a tabbed notebook with separate tabs for Files, Logging, and Errors. The default tab is Files. This shows the NAND device settings as configured via CDL options. If a current NAND image file exists then it can be deleted with a single click, allowing the test run to proceed with a new blank NAND device. The current image file can be saved away to an archive, either using a default name based on the current image name or by letting the user select a file. It is also possible to restore a previously-saved archive, again using the default name if that archive exists, or by selecting an alternative file. If any of the relevant files are changed outside this dialog then the refresh button forces the dialog to check the filesystem again. The various operations will be reported at the bottom of the dialog.

If the Logging tab is selected then the dialog changes to the following:

Figure 3. I/O auxiliary Dialog, Logging



Logfile: /tmp/synthnand.log

Maximum logfile size 16

Number of logfiles 4

Generate checkpoint files □

```
Checking for existing image file... yes
Checking for existing archive file... no
```

There are separate checkbuttons for the six types of events that can be logged. It is also possible to change the current logfile and to edit the other settings related to logfiles.

The Errors tab is not yet implemented.

The dialog can be dismissed using the OK button or by hitting the ESC key. At that point the eCos application will resume running with the selected settings.

File Formats

A NAND image is a binary file with six sections: a 64-byte header block; an array of erase counts; an array of write counts; an array of the factory-bad blocks; a bitmap of the good and bad blocks; and the actual data. The header consists of the following data structure:

```
struct header {
  cyg_uint32 magic;           // 0xEC05A11F
  cyg_uint32 page_size;       // CYGNUM_DEVS_NAND_SYNTH_PAGESIZE
  cyg_uint32 spare_size;      // CYGNUM_DEVS_NAND_SYNTH_SPARE_PER_PAGE
  cyg_uint32 pages_per_block; // CYGNUM_DEVS_NAND_SYNTH_PAGES_PER_BLOCK
  cyg_uint32 number_of_blocks;// CYGNUM_DEVS_NAND_SYNTH_BLOCK_COUNT
  cyg_uint32 tv_sec;
  cyg_uint32 tv_usec;
  cyg_uint32 spare[9];
};
```

All integers in the header and in the following three sections are stored in bigendian format. The *`magic`* field is used to check that a file really holds a NAND image. The next four fields specify the emulated device size and layout, as per the corresponding CDL options. The *`tv_sec`* and *`tv_usec`* fields are filled in with the result of a `gettimeofday()` system call during driver initialization. These fields also appear in logfiles, so code can check that a logfile and an image file correspond to the same test run.

Following the header is an array of BLOCK_COUNT integers holding the number of erase calls for each block. Next is an array of (BLOCK_COUNT * PAGES_PER_BLOCK) integers holding the number of write calls for each page. These arrays can be used to check that higher-level code is performing wear levelling.

The array of factory-bad blocks consists of 32 integers holding the settings of the target definition's file `factory_bad` setting at the time that the image file was created.

The bitmap following the factory-bad block array holds the current state of each erase block. Bit 0 of byte 0 corresponds to erase block 0, bit 0 of byte 1 corresponds to erase block 8, and so on. If a bit is set then the erase block is ok. If a bit is clear then the erase block is bad, either factory-bad or because it has failed subsequently due to a bad block injection.

The bad block bitmap is followed by the actual data. This consists of all erase blocks concatenated without padding, starting with erase block 0. Each erase block is stored starting from page 0 within that block, and again all the pages are concatenated without padding. For each page the actual data is stored first, followed by the spare (OOB) data.

A logfile is a plain text file, not a binary file. It holds one log event per line. Some of these lines can be rather long if READ or WRITE logging is enabled. The fields within each line are separated by a single space. The first field indicates the type of record. The next two fields are call counts, one for the event in question and one for the total number of calls into the driver. The remaining fields depend on the record type.

```
I 0 0 1247514233 562000 synth_nand.dat 2048 64 32 1024
```

This is an initialization record when the logfile is created. There have been no calls to the driver yet so the two counts are 0. The next two fields are the *`tv_sec`* and *`tv_usec`* timestamp values which are also written into the NAND image file. This allows logfiles and image files to be matched up. The remaining fields identify the page size, the spare size, the number of pages per erase block, and the number of erase blocks.

```
F 2 4 43 0
```

This is a call into the driver's `factorybad` function. It is the second such call, and the fourth call into the driver. The query is for erase block 42, and that block has not been marked as factory-bad.

```
 r 5 12 32736 0x0200f0c0 2048 0x0200eeb0 64
```

This is a `read` event. It is the fifth page read into the driver and the 12th call. The request is for page number 32736. 2048 bytes of data should be read into a buffer at location 0x0200f0c0, and 64 bytes of OOB data should be read into 0x0200eeb0.

```
Rd 5 12 32736 0x0200f0c0 2048 FFFFFFFFFFFF…
Ro 5 12 32736 0x0200eeb0 64   FFFFFFFFFFFF…
```

These are two `READ` log events corresponding to the previous `read`. The first line `Rd` is for the data part, and the final field consists of 4096 bytes of hexadecimal data. The second line `Ro` is for the OOB part and the final field consists of 128 bytes of hexadecimal data.

```
w 1 1030 32736 0x0200f0c0 2048 0x0200eed0 64
Wd 1 1030 32736 0x0200f0c0 2048 FFFFFFFF3FFFFFFFFFCF…
Wo 1 1030 32736 0x0200eed0 64 FFFFFFFFFFFFFFFF426274…
```

These lines show a `write` log event and a `WRITE` log event for the same call into the driver. The fields are the same as for `read` and `READ`.

```
E 2 1031 1022
```

This logs an erase call into the library for block 1022. It is the second erase call, and by this time there have been 1031 calls into the driver.

```
Bb 1 2857 631
…
Bp 2 4012 3189 99
```

These lines show bad block injections. The first is for an erase operation for block 631. That erase call is about to fail with EIO and the block will be marked bad. The second is for a write operation for page 3189, which is part of erase block 99. That write operation is about to fail and all of erase block 99 will be marked bad.

Installation

Before a synthetic target eCos application can use a NAND device it is necessary to build and install host-side support. The relevant code resides in the `host` subdirectory of the synthetic target NAND package and building it involves the standard **configure**, **make** and **make install** steps. The implementation of the NAND support does not require any executables, just a Tcl script `nand.tcl` and some support files, so the **make** step is a no-op.

There are two main ways of building the host-side software. It is possible to build both the generic host-side software and all package-specific host-side software, including the NAND support, in a single build tree. This involves using the **configure** script at the toplevel of the eCos repository. For more information on this, see the `README.host` file at the top of the repository. Note that if you have an existing build tree which does not include the synthetic target NAND support then it will be necessary to rerun the toplevel configure script: the search for appropriate packages happens at configure time.

The alternative is to build just the host-side for this package. This requires a separate build directory, building directly in the source tree is disallowed. The **configure** options are much the same as for a build from the toplevel, and the `README.host` file can be consulted for more details. It is essential that the NAND support be configured with the same `--prefix` option as other eCos host-side software, especially the I/O auxiliary provided by the synthetic target architectural HAL package, otherwise the I/O auxiliary will be unable to locate the NAND support.

Test programs

bbt

Bad Block Table unit test. Finds a readable block, then fiddles with its status in the BBT confirming expected behaviour. Requires the synthetic NAND device.

multipagebbt

As for *bbt* but insists that the device parameters mean that the BBT spans multiple pages on-chip. (This is perhaps a contrived case, but might crop up in future with larger devices, so needed to be tested.)

eccdamage

An ECC error fuzzing exercise. Requires `CYGSEM_NAND_SYNTH_RANDOMLY_LOSE`, which induces pseudo-random bit errors; after 1,000 runs, the number of errors corrected is reported.

Part XVII. Journalling Flash File System v2 (JFFS2)

Name

CYGPKG_FS_JFFS2 — Provides Journalling file system for Flash

Description

The Journalling Flash File System version 2 (JFFS2) provides a robust file system to allow reliable use of NOR Flash devices as data storage. The eCos implementation is greatly shared with the Linux kernel implementation, thus ensuring compatibility and encouraging development.

JFFS2 was designed from the outset for embedded devices. It allows recovery when the system has failed abnormally, without the file system itself being left in an unusable state, even if power is disconnected at the moment the Flash device is in the middle of being written to. It also offers features such as compression for efficient data storage, and garbage collection to improve capacity. Most importantly it is fully integrated into the eCos file I/O infrastructure as a plug-in filesystem.

External references

There are a number of external resources containing information about JFFS2 on the internet, other than the usual eCos-specific general resources. The key site is the Linux MTD website which, although clearly having a Linux focus, contains lots of useful documentation on JFFS2 as well as a mailing list with searchable archives. The mailing list welcomes questions on using JFFS2 on eCos. Note that the eCos JFFS2 port does not use the MTD layer itself.

Another useful site is the Red Hat JFFS2 website, which contains a very useful paper presented to a Linux symposium covering the internals and some of the design of JFFS2.

Name

JFFS2 usage — Description of how to use JFFS2

Mounting

A JFFS2 filesystem can be mounted just like any normal eCos filesystem, using the `mount()` function from the POSIX file I/O package (`CYGPKG_FILEIO`). You must choose an appropriate Flash I/O block device to use. Documentation on Flash I/O block devices can be found in the Generic Flash package documentation.

Example 1. Mounting and unmounting a JFFS2 filesystem

```
#include <cyg/fileio/fileio.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
    …
    int rc;
    rc = mount( "/dev/flash/fis/jffs2test", "/fs", "jffs2" );
    if (rc < 0)
       printf( "mount returned error: %s\n", strerror(errno) );
    …
    rc = umount( "/fs" );
    if (rc < 0)
       printf( "umount returned error: %s\n", strerror(errno) );
    …
```

No file system needs to be created in advance for JFFS2. A new file system image will be instantiated if JFFS2 is pointed at an erased Flash area. Similarly if JFFS2 is pointed at a non-erased Flash area that does not contain valid JFFS2 markers, it will refuse to mount it to prevent destruction of data.

If the Flash device supports locking, you should ensure that the flash region for the JFFS2 filesystem is unlocked before mounting, otherwise it will not be possible to write to the Flash. Alternatively, enabled the CDL option `CYGSEM_FS_JFFS2_UNLOCK_FLASH` to unlock Flash on mounting. This option is disabled by default for safety reasons — if Flash has been locked, it may be for an important reason.

If mounting an existing JFFS2 filesystem, the mount procedure will search for any unused blocks that have not already been erased, and erase them. This can result in an extended mount time, and so this feature can be disabled with the CDL option `CYGOPT_FS_JFFS2_ERASE_PENDING_ON_MOUNT`.

Normally it is sufficient to prepare a clean JFFS2 partition as above, and load files into it using RedBoot or an eCos application. But if you do wish to use a pregenerated file system image generated on your host PC, there is a utility named **mkfs.jffs2** which can be used to generate an image. On Ubuntu, Fedora and Red Hat-based distributions of Linux it can be found in a package named mtd-utils; in Windows with Cygwin, in the mtd package. Be sure to use the `-l` or `-b` options to select the endianness so it corresponds with your target, and you may need to set other options according to the requirements of your Flash hardware such as erase block size. The erase block size to use on the **mkfs.jffs2** command line should be the largest flash block size used by your JFFS2 partition on the flash device. Note that the `-s` or `--pagesize` options are unrelated to the flash block size, and instead control JFFS2's view of the page size - see here for information on changing the value. Use the `--help` option for a complete list of **mkfs.jffs2** parameters.

Note that JFFS2's memory requirements are not static, and so they may increase over time before stabilising. Larger Flash partitions may require non-trivial amounts of memory, especially at mount time. Memory use may be controlled by removing features such as compression, or by constraining the size of the Flash partition. Configuration options controlling optional features may be found in the JFFS2 package CDL configuration.

JFFS2 has built-in tolerance of Flash errors when erasing and should adapt and work around bit errors that arise as Flash reaches the end of its working life. Obviously this comes at the expense of device capacity.

Garbage collection

By default, JFFS2 performs garbage collection on an as-needed basis. This means that when there are insufficient spare clean flash blocks remaining, JFFS2 will perform repeated garbage collection until a block can be erased and then used for future writes.

Garbage collection involves scanning a flash block and determining which nodes are still used for valid data and which are obsolete - valid data is then relocated to a new alternative block, until only obsolete data remains, at which point the block can be erased. The algorithms which choose which block is nominated for garbage collection do so to ensure wear levelling over the life of the flash device. Nodes representing individual file fragments are able to be coalesced and merged with adjacent file fragments leading to reduced flash use overall due to eliminating some of the overhead caused by node metadata on the Flash. There will also be a consequent simplification in the internal filesystem data structures, resulting in reduced memory consumption and fewer data structures which JFFS2 needs to trawl through when locating data. It will also greatly augment the benefit of compression: when compressing, nodes are considered in isolation, and so small nodes are unlikely to compress well, whereas larger coalesced nodes are more likely to occupy less flash space.

Garbage collection thread

As a result of only performing garbage collection when needed, it can mean that individual writes to files may occasionally take a lengthy period of time to run if a new flash block is required - a whole flash block may need to be garbage collected from scratch, have its live data written to a new block, and be erased, the latter in particular being a very lengthy procedure. As such, JFFS2 offers the option of using a *garbage collection thread*, which can run in the background to advance the garbage collection process when the filesystem is not otherwise being used.

Of course if a filesystem is going to change too rapidly, or the application is CPU bound by higher priority threads, then the garbage collection thread may not be able to keep up. But for the majority of applications, it can considerably reduce or even virtually eliminate the delays caused by the requirement for occasional garbage collection.

The garbage collection thread can be enabled in the JFFS2 package's configuration using the "Garbage collection background thread" CDL option (`CYGOPT_FS_JFFS2_GC_THREAD`). The priority of that thread defaults to the lowest possible - 1 above that of the idle thread, but this can be changed with the `CYGNUM_JFFS2_GC_THREAD_PRIORITY` option. The thread of course requires a stack for execution, the value of which can be optimised with the `CYGNUM_JFFS2_GC_THREAD_STACK_SIZE` option. Note that one garbage collection thread is started for each mounted JFFS2 filesystem, and so one thread stack is allocated for each.

Usually the garbage collection thread will gradually garbage collect nodes from erase blocks until the block is completely unused by valid data. However it would not actually erase it, leaving that to the usual code path that erases blocks only at the point they are needed. This is because otherwise the Flash system may get locked for the duration of the erase process, preventing any threads, including high priority threads, access to it at that time. Yet the garbage collection thread is intended to be a low priority background thread. Nevertheless, enabling the option `CYGSEM_JFFS2_GC_THREAD_CAN_ERASE` allows the garbage collection thread to erase blocks as well, if blocks are available for erasure.

Finally, the garbage collection thread may run continuously but does not have to run constantly. By default, the thread will be inactive unless the amount of free space in the filesystem is beginning to run low. The filesystem code makes a judgement based on a myriad of factors to decide when to activate this background GC thread (although its low thread priority may result in it not running in practice, so choose its thread priority wisely in relation to the other running threads). Once active, the garbage collection thread will run until sufficient free space is now available, even if more garbage collection may be possible.

As a variation on this behaviour, it is possible to periodically wake the GC thread to advance garbage collection even when the available free space is not yet low; thus preventing it getting close to running out in the first place. This feature is controlled with the `CYGNUM_JFFS2_GC_THREAD_TICKS` configuration option, which gives the number of ticks between garbage collection passes, even when space is not low.

How ticks corresponds to real time is unspecified and depends on the HAL and kernel clock configuration (although most ports have traditionally defaulted to 10ms ticks). The value of this option can even be set to 0. Note though, that garbage collection is never considered "done" - the thread will run continuously until the filesystem is unmounted, therefore making it run too frequently may in fact cause unnecessary flash operations, increasing flash wear (even though it will be levelled wear). The value should be chosen according to the expected write pattern.

The number of ticks between garbage collection passes can also be set at runtime, by invoking a `cyg_fs_setinfo()` call on a filesystem object. There is a corresponding `cyg_fs_getinfo()` call to retrieve the current delay ticks. For example:

```
#include <cyg/fs/jffs2/jffs2.h>
    …
    {
        int err;
        cyg_tick_count_t old_delay_ticks;
        cyg_tick_count_t new_delay_ticks;

        err = cyg_fs_getinfo( "/jffs2",
                              FS_INFO_JFFS2_GET_GC_THREAD_TICKS,
                              &old_delay_ticks,
                              sizeof(cyg_tick_count));
        assert(err == 0);
        new_delay_ticks = old_delay_ticks*10; // slow down GC
        err = cyg_fs_setinfo( "/jffs2",
                              FS_INFO_JFFS2_SET_GC_THREAD_TICKS,
                              &new_delay_ticks,
                              sizeof(cyg_tick_count));
        assert(err == 0);
    }
```

One application of dynamically setting the wakeup delay for the garbage collection thread is so that the thread is more active in times of relative system activity, but operates more slowly when quiescent. This may improve flash life, while still retaining benefits of the garbage collection thread.

The value of the wakeup delay can be set to a special value of `(cyg_tick_count_t)-1`, which indicates that the thread should not wake up periodically any more, but only when space is running low (the behaviour when `CYGNUM_JFFS2_GC_THREAD_TICKS` is disabled). Similarly retrieving the current wakeup tick value with `cyg_fs_getinfo()` may return this special value.

Efficiency

Write size

JFFS2 does not guarantee 100% optimal use of Flash space due to its journalling nature, and the granularity of Flash blocks. It is possible for it to fill up even when not all file space appears to have been used, especially if files have had many small operations performed on them and the Flash partition is small compared to the size of the Flash blocks. It is strongly recommended to have at least 5 or 6 Flash blocks spare, over and above space requirements for data, in order to allow the JFFS2 garbage collector to operate.

It is certainly the case that JFFS2 will work very inefficiently if using many small writes. Unless and until garbage collected, each write will occupy its own JFFS2 node on Flash, and so will incur overhead from the node header. A filesystem is likely to "fill" quickly if written a few bytes at a time, rather than in large chunks, so caution in advised, and more space reserved if that write pattern is anticipated. A common application that can do this is logging. Small writes can also remove the benefits of compression, as there is too little data to compress effectively.

Garbage collection

Use of the garbage collection thread will provide a level of continuous garbage collection. As indicated above, garbage collection can reduce flash usage and memory consumption, and improve performance. Therefore doing so on a continuous basis is a wise idea.

Extra spare space is required in order to allow the JFFS2 garbage collector to operate, over and above space requirements for data. A rule of thumb is to use the following formula:

```
Recommended overhead == 2 + (( flashsize/50 ) + (flashsectors*100) + (sectorsize-1)) / sectorsize
```

So for example, a 16Mbyte flash with 64Kbyte blocks given over entirely to JFFS2 would actually require an overhead of 6 blocks. Or to look at it another way, trying to write data when you have used up all but 6 blocks worth may result in an ENOSPC error being reported. Due to metadata and write characteristics (e.g. lots of 1 byte writes) it's not possible to easily calculate what that actually translates to in terms of maximum file size. Even the above formula is only a rule of thumb, and it has not been proven to be guaranteed to work in all circumstances. It is recommended to be conservative if possible.

Compression

JFFS2 will default to trying to compress files. However, it may be more memory efficient to disable JFFS2 compression entirely in the CDL configuration, and instead ensure that images are stored compressed when they are downloaded, and use the standard RedBoot mechanism to decompress the files upon loading.

Maximum data node size

By default JFFS2 will operate on chunks of files up to 4 kilobytes in size, but larger chunks may be able to be compressed more efficiently, and have lower metadata overheads. To increase the size, you must change JFFS2's view of the machine page size - the eCos JFFS2 port's view of the page size does not actually need to reflect any real underlying page size of the memory management system, and the notion of the page size is a hangover from the Linux origins of JFFS2 which would be too disruptive to remove. Changing the page size can be performed by changing the page size exponent configuration option (`CYGNUM_LINUX_COMPAT_PAGE_SIZE_EXPONENT`) in the Linux compatibility layer package (`CYGPKG_LINUX_COMPAT`). A value of 12 indicates 2^{12} which is 4 kilobytes. For example this could be changed to 16, corresponding to 2^{16} which is 64 kilobytes. If using **mkfs.jffs2**, make sure that its value for the page size, using the `-s` or `--pagesize` options, is the same or lower than the page size given by `CYGNUM_LINUX_COMPAT_PAGE_SIZE_EXPONENT`.

Configuration dependencies

JFFS2 has a number of package dependencies. As such it may be helpful to use the below eCos minimal configuration (.ecm) file and import it into your configuration to satisfy most dependencies quickly without conflict. This minimal configuration file is usable for building both eCos and RedBoot with JFFS2 included. Note you may need to modify the package versions from *`current`* to the version of your release, e.g. *`v2_0_64`*.

```
cdl_configuration eCos {
    package CYGPKG_IO_FLASH current ;
    package CYGPKG_MEMALLOC current ;
    package CYGPKG_COMPRESS_ZLIB current ;
    package CYGPKG_IO_FILEIO current ;
    package CYGPKG_FS_JFFS2 current ;
    package CYGPKG_ERROR current ;
    package CYGPKG_LINUX_COMPAT current ;
    package CYGPKG_IO current ;
    package CYGPKG_CRC current ;
    package CYGPKG_LIBC_STRING current ;
};

cdl_option CYGPKG_IO_FILEIO_DEVFS_SUPPORT {
    user_value 1
};
cdl_component CYGPKG_IO_FLASH_BLOCK_DEVICE {
    user_value 1
};
```

For example:

```
$ ecosconfig new adderII
$ ecosconfig import jffs2.ecm
$ ecosconfig tree
$ make tests
```

Use with RedBoot

JFFS2 support can be built into RedBoot using the above minimal configuration file. In most cases, the configuration settings will then make all the adjustments necessary.

However note that a build of RedBoot which includes JFFS2 with RedBoot, is likely to require more Flash space for its own image, as well as much more RAM space to run. The latter is particularly important to note given that this can reduce the size of the program image which can be loaded into RAM from a JFFS2 filesystem.

Particularly large JFFS2 filesystems, or filesystems with a large number of nodes, require more RAM to be used for JFFS2's in-memory data structures. As such, the value of the configuration option controlling the size of the RedBoot heap (`CYGMEM_REDBOOT_WORKSPACE_HEAP_SIZE`) may need to be increased in such cases. JFFS2 will already make the default size of this heap occupy 192KiB of RAM.

Secure Erase

The eCosPro® port of JFFS2 includes a *Secure Erase* feature. This feature allows the application to ensure that when a file is deleted, its contents are fully erased from the flash.

Usually deleted files persist in Flash for an indeterminate period of time, marked as obsolete. The possible solution taken with other filesystems of trying to overwrite the file data before deletion does not work with JFFS2, as JFFS2 will still retain the old file data in Flash, but writes additional nodes with a higher node version number, and rendering the previous data obsolete. Therefore the Secure Erase functionality can be used to guarantee that a deleted file will have its past contents wiped from the Flash.

Methodology

Because obsolete data for a file could exist in any block, the only way to achieve this is to ensure that every block (other than bad or completely free blocks) is wiped, taking care to relocate live data. This effectively means methodically garbage collecting and erasing every flash block used by the filesystem.

Operation time

For a filesystem which almost completely consists of used flash blocks the secure erase process could take a considerable amount of time during which the filesystem cannot be used for other operations. Therefore it is strongly recommended that the garbage collection thread support is enabled with an appropriate value for `CYGNUM_JFFS2_GC_THREAD_TICKS`. With the garbage collection thread running, more blocks are likely to be completely clean, or at least partially garbage collected, thus reducing the time for secure erasure.

Usage

Support for the secure erase functionality must first be enabled with a CDL configuration option - `CYGOPT_FS_JFFS2_SECERASE`.

Then a secure erase operation can be performed on the filesystem with a `cyg_fs_setinfo()` function call using the `FS_INFO_SECURE_ERASE` config key. This call can be invoked specifying any file or directory within the filesystem including its mount point, although the operation itself will take place on the entire filesystem.

Example 2. Secure erase usage

```
#include <cyg/fileio/fileio.h>
#include <errno.h>
#include <stdio.h>
    …
    int err;
    err = cyg_fs_setinfo("/fs", FS_INFO_SECURE_ERASE, NULL, 0);
    if (ENOERR != err)
    {
      printf( "Secure erase failed: %d\n", strerror(err) );
      …
```

Testing JFFS2

JFFS2 comes with a number of tests that may be run as normal eCos test applications. If you are running the tests under the RedBoot ROM monitor, you should create a FIS partition in RedBoot named “jffs2test”.

Alternatively, for standalone applications, if the platform HAL defines the `CYG_HAL_IO_FLASH_TEST_DEVICES` macro, the code will step through the vector entries supplied by that macro using the referenced flash device entry in conjunction with the `offset` and `size` values supplied.

Without the “jffs2test” named FIS partition, or a platform supplied vector, you must set the CDL configuration options `CYGNUM_FS_JFFS2_TEST_OFFSET` and `CYGNUM_FS_JFFS2_TEST_LENGTH`. In this case the first valid flash device (“`/dev/flash/0`”) will be used.

The tests will attempt to use the region identified by the offset/length combination, but will first check if the area is blank, and will report a test failure if it is not.

When the tests run, they will erase the Flash test area (usually the “jffs2test” FIS partition) in its entirety, so do not use an existing JFFS2 partition in this space.

The tests are designed to test both general features of JFFS2, as well as do a limited stress-test JFFS2 in the presence of multiple threads.

More specifically, the `jffs2-fileio1` test checks a wide variety of file system operations including creating and removing files and directories, scanning directories, and reading and writing file contents. It also repeats to verify that unmounting and remounting works.

The `jffs2-fseek1` test verifies file seek operations on JFFS2 files, using standard I/O C library calls.

Test files with names of the form `jffs2-`*`NtNf`* verify operation with varying numbers of threads, and varying numbers of files.

The test `jffs2_3` is specifically to verify operation of the garbage collection code, and performs a small set of operations repeatedly to do so. It also gives an opportunity for the garbage collection thread to be tested, if enabled.

The test `jffs2-secerase1` is specifically to verify operation of the secure erase facility, if the `CYGOPT_FS_JFFS2_SECERASE` CDL configuration option has been enabled. It also provides further testing of the garbage collection code and the garbage collection thread.

Part XVIII. NOR Flash Support

Table of Contents

Chapter 61. The eCos NOR FLASH Library

The NOR FLASH library is an optional part of eCos, and is only applicable to some platforms.

The eCos NOR FLASH library provides the following functionality:

1. Identifying installed device of a FLASH family.

2. Read, erasing and writing to FLASH blocks.

3. Validating an address is within the FLASH.

4. Determining the number and size of FLASH blocks.

There are two APIs with the flash library. The old API is retained for backwards compatibility reasons, but should slowly be replaced with the new API which is much more flexible and does not pollute the name space as much.

Notes on using the NOR FLASH library

FLASH devices cannot be read from when an erase or write operation is active. This means it is not possible to execute code from flash while an erase or write operation is active. It is possible to use the library when the executable image is resident in FLASH. The low level drivers are written such that the linker places the functions that actually manipulate the flash into RAM. However the library may not be interrupt safe. An interrupt must not cause execution of code that is resident in FLASH. This may be the image itself, or RedBoot. In some configurations of eCos, ^C on the serial port or debugging via Ethernet may cause an interrupt handler to call RedBoot. If RedBoot is resident in FLASH this will cause a crash. Similarly, if another thread invokes a virtual vector function to access RedBoot, eg to perform a `diag_printf()` a crash could result.

Thus with a ROM based image or a ROM based Redboot it is recommended to disable interrupts while erasing or programming flash. Using both a ROMRAM or RAM images and a ROMRAM or RAM RedBoot are safe and there is no need to disable interrupts.

Danger, Will Robinson! Danger!

Unlike nearly every other aspect of embedded system programming, getting it wrong with FLASH devices can render your target system useless. Most targets have a boot loader in the FLASH. Without this boot loader the target will obviously not boot. So before starting to play with this library its worth investigating a few things. How do you recover your target if you delete the boot loader? Do you have the necessary JTAG cable? Or is specialist hardware needed? Is it even possible to recover the target boards or must it be thrown into the rubbish bin? How does killing the board affect your project schedule?

Chapter 62. The Version 2 eCos FLASH API

There are two APIs described here. The first is the application API which programs should use. The second API is that between the FLASH IO library and the device drivers.

FLASH user API

All of the functions described below are declared in the header file `<cyg/io/flash.h>` which all users of the FLASH library should include.

Initializing the FLASH library

The FLASH library needs to be initialized before other FLASH operations can be performed. This only needs to be done once. The following function will only do the initialization once so it's safe to call multiple times:

```
__externC int cyg_flash_init(cyg_flash_printf *pf);
```

The parameter *`pf`* must always be set to NULL. It exists solely for backward compatibility and other settings are deprecated and obsolete. Past use of this parameter has now been replaced with use of the `cyg_flash_set_global_printf` function.

Retrieving information about FLASH devices

The following five functions return information about the FLASH.

```
__externC int cyg_flash_get_info(cyg_uint32 devno, cyg_flash_info_t * info);
__externC int cyg_flash_get_info_addr(cyg_flashaddr_t flash_base, cyg_flash_info_t * info);
__externC int cyg_flash_verify_addr(const flashaddr_t address);
__extern size_t cyg_flash_block_size(const cyg_flashaddr_t flash_base);
typedef struct cyg_flash_block_info
{
  size_t                      block_size;
  cyg_uint32                  blocks;
} cyg_flash_block_info_t;

typedef struct {
  cyg_flashaddr_t              start;          // First address
  cyg_flashaddr_t              end;            // Last address
  cyg_uint32                   num_block_infos;// Number of entries
  const cyg_flash_block_info_t *block_info;    // Info about one block size
} cyg_flash_info_t;
```

`cyg_flash_get_info()` is the main function to get information about installed flash devices. Parameter *`devno`* is used to iterate over the available flash devices, starting from 0. If the devno'th device exists, the structure pointed to by *`info`* is filled in and `CYG_FLASH_ERR_OK` is returned, otherwise `CYG_FLASH_ERR_INVALID`. `cyg_flash_get_info_addr()` is similar, but returns the information about the flash device at the given address. `cyg_flash_block_size()` returns the size of the block at the given address. `cyg_flash_verify_addr()` tests if the target addresses is within one of the FLASH devices, returning `CYG_FLASH_ERR_OK` if so.

Reading from FLASH

There are two methods for reading from FLASH. The first is to use the following function.

```
__externC int cyg_flash_read(cyg_flashaddr_t flash_base, void *ram_base, size_t len, cyg_flashaddr_t *err_address);
```

`flash_base` is where in the flash to read from. *`ram_base`* indicates where the data read from flash should be placed into RAM. *`len`* is the number of bytes to be read from the FLASH and *`err_address`* is used to return the location in FLASH that any error occurred while reading.

The second method is to simply `memcpy()` directly from the FLASH. This is not recommended since some types of device cannot be read in this way, eg NAND FLASH. Using the FLASH library function to read the FLASH will always work so making it easy to port code from one FLASH device to another.

Erasing areas of FLASH

Blocks of FLASH can be erased using the following function:

```
__externC int cyg_flash_erase(cyg_flashaddr_t flash_base,
                              size_t          len,
                              cyg_flashaddr_t *err_address);
```

`flash_base` is where in the flash to erase from. *`len`* is the minimum number of bytes to erase in the FLASH and *`err_address`* is used to return the location in FLASH that any error occurred while erasing. It should be noted that FLASH devices are block oriented when erasing. It is not possible to erase a few bytes within a block, the whole block will be erased. *`flash_base`* may be anywhere within the first block to be erased and *`flash_base+len`* may be anywhere in the last block to be erased.

Programming the FLASH

Programming of the flash is achieved using the following function.

```
__externC int cyg_flash_program(cyg_flashaddr_t flash_base,
                                void            *ram_base,
                                size_t          len,
                                cyg_flashaddr_t *err_address);
```

`flash_base` is where in the flash to program from. *`ram_base`* indicates where the data to be programmed into FLASH should be read from in RAM. *`len`* is the number of bytes to be program into the FLASH and *`err_address`* is used to return the location in FLASH that any error occurred while programming.

Locking and unlocking blocks

Some flash devices have the ability to lock and unlock blocks. A locked block cannot be erased or programmed without it first being unlocked. For devices which support this feature and when `CYGHWR_IO_FLASH_BLOCK_LOCKING` is enabled then the following two functions are available:

```
__externC int cyg_flash_lock(   const cyg_flashaddr_t flash_base,
                                size_t                len,
                                cyg_flashaddr_t       *err_address);

__externC int cyg_flash_unlock( const cyg_flashaddr_t flash_base,
                                size_t                len,
                                cyg_flashaddr_t       *err_address);
```

For some devices the granularity will be at the whole device level, where the code will lock or unlock all of the blocks at the same time.

Locking FLASH mutexes

When the eCos kernel package is included in the eCos configuration, the FLASH IO library will perform mutex locking on FLASH operations. This makes the API defined here thread safe. However applications may wish to directly access the contents of the FLASH. In order for this to be thread safe it is necessary for the application to use the following two functions to inform the FLASH IO library that the FLASH devices are being used and other API calls should be blocked.

```
__externC int cyg_flash_mutex_lock(const cyg_flashaddr_t from, size_t len);
__externC int cyg_flash_mutex_unlock(const cyg_flashaddr_t from, size_t len);
```

Configuring diagnostic output

Each FLASH device can have an associated function which is called to perform diagnostic output. The function to be used can be configured with the following functions:

```
__externC int cyg_flash_set_printf(const cyg_flashaddr_t  flash_base,
                                   cyg_flash_printf       *pf);
__externC void cyg_flash_set_global_printf(cyg_flash_printf *pf);
typedef int cyg_flash_printf(const char *fmt, …);
```

The parameter *`pf`* is a pointer to a function which is to be used for diagnostic output. Typically the function `diag_printf()` will be passed. Normally this function is not used by the higher layer of the library unless `CYGSEM_IO_FLASH_CHATTER` is enabled. Passing a *`NULL`* causes diagnostic output from lower level drivers to be discarded.

`cyg_flash_set_printf` is used to set a diagnostic output function which will be used specifically when diagnostic output is attempted from the FLASH device driver associated with the base address of *`flash_base`*. An error will be returned if no FLASH device is found for this address, or the FLASH subsystem has not yet been initialised with `cyg_flash_init`.

`cyg_flash_set_global_printf` sets a diagnostic output function for all available FLASH devices. Any previous setting of a diagnostic output function (including with `cyg_flash_set_printf`) will be discarded. This function may be called prior to `cyg_flash_init`.

Return values and errors

All the functions above return one of the following return values.

```
CYG_FLASH_ERR_OK              No error - operation complete
CYG_FLASH_ERR_INVALID         Invalid FLASH address
CYG_FLASH_ERR_ERASE           Error trying to erase
CYG_FLASH_ERR_LOCK            Error trying to lock/unlock
CYG_FLASH_ERR_PROGRAM         Error trying to program
CYG_FLASH_ERR_PROTOCOL        Generic error
CYG_FLASH_ERR_PROTECT         Device/region is write-protected
CYG_FLASH_ERR_NOT_INIT        FLASH info not yet initialized
CYG_FLASH_ERR_HWR             Hardware (configuration?) problem
CYG_FLASH_ERR_ERASE_SUSPEND   Device is in erase suspend mode
CYG_FLASH_ERR_PROGRAM_SUSPEND Device is in program suspend mode
CYG_FLASH_ERR_DRV_VERIFY      Driver failed to verify data
CYG_FLASH_ERR_DRV_TIMEOUT     Driver timed out waiting for device
CYG_FLASH_ERR_DRV_WRONG_PART  Driver does not support device
CYG_FLASH_ERR_LOW_VOLTAGE     Not enough juice to complete job
```

To turn an error code into a human readable string the following function can be used:

```
__externC const char *cyg_flash_errmsg(const int err);
```

FLASH device API

This section describes the API between the FLASH IO library the FLASH device drivers.

The FLASH device Structure

This structure keeps all the information about a single driver.

```
struct cyg_flash_dev {
  const struct cyg_flash_dev_funs *funs;            // Function pointers
  cyg_uint32                      flags;            // Device characteristics
  cyg_flashaddr_t                 start;            // First address
  cyg_flashaddr_t                 end;              // Last address
  cyg_uint32                      num_block_infos;  // Number of entries
  const cyg_flash_block_info_t    *block_info;      // Info about one block size
  const void                      *priv;            // Devices private data
  // The following are only written to by the FLASH IO layer.
  cyg_flash_printf                *pf;              // Pointer to diagnostic printf
  bool                            init;             // Device has been initialised
#ifdef CYGPKG_KERNEL
  cyg_mutex_t                     mutex;            // Mutex for thread safeness
#endif
#if (CYGHWR_IO_FLASH_DEVICE > 1)
  struct cyg_flash_dev            *next;            // Pointer to next device
#endif
};

struct cyg_flash_dev_funs {
  int     (*flash_init) (   struct cyg_flash_dev   *dev );

  size_t  (*flash_query) (  struct cyg_flash_dev  *dev,
                            void                  *data,
                            size_t                len );

  int     (*flash_erase_block) (struct cyg_flash_dev  *dev,
                                cyg_flashaddr_t       block_base );

  int     (*flash_program) (struct cyg_flash_dev  *dev,
                            cyg_flashaddr_t       base,
                            const void            *data,
                            size_t                len );

  int     (*flash_read) (   struct cyg_flash_dev  *dev,
                            const cyg_flashaddr_t base,
                            void                  *data,
                            size_t                len );
#ifdef CYGHWR_IO_FLASH_BLOCK_LOCKING
  int     (*flash_block_lock) (   struct cyg_flash_dev  *dev,
                                  const cyg_flashaddr_t block_base );

  int     (*flash_block_unlock) ( struct cyg_flash_dev  *dev,
                                  const cyg_flashaddr_t block_base);
#endif
};
```

The FLASH IO layer will only pass requests for operations on a single block.

Chapter 63. The legacy Version 1 eCos FLASH API

The library has a number of limitations:

1. Only one family of FLASH device may be supported at once.

2. Multiple devices of one family are supported, but they must be contiguous in memory.

3. The library is not thread or interrupt safe under some conditions.

4. The library currently does not use the eCos naming convention for its functions. This may change in the future but backward compatibility is likely to be kept.

There are two APIs described here. The first is the application API which programs should use. The second API is that between the FLASH io library and the device drivers.

FLASH user API

All of the functions described below are declared in the header file `<cyg/io/flash.h>` which all users of the FLASH library should include.

Initializing the FLASH library

The FLASH library needs to be initialized before other FLASH operations can be performed. This only needs to be done once. The following function will only do the initialization once so it's safe to call multiple times:

```
externC int flash_init( _printf *pf );
typedef int _printf(const char *fmt, …);
```

The parameter *`pf`* is a pointer to a function which is to be used for diagnostic output. Typically the function `diag_printf()` will be passed. Normally this function is not used by the higher layer of the library unless `CYGSEM_IO_FLASH_CHATTER` is enabled. Passing a *`NULL`* is not recommended, even when CYGSEM_IO_FLASH_CHATTER is disabled. The lower layers of the library may unconditionally call this function, especially when errors occur, probably resulting in a more serious error/crash!.

Retrieving information about the FLASH

The following four functions return information about the FLASH.

```
externC int flash_get_block_info(int *block_size, int *blocks);
externC int flash_get_limits(void *target, void **start, void **end);
externC int flash_verify_addr(void *target);
externC bool flash_code_overlaps(void *start, void *end);
```

The function `flash_get_block_info()` returns the size and number of blocks. When the device has a mixture of block sizes, the size of the "normal" block will be returned. Please read the source code to determine exactly what this means. `flash_get_limits()` returns the lower and upper memory address the FLASH occupies (NOTE: For the upper memory address this is the last valid FLASH location, and not the first memory address after the FLASH). The *`target`* parameter is currently unused. `flash_verify_addr()` tests if the target addresses is within the flash, returning `FLASH_ERR_OK` if so. Lastly, `flash_code_overlaps()` checks if the executing code is resident in the section of flash indicated by *`start`* and *`end`*. If this function returns true, erase and program operations within this range are very likely to cause the target to crash and burn horribly. Note the FLASH library does allow you to shoot yourself in the foot in this way.

Reading from FLASH

There are two methods for reading from FLASH. The first is to use the following function.

```
externC int flash_read(void *flash_base, void *ram_base, int len, void **err_address);
```

flash_base is where in the flash to read from. *ram_base* indicates where the data read from flash should be placed into RAM. *len* is the number of bytes to be read from the FLASH and *err_address* is used to return the location in FLASH that any error occurred while reading.

The second method is to simply `memcpy()` directly from the FLASH. This is not recommended since some types of device cannot be read in this way, eg NAND FLASH. Using the FLASH library function to read the FLASH will always work so making it easy to port code from one FLASH device to another.

Erasing areas of FLASH

Blocks of FLASH can be erased using the following function:

```
externC int flash_erase(void *flash_base, int len, void **err_address);
```

flash_base is where in the flash to erase from. *len* is the minimum number of bytes to erase in the FLASH and *err_address* is used to return the location in FLASH that any error occurred while erasing. It should be noted that FLASH devices are block oriented when erasing. It is not possible to erase a few bytes within a block, the whole block will be erased. *flash_base* may be anywhere within the first block to be erased and *flash_base+len* may be anywhere in the last block to be erased.

Programming the FLASH

Programming of the flash is achieved using the following function.

```
externC int flash_program(void *flash_base, void *ram_base, int len, void **err_address);
```

flash_base is where in the flash to program from. *ram_base* indicates where the data to be programmed into FLASH should be read from in RAM. *len* is the number of bytes to be program into the FLASH and *err_address* is used to return the location in FLASH that any error occurred while programming.

Locking and unlocking blocks

Some flash devices have the ability to lock and unlock blocks. A locked block cannot be erased or programmed without it first being unlocked. For devices which support this feature and when `CYGHWR_IO_FLASH_BLOCK_LOCKING` is enabled then the following two functions are available:

```
externC int flash_lock(void *flash_base, int len, void **err_address);
externC int flash_unlock(void *flash_base, int len, void **err_address);
```

Return values and errors

All the functions above, except `flash_code_overlaps()` return one of the following return values.

```
FLASH_ERR_OK              No error - operation complete
FLASH_ERR_INVALID         Invalid FLASH address
FLASH_ERR_ERASE           Error trying to erase
FLASH_ERR_LOCK            Error trying to lock/unlock
FLASH_ERR_PROGRAM         Error trying to program
FLASH_ERR_PROTOCOL        Generic error
FLASH_ERR_PROTECT         Device/region is write-protected
FLASH_ERR_NOT_INIT        FLASH info not yet initialized
FLASH_ERR_HWR             Hardware (configuration?) problem
```

```
FLASH_ERR_ERASE_SUSPEND   Device is in erase suspend mode
FLASH_ERR_PROGRAM_SUSPEND Device is in program suspend mode
FLASH_ERR_DRV_VERIFY      Driver failed to verify data
FLASH_ERR_DRV_TIMEOUT     Driver timed out waiting for device
FLASH_ERR_DRV_WRONG_PART  Driver does not support device
FLASH_ERR_LOW_VOLTAGE     Not enough juice to complete job
```

To turn an error code into a human readable string the following function can be used:

```
externC char *flash_errmsg(int err);
```

Notes on using the FLASH library

The FLASH library evolved from the needs and environment of RedBoot rather than being a general purpose eCos component. This history explains some of the problems with the library.

The library is not thread safe. Multiple simultaneous calls to its library functions will likely fail and may cause a crash. It is the callers responsibility to use the necessary mutex's if needed.

FLASH device API

This section describes the API between the FLASH IO library the FLASH device drivers.

The flash_info structure

The *flash_info*structure is used by both the FLASH IO library and the device driver.

```
struct flash_info {
  int   block_size;   // Assuming fixed size "blocks"
  int   blocks;       // Number of blocks
  int   buffer_size;  // Size of write buffer (only defined for some devices)
  unsigned long block_mask;
  void *start, *end;  // Address range
  int   init;         // FLASH API initialised
  _printf *pf;        // printf like function for diagnostics
};
```

block_mask is used internally in the FLASH IO library. It contains a mask which can be used to turn an arbitrary address in flash to the base address of the block which contains the address.

There exists one global instance of this structure with the name *flash_info*. All calls into the device driver makes use of this global structure to maintain state.

Initializing the device driver

The FLASH IO library will call the following function to initialize the device driver:

```
externC int  flash_hwr_init(void);
```

The device driver should probe the hardware to see if the FLASH devices exist. If it does it should fill in *start, end, blocks and block_size.* If the FLASH contains a write buffer the size of this should be placed in *buffer_size*. On successful probing the function should return `FLASH_ERR_OK`. When things go wrong it can be assumed that *pf* points to a printf like function for outputting error messages.

Querying the FLASH

FLASH devices can be queried to return there manufacture ID, size etc. This function allows this information to be returned.

```
int flash_query(unsigned char *data);
```

The caller must know the size of data to be returned and provide an appropriately sized buffer pointed to be parameter `data`. This function is generally used by `flash_hwr_init()`.

Erasing a block of FLASH

So that the FLASH IO layer can erase a block of FLASH the following function should be provided.

```
int flash_erase_block(volatile flash_t *block, unsigned int block_size);
```

Programming a region of FLASH

The following function must be provided so that data can be written into the FLASH.

```
int flash_program_buf(volatile flash_t *addr, flash_t *data, int len,
unsigned long block_mask, int buffer_size);
```

The device will only be asked to program data in one block of the flash. The FLASH IO layer will break longer user requests into a smaller writes.

Reading a region from FLASH

Some FLASH devices are not memory mapped so it is not possible to read there contents directly. The following function read a region of FLASH.

```
int flash_read_buf(volatile flash_t* addr, flash_t* data, int len);
```

As with writing to the flash, the FLASH IO layer will break longer user requests for data into a number of reads which are at maximum one block in size.

A device which cannot be read directy should set `CYGSEM_IO_FLASH_READ_INDIRECT` so that the IO layer makes use of the `flash_read_buf()`function.

Locking and unlocking FLASH blocks

Some flash devices allow blocks to be locked so that they cannot be written to. The device driver should provide the following functions to manipulate these locks.

```
int flash_lock_block(volatile flash_t *block);
int flash_unlock_block(volatile flash_t *block, int block_size, int blocks);
```

These functions are only used if `CYGHWR_IO_FLASH_BLOCK_LOCKING`

Mapping FLASH error codes to FLASH IO error codes

The functions `flash_erase_block(), flash_program_buf(), flash_read_buf(), flash_lock_block() and flash_unlock_block()` return an error code which is specific to the flash device. To map this into a FLASH IO error code, the driver should provide the following function:

```
int flash_hwr_map_error(int err);
```

Determining if code is in FLASH

Although a general function, the device driver is expected to provide the implementation of the function `flash_code_overlaps()`.

Implementation Notes

The FLASH IO layer will manipulate the caches as required. The device drivers do not need to enable/disable caches when performing operations of the FLASH.

Device drivers should keep all chatter to a minimum when `CYGSEM_IO_FLASH_CHATTER` is not defined. All output should use the print function in the *`pf`* in *`flash_info`* and not `diag_printf()`

Device driver functions which manipulate the state of the flash so that it cannot be read from for program execute need to ensure there code is placed into RAM. The linker will do this if the appropriate attribute is added to the function. e.g:

```
int flash_program_buf(volatile flash_t *addr, flash_t *data, int len,
                      unsigned long block_mask, int buffer_size)
__attribute__ ((section (".2ram.flash_program_buf")));
```

Chapter 64. FLASH I/O devices

It can be useful to be able to access FLASH devices using the generic I/O infrastructure found in `CYGPKG_IO`, and the generic FLASH layer provides an optional ability to do so. This allows the use of functions like `cyg_io_lookup()`, `cyg_io_read()`, `cyg_io_write()` etc.

Additionally it means that, courtesy of the “devfs” pseudo-filesystem in the file I/O layer (`CYGPKG_IO_FILEIO`), functions like `open()`, `read()`, `write()` etc. can even be used directly on the FLASH devices.

Overview and CDL Configuration

This package implements support for FLASH as an I/O device by exporting it as if it is a block device. To enable this support, the CDL option titled “Provide /dev block devices”, also known as `CYGPKG_IO_FLASH_BLOCK_DEVICE`, must be enabled. (There is also a legacy format alternative which is now deprecated).

There are two methods of addressing FLASH as a block device:

1. Using the FLASH Information System (FIS) - this is a method of defining and naming FLASH partitions, usually in RedBoot. This option is only valid if RedBoot is resident and was used to boot the application. To reference FLASH partitions in this way, you would use a device name of the form `/dev/flash/fis/`*`partition-name`* , for example `/dev/flash/fis/jffs2` to reference a FIS partition named JFFS2.

   The CDL option `CYGFUN_IO_FLASH_BLOCK_FROM_FIS` must be enabled for this support.

2. Referencing by device number, offset and length - this method extracts addressing information from the name itself. The form of the device would be `/dev/flash/`*`device-number`*`/`*`offset`*`[,`*`length`* `]`

   `device-number` This is a fixed number allocated to identify each FLASH region in the system. The first region is numbered 0, the second 1, and so on. If you have only one FLASH device, it will be numbered 0.

   `offset` This is the index into the FLASH region in bytes to use. It may be specified as decimal, or if prefixed with `0x`, then hexadecimal.

   `length` This field is optional and defaults to the remainder of the FLASH region. Again it may be specified in decimal or hexadecimal.

   Some examples:

   /dev/flash/0/0 This defines a block device that uses the entirety of FLASH region 0.

   /dev/flash/1/0x20000,65536 This defines a block device which points inside FLASH region 1, starting at offset 0x20000 (128Kb) and extending for 64Kb.

   /dev/flash/0/65536 This defines a block device which points inside FLASH region 0, starting at offset 64Kb and continuing up to the end of the device.

   Obviously great care is required when constructing the device names as using the wrong specification may subsequently overwrite important areas of FLASH, such as RedBoot. Using the alternative via FIS names is preferable as these are less error-prone to configure, and also allows for the FLASH region to be relocated without requiring program recompilation.

Using FLASH I/O devices

The FLASH I/O block devices can be accessed, read and written using the standard interface supplied by the generic I/O (CYGPKG_IO) package. These include the functions: `cyg_io_lookup()` to access the device and get a handle, `cyg_io_read()` and `cyg_io_write()` for sequential read and write operations, `cyg_io_bread()` and `cyg_io_bwrite()` for random access read and write operations, and `cyg_io_get_config()` and `cyg_io_setconfig()` for run-time configuration inspection and control.

However there are two aspects that differ from some other I/O devices accessed this way:

1. The first is that the lookup operation uses up resources which must be subsequently freed when the last user of the I/O handle is finished. The number of FLASH I/O devices that may be simultaneously opened is configured with the CYGNUM_IO_FLASH_BLOCK_DEVICES CDL option. After the last user is finished, the device may be closed using `cyg_io_setconfig()` with the CYG_IO_SET_CONFIG_CLOSE key. Reference counting to ensure that it is only the last user that causes a close, is left to higher layers.

2. The second is that write operations assume that the flash is already erased. Attempting to write to Flash that has already been written to may result in errors. Instead FLASH must be erased before it may be written.

FLASH block devices can also be read and written using the standard POSIX primitives, `open()`, `close()`, `read()`, `write()`, `lseek()`, and so on if the POSIX file I/O package (CYGPKG_FILEIO) is included in the configuration. As with the eCos generic I/O interface you must call `close()` to ensure resources are freed when the device is no longer used.

Other configuration keys are provided to perform FLASH erase operations, and to retrieve device sizes, and FLASH block sizes at a particular address. These operations are accessed with `cyg_io_get_config()` (or if using the POSIX file I/O API, `cyg_fs_getinfo()`) with the following keys:

CYG_IO_GET_CONFIG_FLASH_ERASE

This erases a region of FLASH. `cyg_io_get_config()` must be passed a structure defined as per the following, which is also supplied in `<cyg/io/flash.h>`:

```
typedef struct {
  CYG_ADDRESS offset;
  size_t len;
  int flasherr;
  cyg_flashaddr_t err_address;
} cyg_io_flash_getconfig_erase_t;
```

In this structure, *`offset`* specifies the offset within the block device to erase, *`len`* specifies the amount to address, *`flasherr`* is set on return to specify an error with the FLASH erase operation itself, and *`err_address`* is used if there was an error to specify at which address the error happened.

CYG_IO_GET_CONFIG_FLASH_LOCK

This protects a region of FLASH using the locking facilities available on the card, if provided by the underlying driver. `cyg_io_get_config()` must be passed a structure defined as per the following:

```
typedef struct {
  CYG_ADDRESS offset;
  size_t len;
  int flasherr;
  cyg_flashaddr_t err_address;
} cyg_io_flash_getconfig_lock_t;
```

In this structure, *`offset`* specifies the offset within the block device to lock, *`len`* specifies the amount to address, *`flasherr`* is set on return to specify an error with the FLASH lock operation itself, and *`err_address`* is used if there was an error to specify at which address the error happened. If locking support is not available -EINVAL will be returned from `cyg_io_get_config()`.

CYG_IO_GET_CONFIG_FLASH_UNLOCK

This disables protection for a region of FLASH using the unlocking facilities available on the card, if provided by the underlying driver. `cyg_io_get_config()` must be passed a structure defined as per the following:

```
typedef struct {
  CYG_ADDRESS offset;
  size_t len;
  int flasherr;
  cyg_flashaddr_t err_address;
} cyg_io_flash_getconfig_unlock_t;
```

In this structure, *offset* specifies the offset within the block device to unlock, *len* specifies the amount to address, *flasherr* is set on return to specify an error with the FLASH unlock operation itself, and *err_address* is used if there was an error to specify at which address the error happened. If unlocking support is not available -EINVAL will be returned from `cyg_io_get_config()`.

CYG_IO_GET_CONFIG_FLASH_DEVSIZE

This returns the size of the FLASH block device. The `cyg_io_get_config()` function must be passed a structure defined as per the following, which is also supplied in `<cyg/io/flash.h>`:

```
typedef struct {
  size_t dev_size;
} cyg_io_flash_getconfig_devsize_t;
```

In this structure, *dev_size* is used to return the size of the FLASH device.

CYG_IO_GET_CONFIG_FLASH_DEVADDR

This returns the address in the virtual memory map that the generic flash layer has been informed that this FLASH device is mapped to. Note that some flash devices such as dataflash are not truly memory mapped, and so this function only returns useful information when used with a true memory mapped FLASH device. The `cyg_io_get_config()` function must be passed a structure defined as per the following, which is also supplied in `<cyg/io/flash.h>`:

```
typedef struct {
  cyg_flashaddr_t dev_addr;
} cyg_io_flash_getconfig_devaddr_t;
```

In this structure, *dev_addr* is used to return the address corresponding to the base of the FLASH device in the virtual memory map.

CYG_IO_GET_CONFIG_FLASH_BLOCKSIZE

This returns the size of a FLASH block at a supplied offset in the FLASH block device. The `cyg_io_get_config()` function must be passed a structure defined as per the following, which is also supplied in `<cyg/io/flash.h>`:

```
typedef struct {
  CYG_ADDRESS offset;
  size_t block_size;
} cyg_io_flash_getconfig_blocksize_t;
```

In this structure, *offset* specifies the address within the block device of which the FLASH block size is required - a single FLASH device may contain blocks of differing sizes. The *block_size* field is used to return the block size at the specified offset.

Chapter 65. Common SPI Flash Memory Device Driver

Name

eCos Common Support for SPI Flash Memory Devices — Overview

Description

The CYGPKG_DEVS_FLASH_SPI_COMMON package provides an abstraction layer between the standard eCos I/O Flash API package (CYGPKG_IO_FLASH) and hardware specific *x*SPI controller drivers. This allows for serial memory device support to be shared across architectures and platforms, avoiding the need for the H/W specific device drivers to duplicate manufacturer or device specific information in each H/W *x*SPI driver implementation.

Note

Many eCos targets will still use the older driver model where the architecture specific device driver will implement direct support for a specific subset of SPI memory devices. Only newer ports, and some ports that have been explicitly updated, will reference this common approach. The goal is to bring as much support as is relevant for SPI memory devices into this single package, to aid maintenance and porting, with a simpler H/W device driver implementation for the platform specific component.

This common package presents as a Flash V2 device driver to the Flash I/O layer.

The package allows for JEDEC Serial Flash Discoverable Parameters (SFDP) device supplied parameter tables to be used to configure the required device access. The package currently supports JESD216D.01 and earlier devices. It will work with devices that implement newer versions of the standard, but will be limited to the backwards compatible features .

Caution

At the time of writing not all of the SFDP table declared configurations have been tested.

The following table is not an exhaustive list of tested platforms/devices, but an example set:

Platform	Device	*x*SPI	Notes
mimxrt1050_evk	IS25WP064AJBLE	Quad	64-Mbit
samv71_xult	S25FL116K	Quad	16-Mbit
stm32h735_disco	MX25LM51245GXDI00	Octal	512-Mbit
-	W25Q32JV	Quad	32-Mbit
-	W25Q512JV-DTR	Quad	512-Mbit

Configuration Options

The common SPI memory driver package will be loaded automatically when configuring eCos for a target with suitable hardware. However the driver will be inactive unless the generic flash package CYGPKG_IO_FLASH is loaded. It may be necessary to add the generic CYGPKG_IO_FLASH package to the configuration explicitly before the driver functionality becomes available. There should never be any requirement to load or unload the CYGPKG_DEVS_FLASH_SPI_COMMON driver package.

The flash driver provides a small number of configuration options which application developers may use to control features provided by the package.

CYGFUN_DEVS_FLASH_SPI_COMMON_MEMMAPPED

> If this option is enabled then the flash device is configured for memory mapped mode **when** the underlying H/W driver and platform HAL support such use.

Memory mapped access allows the CPU to directly read data or execute code from the flash area. The default is for the feature to be enabled, which is desired for most configurations. However, in some situations, indirect (e.g. DMA) access may be preferred for performance reasons, in which case this feature can be disabled.

When the option is enabled some further configuration options are presented:

`CYGIMP_DEVS_FLASH_SPI_COMMON_MEMMAPPED_XIPISR`

This option should be enabled if ISRs or DSRs are to execute from the memory mapped xSPI space.

Note

This will adversely affect the interrupt latency of the system, since certain xSPI operations will need to disable interrupts when switching out of memory mapped mode (e.g. erasing). So this feature should only be enabled if actually required.

`CYGIMP_DEVS_FLASH_SPI_COMMON_MEMMAPPED_DRIVER`

If the eCos application providing the xSPI flash driver is executing from the flash device (using memory mapped mode) then some critical functionality **must** execute from a different memory space (e.g. SRAM).

`CYGFUN_DEVS_FLASH_SPI_COMMON_SOFTRESET`

This option controls whether support for device soft reset is enabled. The developer is not normally required to modify this option

Fixed settings

If the target system needs to support devices without SFDP tables, or where the tables provide inconsistent or incomplete information, then a hook mechanism is provided based on the standard `RDID` command (`0x9F`). The `flash_csm_fixedset.h` provides the `CYG_CSM_FIXEDSET(localname, mask, id, init_function)` macro to allow an instantation of a `cyg_csm_family_t` structure to be added to the table scanned by this common flash driver.

This provides a mechanism for partially or completely updating the internal context used to describe a device to the common flash code and the underlying H/W driver layer.

Currently SFDP fix-up support is provided for the Winbond `W25Q512JV` parts when the `CYGINT_DEVS_FLASH_SPI_COMMON_HARDWIRED_W25Q512` interface is implemented. Normally the required `implements` is performed by the target platform CDL as required, and should not need to be managed by application developers.

Name

Common SPI Memory Device Hardware Driver — Interface to a hardware device driver

Description

Note

Application developers should not normally need to concern themselves with the internal API between this common layer and the H/W specific device drivers. The following information is primarily for H/W device driver developers.

In most cases the platform (PLF) declares the individual flash driver instances. The top-level descriptor as used with the flash API (`CYGPKG_IO_FLASH`) should reference the flash API functions provided by this package (`cyg_devs_flash_common_funs`) as well as provide a per-instance `cyg_flash_csm_context_t` structure initialised with a reference to the instance-specific hardware driver descriptor in the `p_hwdriver` field.

The driver instance specific `cyg_spi_common_hwdriver_t` descriptor is used to describe the hardware driver specific features to this common layer.

```
Flash API                      Common                                   H/W Driver
(struct cyg_flash_dev).priv -> (cyg_flash_csm_context_t).p_hwdriver -> (cyg_spi_common_hwdriver_t)
```

All architecture/platform/HAL eCos *x*SPI device drivers using the `CYGPKG_DEVS_FLASH_SPI_COMMON` package must implement a standard interface defined by the header `<cyg/io/flash_csm_dev.h>`. The interface descriptor structure includes a private pointer for the H/W driver context, a “features” set and a set of function pointers for various operations: initialization, memory operation, memory-mapped access and general configuration.

```
struct cyg_spi_common_hwdriver {
    // H/W driver private (opaque) context:
    const void *p_io; // H/W driver specific I/O information

    // H/W driver feature set descriptor:
    const cyg_flash_csm_features_t * const p_features;

    // Common H/W driver API
    cyg_spi_common_hwdriver_init *init; // initialisation function
    cyg_spi_common_hwdriver_op *op; // single command operation function
    cyg_spi_common_hwdriver_mm_start *mm_start; // memory-mapped start/enable
    cyg_spi_common_hwdriver_mm_stop *mm_stop; // memory-mapped stop/disable
    cyg_spi_common_hwdriver_config *config; // get/set config+control
};
```

Hardware Driver Features

The `p_features` structure provides fixed information used to describe to this common layer the features and settings of the H/W driver instance:

```
typedef struct cyg_flash_csm_features {
    cyg_uint32 avail; // bitmask of H/W driver available features
    cyg_uint32 mmaddr; // if MM capable, base address for MM region
    // Since early JESD216 standards do not provide a mechanism for the device
    // to report its maximum frequency we allow the platform/variant HAL to be
    // configured with maximum rates.
    cyg_uint32 max_sdr; // if non-zero platform/variant HAL provided maximum SDR baudrate
    cyg_uint32 max_ddr; // if non-zero platform/variant HAL provided maximum DDR baudrate
    cyg_uint32 nmodes; // number of modes present in modes vector
    const cyg_uint32 * const p_modes; // pointer to vector of FLASH_CSM_OP_MODE_MASK
                                      // covered bitmasks for available modes
} cyg_flash_csm_features_t;
```

The `p_features` structure allows the H/W driver to report the SPI modes capable by the device driver. This can, for example, be used in conjunction with information gathered from the device using SFDP to select the common subset of supported access methods: e.g. Quad (QSPI) vs Octal (OSPI).

The current set of **avail**able feature flags indicating H/W driver support is:

Flag	Description
`FLASH_CSM_FEATURE_ADDR3`	3-byte addressing supported
`FLASH_CSM_FEATURE_ADDR4`	4-byte addressing supported
`FLASH_CSM_FEATURE_ADDR5`	5-byte addressing supported
`FLASH_CSM_FEATURE_MODEBITS`	Driver supports writing mode bits (sometimes referred to as OPT or Alternate) bits
`FLASH_CSM_FEATURE_CMD8`	8-bit commands supported
`FLASH_CSM_FEATURE_CMD16`	16-bit commands supported
`FLASH_CSM_FEATURE_MM`	Memory mapped access supported
`FLASH_CSM_FEATURE_MM_XIP`	eXecute-In-Place supported
`FLASH_CSM_FEATURE_MM_SM`	Driver can memory-map serial-memory as well as data-memory
`FLASH_CSM_FEATURE_MM_SM_RA`	Random Access supported for memory mapped serial-memory
`FLASH_CSM_FEATURE_CR`	Continuous Read supported
`FLASH_CSM_FEATURE_DS`	Data Strobe signalling available

The `p_modes` pointer references the `nmodes` deep vector of access modes supported, encoded using the same OP bitmask encoding as used for the individual memory operations. For example an Octal (OSPI) capable driver might define:

```
// List of possible modes for this driver:
static const cyg_uint32 cyg_hwdriver_modes[] = {
    // 8-line Octal (OSPI)
    FLASH_CSM_OP_MODE_1S1S8S, //  0
    FLASH_CSM_OP_MODE_1S8S8S, //  1
    FLASH_CSM_OP_MODE_8S8S8S, //  2
    FLASH_CSM_OP_MODE_8D8D8D, //  3
    // 4-line QSPI
    FLASH_CSM_OP_MODE_1S1S4S, //  4
    FLASH_CSM_OP_MODE_1S4S4S, //  5
    FLASH_CSM_OP_MODE_4S4S4S, //  6
    FLASH_CSM_OP_MODE_4S4D4D, //  7
    // 2-line
    FLASH_CSM_OP_MODE_1S1S2S, //  8
    FLASH_CSM_OP_MODE_1S2S2S, //  9
    FLASH_CSM_OP_MODE_2S2S2S, // 10
    // 1-line
    FLASH_CSM_OP_MODE_1S1S1S, // 11
};
```

Referencing the vector in its feature descriptor:

```
static const cyg_flash_csm_features_t hwdriver1_features = {
    ..elided..
    .nmodes = NUMOF_(cyg_hwdriver_modes),
    .p_modes = &cyg_hwdriver_modes[0]
};
```

Hardware Driver-Specific Structure

The `p_io` pointer allows the H/W device driver to hold per-instance private data as needed for the operation of the driver.

Normally the driver context would be split into read-only, constant, data that could be held in the code with only the truly dynamic context occupying RAM space. See the Hardware Example below for an outline.

Functions

The H/W driver provides its common driver API via the `cyg_spi_common_hwdriver_t` descriptor. For the function pointers the `NULL` value can be used to indicate that the relevant support is not required. Only the `op` function **must** be provided, though it is unlikely that the driver would not require a `init` function to be called at startup.

Initialization

```
typedef int cyg_spi_common_hwdriver_init(const void *p_info, cyg_bool do_reset, cyg_uint32 baudrate);
```

This function allows the H/W driver to complete the run-time initialisation of any dynamic context needed, along with setting up the controller in preperation for the first operation.

This can consist of attaching any ISR/DSR or DMA handlers needed, setting up I/O pin configurations (if the platform/architecture uses pin multiplexing), etc.

The `reset` parameter indicates whether the upper layer requires the hardware to be “reset” back to a known state.

The `baudrate` is the clock frequency that will be used for the initial operations. Normally (for SFDP devices) this will be 50MHz.

The function call should return standard flash API status code. e.g. `CYG_FLASH_ERR_OK` to indicate success.

Memory Operation

```
typedef cyg_bool cyg_spi_common_hwdriver_op(const void *p_info, const cyg_flash_csm_op_t *p_op);
```

This is the core operation of the H/W driver interface. The referenced `cyg_flash_csm_op_t` pointer `p_op` describes the basic operation to be performed on the serial memory device.

The `p_info` pointer is a reference to the H/W driver specific context supplied when declaring the flash descriptor.

The function should return a simple boolean `true` success indication, or `false` if an error occurred.

The `<cyg/io/flash_csm_dev.h>` header defines the referenced `p_op` structure:

```
typedef struct cyg_flash_csm_op {
    cyg_uint32 mode; // Encoded bus information and control
    cyg_uint32 cmdflags; // Instruction CMD and extra control flags
    cyg_uint32 address; // Device relative address
    cyg_uint32 opt; // Upto 32-bits of OPT (mode-bits; alternate) data
    cyg_uint32 timeout; // Millisecond timeout for operation
    cyg_uint32 nbytes; // Non-zero is number of valid bytes from p_buff
    cyg_uint8 *p_buff; // Pointer to data buffer for transfer
} cyg_flash_csm_op_t;
```

The `mode` and `cmdflags` fields define whether the other fields are used/required. For example, a simple device operation to enable the write-latch will not normally require any data to be written (or read) and so the `nbytes` and `p_buff` fields would not be referenced for that operation.

The `<cyg/io/flash_csm_dev.h>` header is the definitive source for the bitmask use for the `mode` and `cmdflags` fields and should be examined by the developer writing a H/W driver. The header contains helper manifests and macros to aid the decoding of the fields.

mode

The `mode` bitmask encodes the operation. It contains some single-bit boolean flags as well as some multi-bit values with specific encodings.

The value `0x00000000` (`CSM_OP_INVALID`) is never a valid descriptor since we should always have at least one of the instruction, address, opt or data phases defined.

An operation compromises one or more phases in the order: Instruction, Address, Mode and Data. The `mode` bitmask encodes which phases are enabled, and hence their associated bitmask flags and values are valid, as well as some general operation control flags.

1. **Instruction phase**

   If an **instruction** phase is required then the `FLASH_CSM_OP_IP` mask will have the value `FLASH_CSM_OP_IP_ACTIVE`, otherwise the flag will have the value `FLASH_CSM_OP_IP_NONE`.

   If `FLASH_CSM_OP_IP_ACTIVE` then the `FLASH_CSM_OP_IW_MASK` bits encode the number of lines to be used for the instruction phase:

Value	Description
`FLASH_CSM_OP_IW_LINE1`	1-line (SPI)
`FLASH_CSM_OP_IW_LINE2`	2-lines (Dual)
`FLASH_CSM_OP_IW_LINE4`	4-lines (Quad)
`FLASH_CSM_OP_IW_LINE8`	8-lines (Octal)

   If `FLASH_CSM_OP_IP_ACTIVE` then the `p_op` field `cmdflags` holds the command instruction.

   The command length is encoded by the `FLASH_CSM_OP_CL` mask. The value should be `FLASH_CSM_OP_CL_8BIT` for 8-bit commands, and `FLASH_CSM_OP_CL_16BIT` for 16-bit commands.

   If `FLASH_CSM_OP_IP_ACTIVE` then the `FLASH_CSM_OP_IP_IR` mask encodes whether the instruction phase is Single-Data-Rate (`FLASH_CSM_OP_IR_SDR`) or Dual-Data-Rate (`FLASH_CSM_OP_IR_DDR`).

2. **Address phase**

   If an **address** phase is required then the `FLASH_CSM_OP_AP` mask will have the value `FLASH_CSM_OP_AP_ACTIVE`, otherwise the flag will have the value `FLASH_CSM_OP_AP_NONE`.

   If `FLASH_CSM_OP_AP_ACTIVE` then the `FLASH_CSM_OP_AB_MASK` bits encode the number of bytes used for an address:

Value	Description
`FLASH_CSM_OP_AB_3BYTE`	3-byte (24-bit) address
`FLASH_CSM_OP_AB_4BYTE`	4-byte (32-bit) address
`FLASH_CSM_OP_AB_5BYTE`	5-byte (40-bit) address (not currently supported)

   If `FLASH_CSM_OP_AP_ACTIVE` then the `FLASH_CSM_OP_AW_MASK` bits encode the number of lines to be used for the address phase:

Value	Description
`FLASH_CSM_OP_AW_LINE1`	1-line (SPI)
`FLASH_CSM_OP_AW_LINE2`	2-lines (Dual)
`FLASH_CSM_OP_AW_LINE4`	4-lines (Quad)
`FLASH_CSM_OP_AW_LINE8`	8-lines (Octal)

   If `FLASH_CSM_OP_AP_ACTIVE` then the `p_op` field `address` should define the **device** relative address for the operation.

If FLASH_CSM_OP_AP_ACTIVE then the FLASH_CSM_OP_AR mask encodes whether the address phase is Single-Data-Rate (FLASH_CSM_OP_AR_SDR) or Dual-Data-Rate (FLASH_CSM_OP_AR_DDR).

3. **Mode phase**

   Different hardware implementations support OPT/Alternate bytes of differing sizes and limitations. The JESD216D.01 standard describes these as “mode bits” and are sent after the address phase.

   If a **mode** phase is required then the FLASH_CSM_OP_MP mask will have the value FLASH_CSM_OP_MP_ACTIVE, otherwise the flag will have the value FLASH_CSM_OP_MP_NONE.

   If FLASH_CSM_OP_MP_ACTIVE then the FLASH_CSM_OP_MB_MASK bits encode the number of **bits** (range 1..32). The mode bits are signalled on the same number of SPI lines as the address phase.

   If FLASH_CSM_OP_MP_ACTIVE then the p_op field opt holds the mode bits value.

   If FLASH_CSM_OP_MP_ACTIVE then the FLASH_CSM_OP_MR mask encodes whether the mode bits phase is Single-Data-Rate (FLASH_CSM_OP_MR_SDR) or Dual-Data-Rate (FLASH_CSM_OP_MR_DDR).

4. **Data phase**

   If an **data** phase is required then the FLASH_CSM_OP_DP mask will have the value FLASH_CSM_OP_DP_ACTIVE, otherwise the flag will have the value FLASH_CSM_OP_DP_NONE.

   If FLASH_CSM_OP_DP_ACTIVE then the FLASH_CSM_OP_DW_MASK bits encode the number of lines to be used for the data phase:

Value	**Description**
FLASH_CSM_OP_DW_LINE1	1-line (SPI)
FLASH_CSM_OP_DW_LINE2	2-lines (Dual)
FLASH_CSM_OP_DW_LINE4	4-lines (Quad)
FLASH_CSM_OP_DW_LINE8	8-lines (Octal)

   If FLASH_CSM_OP_DP_ACTIVE then the p_op field nbytes should define the number of valid memory bytes referenced by the p_buff pointer.

   If FLASH_CSM_OP_DP_ACTIVE then the FLASH_CSM_OP_DR mask encodes whether the address phase is Single-Data-Rate (FLASH_CSM_OP_DR_SDR) or Dual-Data-Rate (FLASH_CSM_OP_DR_DDR).

The mode field also encodes other information that may be required by the H/W driver.

The FLASH_CSM_OP_SD mask has the value FLASH_CSM_OP_SD_DATA when the operation is accessing the data-memory of the flash device, and the value FLASH_CSM_OP_SD_DEVICE if accessing device-internal “memory” (e.g. SFDP tables, unique IDs, etc.).

The FLASH_CSM_OP_TD mask encodes the Transfer Direction, whether the operation is a read (FLASH_CSM_OP_TD_READ) or a write (FLASH_CSM_OP_TD_WRITE).

The FLASH_CSM_OP_DA indicates whether the transfer should be undertaken by the operation call directly (FLASH_CSM_OP_DA_XFER) or should be deferred for subsequent memory-mapped access (FLASH_CSM_OP_DA_DEFER).

When reading the FLASH_CSM_OP_CR mask indicates that the device is configured for Continuous Read. If FLASH_CSM_OP_CR_NONE then continuous read is not configured. If FLASH_CSM_OP_CR_ACTIVE then it indicates that the controller can setup access for continuous read mode.

The FLASH_CSM_OP_DS encodes whether the operation requires the Data Strobe signal (FLASH_CSM_OP_DS_ACTIVE) or the signal is not required (FLASH_CSM_OP_DS_NONE).

cmdflags

When required by the mode bitmask the cmdflags field encodes the command code (8- or 16-bit), the number of Delay Cycles and whether the timeout is valid.

timeout

The timeout value is only valid if the cmdflag flag FLASH_CSM_CMD_TO_VALID is set, otherwise the field is ignored.

The timeout field is a millisecond operation timeout, or one of the special values: CYG_FLASH_CSM_TO_NOWAIT or CYG_FLASH_CSM_TO_INFINITY. The ...NOWAIT value is for an immediate, polled, return without waiting, operation. The ...INFINITY value is for when the operation should block indefinately until completion (success or error indicated).

Memory Mapped

```
typedef cyg_bool cyg_spi_common_hwdriver_mm_start(const void *p_info);
```

```
typedef cyg_bool cyg_spi_common_hwdriver_mm_stop(const void *p_info);
```

The optional mm_start and mm_stop functions are used to notify the H/W driver when memory-mapped state is being changed.

Since most devices cannot continue to provide memory-mapped access whilst being erased or programmed the common driver layer allows the H/W driver to perform any controller operations needed to ensure the hardware is in the correct mode. For example, this may include changing the cached/uncached state for the memory covered by flash device, or require specific controller operations to abort any active memory-mapped pre-fetching that may be occurring.

Configuration

```
typedef cyg_bool cyg_spi_common_hwdriver_config(const void *p_info,
                                                cyg_uint32 key, void *p_buff, cyg_uint32 *p_len);
```

The optional H/W driver supplied config function is used with specific configuration key values to interact with the H/W driver:

CYG_CSM_CFG_SET_BAUDRATE
CYG_CSM_CFG_GET_BAUDRATE

> Used by the common layer to control the clock frequency (normally named SCK) of the H/W driver instance. The max_sdr and max_ddr fields of the H/W driver supplied (cyg_flash_csm_features_t) descriptor allow the H/W driver to limit the upper frequency to that supported by the specific controller, with the common layer device support being used for the actual flash memory device maximum rates possible.
>
> The frequency setting is a simple 32-bit unsigned integer (e.g. cyg_uint32).

CYG_CSM_CFG_SET_MEMTYPE
CYG_CSM_CFG_GET_MEMTYPE

> If required, for OCTOSPI devices, these options provide common layer control of the memory type as used by the H/W driver.
>
> The memory type setting is currently a simple 32-bit unsigned integer (e.g. cyg_uint32):
>
> CYG_CSM_MEMTYPE_DTR_D0D1
>
> > This option indicates Micron style byte-ordering.

CYG_CSM_MEMTYPE_DTR_D1D0

This option indicates Macronix style byte-ordering.

CYG_CSM_MEMTYPE_STANDARD

Indicates normal SPI access.

CYG_CSM_MEMTYPE_UNDEFINED

This value indicates that no specific memory type has been set.

CYG_CSM_CFG_SET_DATASTROBE
CYG_CSM_CFG_GET_DATASTROBE

For configurarions that use a data strobe signal (DQS) these config options provide the mechanism for informing the H/W driver of the device data strobe timing.

The data strobe settings are a simple 32-bit unsigned integer (e.g. cyg_uint32):

CYG_CSM_DATASTROBE_START

Start of first data bit aligned with the first rising edge of DQS.

CYG_CSM_DATASTROBE_MIDDLE

First rising edge of DQS in the middle of the first data bit.

CYG_CSM_DATASTROBE_HALF

First rising edge of DQS is half a clock cycle before the start of the first data bit.

CYG_CSM_DATASTROBE_UNDEFINED

This setting is used to indicate that the data strobe timing is not defined or unknown.

Example

The following section provides an example skeleton of how a H/W driver instance can be declared.

Since device drivers normally have a requirement for some fixed (constant) information describing the hardware configuration as well as possibly some dynamic state to hold run-time information (e.g. ISR or DSR state) the example below shows a simple framework. The example hw_driver_ctx_t structure used for the dynamic context and the hw_driver_io_t structure holding the constant/fixed information are specific to the H/W driver implementation and the underlying H/W controller requirements.

For our example instance in the H/W driver source we can provide a RAM based private context for the dynamic state:

```
static hw_driver_ctx_t hw_dynamic1 = {};
```

This can then be referenced from a constant (normally placed in read-only memory by the linker) structure with the fixed information for the driver along with a pointer to the RAM based dynamic run-time context structure:

```
static const hw_driver_io_t hw_context1 {
    .p_ctx = &hw_driver_dynamic1, // dynamic H/W driver state
    // The fixed information needed by the H/W driver
    .intr_vec = <HAL_INTERRUPT_NUMBER>
    .pin_sclk = <HAL/PLF_SCLK_PIN_DESCRIPTOR>
    ..elided..
};
```

The driver can then provide a per-instance common H/W driver API structure referencing the H/W driver context and the features and functions provided by the driver:

```
const cyg_spi_common_hwdriver_t cyg_dev_flash_csm_example1= {
    .p_io = &hw_context1,
    .p_features = &hw_features,
    .init = hw_init,
    .op = hw_op,
    .mm_start = NULL, // op interface sufficient for this driver
    .mm_stop = NULL, // op interface sufficient for this driver
    .config = hw_config
};
```

With the H/W driver providing the `cyg_spi_common_hwdriver_t` structure the platform specific sources would then reference the H/W driver instance when declaring the flash object in the platform/HAL specific source file:

```
static struct cyg_flash_block_info cyg_flash_common_block_info;

static cyg_flash_csm_context_t cyg_flash_common_ctx = {
    .p_hwdriver = &cyg_dev_flash_csm_example1, // reference H/W instance
};

CYG_FLASH_DRIVER(cyg_common_device,
                 &cyg_devs_flash_common_funs,
                 0,
                 <BASE_ADDR>, // start (normally same as p_features.mmaddr field
                 <BASE_ADDR>, // end (depends on detected device, so filled in at run-time)
                 1, // number of flash block info structures
                 &cyg_flash_common_block_info,
                 &cyg_flash_common_ctx);
```

After successful flash device initialisation the `struct cyg_flash_dev` field `end` will hold the end address for the flash area. The start and end addresses are used by the flash API to select the relevant device descriptor when accessing a flashg area.

When called, the common layer code can use the `p_hwdriver` field to access the specific H/W instance used to access the actual flash device for the area, with the H/W driver subsequently de-referencing its own structures to access the fixed and dynamic portions of its context.

Chapter 66. AMD AM29xxxxx Flash Device Driver

Name

eCos Support for AMD AM29xxxxx Flash Devices and Compatibles — Overview

Description

The CYGPKG_DEVS_FLASH_AMD_AM29XXXXX_V2 AMD AM29xxxxx V2 flash driver package implements support for the AM29xxxxx family of flash devices and compatibles. Normally the driver is not accessed directly. Instead application code will use the API provided by the generic flash driver package CYGPKG_IO_FLASH, for example by calling functions like cyg_flash_program.

The driver imposes one restriction on application code which developers should be aware of: when programming the flash the destination addresses must be aligned to a bus boundary. For example if the target hardware has a single flash device attached to a 16-bit bus then program operations must involve a multiple of 16-bit values aligned to a 16-bit boundary. Note that it is the bus width that matters, not the device width. If the target hardware has two 16-bit devices attached to a 32-bit bus then program operations must still be aligned to a 32-bit boundary, even though in theory a 16-bit boundary would suffice. In practice this is rarely an issue, and requiring the larger boundary greatly simplifies the code and improves performance.

Note

Many eCos targets with AM29xxxxx or compatible flash devices will still use the older driver package CYGPKG_DEVS_FLASH_AMD_AM29XXXXX. Only newer ports and some older ports that have been converted will use the V2 driver. This documentation only applies to the V2 driver.

Configuration Options

The AM29xxxxx flash driver package will be loaded automatically when configuring eCos for a target with suitable hardware. However the driver will be inactive unless the generic flash package CYGPKG_IO_FLASH is loaded. It may be necessary to add this generic package to the configuration explicitly before the driver functionality becomes available. There should never be any need to load or unload the AM29xxxx driver package.

The driver contains a small number of configuration options which application developers may wish to tweak. CYGNUM_DEVS_FLASH_AMD_AM29XXXXX_V2_PROGRAM_BURST_SIZE controls the program operation. On typical hardware programming the flash requires disabling interrupts and the cache for an extended period of time. Some or all of the flash hardware will be unusable while each word is programmed, and disabling interrupts is the only reliable way of ensuring that no interrupt handler or other thread will try to access the flash in the middle of an operation. This can have a major impact on the real-time responsiveness of the typical applications. To ameliorate this the driver wil perform writes in small bursts, briefly re-enabling the cache and interrupts between each burst. The number of words written per burst is controlled by this configuration operation: reducing the value will improve real-time response but will add overhead, so the actual flash program operation will take longer; conversely more writes per burst will worsen response times but reduce overhead.

Similarly erasing a block of flash safely requires disabling interrupts and the cache. Erasing a block can easily take a second or so, and disabling interrupts for such a long period of time is usually undesirable. Hence the driver can also perform the erase in bursts, using the hardware's suspend and resume capabilities. CYGNUM_DEVS_FLASH_AM29XXXXX_V2_ERASE_BURST_DURATION controls the number of polling loops during which interrupts are disabled. Reducing its value improves responsiveness at the cost of performance, and increasing its value has the opposite effect. Note that too low a value may prevent the erase operation from working at all: the chip will be spending its time suspending and resuming, rather than actually performing the erase. The minimum value will depend on the specific hardware.

There are a number of other options, relating mostly to hardware characteristics. It is very rare that application developers need to change any of these. For example the option CYGNUM_DEVS_FLASH_AMD_AM29XXXXX_V2_ERASE_REGIONS may need a non-default value if the flash devices used on the target have an unusual boot block layout. If so the platform HAL will impose a requires constraint on this option and the configuration system will resolve the constraint. The only time it might be necessary to change the value manually is if the actual board being used is a variant of the one supported by the platform HAL and uses a different flash chip.

Name

Instantiating — including the driver in an eCos target

Synopsis

#include <cyg/io/am29xxxxx_dev.h>

`int` **`cyg_am29xxxxx_init_check_devid_XX`**`(`*`device`*`);`

`int` **`cyg_am29xxxxx_init_cfi_XX`**`(`*`device`*`);`

`int` **`cyg_am29xxxxx_erase_XX`**`(`*`device`*`, `*`addr`*`);`

`int` **`cyg_am29xxxxx_program_XX`**`(`*`device`*`, `*`addr`*`, `*`data`*`, `*`len`*`);`

`int` **`cyg_at49xxxx_softlock`**`(`*`device`*`, `*`addr`*`);`

`int` **`cyg_at49xxxx_hardlock`**`(`*`device`*`, `*`addr`*`);`

`int` **`cyg_at49xxxx_unlock`**`(`*`device`*`, `*`addr`*`);`

`int` **`cyg_am29xxxxx_read_devid_XX`**`(`*`device`*`);`

Description

The AM29xxxxx family contains some hundreds of different flash devices, all supporting the same basic set of operations but with various common or uncommon extensions. The devices vary in capacity, performance, boot block layout, and width. There are also platform-specific issues such as how many devices are actually present on the board and where they are mapped in the address space. The AM29xxxxx driver package cannot know the details of every chip and every platform. Instead it is the responsibility of another package, usually the platform HAL, to supply the necessary information by instantiating some data structures. Two pieces of information are especially important: the bus configuration and the boot block layout.

Flash devices are typically 8-bits, 16-bits, or 32-bits wide (64-bit devices are not yet in common use). Most 16-bit devices will also support 8-bit accesses, but not all. Similarly 32-bit devices can be accessed 16-bits at a time or 8-bits at a time. A board will have one or more of these devices on the bus. For example there may be a single 16-bit device on a 16-bit bus, or two 16-bit devices on a 32-bit bus. The processor's bus logic determines which combinations are possible, and there will be a trade off between cost and performance: two 16-bit devices in parallel can provide twice the memory bandwidth of a single device. The driver supports the following combinations:

8
: A single 8-bit flash device on an 8-bit bus.

16
: A single 16-bit flash device on a 16-bit bus.

32
: A single 32-bit flash device on an 32-bit bus.

88
: Two parallel 8-bit devices on an 16-bit bus.

8888
: Four parallel 8-bit devices on a 32-bit bus.

1616
: Two parallel 16-bit devices on a 32-bit bus, with one device providing the bottom two bytes of each 32-bit datum and the other device providing the top two bytes.

16as8
: A single 16-bit flash device connected to an 8-bit bus.

32as16 A single 32-bit flash device connected to a 16-bit bus.

These configuration all require slightly different code to manipulate the hardware. The AM29xxxxx driver package provides separate functions for each configuration, for example `cyg_am29xxxxx_erase_16` and `cyg_am29xxxxx_program_1616`.

Caution

At the time of writing not all the configurations have been tested.

The second piece of information is the boot block layout. Flash devices are subdivided into blocks (also known as sectors - both terms are in common use). Some operations such as erase work on a whole block at a time, and for most applications a block is the smallest unit that gets updated. A typical block size is 64K. It is inefficient to use an entire 64K block for small bits of configuration data and similar information, so many flash devices also support a number of smaller boot blocks. A typical 2MB flash device could have a single 16K block, followed by two 8K blocks, then a 32K block, and finally 31 full-size 64K blocks. The boot blocks may appear at the bottom or the top of the device. So-called uniform devices do not have boot blocks, just full-size ones. The driver needs to know the boot block layout. With modern devices it can work this out at run-time, but often it is better to provide the information statically.

Example

In most cases flash support is specific to a platform. Even if two platforms happen to use the same flash device there are likely to be differences such as the location in the address map. Hence there is little possibility of re-using the platform-specific code, and this code should be placed in the platform HAL rather than in a separate package. Typically this involves a separate file and a corresponding compile property in the platform HAL's CDL:

```
cdl_package CYGPKG_HAL_M68K_ALAIA {
    …
    compile -library=libextras.a alaia_flash.c
    …
}
```

The contents of this file will not be accessed directly, only indirectly via the generic flash API, so normally it would be removed by link-time garbage collection. To avoid this the object file has to go into `libextras.a`.

The actual file `alaia_flash.c` will look something like:

```
#include <pkgconf/system.h>
#ifdef CYGPKG_DEVS_FLASH_AMD_AM29XXXXX_V2

#include <cyg/io/flash.h>
#include <cyg/io/flash_dev.h>
#include <cyg/io/am29xxxxx_dev.h>

static const CYG_FLASH_FUNS(hal_alaia_flash_amd_funs,
    &cyg_am29xxxxx_init_check_devid_16,
    &cyg_flash_devfn_query_nop,
    &cyg_am29xxxxx_erase_16,
    &cyg_am29xxxxx_program_16,
    (int (*)(struct cyg_flash_dev*, const cyg_flashaddr_t, void*, size_t))0,
    &cyg_flash_devfn_lock_nop,
    &cyg_flash_devfn_unlock_nop);

static const cyg_am29xxxxx_dev hal_alaia_flash_priv = {
    .devid      = 0x45,
    .block_info = {
        { 0x00004000, 1 },
        { 0x00002000, 2 },
        { 0x00008000, 1 },
        { 0x00010000, 63 }
    }
};
```

```
CYG_FLASH_DRIVER(hal_alaia_flash,
                 &hal_alaia_flash_amd_funs,
                 0,
                 0xFFC00000,
                 0xFFFFFFFF,
                 4,
                 hal_alaia_flash_priv.block_info,
                 &hal_alaia_flash_priv
);
#endif
```

The bulk of the file is protected by an `#ifdef` for the AM29xxxxx flash driver. That driver will only be active if the generic flash support is enabled. Without that support there will be no way of accessing the device so instantiating the data structures would serve no purpose. The rest of the file is split into three structure definitions. The first supplies the functions which will be used to perform the actual flash accesses, using a macro provided by the generic flash code in `cyg/io/flash_dev.h`. The relevant ones have an `_16` suffix, indicating that on this board there is a single 16-bit flash device on a 16-bit bus. The second provides information specific to AM29xxxxx flash devices. The third provides the cyg_flash_dev structure needed by the generic flash code, which contains pointers to the previous two.

Functions

All eCos flash device drivers must implement a standard interface, defined by the generic flash code `CYGPKG_IO_FLASH`. This interface includes a table of seven function pointers for various operations: initialization, query, erase, program, read, locking and unlocking. The query operation is optional and the generic flash support provides a dummy implementation `cyg_flash_devfn_query_nop`. AM29xxxxx flash devices are always directly accessible so there is no need for a separate read function. The remaining functions are more complicated.

Usually the table can be declared `const`. In a ROM startup application this avoids both ROM and RAM copies of the table, saving a small amount of memory. `const` should not be used if the table may be modified by a platform-specific initialization routine.

Initialization

There is a choice of three main initialization functions. The simplest is `cyg_flash_devfn_init_nop`, which does nothing. It can be used if the cyg_am29xxxxx_dev and cyg_flash_dev structures are fully initialized statically and the flash will just work without special effort. This is useful if it is guaranteed that the board will always be manufactured using the same flash chip, since the nop function involves the smallest code size and run-time overheads.

The next step up is `cyg_am29xxxxx_init_check_devid_XX`, where `XX` will be replaced by the suffix appropriate for the bus configuration. It is still necessary to provide all the device information statically, including the *`devid`* field in the cyg_am29xxxxx_dev structure. This initialization function will attempt to query the flash device and check that the provided device id matches the actual hardware. If there is a mismatch the device will be marked uninitialized and subsequent attempts to manipulate the flash will fail.

If the board may end up being manufactured with any of a number of different flash chips then the driver can perform run-time initialization, using a `cyg_am29xxxxx_init_cfi_XX` function. This queries the flash device as per the Common Flash Memory Interface Specification, supported by all current devices (although not necessarily by older devices). The *`block_info`* field in the cyg_am29xxxxx_dev structure and the *`end`* and *`num_block_infos`* fields in the cyg_flash_dev structure will be filled in. It is still necessary to supply the *`start`* field statically since otherwise the driver will not know how to access the flash device. The main disadvantage of using CFI is that it increases the code size.

Caution

If CFI is used then the cyg_am29xxxxx_dev structure must not be declared `const`. The CFI code will attempt to update the structure and will fail if the structure is held in read-only memory. This would leave the flash driver non-functional.

A final option is to use a platform-specific initialization function. This may be useful if the board may be manufactured with one of a small number of different flash devices and the platform HAL needs to adapt to this. The AM29xxxxx driver provides a utility function to read the device id, `cyg_am29xxxxx_read_devid_XX`:

```
static int
alaia_flash_init(struct cyg_flash_dev* dev)
{
    int devid = cyg_am29xxxxx_read_devid_1616(dev);
    switch(devid) {
        case 0x0042 :
          …
        case 0x0084 :
          …
        default:
          return CYG_FLASH_ERR_DRV_WRONG_PART;
    }
}
```

There are many other possible uses for a platform-specific initialization function. For example initial prototype boards might have only supported 8-bit access to a 16-bit flash device rather than 16-bit access, but this problem was fixed in the next revision. The platform-specific initialization function can figure out which model board it is running on and replace the default `16as8` functions with faster `16` ones.

Erase and Program

The AM29xxxxx driver provides erase and program functions appropriate for the various bus configurations. On most targets these can be used directly. On some targets it may be necessary to do some extra work before and after the erase and program operations. For example if the hardware has an MMU then the part of the address map containing the flash may have been set to read-only, in an attempt to catch spurious memory accesses. Erasing or programming the flash requires write-access, so the MMU settings have to be changed temporarily. As another example some flash device may require a higher voltage to be applied during an erase or program operation. or a higher voltage may be desirable to make the operation proceed faster. A typical platform-specific erase function would look like this:

```
static int
alaia_flash_erase(struct cyg_flash_dev* dev, cyg_flashaddr_t addr)
{
    int result;
    …  // Set up the hardware for an erase
    result = cyg_am29xxxxx_erase_32(dev, addr);
    …  // Revert the hardware change
    return result;
}
```

There are two configurations which affect the erase and program functions, and which a platform HAL may wish to change: `CYGNUM_DEVS_FLASH_AMD_AM29XXXXX_V2_ERASE_TIMEOUT` and `CYGNUM_DEVS_FLASH_AMD_AM29XXXXX_V2_PROGRAM_TIMEOUT`. The erase and program operations both involve polling for completion, and these timeout impose an upper bound on the polling loop. Normally these operations should never take anywhere close to the timeout period, so a timeout indicates a catastrophic failure that should really be handled by a watchdog reset. A reset is particularly appropriate because there will be no clean way of aborting the flash operation. The main reason for the timeouts is to help with debugging when porting to new hardware. If there is a valid reason why a particular platform needs different timeouts then the platform HAL's CDL can require appropriate values for these options.

Locking

There is no single way of implementing the block lock and unlock operations on all AM29xxxxx devices. If these operations are supported at all then usually they involve manipulating the voltages on certain pins. This would not be able to be handled by generic driver code since it requires knowing how these pins can be manipulated via the processor's GPIO lines. Therefore the AM29xxxxx driver does not usually provide lock and unlock functions, and instead the generic dummy functions `cyg_flash_devfn_lock_nop` and `cyg_flash_devfn_unlock_nop` should be used. An exception exists for the

AT49xxxx family of devices which are sufficiently AMD compatible in other respects. Otherwise, if a platform does provide a way of implementing the locking then this can be handled by platform-specific functions.

```
static int
alaia_lock(struct cyg_flash_dev* dev, const cyg_flashaddr_t addr)
{
    …
}

static int
alaia_unlock(struct cyg_flash_dev* dev, const cyg_flashaddr_t addr)
{
    …
}
```

If real locking functions are implemented then the platform HAL's CDL script should implement the CDL interface `CYGHWR_IO_FLASH_BLOCK_LOCKING`. Otherwise the generic flash package may believe that none of the flash drivers in the system provide locking functionality and disable the interface functions.

AT49xxxx locking

As locking is standardised across the AT49xxxx family of AMD AM29xxxxx compatible Flash parts, a method supporting this is included within this driver. `cyg_at49xxxx_softlock_XX` provides a means of locking a Flash sector such that it may be subsequently unlocked. `cyg_at49xxxx_hardlock_XX` locks a sector such that it cannot be unlocked until after reset or a power cycle. `cyg_at49xxxx_unlock_XX` unlocks a sector that has previously been softlocked. At power on or Flash device reset, all sectors default to being softlocked.

Other

The driver provides a set of functions `cyg_am29xxxxx_read_devid_XX`, one per supported bus configuration. These functions take a single argument, a pointer to the cyg_flash_dev structure, and return the chip's device id. For older devices this id is a single byte. For more recent devices the id is a 3-byte value, 0x7E followed by a further two bytes that actually identify the device. `cyg_am29xxxxx_read_devid_XX` is usually called only from inside a platform-specific driver initialization routine, allowing the platform HAL to adapt to the actual device present on the board.

Device-Specific Structure

The cyg_am29xxxxx_dev structure provides information specific to AM29xxxxx flash devices, as opposed to the more generic flash information which goes into the cyg_flash_dev structure. There are only two fields: *`devid`* and *`block_info`*.

`devid` is only needed if the driver's initialization function is set to `cyg_am29xxxxx_init_check_devid_XX`. That function will extract the actual device info from the flash chip and compare it with the *`devid`* field. If there is a mismatch then subsequent operations on the device will fail.

The *`block_info`* field consists of one or more pairs of the block size in bytes and the number of blocks of that size. The order must match the actual hardware device since the flash code will use the table to determine the start and end locations of each block. The table can be initialized in one of three ways:

1. If the driver initialization function is set to `cyg_flash_devfn_init_nop` or `cyg_am29xxxxx_init_check_devid_XX` then the block information should be provided statically. This is appropriate if the board will also be manufactured using the same flash chip.

2. If `cyg_am29xxxxx_init_cfi_XX` is used then this will fill in the block info table. Hence there is no need for static initialization.

3. If a platform-specific initialization function is used then either this should fill in the block info table, or the info should be provided statically.

The size of the *block_info* table is determined by the configuration option CYGNUM_DEVS_FLASH_AMD_AM29XXXXX_V2_ERASE_REGIONS. This has a default value of 4, which should suffice for nearly all AM29xxxxx flash devices. If more entries are needed then the platform HAL's CDL script should require a larger value.

If the cyg_am29xxxxx_dev structure is statically initialized then it can be const. This saves a small amount of memory in ROM startup applications. If the structure is updated at run-time, either by cyg_am29xxxxx_init_cfi_XX or by a platform-specific initialization routine, then it cannot be const.

Flash Structure

Internally the generic flash code works in terms of cyg_flash_dev structures, and the platform HAL should define one of these. The structure should be placed in the cyg_flashdev table. The following fields need to be provided:

funs
: This should point at the table of functions.

start
: The base address of the flash in the address map. On some board the flash may be mapped into memory several times, for example it may appear in both cached and uncached parts of the address space. The *start* field should correspond to the cached address.

end
: The address of the last byte in the flash. It can either be statically initialized, or cyg_am29xxxxx_init_cfi_XX will calculate its value at run-time.

num_block_infos
: This should be the number of entries in the *block_info* table. It can either be statically initialized or it will be filled in by cyg_am29xxxxx_init_cfi_XX.

block_info
: The table with the block information is held in the cyg_am29xxxxx_dev structure, so this field should just point into that structure.

priv
: This field is reserved for use by the device driver. For the AM29xxxxx driver it should point at the appropriate cyg_am29xxxxx_dev structure.

The cyg_flash_dev structure contains a number of other fields which are manipulated only by the generic flash code. Some of these fields will be updated at run-time so the structure cannot be declared const.

Multiple Devices

A board may have several flash devices in parallel, for example two 16-bit devices on a 32-bit bus. It may also have several such banks to increase the total amount of flash. If each device provides 2MB, there could be one bank of 2 parallel flash devices at 0xFF800000 and another bank at 0xFFC00000, giving a total of 8MB. This setup can be described in several ways. One approach is to define two cyg_flash_dev structures. The table of function pointers can usually be shared, as can the cyg_am29xxxxx_dev structure. Another approach is to define a single cyg_flash_dev structure but with a larger *block_info* table, covering the blocks in both banks of devices. The second approach makes more efficient use of memory.

Many variations are possible, for example a small slow flash device may be used for initial bootstrap and holding the configuration data, while there is also a much larger and faster device to hold a file system. Such variations are usually best described by separate cyg_flash_dev structures.

If more than one cyg_flash_dev structure is instantiated then the platform HAL's CDL script should implement the CDL interface CYGHWR_IO_FLASH_DEVICE once for every device past the first. Otherwise the generic code may default to the case of a single flash device and optimize for that.

Platform-Specific Macros

The AM29xxxxx driver source code includes the header files cyg/hal/hal_arch.h and cyg/hal/hal_io.h, and hence indirectly the corresponding platform header files (if defined). Optionally these headers can define macros which are used inside the driver, thus giving the HAL limited control over how the driver works.

Cache Management

By default the AM29xxxxx driver assumes that the flash can be accessed uncached, and it will use the HAL CYGARC_UNCACHED_ADDRESS macro to map the cached address in the *start* field of the cyg_flash_dev structure into an uncached address. If for any reason this HAL macro is inappropriate for the flash then an alternative macro HAL_AM29XXXXX_UNCACHED_ADDRESS can be defined instead. However fixing the CYGARC_UNCACHED_ADDRESS macro is normally the better solution.

If there is no way of bypassing the cache then the platform HAL should implement the CDL interface CYGHWR_DEVS_FLASH_AMD_AM29XXXXX_V2_CACHED_ONLY. The flash driver will now disable and re-enable the cache as required. For example a program operation will involve the following:

```
AM29_INTSCACHE_STATE;
AM29_INTSCACHE_BEGIN();
while ( ! finished ) {
    write a burst of CYGNUM_DEVS_FLASH_AMD_AM29XXXXX_V2_PROGRAM_BURST_SIZE
    AM29_INTSCACHE_SUSPEND();
    AM29_INTSCACHE_RESUME();
}
AM29_INTSCACHE_END();
```

The default implementations of these INTSCACHE macros are as follows: STATE defines any local variables that may be needed, e.g. to save the current interrupt state; BEGIN disables interrupts, synchronizes the data caches, disables it, and invalidates the current contents; SUSPEND re-enables the data cache and then interrupts; RESUME disables interrupts and the data cache; END re-enables the cache and then interrupts. The cache is only disabled when interrupts are disabled, so there is no possibility of an interrupt handler running or a context switch occurring while the cache is disabled, potentially leaving the system running very slowly. The data cache synchronization ensures that there are no dirty cache lines, so when the cache is disabled the low-level flash write code will not see stale data in memory. The invalidate ensures that at the end of the operation higher-level code will not pick up stale cache contents instead of the newly written flash data. The SUSPEND and RESUME macros only re-enable and disable the data cache. An interrupt and possibly a context switch may occur between these macros and use the cache normally. It is assumed that any code which runs at this time will not touch the memory being used by the flash operation, so as far as the low-level program code is concerned it can just continue to use the uncached memory contents as set up by the BEGIN macro. If any code modifies the const data currently being written to a flash block or tries to read the flash block being modified then the system's behaviour is undefined. Theoretically a more robust approach is possible, synchronizing and invalidating the cache again in every RESUME. However these cache operations can be expensive and RESUME may get invoked some thousands of times for every flash block, so this alternative approach would cripple the driver's performance.

Some implementations of the HAL cache macros may not provide the exact semantics required by the flash driver. For example HAL_DCACHE_DISABLE may have an unwanted side effect, or it may do more work than is needed here. The driver will check for alternative macros HAL_AM29XXXXX_INTSCACHE_STATE, HAL_AM29XXXXX_INTSCACHE_BEGIN, HAL_AM29XXXXX_INTSCACHE_SUSPEND, HAL_AM29XXXXX_INTSCACHE_RESUME and HAL_AM29XXXXX_INTSCACHE_END, using these instead of the defaults.

Chapter 67. Atmel AT45xxxxxx DataFlash Device Driver

Name

Overview — eCos Support for Atmel AT45xxxxxx DataFlash Devices and Compatibles

Description

The `CYGPKG_DEVS_FLASH_ATMEL_DATAFLASH` Atmel AT45xxxxxx V2 flash driver package implements support for the AT45xxxxxx family of flash devices and compatibles. The driver is not normally accessed directly. Instead application code will use the API provided by the generic flash driver package `CYGPKG_IO_FLASH`, for example by calling functions like `cyg_flash_program`.

DataFlash devices are accessed via the SPI bus. Therefore, any platform on which this driver is to be used must also have an SPI driver. The DataFlash driver accesses the driver via the standard SPI API calls defined in the `CYGPKG_IO_SPI` package.

Configuration

The DataFlash flash driver package will be loaded automatically when configuring eCos for a target with suitable hardware. However the driver will be inactive unless the generic flash package `CYGPKG_IO_FLASH` is loaded. It may be necessary to add this generic package to the configuration explicitly before the driver functionality becomes available. There should never be any need to load or unload the DataFlash driver package.

Name

Instantiating — including the driver in an eCos target

Synopsis

#include <cyg/io/dataflash.h>

`CYG_DATAFLASH_FLASH_DRIVER` (*`name`*, *`sdev`*, *`addr`*, *`start`*, *`end`*);

Description

The DataFlash family contains several different flash devices, all supporting the same basic set of operations The devices vary in capacity, performance and sector layout. There are also platform-specific issues such as which SPI bus the device is connected to, and which chip select it uses. The DataFlash driver package cannot know all this information. Instead it is the responsibility of another package, either the platform HAL or a flash configuration package, to instantiate some DataFlash device structures.

The definition of the parameters is split between two data structures. The first is an SPI specific data structure describing the SPI bus, chip select and signalling characteristics for the device. This is usually defined in an SPI specific configuration package or the platform HAL. The reader is referred to the SPI documentation for details of the contents of this structure, but see later for an example.

The second data structure defines the characteristics of the DataFlash device for use by the flash subsystem. For convenience a macro, `CYG_DATAFLASH_FLASH_DRIVER`, has been defined to automate the generation of this structure. This macro takes a number of arguments:

`name`

This provides a name fragment for distinguishing this DataFlash device from any others. It is concatenated with `cyg_dataflash_priv_` to form a static variable name. Any unique string that obeys the rules of C variable names is sufficient.

`sdev`

A pointer to the SPI device object that describes this flash device. If the SPI device has been declared with the name `spi_dataflash_dev0` then this argument should be `(&spi_dataflash_dev0.spi_device)`.

`addr`

SPI DataFlash devices do not have a physical address in the system memory space. However, the flash subsystem expects all flash devices to have an address. This argument gives the DataFlash device a virtual address in the memory space. This should be allocated to a location that contains no other flash devices, and to avoid confusion, no other memory or devices. Subsequently, the DataFlash may be accessed by performing flash system accesses starting at this address.

`start`

This parameter defines the sector at which the flash device mapping starts. The beginning of this sector is mapped to the virtual address given in the `addr` argument. This value will usually be zero, but may be non-zero if there is reserved data at the beginning of the DataFlash.

`end`

This parameter defines the sector at which the flash device mapping ends. The end of this sector defines the maximum extent in the address space that the DataFlash occupies. The exact size of the mapping will depend on the number and sizes of the flash sectors covered. This value will usually be the number of sectors in the device, but may be less if there is reserved data at the end of the DataFlash.

Example

DataFlash support is usually specific to each platform. Even if two platforms happen to use the same flash device there are likely to be differences such as the SPI bus, chip select and location in the address map. Hence there is little possibility of re-using the platform-specific code, and this code is generally placed in the platform HAL or in a separate platform specific package.

The code to declare a DataFlash device might appear as follows:

```
#include <cyg/io/spi.h>
#include <cyg/io/spi_at91.h>
#include <cyg/io/dataflash.h>

__externC cyg_spi_at91_device_t spi_dataflash_dev0;

CYG_DATAFLASH_FLASH_DRIVER( eb55_dataflash,
                            (&spi_dataflash_dev0.spi_device),
                            0x08000000,
                            0,
                            16 );
```

Here, we are defining a dataflash device that is mapped to virtual address 0x08000000. The start and end sectors cover the entire flash device, 17 sectors. In addition to a DataFlash specific structure, this macro also creates a cyg_flash_dev structure which supplies the driver interface to the flash subsystem. The SPI device structure is defined elsewhere, in an SPI specific package, and has the following format:

```
#include <cyg/infra/cyg_type.h>
#include <cyg/io/spi.h>
#include <cyg/io/spi_at91.h>

// AT45DB321B DataFlash
cyg_spi_at91_device_t spi_dataflash_dev0 CYG_SPI_DEVICE_ON_BUS(0) =
{
    .spi_device.spi_bus = &cyg_spi_at91_bus.spi_bus,

    .dev_num     = 0,       // Device number
    .cl_pol      = 1,       // Clock polarity (0 or 1)
    .cl_pha      = 0,       // Clock phase (0 or 1)
    .cl_brate    = 8192000, // Clock baud rate
    .cs_up_udly  = 1,       // Delay in usec between CS up and transfer start
    .cs_dw_udly  = 1,       // Delay in usec between transfer end and CS down
    .tr_bt_udly  = 1        // Delay in usec between two transfers
};
```

The parameters here attach the device to the only SPI bus in the hardware, use chip select 0 to access it, and set the communication parameters for the clock polarity, phase, baud rate and delays.

Device Info

The exact DataFlash device attached to the SPI bus is discovered by the driver by querying it and matching the device ID against an internal table of supported devices. If a particular device is not currently supported, it must be added to the table in `devs_flash_atmel_dataflash.c`. A typical entry in this table appears as follows:

```
{   // AT45DB321B
    device_id:    0x0D,
    page_size:    528,
    page_count:   8192,
    baddr_bits:   10,
    block_size:   8,
    sector_sizes: {  1, 63, 64, 64, 64, 64, 64, 64,
                    64, 64, 64, 64, 64, 64, 64, 64, 64 },
    sector_count: 17
},
```

The fields of this structure are:

`device_id` This defines the device ID returned as part of the status register. This is the field that is matched to select this DataFlash device.

`page_size` This gives the size of pages in this flash device.

`page_size` This gives the total number of pages in this device.

`baddr_bits` This gives the number of bits used in the SPI command address format for specifying a byte address within a page.

`block_size` This gives the number of pages in a block.

`sector_sizes` This is an array giving the size, in blocks, of each sector of the DataFlash.

`sector_count` This gives the number of entries in the *`sector_sizes`* array.

Chapter 68. Freescale MCFxxxx CFM Flash Device Driver

Name

CYGPKG_DEVS_FLASH_M68K_MCFxxxx_CFM — eCos Flash Driver for MCFxxxx CFM On-chip Flash

Description

Some members of the Freescale MCFxxxx family, for example the MCF5213, come with on-chip flash in the form of a ColdFire Flash Module or CFM. This package `CYGPKG_DEVS_FLASH_M68K_MCFxxxx_CFM` provides an eCos flash device driver for CFM hardware. Normally the driver is not accessed directly. Instead application code will use the API provided by the generic flash driver package `CYGPKG_IO_FLASH`, for example by calling functions like `cyg_flash_program`.

Configuration Options

The CFM flash driver package will be loaded automatically when configuring eCos for a target with suitable hardware. However the driver will be inactive unless the generic flash package `CYGPKG_IO_FLASH` is loaded. It may be necesary to add this generic package to the configuration explicitly before the driver functionality becomes available. There should never be any need to load or unload the CFM driver package.

The driver contains a small number of configuration options which application developers may wish to tweak.

Misaligned Writes

The CFM hardware only supports writes of a whole 32-bit integer at a time. For most applications this is not a problem and the driver imposes this restriction on higher-level code, so when calling `cyg_flash_program` the destination address must be on a 32-bit boundary and the length must be a multiple of four bytes. If this restriction is unacceptable then support for misaligned writes of arbitrary lengths can be enabled via configuration option `CYGIMP_DEVS_FLASH_M68K_MCFxxxx_CFM_MISALIGNED_WRITES`. The implementation involves reading in the existing flash contents and or'ing in the new data. The default behaviour is to leave this disabled since most applications do not require the functionality and it just adds to the code size.

Locking

CFM has a somewhat unusual approach to implementing lock and unlock support. The first 1K of on-chip flash is normally reserved for the M68K exception vectors, on the assumption that the processor will or may boot from here. This is immediately followed by a hal_mcfxxxx_cfm_security_settings data structure at offset 0x400. The structure has a 32-bit field *`cfm_prot`* which determines the initial locking status. The whole CFM flash array is split into 32 sectors. Each bit in *`cfm_prot`* determines the initial locked state of all blocks within that sector, with 1 for locked and 0 for unlocked. Locking and unlocking can only affect a whole sector at a time, not individual flash blocks (unless of course the flash is organized such that there exactly 32 flash blocks).

In typical usage the on-chip flash will be used for bootstrap and hence the security settings are part of the boot image. The default security settings are supplied by the processor or platform HAL and will be such that all flash sectors are unlocked. This is the most convenient setting when developing software. However an application can override this and thus lock part or all of the flash.

The driver provides two configuration options related to flash locking. `CYGIMP_DEVS_FLASH_M68K_MCFxxxx_CFM_AUTO_UNLOCK` causes all of flash to be unlocked automatically during driver initialization. This gives simple and deterministic behaviour irrespective of the current contents of the flash security settings structure. However it leaves the flash more vulnerable to accidental corruption by an errant application. `CYGIMP_DEVS_FLASH_M68K_MCFxxxx_CFM_SUPPORT_LOCKING` enables support for fine-grained locking using the generic functions `cyg_flash_lock` and `cyg_flash_unlock`. This provides more protection against errant applications, at the cost of increased application complexity.

RAM Functions and Interrupts

When performing a flash erase or program operation part or all of the flash may become inaccessible. The exact details of this vary between ColdFire processors. Obviously this means that the low-level functions which manipulate the flash cannot reside in the same area of flash as any blocks that may get erased or programmed. Worse, if an interrupt happens during an erase or program operation and the interrupt handler involves code or data in the same area of flash, or may cause a higher priority thread to wake up which accesses that area of flash, then the system is likely to fail. To avoid problems the flash driver takes two precautions by default:

1. The low-level flash functions are located in RAM. If necessary they are copied from ROM from RAM during system initialization.
2. The low-level flash functions disable interrupts around any code which may leave parts of the flash inaccessible.

This combination avoids problems but at the cost of increased RAM usage and often increased interrupt latency. In some circumstances these precautions are unnecessary. For example suppose there is 512K of flash split into two logical blocks of 256K each, such that an erase or program operation affects all of one logical block but not the other. If the application has been arranged such that all code and constant data resides in the bottom 256K and flash operations are only performed on the remaining 256K then there is no need to place the low-level flash functions in RAM. The configuration option `CYGIMP_DEVS_FLASH_M68K_MCFxxxx_CFM_FUNCTIONS_IN_RAM` can then be disabled. In a similar scenario, if the top 256K of flash are only ever accessed via the flash API then there is no need to disable interrupts: the generic flash layer ensures only one thread can perform flash operations on a given device via a mutex. The configuration option `CYGIMP_DEVS_FLASH_M68K_MCFxxxx_CFM_LEAVE_INTERRUPTS_ENABLED` can then be enabled.

Additional Functionality

The driver exports two functions which offer functionality not accessible via the standard flash API:

```
#include <cyg/io/mcfxxxx_cfm_dev.h>

externC void     cyg_mcfxxxx_cfm_unlockall(void);
externC cyg_bool cyg_mcfxxxx_cfm_is_locked(const cyg_flashaddr_t addr);
```

The first can be used to unlock all flash sectors, effectively bypassing any locking set by the hal_mcfxxxx_cfm_security_settings structure. The second can be used to query whether or not the block containing the specified address is currently locked. Both functions should be called only after flash has been initialized.

Instantiating a CFM Flash device

The CFM package only provides the device driver functions needed for manipulating CFM flash. It does not actually create a device instance. The amount of on-chip flash varies between ColdFire processors and the driver package does not maintain any central repository about the characteristics of each processor. Instead it is left to other code, usually the processor HAL, to instantiate the flash device. That makes it possible to add support for new processors simply by adding a new processor HAL, with no need to change the flash driver package.

The CFM package provides a utility macro for instantiating a device. Typical usage would be:

```
#include <cyg/io/mcfxxxx_cfm_dev.h>

CYG_MCFxxxx_CFM_INSTANCE(0x00000000, 0x0003FFFF, 2048);
```

The first two arguments specify the base and end address of the flash in the memory map. In this example there is 256K of flash mapped to location 0. Typically the base address is set via the FLASHBAR system register and the size is of course determined by the specific ColdFire processor being used. The final argument is the size of a flash block, in other words the unit of erase operations. This will have to obtained from the processor documentation.

The `CYG_MCFxxxx_CFM_INSTANCE` macro may instantiate a device with or without software locking support, as determined by the driver configuration option `CYGIMP_DEVS_FLASH_M68K_MCFxxxx_CFM_SUPPORT_LOCKING` .

If for some reason the `CYG_MCFxxxx_CFM_INSTANCE` is inappropriate for a specific processor then the mcfxxxx_cfm_dev.h header file exports all the device driver functions, so code can create a flash device instance explicitly.

Chapter 69. Intel Strata Flash Device Driver

Name

Overview — eCos Support for Intel Strata Flash Devices and Compatibles

Description

The CYGPKG_DEVS_FLASH_STRATA_V2 flash driver package implements support for the Intel Strata family of flash devices and compatibles. The driver is not normally accessed directly. Instead application code will use the API provided by the generic flash driver package CYGPKG_IO_FLASH, for example by calling functions like cyg_flash_program. There are a small number of additional functions specific to Strata devices.

The driver imposes one restriction on application code which developers should be aware of: when programming the flash the destination addresses must be aligned to a bus boundary. For example if the target hardware has a single flash device attached to a 16-bit bus then program operations must involve a multiple of 16-bit values aligned to a 16-bit boundary. Note that it is the bus width that matters, not the device width. If the target hardware has two 16-bit devices attached to a 32-bit bus then program operations must still be aligned to a 32-bit boundary, even though in theory a 16-bit boundary would suffice. In practice this is rarely an issue, and requiring the larger boundary greatly simplifies the code and improves performance.

Note

Many eCos targets with Strata or compatible flash devices will still use the older driver package CYGPKG_DEVS_FLASH_STRATA. Only newer ports and some older ports that have been converted will use the V2 driver. This documentation only applies to the V2 driver.

Configuration Options

The Strata flash driver package will be loaded automatically when configuring eCos for a target with suitable hardware. However the driver will be inactive unless the generic flash package CYGPKG_IO_FLASH is loaded. It may be necessary to add this generic package to the configuration explicitly before the driver functionality becomes available. There should never be any need to load or unload the Strata driver package.

The Strata flash driver package contains a small number of configuration options which application developers may wish to tweak. CYGNUM_DEVS_FLASH_STRATA_V2_PROGRAM_BURST_SIZE controls the program operation. On typical hardware programming the flash requires disabling interrupts and possibly the cache for an extended period of time. If the hardware does not provide any way of bypassing the cache when writing to the flash then the cache must be disabled or the commands written to the flash may get stuck inside the cache instead of going directly to the flash chip. Some or all of the flash hardware will be unusable while each word is programmed, and disabling interrupts is the only reliable way of ensuring that no interrupt handler or other thread will try to access the flash in the middle of an operation. This can have a major impact on the real-time responsiveness of the typical applications. To ameliorate this the driver will perform writes in small bursts, briefly re-enabling the cache and interrupts between each burst. The number of words written per burst is controlled by this configuration operation: reducing the value will improve real-time response but will add overhead, so the actual flash program operation will take longer; conversely more writes per burst will worsen response times but reduce overhead. For flash devices which support buffered writes the driver will always try to use a full buffer so there is no point in reducing the burst size to less than the buffer size, but setting the burst size to a larger value is permitted.

Similarly erasing a block of flash safely requires disabling interrupts and possibly the cache. Erasing a block can easily take a second or so, and disabling interrupts for such a long period of time is usually undesirable. Hence the driver can also perform the erase in bursts, using the hardware's suspend and resume capabilities. CYGNUM_DEVS_FLASH_STRATA_V2_ERASE_BURST_DURATION controls the number of polling loops during which interrupts are disabled. Reducing its value improves responsiveness at the cost of performance, and increasing its value has the opposite effect. Note that too low a value may prevent the erase operation from working at all: the chip will be spending its time suspending and resuming, rather than actually performing the erase. The minimum value will depend on the specific hardware.

There are a number of other options, relating mostly to hardware characteristics. It is very rare that application developers need to change any of these. For example the option CYGNUM_DEVS_FLASH_STRATA_V2_ERASE_REGIONS may need a non-default value if the flash devices used on the target have an unusual boot block layout. If so the platform HAL will impose a requires constraint on this option and the configuration system will resolve the constraint. The only time it might be necessary to change the value manually is if the actual board being used is a variant of the one supported by the platform HAL and uses a different flash chip.

Name

Instantiating — including the driver in an eCos target

Synopsis

#include <cyg/io/strata_dev.h>

```
int cyg_strata_init_nop(device);

int cyg_strata_init_check_devid_XX(device);

int cyg_strata_init_cfi_XX(device);

int cyg_strata_erase_XX(device, addr);

int cyg_strata_program_XX(device, addr, data, len);

int cyg_strata_bufprogram_XX(device, addr, data, len);

int cyg_strata_lock_j3_XX(device, addr);

int cyg_strata_unlock_j3_XX(device, addr);

int cyg_strata_lock_k3_XX(device, addr);

int cyg_strata_unlock_k3_XX(device, addr);
```

Description

The Strata family contains a number of different devices, all supporting the same basic set of operations but with various common or uncommon extensions. The range includes:

28FxxxB3 Boot Block
: These support 8 8K boot blocks as well as the usual 64K blocks. There is no buffered write capability. The only locking mechanism available involves manipulating voltages on certain pins.

28FxxxC3
: These also have boot blocks. There is no buffered write capability. Individual blocks can be locked and unlocked in software.

28FxxxJ3
: These are uniform devices where all blocks are 128K. Buffered writes are supported. Blocks can be locked individually, but the only unlock operation is a global unlock-all.

28FxxxK3
: These are also uniform devices with 128K blocks. Buffered writes are supported. Individual blocks can be locked and unlocked in software.

Each of these comes in a range of sizes and bus widths. There are also platform-specific issues such as how many devices are actually present on the board and where they are mapped in the address space. The Strata driver package cannot know all this information. Instead it is the responsibility of another package, usually the platform HAL, to instantiate some flash device structures. Two pieces of information are especially important: the bus configuration and the boot block layout.

Flash devices are typically 8-bits, 16-bits, or 32-bits wide (64-bit devices are not yet in common use). Most 16-bit devices will also support 8-bit accesses, but not all. Similarly 32-bit devices can be accessed 16-bits at a time or 8-bits at a time. A board will have one or more of these devices on the bus. For example there may be a single 16-bit device on a 16-bit bus, or two 16-bit devices on a 32-bit bus. The processor's bus logic determines which combinations are possible, and usually there will be a trade off between cost and performance. For example two 16-bit devices in parallel can provide twice the memory bandwidth of a single device. The driver supports the following combinations:

8 A single 8-bit flash device on an 8-bit bus.

16 A single 16-bit flash device on a 16-bit bus.

32 A single 32-bit flash device on an 32-bit bus.

88 Two parallel 8-bit devices on an 16-bit bus.

8888 Four parallel 8-bit devices on a 32-bit bus.

1616 Two parallel 16-bit devices on a 32-bit bus, with one device providing the bottom two bytes of each 32-bit datum and the other device providing the upper two bytes.

16as8 A single 16-bit flash device connected to an 8-bit bus.

These configuration all require slightly different code to manipulate the hardware. The Strata driver package provides separate functions for each configuration, for example `cyg_strata_erase_16` and `cyg_strata_program_1616`.

Caution

At the time of writing not all the configurations have been tested.

The second piece of information is the boot block layout. Flash devices are subdivided into blocks (also known as sectors, both terms are in common use). Some operations such as erase work on a whole block at a time, and for most applications a block is the smallest unit that gets updated. A typical block size is 64K. It is inefficient to use an entire 64K block for small bits of configuration data and similar information, so some flash devices also support a number of smaller boot blocks. A typical 2MB flash device could have eight 8K blocks and 31 full-size 64K blocks. The boot blocks may appear at the bottom or the top of the device. So-called uniform devices do not have boot blocks, just full-size ones. The driver needs to know the boot block layout. With modern devices it can work this out at run-time, but often it is better to provide the information statically.

Example

Flash support is usually specific to each platform. Even if two platforms happen to use the same flash device there are likely to be differences such as the location in the address map. Hence there is little possibility of re-using the platform-specific code, and this code is generally placed in the platform HAL rather than in a separate package. Typically this involves a separate file and a corresponding compile property in the platform HAL's CDL:

```
cdl_package CYGPKG_HAL_M68K_KIKOO {
    …
    compile -library=libextras.a kikoo_flash.c
    …
}
```

The contents of this file will not be accessed directly, only indirectly via the generic flash API, so normally it would be removed by link-time garbage collection. To avoid this the object file has to go into `libextras.a`.

The actual file `kikoo_flash.c` will look something like:

```
#include <pkgconf/system.h>
#ifdef CYGPKG_DEVS_FLASH_STRATA_V2

#include <cyg/io/flash.h>
#include <cyg/io/strata_dev.h>

static const CYG_FLASH_FUNS(hal_kikoo_flash_strata_funs,
    &cyg_strata_init_check_devid_16,
    &cyg_flash_devfn_query_nop,
    &cyg_strata_erase_16,
    &cyg_strata_bufprogram_16,
    (int (*)(struct cyg_flash_dev*, const cyg_flashaddr_t, void*, size_t))0,
    &cyg_strata_lock_j3_16,
    &cyg_strata_unlock_j3_16);

static const cyg_strata_dev hal_kikoo_flash_priv = {
    .manufacturer_code = CYG_FLASH_STRATA_MANUFACTURER_INTEL,
    .device_code = 0x0017,
    .bufsize    = 16,
    .block_info = {
        { 0x00020000, 64 } // 64 * 128K blocks
    }
};

CYG_FLASH_DRIVER(hal_kikoo_flash,
                 &hal_kikoo_flash_strata_funs,
                 0,
                 0x60000000,
                 0x601FFFFF,
                 1,
                 hal_kikoo_flash_priv.block_info,
                 &hal_kikoo_flash_priv
);
#endif
```

The bulk of the file is protected by an ifdef for the Strata flash driver. That driver will only be active if the generic flash support is enabled. Without that support there will be no way of accessing the device so there is no point in instantiating the device. The rest of the file is split into three definitions. The first supplies the functions which will be used to perform the actual flash accesses, using a macro provided by the generic flash code in `cyg/io/flash_dev.h`. The relevant ones have an `_16` suffix, indicating that on this board there is a single 16-bit flash device on a 16-bit bus. The second definition provides information specific to Strata flash devices. The third provides the cyg_flash_dev structure needed by the generic flash code, which contains pointers to the previous two.

Functions

All eCos flash device drivers must implement a standard interface, defined by the generic flash code `CYGPKG_IO_FLASH`. This interface includes a table of 7 function pointers for various operations: initialization, query, erase, program, read, locking and unlocking. The query operation is optional and the generic flash support provides a dummy implementation `cyg_flash_devfn_query_nop`. Strata flash devices are always directly accessible so there is no need for a separate read function. The remaining functions are more complicated.

Usually the table can be declared `const`. In a ROM startup application this avoids both ROM and RAM copies of the table, saving a small amount of memory. `const` should not be used if the table may be modified by a platform-specific initialization routine.

Initialization

There is a choice of three main initialization functions. The simplest is `cyg_flash_devfn_init_nop`, which does nothing. It can be used if the cyg_strata_dev and cyg_flash_dev structures are fully initialized statically and the flash will just work without special effort. This is useful if it is guaranteed that the board will always be manufactured using the same flash chip, since the nop function involves the smallest code size and run-time overheads.

The next step up is `cyg_strata_init_check_devid_XX`, where `XX` will be replaced by the suffix appropriate for the bus configuration. It is still necessary to provide all the device information statically, including the *`devid`* field in the cyg_strata_dev structure. However this initialization function will attempt to query the flash device and check that the provided manufacturer and device codes matches the actual hardware. If there is a mismatch the device will be marked uninitialized and subsequent attempts to manipulate the flash will fail.

If the board may end up being manufactured with any of a number of different flash chips then the driver can perform run-time initialization, using a `cyg_strata_init_cfi_XX` function. This queries the flash device as per the Common Flash Memory

Interface Specification, supported by all current devices (although not necessarily by older devices). The *block_info* field in the cyg_strata_dev structure and the *end* and *num_block_infos* fields in the cyg_flash_dev structure will be filled in. It is still necessary to supply the *start* field statically since otherwise the driver will not know how to access the flash device. The main disadvantage of using CFI is that it will increase the code size.

A final option is to use a platform-specific initialization function. This may be useful if the board may be manufactured with one of a small number of different flash devices and the platform HAL needs to adapt to this. The Strata driver provides a utility function to read the device id, `cyg_strata_read_devid_XX`:

```
static int
kikoo_flash_init(struct cyg_flash_dev* dev)
{
    int manufacturer_code, device_code;
    cyg_strata_read_devid_1616(dev, &manufacturer_code, &device_code);
    if (manufacturer_code != CYG_FLASH_STRATA_MANUFACTURER_STMICRO) {
        return CYG_FLASH_ERR_DRV_WRONG_PART;
    }
    switch(device_code) {
        case 0x0042 :
          …
        case 0x0084 :
          …
        default:
          return CYG_FLASH_ERR_DRV_WRONG_PART;
    }
}
```

There are many other possible uses for a platform-specific initialization function. For example initial prototype boards might have only supported 8-bit access to a 16-bit flash device rather than 16-bit access, but this was fixed in the next revision. The platform-specific initialization function could figure out which model board it is running on and replace the default `16as8` functions with `16` ones.

Erase and Program

The Strata driver provides erase and program functions appropriate for the various bus configurations. On most targets these can be used directly. On some targets it may be necessary to do some extra work before and after the erase and program operations. For example if the hardware has an MMU then the part of the address map containing the flash may have been set to read-only, in an attempt to catch spurious memory accesses. Erasing or programming the flash requires write-access, so the MMU settings have to be changed temporarily. For another example some flash device may require a higher voltage to be applied during an erase or program operation. or a higher voltage may be desirable to make the operation proceed faster. A typical platform-specific erase function would look like this:

```
static int
kikoo_flash_erase(struct cyg_flash_dev* dev, cyg_flashaddr_t addr)
{
    int result;
    …  // Set up the hardware for an erase
    result = cyg_strata_erase_32(dev, addr);
    …  // Revert the hardware change
    return result;
}
```

There are two versions of the program function. `cyg_strata_bufprogram_xx` uses the buffered write capability of some strata chips. This allows the flash chip to perform the writes in parallel, thus greatly improving performance. It requires that the *bufsize* field of the cyg_strata_dev structure is set correctly to the number of words in the write buffer. The usual value for this is 16, corresponding to a 32-byte write buffer. The alternative `cyg_strata_program_xx` writes the data one word at a time so is significantly slower. It should be used only with strata chips that do not support buffered writes, for example the b3 and c3 series.

There are two configuration options which affect the erase and program functions, and which a platform HAL may wish to change: `CYGNUM_DEVS_FLASH_STRATA_V2_ERASE_TIMEOUT` and `CYGNUM_DEVS_FLASH_STRATA_V2_PROGRAM_TIME-`

OUT. The erase and program operations both involve polling for completion, and these timeout impose an upper bound on the polling loop. Normally these operations should never take anywhere close to the timeout period, and hence a timeout probably indicates a catastrophic failure that should really be handled by a watchdog reset. A reset is particularly appropriate because there will be no clean way of aborting the flash operation. The main reason for the timeouts is to help with debugging when porting to new hardware. If there is a valid reason why a particular platform needs different timeouts then the platform HAL's CDL can require appropriate values for these options.

Locking

Current Strata devices implement locking in three different ways, requiring different sets of functions:

28FxxxB3 There is no software locking support. The `cyg_flash_devfn_lock_nop` and `cyg_flash_devfn_unlock_nop` functions should be used.

28FxxxC3
28FxxxK3 These support locking and unlocking individual blocks. The `cyg_strata_lock_k3_XX` and `cyg_strata_unlock_k3_XX` functions should be used. All blocks are locked following power-up or reset, so the unlock function must be used before any erase or program operation. Theoretically the lock function is optional and `cyg_flash_devfn_lock_nop` can be used instead, saving a small amount of code space.

28FxxxJ3 Individual blocks can be locked using `cyg_strata_lock_j3_XX`, albeit using a slightly different algorithm from the C3 and K3 series. However the only unlock support is a global unlock of all blocks. Hence the only way to unlock a single block is to check the locked status of every block, unlock them all, and relock the ones that should still be locked. This time-consuming operation is implemented by `cyg_strata_unlock_j3_XX`. Worse, unlocking all blocks can take approximately a second. During this time the flash is unusable so normally interrupts have to be disabled, affecting real-time responsiveness. There is no way of suspending this operation.

Unlike the C3 and K3 chips, on a J3 blocks are not automatically locked following power-up or reset. Hence lock and unlock support is optional, and `cyg_flash_devfn_lock_nop` and `cyg_flash_devfn_unlock_nop` can be used.

If real locking functions are used then the platform HAL's CDL script should implement the CDL interface `CYGHWR_IO_FLASH_BLOCK_LOCKING`. Otherwise the generic flash package may believe that none of the flash drivers in the system provide locking functionality and disable the interface functions.

Device-Specific Structure

The cyg_strata_dev structure provides information specific to Strata flash devices, as opposed to the more generic flash information which goes into the cyg_flash_dev structure. There are only two fields: *`devid`* and *`block_info`*.

`manufacturer_code` and *`device_code`* are needed only if the driver's initialization function is set to `cyg_strata_init_check_devid_XX`. That function will extract the actual device info from the flash chip and compare it with these fields. If there is a mismatch then subsequent operations on the device will fail. Definitions of `CYG_FLASH_STRATA_MANUFACTURER_INTEL` and `CYG_FLASH_STRATA_MANUFACTURER_STMICRO` are provided for convenience.

The *`bufsize`* field is needed only if a buffered program function `cyg_strata_bufprogram_XX` is used. It should give the size of the buffer in words. Typically Strata devices have a 32-byte buffer, so when attached to an 8-bit bus *`bufsize`* should be 32 and when attached to a 16-bit bus it should be 16.

The *`block_info`* field consists of one or more pairs of the block size in bytes and the number of blocks of that size. The order must match the actual hardware device since the flash code will use the table to determine the start and end locations of each block. The table can be initialized in one of three ways:

1. If the driver initialization function is set to `cyg_strata_init_nop` or `cyg_strata_init_check_devid_XX` then the block information should be provided statically. This is appropriate if the board will also be manufactured using the same flash chip.

2. If cyg_strata_init_cfi_XX is used then this will fill in the block info table. Hence there is no need for static initialization.
3. If a platform-specific initialization function is used then either this should fill in the block info table, or the info should be provided statically.

The size of the *block_info* table is determined by the configuration option CYGNUM_DEVS_FLASH_STRATA_V2_ERASE_REGIONS. This has a default value of 2, which should suffice for nearly all Strata flash devices. If more entries are needed then the platform HAL's CDL script should require a larger value.

If the cyg_strata_dev structure is statically initialized then it can be const. This saves a small amount of memory in ROM startup applications. If the structure may be updated at run-time, either by cyg_strata_init_cfi_XX or by a platform-specific initialization routine, then it cannot be const.

Flash Structure

Internally the flash code works in terms of cyg_flash_dev structures, and the platform HAL should define one of these. The structure should be placed in the cyg_flashdev table. The following fields need to be provided:

funs
This should point at the table of functions.

start
The base address of the flash in the address map. On some board the flash may be mapped into memory several times, for example it may appear in both cached and uncached parts of the address space. The *start* field should correspond to the cached address.

end
The address of the last byte in the flash. It can either be statically initialized, or cyg_strata_init_cfi_XX will calculate its value at run-time.

num_block_infos
This should be the number of entries in the *block_info* table. It can either be statically initialized or it will be filled in by cyg_strata_init_cfi_XX.

block_info
The table with the block information is held in the cyg_strata_dev structure, so this field should just point into that structure.

priv
This field is reserved for use by the device driver. For the Strata driver it should point at the appropriate cyg_strata_dev structure.

The cyg_flash_dev structure contains a number of other fields which are manipulated only by the generic flash code. Some of these fields will be updated at run-time so the structure cannot be declared const.

Multiple Devices

A board may have several flash devices in parallel, for example two 16-bit devices on a 32-bit bus. It may also have several such banks to increase the total amount of flash. If each device provides 2MB, there could be one bank of 2 parallel flash devices at 0xFF800000 and another bank at 0xFFC00000, giving a total of 8MB. This setup can be described in several ways. One approach is to define two cyg_flash_dev structures. The table of function pointers can usually be shared, as can the cyg_strata_dev structure. Another approach is to define a single cyg_flash_dev structure but with a larger *block_info* table, covering the blocks in both banks of devices. The second approach makes more efficient use of memory.

Many variations are possible, for example a small slow flash device may be used for initial bootstrap and holding the configuration data, while there is also a much larger and faster device to hold a file system. Such variations are usually best described by separate cyg_flash_dev structures.

If more than one cyg_flash_dev structure is instantiated then the platform HAL's CDL script should implement the CDL interface CYGHWR_IO_FLASH_DEVICE once for every device past the first. Otherwise the generic code may default to the case of a single flash device and optimize for that.

Platform-Specific Macros

The Strata driver source code includes the header files cyg/hal/hal_arch.h and cyg/hal/hal_io.h, and hence indirectly the corresponding platform header files (if defined). Optionally these headers can define macros which are used inside the driver, thus giving the HAL limited control over how the driver works.

Cache Management

By default the strata driver assumes that the flash can be accessed uncached, and it will use the HAL CYGARC_UNCACHED_ADDRESS macro to map the cached address in the *start* field of the cyg_flash_dev structure into an uncached address. If for any reason this HAL macro is inappropriate for the flash then an alternative macro HAL_STRATA_UNCACHED_ADDRESS can be defined instead. However fixing the CYGARC_UNCACHED_ADDRESS macro is normally the better solution.

If there is no way of bypassing the cache then the platform HAL should implement the CDL interface CYGHWR_DEVS_FLASH_STRATA_V2_CACHED_ONLY. The flash driver will now disable and re-enable the cache as required. For example a program operation will involve the following:

```
STRATA_INTSCACHE_STATE;
STRATA_INTSCACHE_BEGIN();
while ( ! finished ) {
    write a burst of CYGNUM_DEVS_FLASH_STRATA_V2_PROGRAM_BURST_SIZE
    STRATA_INTSCACHE_SUSPEND();
    STRATA_INTSCACHE_RESUME();
}
STRATA_INTSCACHE_END();
```

The default implementations of these INTSCACHE macros are as follows: STATE defines any local variables that may be needed, e.g. to save the current interrupt state; BEGIN disables interrupts, synchronizes the data caches, disables it, and invalidates the current contents; SUSPEND re-enables the data cache and then interrupts; RESUME disables interrupts and the data cache; END re-enables the cache and then interrupts. The cache is only disabled when interrupts are disabled, so there is no possibility of an interrupt handler running or a context switch occurring while the cache is disabled, potentially leaving the system running very slowly. The data cache synchronization ensures that there are no dirty cache lines, so when the cache is disabled the low-level flash write code will not see stale data in memory. The invalidate ensures that at the end of the operation higher-level code will not pick up stale cache contents instead of the newly written flash data. The SUSPEND and RESUME macros only re-enable and disable the data cache. An interrupt and possibly a context switch may occur between these macros and use the cache normally. It is assumed that any code which runs at this time will not touch the memory being used by the flash operation, so as far as the low-level program code is concerned it can just continue to use the uncached memory contents as set up by the BEGIN macro. If any code modifies the const data currently being written to a flash block or tries to read the flash block being modified then the system's behaviour is undefined. Theoretically a more robust approach is possible, synchronizing and invalidating the cache again in every RESUME. However these cache operations can be expensive and RESUME may get invoked some thousands of times for every flash block, so this alternative approach would cripple the driver's performance.

Some implementations of the HAL cache macros may not provide the exact semantics required by the flash driver. For example HAL_DCACHE_DISABLE may have an unwanted side effect, or it may do more work than is needed here. The driver will check for alternative macros HAL_STRATA_INTSCACHE_STATE, HAL_STRATA_INTSCACHE_BEGIN, HAL_STRATA_INTSCACHE_SUSPEND, HAL_STRATA_INTSCACHE_RESUME and HAL_STRATA_INTSCACHE_END, using these instead of the defaults.

Polling Support

On some platforms it may be necessary to perform some additional action in the middle of a lengthy erase or program operation. For example, consider a platform with a watchdog device that cannot be disabled: when running in a polled environment such as RedBoot the flash operation run will run to completion, and there will be no opportunity for higher-level code to reset the watchdog; if the flash operation takes long enough the watchdog will trigger. To avoid problems in such scenarios, the platform HAL can define a macro HAL_STRATA_POLL. The macro is optional: if absent then the driver will automatically substitute a no-op.

Name

Strata — driver-specific functions

Synopsis

#include <cyg/io/strata_dev.h>

`void cyg_strata_read_devid_XX(device, manufacturer, device);`

`int cyg_strata_unlock_all_j3_XX(device);`

Description

The driver provides two sets of functions specific to Strata devices and not accessible via the standard eCos flash API. Both may be used safely before the flash subsystem is initialized using `cyg_flash_init`.

`cyg_strata_read_devid_XX` can be used to get the manufacturer and device codes. Typically it is called from a platform-specific driver initialization routine, allowing the platform HAL to adapt to the actual device present on the board. This may be useful if a board may get manufactured with several different and somewhat incompatible chips, although usually `cyg_strata_init_cfi` is the better approach. It may also be used during testing and porting to check that the chip is working correctly.

`cyg_strata_unlock_all_j3_XX` is only useful with 28FxxxJ3 chips and compatibles. These do not allow individual blocks to be unlocked. Hence the standard block unlock functionality is expensive: it requires checking the locked state of every block, unlocking every block, and then relocking all the blocks that should still be blocked. Worse, unlocking every block is a time-consuming operation, taking approximately a second, that needs to run with interrupts disabled. For many applications it is better to just ignore the chip's locking capabilities and run with all blocks permanently unlocked. Invoking `cyg_strata_unlock_all_j3_XX` during manufacture or when the board is commissioned achieves this.

Chapter 70. SST 39VFXXX Flash Device Driver

Name

Overview — eCos Support for SST 39VFXXX Flash Devices and Compatibles

Description

The CYGPKG_DEVS_FLASH_SST_39VFXXX_V2 SST 39VFXXX V2 flash driver package implements support for the 39VFXXX family of flash devices and compatibles. Normally the driver is not accessed directly. Instead application code will use the API provided by the generic flash driver package CYGPKG_IO_FLASH, for example by calling functions like cyg_flash_program.

The driver imposes one restriction on application code which developers should be aware of: when programming the flash the destination addresses must be aligned to a bus boundary. For example if the target hardware has a single flash device attached to a 16-bit bus then program operations must involve a multiple of 16-bit values aligned to a 16-bit boundary. Note that it is the bus width that matters, not the device width. If the target hardware has two 16-bit devices attached to a 32-bit bus then program operations must still be aligned to a 32-bit boundary, even though in theory a 16-bit boundary would suffice. In practice this is rarely an issue, and requiring the larger boundary greatly simplifies the code and improves performance.

Note

Many eCos targets with 39vfxxx or compatible flash devices will still use the older driver package CYGPKG_DEVS_FLASH_SST_39VFXXX. Only newer ports and some older ports that have been converted will use the V2 driver. This documentation only applies to the V2 driver.

Configuration Options

The 39vfxxx flash driver package will be loaded automatically when configuring eCos for a target with suitable hardware. However the driver will be inactive unless the generic flash package CYGPKG_IO_FLASH is loaded. It may be necessary to add this generic package to the configuration explicitly before the driver functionality becomes available. There should never be any need to load or unload the AM29xxxx driver package.

The driver contains a small number of configuration options which application developers may wish to tweak. CYGNUM_DEVS_FLASH_SST_39VFXXX_V2_PROGRAM_BURST_SIZE controls the program operation. On typical hardware programming the flash requires disabling interrupts and the cache for an extended period of time. Some or all of the flash hardware will be unusable while each word is programmed, and disabling interrupts is the only reliable way of ensuring that no interrupt handler or other thread will try to access the flash in the middle of an operation. This can have a major impact on the real-time responsiveness of the typical applications. To ameliorate this the driver wil perform writes in small bursts, briefly re-enabling the cache and interrupts between each burst. The number of words written per burst is controlled by this configuration operation: reducing the value will improve real-time response but will add overhead, so the actual flash program operation will take longer; conversely more writes per burst will worsen response times but reduce overhead.

Similarly erasing a block of flash safely requires disabling interrupts and the cache. Erasing a block can easily take a second or so, and disabling interrupts for such a long period of time is usually undesirable. Hence the driver can also perform the erase in bursts, using the hardware's suspend and resume capabilities. CYGNUM_DEVS_FLASH_STRATA_V2_ERASE_BURST_DURATION controls the number of polling loops during which interrupts are disabled. Reducing its value improves responsiveness at the cost of performance, and increasing its value has the opposite effect. Note that too low a value may prevent the erase operation from working at all: the chip will be spending its time suspending and resuming, rather than actually performing the erase. The minimum value will depend on the specific hardware.

There are a number of other options, relating mostly to hardware characteristics. It is very rare that application developers need to change any of these. For example the option CYGNUM_DEVS_FLASH_SST_39VFXXX_V2_ERASE_REGIONS may need a non-default value if the flash devices used on the target have an unusual boot block layout. If so the platform HAL will impose a requires constraint on this option and the configuration system will resolve the constraint. The only time it might be necessary to change the value manually is if the actual board being used is a variant of the one supported by the platform HAL and uses a different flash chip.

Name

Instantiating — including the driver in an eCos target

Synopsis

#include <cyg/io/39vfxxx_dev.h>

int **cyg_39vfxxx_init_check_devid_XX**(*device*);

int **cyg_39vfxxx_init_cfi_XX**(*device*);

int **cyg_39vfxxx_erase_XX**(*device*, *addr*);

int **cyg_39vfxxx_program_XX**(*device*, *addr*, *data*, *len*);

int **cyg_at49xxxx_softlock**(*device*, *addr*);

int **cyg_at49xxxx_hardlock**(*device*, *addr*);

int **cyg_at49xxxx_unlock**(*device*, *addr*);

int **cyg_39vfxxx_read_devid_XX**(*device*);

Description

The 39VFXXX family contains several different flash devices, all supporting the same basic set of operations but with various common or uncommon extensions. The devices vary in capacity, performance, boot block layout, and width. There are also platform-specific issues such as how many devices are actually present on the board and where they are mapped in the address space. The 39vfxxx driver package cannot know the details of every chip and every platform. Instead it is the responsibility of another package, usually the platform HAL, to supply the necessary information by instantiating some data structures. Two pieces of information are especially important: the bus configuration and the boot block layout.

Flash devices are typically 8-bits, 16-bits, or 32-bits wide (64-bit devices are not yet in common use). Most 16-bit devices will also support 8-bit accesses, but not all. Similarly 32-bit devices can be accessed 16-bits at a time or 8-bits at a time. A board will have one or more of these devices on the bus. For example there may be a single 16-bit device on a 16-bit bus, or two 16-bit devices on a 32-bit bus. The processor's bus logic determines which combinations are possible, and there will be a trade off between cost and performance: two 16-bit devices in parallel can provide twice the memory bandwidth of a single device. The driver supports the following combinations:

8 A single 8-bit flash device on an 8-bit bus.

16 A single 16-bit flash device on a 16-bit bus.

32 A single 32-bit flash device on an 32-bit bus.

88 Two parallel 8-bit devices on an 16-bit bus.

8888 Four parallel 8-bit devices on a 32-bit bus.

1616 Two parallel 16-bit devices on a 32-bit bus, with one device providing the bottom two bytes of each 32-bit datum and the other device providing the top two bytes.

16as8 A single 16-bit flash device connected to an 8-bit bus.

These configuration all require slightly different code to manipulate the hardware. The 39vfxxx driver package provides separate functions for each configuration, for example `cyg_39vfxxx_erase_16` and `cyg_39vfxxx_program_1616`.

Caution

At the time of writing not all the configurations have been tested.

The second piece of information is the boot block layout. Flash devices are subdivided into blocks (also known as sectors - both terms are in common use). Some operations such as erase work on a whole block at a time, and for most applications a block is the smallest unit that gets updated. A typical block size is 64K. It is inefficient to use an entire 64K block for small bits of configuration data and similar information, so many flash devices also support a number of smaller boot blocks. A typical 2MB flash device could have a single 16K block, followed by two 8K blocks, then a 32K block, and finally 31 full-size 64K blocks. The boot blocks may appear at the bottom or the top of the device. So-called uniform devices do not have boot blocks, just full-size ones. The driver needs to know the boot block layout. With modern devices it can work this out at run-time, but often it is better to provide the information statically.

Example

In most cases flash support is specific to a platform. Even if two platforms happen to use the same flash device there are likely to be differences such as the location in the address map. Hence there is little possibility of re-using the platform-specific code, and this code should be placed in the platform HAL rather than in a separate package. Typically this involves a separate file and a corresponding compile property in the platform HAL's CDL:

```
cdl_package CYGPKG_HAL_M68K_ALAIA {
    …
    compile -library=libextras.a alaia_flash.c
    …
}
```

The contents of this file will not be accessed directly, only indirectly via the generic flash API, so normally it would be removed by link-time garbage collection. To avoid this the object file has to go into `libextras.a`.

The actual file `alaia_flash.c` will look something like:

```
#include <pkgconf/system.h>
#ifdef CYGPKG_DEVS_FLASH_SST_39VFXXX_V2

#include <cyg/io/flash.h>
#include <cyg/io/flash_dev.h>
#include <cyg/io/39vfxxx_dev.h>

static const CYG_FLASH_FUNS(hal_alaia_flash_amd_funs,
    &cyg_39vfxxx_init_check_devid_16,
    &cyg_flash_devfn_query_nop,
    &cyg_39vfxxx_erase_16,
    &cyg_39vfxxx_program_16,
    (int (*)(struct cyg_flash_dev*, const cyg_flashaddr_t, void*, size_t))0,
    &cyg_flash_devfn_lock_nop,
    &cyg_flash_devfn_unlock_nop);

static const cyg_39vfxxx_dev hal_alaia_flash_priv = {
    .devid      = 0x45,
    .block_info = {
        { 0x00004000, 1 },
        { 0x00002000, 2 },
        { 0x00008000, 1 },
        { 0x00010000, 63 }
    }
};

CYG_FLASH_DRIVER(hal_alaia_flash,
                 &hal_alaia_flash_amd_funs,
                 0,
                 0xFFC00000,
                 0xFFFFFFFF,
                 4,
                 hal_alaia_flash_priv.block_info,
                 &hal_alaia_flash_priv
);
#endif
```

The bulk of the file is protected by an `#ifdef` for the 39vfxxx flash driver. That driver will only be active if the generic flash support is enabled. Without that support there will be no way of accessing the device so instantiating the data structures would serve no purpose. The rest of the file is split into three structure definitions. The first supplies the functions which will be used to perform the actual flash accesses, using a macro provided by the generic flash code in `cyg/io/flash_dev.h`. The relevant ones have an `_16` suffix, indicating that on this board there is a single 16-bit flash device on a 16-bit bus. The second provides information specific to 39vfxxx flash devices. The third provides the cyg_flash_dev structure needed by the generic flash code, which contains pointers to the previous two.

Functions

All eCos flash device drivers must implement a standard interface, defined by the generic flash code `CYGPKG_IO_FLASH`. This interface includes a table of seven function pointers for various operations: initialization, query, erase, program, read, locking and unlocking. The query operation is optional and the generic flash support provides a dummy implementation `cyg_flash_devfn_query_nop`. 39vfxxx flash devices are always directly accessible so there is no need for a separate read function. The remaining functions are more complicated.

Usually the table can be declared `const`. In a ROM startup application this avoids both ROM and RAM copies of the table, saving a small amount of memory. `const` should not be used if the table may be modified by a platform-specific initialization routine.

Initialization

There is a choice of three main initialization functions. The simplest is `cyg_flash_devfn_init_nop`, which does nothing. It can be used if the cyg_39vfxxx_dev and cyg_flash_dev structures are fully initialized statically and the flash will just work without special effort. This is useful if it is guaranteed that the board will always be manufactured using the same flash chip, since the nop function involves the smallest code size and run-time overheads.

The next step up is `cyg_39vfxxx_init_check_devid_XX`, where `XX` will be replaced by the suffix appropriate for the bus configuration. It is still necessary to provide all the device information statically, including the *`devid`* field in the cyg_39vfxxx_dev structure. This initialization function will attempt to query the flash device and check that the provided device id matches the actual hardware. If there is a mismatch the device will be marked uninitialized and subsequent attempts to manipulate the flash will fail.

If the board may end up being manufactured with any of a number of different flash chips then the driver can perform run-time initialization, using a `cyg_39vfxxx_init_cfi_XX` function. This queries the flash device as per the Common Flash Memory Interface Specification, supported by all current devices (although not necessarily by older devices). The *`block_info`* field in the cyg_39vfxxx_dev structure and the *`end`* and *`num_block_infos`* fields in the cyg_flash_dev structure will be filled in. It is still necessary to supply the *`start`* field statically since otherwise the driver will not know how to access the flash device. The main disadvantage of using CFI is that it increases the code size.

Caution

If CFI is used then the cyg_39vfxxx_dev structure must not be declared `const`. The CFI code will attempt to update the structure and will fail if the structure is held in read-only memory. This would leave the flash driver non-functional.

A final option is to use a platform-specific initialization function. This may be useful if the board may be manufactured with one of a small number of different flash devices and the platform HAL needs to adapt to this. The 39vfxxx driver provides a utility function to read the device id, `cyg_39vfxxx_read_devid_XX`:

```
static int
alaia_flash_init(struct cyg_flash_dev* dev)
{
    int devid = cyg_39vfxxx_read_devid_1616(dev);
    switch(devid) {
        case 0x0042 :
          …
        case 0x0084 :
          …
        default:
          return CYG_FLASH_ERR_DRV_WRONG_PART;
    }
}
```

There are many other possible uses for a platform-specific initialization function. For example initial prototype boards might have only supported 8-bit access to a 16-bit flash device rather than 16-bit access, but this problem was fixed in the next revision. The platform-specific initialization function can figure out which model board it is running on and replace the default `16as8` functions with faster `16` ones.

Erase and Program

The 39vfxxx driver provides erase and program functions appropriate for the various bus configurations. On most targets these can be used directly. On some targets it may be necessary to do some extra work before and after the erase and program operations. For example if the hardware has an MMU then the part of the address map containing the flash may have been set to read-only, in an attempt to catch spurious memory accesses. Erasing or programming the flash requires write-access, so the MMU settings have to be changed temporarily. As another example some flash device may require a higher voltage to be applied during an erase or program operation. or a higher voltage may be desirable to make the operation proceed faster. A typical platform-specific erase function would look like this:

```
static int
alaia_flash_erase(struct cyg_flash_dev* dev, cyg_flashaddr_t addr)
{
    int result;
    …  // Set up the hardware for an erase
    result = cyg_39vfxxx_erase_32(dev, addr);
    …  // Revert the hardware change
    return result;
}
```

There are two configurations which affect the erase and program functions, and which a platform HAL may wish to change: `CYGNUM_DEVS_FLASH_SST_39VFXXX_V2_ERASE_TIMEOUT` and `CYGNUM_DEVS_FLASH_SST_39VFXXX_V2_PROGRAM_TIMEOUT`. The erase and program operations both involve polling for completion, and these timeout impose an upper bound on the polling loop. Normally these operations should never take anywhere close to the timeout period, so a timeout indicates a catastrophic failure that should really be handled by a watchdog reset. A reset is particularly appropriate because there will be no clean way of aborting the flash operation. The main reason for the timeouts is to help with debugging when porting to new hardware. If there is a valid reason why a particular platform needs different timeouts then the platform HAL's CDL can require appropriate values for these options.

Locking

There is no single way of implementing the block lock and unlock operations on all 39vfxxx devices. If these operations are supported at all then usually they involve manipulating the voltages on certain pins. This would not be able to be handled by generic driver code since it requires knowing how these pins can be manipulated via the processor's GPIO lines. Therefore the 39vfxxx driver does not usually provide lock and unlock functions, and instead the generic dummy functions `cyg_flash_devfn_lock_nop` and `cyg_flash_devfn_unlock_nop` should be used. An exception exists for the AT49xxxx family of devices which are sufficiently SST compatible in other respects. Otherwise, if a platform does provide a way of implementing the locking then this can be handled by platform-specific functions.

```
static int
alaia_lock(struct cyg_flash_dev* dev, const cyg_flashaddr_t addr)
{
    …
}
```

```
static int
alaia_unlock(struct cyg_flash_dev* dev, const cyg_flashaddr_t addr)
{
    …
}
```

If real locking functions are implemented then the platform HAL's CDL script should implement the CDL interface CYGHWR_IO_FLASH_BLOCK_LOCKING. Otherwise the generic flash package may believe that none of the flash drivers in the system provide locking functionality and disable the interface functions.

AT49xxxx locking

As locking is standardised on across the AT49xxxx family of SST 39vfxxx compatible Flash parts, a method is supporting this is included within this driver. cyg_at49xxxx_softlock_XX provides a means of locking a Flash sector such that it may be subsequently unlocked. cyg_at49xxxx_hardlock_XX locks a sector such that it cannot be unlocked until after reset or a power cycle. cyg_at49xxxx_unlock_XX unlocks a sector that has previously been softlocked. At power on or Flash device reset, all sectors default to being softlocked.

Other

The driver provides a set of functions cyg_39vfxxx_read_devid_XX, one per supported bus configuration. These functions take a single argument, a pointer to the cyg_flash_dev structure, and return the chip's device id. For older devices this id is a single byte. For more recent devices the id is a 3-byte value, 0x7E followed by a further two bytes that actually identify the device. cyg_39vfxxx_read_devid_XX is usually called only from inside a platform-specific driver initialization routine, allowing the platform HAL to adapt to the actual device present on the board.

Device-Specific Structure

The cyg_39vfxxx_dev structure provides information specific to 39vfxxx flash devices, as opposed to the more generic flash information which goes into the cyg_flash_dev structure. There are only two fields: *devid* and *block_info*.

devid is only needed if the driver's initialization function is set to cyg_39vfxxx_init_check_devid_XX. That function will extract the actual device info from the flash chip and compare it with the *devid* field. If there is a mismatch then subsequent operations on the device will fail.

The *block_info* field consists of one or more pairs of the block size in bytes and the number of blocks of that size. The order must match the actual hardware device since the flash code will use the table to determine the start and end locations of each block. The table can be initialized in one of three ways:

1. If the driver initialization function is set to cyg_flash_devfn_init_nop or cyg_39vfxxx_init_check_devid_XX then the block information should be provided statically. This is appropriate if the board will also be manufactured using the same flash chip.

2. If cyg_39vfxxx_init_cfi_XX is used then this will fill in the block info table. Hence there is no need for static initialization.

3. If a platform-specific initialization function is used then either this should fill in the block info table, or the info should be provided statically.

The size of the *block_info* table is determined by the configuration option CYGNUM_DEVS_FLASH_SST_39VFXXX_V2_ERASE_REGIONS. This has a default value of 4, which should suffice for nearly all 39vfxxx flash devices. If more entries are needed then the platform HAL's CDL script should require a larger value.

If the cyg_39vfxxx_dev structure is statically initialized then it can be const. This saves a small amount of memory in ROM startup applications. If the structure is updated at run-time, either by cyg_39vfxxx_init_cfi_XX or by a platform-specific initialization routine, then it cannot be const.

Flash Structure

Internally the generic flash code works in terms of cyg_flash_dev structures, and the platform HAL should define one of these. The structure should be placed in the `cyg_flashdev` table. The following fields need to be provided:

funs
: This should point at the table of functions.

start
: The base address of the flash in the address map. On some board the flash may be mapped into memory several times, for example it may appear in both cached and uncached parts of the address space. The *start* field should correspond to the cached address.

end
: The address of the last byte in the flash. It can either be statically initialized, or `cyg_39vfxxx_init_cfi_XX` will calculate its value at run-time.

num_block_infos
: This should be the number of entries in the *block_info* table. It can either be statically initialized or it will be filled in by `cyg_39vfxxx_init_cfi_XX`.

block_info
: The table with the block information is held in the cyg_39vfxxx_dev structure, so this field should just point into that structure.

priv
: This field is reserved for use by the device driver. For the 39vfxxx driver it should point at the appropriate cyg_39vfxxx_dev structure.

The cyg_flash_dev structure contains a number of other fields which are manipulated only by the generic flash code. Some of these fields will be updated at run-time so the structure cannot be declared `const`.

Multiple Devices

A board may have several flash devices in parallel, for example two 16-bit devices on a 32-bit bus. It may also have several such banks to increase the total amount of flash. If each device provides 2MB, there could be one bank of 2 parallel flash devices at 0xFF800000 and another bank at 0xFFC00000, giving a total of 8MB. This setup can be described in several ways. One approach is to define two cyg_flash_dev structures. The table of function pointers can usually be shared, as can the cyg_39vfxxx_dev structure. Another approach is to define a single cyg_flash_dev structure but with a larger *block_info* table, covering the blocks in both banks of devices. The second approach makes more efficient use of memory.

Many variations are possible, for example a small slow flash device may be used for initial bootstrap and holding the configuration data, while there is also a much larger and faster device to hold a file system. Such variations are usually best described by separate cyg_flash_dev structures.

If more than one cyg_flash_dev structure is instantiated then the platform HAL's CDL script should implement the CDL interface `CYGHWR_IO_FLASH_DEVICE` once for every device past the first. Otherwise the generic code may default to the case of a single flash device and optimize for that.

Platform-Specific Macros

The 39vfxxx driver source code includes the header files `cyg/hal/hal_arch.h` and `cyg/hal/hal_io.h`, and hence indirectly the corresponding platform header files (if defined). Optionally these headers can define macros which are used inside the driver, thus giving the HAL limited control over how the driver works.

Cache Management

By default the 39vfxxx driver assumes that the flash can be accessed uncached, and it will use the HAL `CYGARC_UNCACHED_ADDRESS` macro to map the cached address in the *start* field of the cyg_flash_dev structure into an uncached address. If for any reason this HAL macro is inappropriate for the flash then an alternative macro HAL_39VFXXX_UNCACHED_ADDRESS can be defined instead. However fixing the CYGARC_UNCACHED_ADDRESS macro is normally the better solution.

If there is no way of bypassing the cache then the platform HAL should implement the CDL interface CYGHWR_DEVS_FLASH_SST_39VFXXX_V2_CACHED_ONLY. The flash driver will now disable and re-enable the cache as required. For example a program operation will involve the following:

```
AM29_INTSCACHE_STATE;
AM29_INTSCACHE_BEGIN();
while ( ! finished ) {
    write a burst of CYGNUM_DEVS_FLASH_SST_39VFXXX_V2_PROGRAM_BURST_SIZE
    AM29_INTSCACHE_SUSPEND();
    AM29_INTSCACHE_RESUME();
}
AM29_INTSCACHE_END();
```

The default implementations of these INTSCACHE macros are as follows: STATE defines any local variables that may be needed, e.g. to save the current interrupt state; BEGIN disables interrupts, synchronizes the data caches, disables it, and invalidates the current contents; SUSPEND re-enables the data cache and then interrupts; RESUME disables interrupts and the data cache; END re-enables the cache and then interrupts. The cache is only disabled when interrupts are disabled, so there is no possibility of an interrupt handler running or a context switch occurring while the cache is disabled, potentially leaving the system running very slowly. The data cache synchronization ensures that there are no dirty cache lines, so when the cache is disabled the low-level flash write code will not see stale data in memory. The invalidate ensures that at the end of the operation higher-level code will not pick up stale cache contents instead of the newly written flash data. The SUSPEND and RESUME macros only re-enable and disable the data cache. An interrupt and possibly a context switch may occur between these macros and use the cache normally. It is assumed that any code which runs at this time will not touch the memory being used by the flash operation, so as far as the low-level program code is concerned it can just continue to use the uncached memory contents as set up by the BEGIN macro. If any code modifies the const data currently being written to a flash block or tries to read the flash block being modified then the system's behaviour is undefined. Theoretically a more robust approach is possible, synchronizing and invalidating the cache again in every RESUME. However these cache operations can be expensive and RESUME may get invoked some thousands of times for every flash block, so this alternative approach would cripple the driver's performance.

Some implementations of the HAL cache macros may not provide the exact semantics required by the flash driver. For example HAL_DCACHE_DISABLE may have an unwanted side effect, or it may do more work than is needed here. The driver will check for alternative macros HAL_39VFXXX_INTSCACHE_STATE, HAL_39VFXXX_INTSCACHE_BEGIN, HAL_39VFXXX_INTSCACHE_SUSPEND, HAL_39VFXXX_INTSCACHE_RESUME and HAL_39VFXXX_INTSCACHE_END, using these instead of the defaults.

Part XIX. ecoflash Flash Programming Utility

Name

ecoflash — Flash Programming Utility

Synopsis

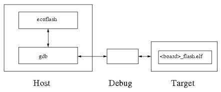
`ecoflash` ---help [subcommand]`ecoflash` [*(1) options*] boards `ecoflash` [*(1) options*] info `ecoflash` [*(1) options*] program [[-r] | [--raw]] [[-n] | [--no-erase]] { *`file`* } [*`address`*]`ecoflash` [*(1) options*] dump [[-a] | [--append]] { *`file`* } [*`address`*] [*`length`*]`ecoflash` [*(1) options*] erase { *`address`* } [*`length`*]`ecoflash` [*(1) options*] write [[-n] | [--no-erase]] [[-o *`offset`*] | [--offset=*`offset`*]] { *`file`* } { *`address`* } [*`length`*]`ecoflash` [*(1) options*] lock { *`address`* } [*`length`*]`ecoflash` [*(1) options*] unlock { *`address`* } [*`length`*]

(1)
[[-h] | [--help]]
[--version]
[--dry-run]
[[-q] | [--quiet]]
[[-v] | [--verbose]]
[[-b *`board`*] | [--board=*`board`*]]
[[-g *`gdb executable`*] | [--gdb=*`gdb executable`*]]
[[-o *`objcopy executable`*] | [--objcopy=*`objcopy executable`*]]
[[-t *`target command`*] | [--target=*`target command`*]]

Description

ecoflash is a utility for programming on-board flash via BDM, jtag, or similar hardware debug technology. There is nothing particularly original about this. Most manufacturers will provide similar utilities, and in fact those are likely to offer better performance because they operate at a lower level. The main advantages of ecoflash are: it provides a common user experience across a range of hardware; it is designed to work within a typical eCos development environment; and it has some built-in knowledge of how eCos systems work.

ecoflash works by running a suitable version of **gdb** on the host machine and running gdb commands. That version of gdb must either be able to drive the hardware debug technology directly, or more commonly it will in turn interact with some stub or daemon that knows how to drive the debug hardware. The target is initialized and a small target-side executable, for example `m5213evb_flash.elf`, is then loaded on to the target. The target-side executable is a simple eCos application that is linked with the eCos flash driver support, so it can be readily ported to any target for which a suitable flash driver is available. Manipulating the flash involves setting target-side variables used gdb commands, letting the target-side executable run until a breakpoint is hit, and then examining more target-side variables to determine the status.



The basic syntax is: ecoflash [standard options] subcommand [options] args [optional args]. The standard options provide information such as the target board, and most of these can be specified instead using environment variables. The subcommands specify the exact operation to be performed, for example to program an executable at the default location within the flash. Some subcommands take additional options, for example to suppress automatic erasure of flash blocks. These are followed by required arguments such as the executable filename, and possibly optional arguments. **ecoflash -h** with no additional argument provides help information for the utility as a whole. Additional information for a specific subcommand can be obtained using e.g. **ecoflash -h program**.

Standard Options

ecoflash accepts a number of standard options, for example the target board can be specified using `-b board`. These options will be used by several but not all of the subcommands.

-h
--help

Obtain help information about ecoflash or one of its subcommands.

--version

Display the ecoflash version string.

--dry-run

This suppresses the low-level block erase and write operations so the flash state does not actually change, but otherwise ecoflash operates normally including initializing the board, downloading the target-side executable, and having the latter initialize the eCos flash driver. It can be used to verify that the hardware is operating correctly.

-q
--quiet
-v
--verbose

These can be used to reduce or increase the amount of feedback generated by ecoflash. `-v` can be specified several times for increased verbosity.

-b <board>
--board=<board>

ecoflash needs to know which target board it should access, so that it can perform appropriate initialization and download the right target-side executable. **ecoflash boards** can be used to get a list of supported boards. It is possible to set an environment variable `ECOFLASH_BOARD` as an alternative to specifying this option on the command line every time:

```
$ export ECOFLASH_BOARD=m5213evb
```

The board name has two effects. It causes ecoflash to load a configuration file `<board>.ecf` with hardware-specific information, for example how to initialize the board using gdb commands. It also determines the target-side executable `<board>_flash.elf`. For both files ecoflash will first look in the current directory. If an `ECOFLASH_DIR` environment variable is defined then it will look in that directory. Finally it will look in the directory `../share/ecos/ecoflash` relative to the ecoflash executable.

-g <gdb executable>
--gdb=<gdb executable>

All subcommands except **boards** involve starting a gdb session on the host and downloading a target-side executable. Usually the gdb executable, for example **m68k-elf-gdb**, is specified in the board `.ecf` configuration file and found in the current `PATH`, so there is no need to use this option. However an alternative gdb can be specified if desired, for example when building and debugging an experimental version of gdb. The environment variable `ECOFLASH_GDB` can also be used instead of the command line option.

-o <objcopy executable>
--objcopy=<objcopy executable>

The **program** subcommand can take an eCos executable in ELF format and automatically convert it to raw binary to program into flash. This involves running the appropriate host-side objcopy utility, for example **m68k-elf-objcopy**. By default ecoflash will munge the gdb file name to generate the objcopy name and will look for it on the `PATH`, so there is no need to use this option. An alternative objcopy can be specified on the command line or via the `ECOFLASH_OBJCOPY` environment variable.

-t <target command>
--target=<target command

ecoflash needs to know how to get gdb to interact with the debug hardware. The exact details of this depend not just on the hardware but also on the developer's setup, so cannot be provided by the board `.ecf` configuration file. Instead it must be provided by the user, in the form of a gdb target command. For example, if the debug hardware is accessed via a daemon on the local machine and that daemon listens on TCP/IP port 9000 for gdb remote protocol traffic then the gdb command to connect to the target hardware would be: `target remote localhost:9000`. The ecoflash `-t` option should consist of everything after `target`, for example:

```
$ ecoflash -b alaia -t 'remote localhost:9000' info
```

Note the use of quote marks to make the shell treat it as a single argument, even though it contains spaces, and to prevent any processing of special characters like $ and |. The `ECOFLASH_TARGET` environment variable can be set instead:

```
$ export ECOFLASH_TARGET='remote localhost:9000'
```

Supported Boards

ecoflash boards can be used to get the names of the boards supported in the current installation, in other words what `-b` options are valid. A board is considered supported if ecoflash can find a `<board>.ecf` configuration file and a target-side executable `<board>_flash.elf`. It will search in the current directory, in the directory specified by the `ECOFLASH_DIR` environment variable if that is defined, and in a directory relative to the ecoflash executable. For example if ecoflash is installed in `/usr/local/ecos/bin` then it will search in `/usr/local/ecos/share/ecos/ecoflash`.

Note that **ecoflash boards** only examines the file system and does not attempt to start gdb or interact in any way with target hardware.

Board Information

ecoflash info can be used to get information about a particular board, and also to verify that everything is working correctly.

```
% ecoflash -b m5213evb -t 'remote localhost:9000' info
Target board is m5213evb.
  gdb is "m68k-elf-gdb", gdb target is "remote localhost:9000"
  Target-side executable is version 1.
  Detected 1 bank of flash.
    Start 0x00000000, end 0x0003ffff -> 256K.
      128 blocks of 2K.
  Flash block locking is not supported.
  Default program location for executables is 0x00000000.
  Target-side buffer for read and write operation is 16K.
```

This shows the gdb command and the target string, which can be useful if that information comes from environment variables rather than the command line. ecoflash will run gdb and download the target-side executable, which reports itself as version 1. The target-side executable initializes the eCos flash driver and has detected the amount of flash reported and that lock and unlock operations are not supported. By default executables will be programmed at location 0x0, and transfers between host and target use a 16k buffer (the M5213EVB only has a small amount of RAM, usually a larger buffer will be used).

Programming an Executable

ecoflash program can be used to install an eCos executable at the boot location within the flash. The executable should be linked against an eCos configuration with a suitable startup type, usually ROM or ROMRAM although this may vary between platforms. In its simplest form the subcommand just takes a single argument, a filename for the executable:

```
$ ecoflash program redboot.elf
Erasing 0x00000000 - 0x0000be57
Writing 0x00000000 - 0x00003fff (16384 bytes) from file "/tmp/redboot.1495", offset 0
Writing 0x00004000 - 0x00007fff (16384 bytes) from file "/tmp/redboot.1495", offset 16384
Writing 0x00008000 - 0x0000be57 (15960 bytes) from file "/tmp/redboot.1495", offset 32768
```

This assumes the `ECOFLASH_BOARD` and `ECOFLASH_TARGET` environment variables are set. ecoflash will examine the specified file. ELF executables will be automatically converted to a temporary raw binary file before being programmed into flash, using the objcopy utility. The default address within the flash is supplied by the target-side executable. Usually this will be the processor's boot location but the exact boot mechanism varies widely between processors and platforms.

ecoflash program takes two options. `-r` or `--raw` can be used to suppress the detection of ELF format files. Instead the file will be treated as a raw binary and no conversion is performed. This may be useful if the board has its own primary bootloader which expects to find an ELF executable at a particular address within the flash. `-n` or `--no-erase` can be used to suppress the automatic erase of the flash blocks prior to programming the flash. This may be useful on hardware where flash erase is optional, or if ecoflash is used in a batch environment where a previous step will have already erased the required amount of flash.

ecoflash program takes an optional additional argument, an alternative address within the flash. This may be useful if for example the board can boot from one of two locations depending on the state of a jumper.

Dumping Flash Contents

ecoflash dump can be used to read part or all of the flash and dump the data to a file on the host. This can be particularly useful when saving a known working image prior to replacing it with an experimental version. The default behaviour is to dump the entire flash contents:

```
$ ecoflash dump /tmp/working.bin
Dumping 0x00000000 - 0x00003fff (16384 bytes) to file "/tmp/working.bin"
Dumping 0x00004000 - 0x00007fff (16384 bytes) to file "/tmp/working.bin"
…
```

Optionally the starting address and the length can be specified:

```
$ ecoflash dump /tmp/working.bin 0xFFF00000 128K
```

Lengths can be specified in bytes, kilobytes using a K suffix, megabytes using an M suffix, or flash blocks using a B suffix. Note that some flash devices have boot blocks of varying sizes so specifying a size in terms of blocks can be confusing.

ecoflash dump takes a single option, `-a` or `--append`. This causes ecoflash to append to the specified file instead of overwriting it.

Erasing Flash Blocks

ecoflash erase can be used to erase one or more flash blocks. This command is rarely needed since both the program and write subcommands will erase the required number of flash blocks by default, but may prove useful if the flash contains data other than an eCos executable and that data should be reset to uninitialized. In its simplest form the erase subcommand just takes an address:

```
$ ecoflash erase 0x40000
```

This causes ecoflash to erase the single flash block containing the specified address. Optionally a length can be specified, for example to erase 8 flash blocks:

```
$ ecoflash erase 0x40000 8B
```

The length can be specified in bytes, in kilobytes using a K suffix, in megabytes using an M suffix, or in flash blocks using a B suffix. Care must be taken if the specified address is not at the start of a flash block. For example if the address is 0x48000, the length is 128K, and flash blocks are 64K, then this is treated as a request to erase flash from 0x48000 to 0x67FFF. Since erase operations always involve whole flash blocks the actual erase affects all memory from 0x40000 to 0x6FFFF, so a total of 192K gets erased.

The erase subcommand does not have any options of its own, just the standard ones for all subcommands.

Writing Raw Data

ecoflash write can be used to write a raw data file to an arbitrary location within the flash. It is intended for installing additional data rather than the main executable, since the **program** subcommand is more appropriate for the latter. At least a filename and an address within the flash should be specified:

```
$ ecoflash write data.bin 0x00040000
Erasing 0x00040000 - 0x0004297c
Writing 0x00040000 - 0x0004297c (10621 bytes) from file "data.bin", offset 0
```

Optionally a length can be specified, for example:

```
$ ecoflash write data.bin 0x00040000 64K
```

This will write only the first 64K of data.bin rather than the whole file. The length can be specified in bytes, in kilobytes using a K suffix, in megabytes using an M suffix, or in flash blocks using a B suffix.

ecoflash write takes two options. `-n` or `--no-erase` can be used to suppress the automatic erase before the data is written to flash. This can be useful if a single flash block should contain data from more than one file: **ecoflash erase** would be used to erase the whole flash black, then two **ecoflash write -n** commands would program the two files at the appropriate locations; alternatively the erase step can be skipped in subsumed by the first write, with only the second write using a `-n` option.

`-o <offset>` or `-offset=<offset>` can be used to skip part of a file. For example the following writes 12K of a file starting at a 4K offset:

```
$ ecoflash write --offset=4096 data.bin 12K
```

The offset can be specified in bytes, kilobytes using a K suffix, or megabytes using an M suffix.

Locking and Unlocking

On targets where the hardware and the flash driver support locking, the **lock** and **unlock** subcommands can be used to manipulate the locked status of one or more flash blocks. Both subcommands take an address and an optional length:

```
$ ecoflash lock 0x40000
…
$ ecoflash unlock 0x50000 256K
```

If no length is specified then just a single flash block is affected, unless the hardware implements locking at a coarser grain than individual flash blocks. The length can be specified in bytes, in kilobytes using a K suffix, in megabytes using an M suffix, or in flash blocks using a B suffix.

With some flash hardware locking is not persistent. Instead the locks are set to a default state when the flash chips are powered up or reset, usually locked. On such hardware the ecoflash lock and unlock subcommands are of little use since the locks would revert to their default state when ecoflash exits. Instead the target-side executable will either unlock all flash blocks during initialization or take whatever action is needed at run-time to handle erase and write operations.

Installation

Depending on your eCos distribution ecoflash may already be installed on your system. If not, installation is straightforward. The host-side consists of a single executable `ecoflash` in the package's `host` subdirectory. This is actually a platform-independent script written in the expect scripting language. It needs to be installed in a suitable location on the user's search `PATH`. The file can just be copied manually, or alternatively the `host` subdirectory contains a suitable configure script and support files:

```
$ <package path>/host/configure --prefix=/usr/local
$ make
$ make install
```

This will install ecoflash in the directory `/usr/local/bin`. Note that eCos also has a top-level `configure` script which will find subsidiary `configure` scripts inside the individual packages. A top-level configure/make/make install sequence will automatically install ecoflash as well as host-side support from other packages.

The ecoflash package contains only the generic support. It should be complemented by a `.ecf` configuration file and a `_flash.elf` target-side executable for every supported platform. The platform HAL's `misc` subdirectory usually holds a suitable `.ecf` file. The target-side executable will need to be rebuilt:

```
$ ecosconfig new <target> minimal
$ ecosconfig import <path>/ecoflash.ecm
$ ecosconfig tree
$ make
```

For an existing port there should be an `ecoflash.ecm` file in the platform HAL's misc subdirectory. Importing this will add the ecoflash package and any necessary support packages, set any platform-specific configuration options, and resolve any conflicts. After the make there should be a file `install/bin/flash.elf`, the target-side executable, and this should get renamed to `<board>_flash.elf` and installed in a location where ecoflash will find it.

Porting

Typically the only hard part of porting ecoflash to a new platform is to get gdb to interact with the jtag or BDM hardware and initialize the board. The porting process involves three steps: adding appropriate definitions to the platform HAL; building the target-side executable `<board>_flash.elf`; and writing the platform configuration file `<board>.ecf`.

The platform HAL must supply a single #define'd symbol corresponding to the default base address for **ecoflash program** operations. Usually this symbol gets defined in `cyg/hal/plf_io.h`, but the details may vary between architectures.

```
#define HAL_ECOFLASH_PROGRAM_BASE        0x00000000
```

Optionally the platform HAL can define a buffer size using `HAL_ECOFLASH_BUFLEN`, an additional header file to include using `HAL_ECOFLASH_HEADER`, and an initialization macro `HAL_ECOFLASH_EXTRA_INIT()`. The latter may perform operations such as unlocking all flash blocks on hardware where locks are transient.

The target-side executable is a very simple eCos application that uses the generic flash driver support to interact with the the hardware. Hence it assumes that a suitable flash driver is already available. Code and data sizes are both of the order of 4K, although obviously that will depend on the processor architecture. Usually the code will be RAM-resident and linked with a JTAG or RAM startup configuration. In addition a buffer is needed for transferring data between host and target. By default that buffer is 64K, corresponding to typical flash block sizes, but can be smaller if there is not enough RAM for a buffer that size. Building the target-side executable is straightforward:

```
$ ecosconfig new <target> minimal
$ ecosconfig add CYGPKG_IO_FLASH CYGPKG_LIBC_STRING CYGPKG_ECOFLASH
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

It may be necessary to tweak the configuration data before generating the build tree, for example to change the startup type to JTAG or RAM. Adding the ecoflash package will result in one conflict related to the global compiler flags: by default ecoflash is built with no eCos debugging information, except for the ecoflash application itself. The target-side executable may get checked into the source code control system as a binary, so avoiding debug information helps to keep the size down. Stripping out all debug information after the build is not possible because it would interfere with some of the gdb commands that ecoflash uses to interact with the target.

The result of the make is an executable `flash.elf` in the `install/bin` subdirectory. This should get renamed to `<target>_flash.elf` and installed to a directory where ecoflash will find it. Optionally an `ecoflash.ecm` file containing the configuration settings can be exported to facilitate future rebuilding.

Next it is necessary to write the configuration file `<target>.ecf`. This is a straightforward expect script that gets included by the main ecoflash executable. It should set variables `::gdb_executable` and `::command_prefix`. Optionally it may also define procedures `target_init0`, `target_init1`, and `target_kill`. `target_init0` is invoked after gdb has been started but before the gdb target command has been issued. `target_init1` is invoked after the gdb target command has been issued, and typically takes care of initializing the board via a sequence of gdb commands. To facilitate this the main ecoflash script provides procedures gdb_run_command and gdb_run_quiet_command which will do the hard work. `target_kill` is invoked just before shutting down the gdb session. The `<target>.ecf` file is typically placed in the platform HAL's `misc` subdirectory.

Part XX. Flash Safe

Name

CYGPKG_FLASHSAFE — provide safe storage for data in flash memory

Description

The Flash Safe package provides a robust and simple mechanism for storing data in flash memory. It is intended for relatively small quantities of data such as configuration and customization data. For larger amounts of data, the JFFS2 flash filesystem is available.

The Flash Safe operates by dividing a region of the flash into a number of equal sized blocks. Each block is given a sequence number and is checksummed. A header containing these is stored at the start and end of each block. At startup the Flash Safe searches the available blocks for one with the latest sequence number and a valid checksum. The application can now retrieve data stored in the Flash Safe against a numeric key.

To store new data, the application opens a block, which will cause the block with the oldest sequence number to be erased and prepared for writing. Data can now be written to the block with a numeric key identifying each write. When all the data has been written, the block is committed, which will cause the headers to be written with valid checksums. This block now becomes the source of all subsequent data retrieval, freeing the original valid block for reuse.

This approach provides a simple transactional mechanism for storing data across power failures and crashes. At any time at least one committed block is valid and is released only after a new valid block has been committed to replace it. Any interruption during the creation of a new block will leave it invalid and data retrieval will fall back to the last committed block. The use of keys to identify data makes retrieval independent of the order in which the items are stored, of any change in size of the data and of any alignment requirements of the underlying flash device and driver.

The Flash Safe needs a minimum of two blocks to be defined, which must each be a multiple of the block size of the underlying flash device. A single Flash Safe block per flash device block would be the normal approach. More Flash Safe blocks may be used to implement a crude wear levelling mechanism, since under normal circumstances the Flash Safe will use the blocks in a round-robin manner.

Configuration

The flashsafe is mostly configured at runtime. The following CDL configuration options are present:

CYGNUM_FLASHSAFE_BUFFER_SIZE

> This option defines the size of the buffer that the flashsafe uses to store data prior to writing it to disk. Different flash devices have different alignment and minimum sizes for writes to the flash. This buffer collects data items into segments that can be written in single operations.
>
> A flashsafe block can be viewed as an array of segments of `CYGNUM_FLASHSAFE_BUFFER_SIZE` bytes each. The first and last segments are reserved for the flashsafe system's use, and the rest are available for data storage. So the buffer size may be at most one third the size of the block. The buffer size should also be chosen to be an integer fraction of the block size. It is not possible to change the buffer size once a flashsafe has been initialized, since it plays a part in defining the format of the stored data.

CYGPKG_FLASHSAFE_TESTS

> This lists the set of test programs. At present there is only one test, which runs on the synthetic target.

Interaction with RedBoot

The Flash Safe is mainly targetted at small systems where a FIS directory or JFFS2 would not be appropriate. Consequently it does not try to look up a named entry in the FIS or work via the flash IO device. Instead it uses raw flash block addresses. The flashsafe parameters can be set up at runtime by the application querying the RedBoot FIS interface, and the flash subsystem. The following code extract demonstrates how this might be done.

```
cyg_flashsafe flashsafe;
cyg_flash_info_t flashinfo;
cyg_uint32 size;
int err;

// Initialize flash system
err = cyg_flash_init(diag_printf);
if( err != CYG_FLASH_ERR_OK ) …

// Fetch flashsafe region base and size from FIS directory.
err = CYGACC_CALL_IF_FLASH_FIS_OP(CYGNUM_CALL_IF_FLASH_FIS_GET_FLASH_BASE,
                                  "flashsafe",
                                  &flashsafe.base);
if( err == 0 ) …

err = CYGACC_CALL_IF_FLASH_FIS_OP(CYGNUM_CALL_IF_FLASH_FIS_GET_SIZE,
                                  "flashsafe",
                                  &size);
if( err == 0 ) …

// Fetch flash device info from flash system
err = cyg_flash_get_info_addr( flashsafe.base, &flashinfo );
if( err != CYG_FLASH_ERR_OK ) …

// Calculate block number and size.
flashsafe.block_size = flashinfo.block_info[0].block_size;
flashsafe.block_count = size/flashsafe.block_size;

err = cyg_flashsafe_init( &flashsafe );
if(err != CYG_FLASHSAFE_ERR_OK) …
```

Name

Flash Safe — API Details

Synopsis

#include <cyg/flashsafe/flashsafe.h>

```
int cyg_flashsafe_init(*flashsafe);

int cyg_flashsafe_data(*flashsafe, key, **data);

int cyg_flashsafe_read(*flashsafe, key, *data, *len);

int cyg_flashsafe_open(*flashsafe);

int cyg_flashsafe_write(*flashsafe, key, *data, len);

int cyg_flashsafe_commit(*flashsafe);

const char *cyg_flashsafe_errmsg(err);
```

Description

The flash safe is accessed through this API. The flashsafe is initialized by calling `cyg_flashsafe_init()` passing it a pointer to a cyg_flashsafe structure. Within this structure the *`base`*, *`block_count`* and *`block_size`* fields must have been initialized to describe the area of flash to be managed. Typically, the structure would be defined statically as follows:

```
cyg_flashsafe flashsafe =
{
    .base               = 0x40000000,
    .block_count        = 3,
    .block_size         = 0x2000
};
```

If the flashsafe already contains data, then items may be retrieved using `cyg_flashsafe_data()`. The *`key`* argument identifies the data item to be retrieved. The *`data`* argument is a pointer to a location where a pointer to the data will be stored. Typically the pointer returned will refer directly to the flash device, and will thus be read-only. No size is returned, the application should either know what size the data is (e.g a structure or fixed sized array), or arrange for the data to be self-sized (e.g. a zero terminated string or contains a size field).

The function `cyg_flashsafe_read()` performs a similar job to `cyg_flashsafe_data()` except that the data is copied out of the flash memory. The *`data`* argument points to a buffer into which the data will be copied, and the *`len`* points to a location where the size of the buffer is stored when the call is made, and will contain the size of data copied in on return. This function is useful where it is intended to update the data read from the flashsafe, possibly to write it back to flashsafe. `cyg_flashsafe_data()` is useful where the data only needs to be read, and saves valuable RAM space.

To open a flashsafe block for update the function `cyg_flashsafe_open()` should be called. This will select the oldest block in the safe, erase it and prepare it for writing.

The function `cyg_flashsafe_write()` is used to write new data to the current open block. The *`key`* argument should be an application selected 16 bit value that is used to identify this data item. This value should be unique for each item, otherwise the behaviour is undefined. The *`data`* and *`len`* arguments describe a buffer containing the data to be written. The size of the data written must be less than or equal to the value of `CYGNUM_FLASHSAFE_BUFFER_SIZE`.

The function `cyg_flashsafe_commit()` causes the current open block to be committed to the flash. This involves calculating the checksum and writing the header and trailer with the current sequence number.

Each of these functions may return one of a number of error codes. These may include the following:

CYG_FLASHSAFE_ERR_OK

This is returned when the operation succeeded.

CYG_FLASHSAFE_ERR_FLASH

This is returned when the flash device failed to initialize.

CYG_FLASHSAFE_ERR_MISMATCH

This is returned when there is a mismatch between the size any layout of the flashsafe described in the initialized cyg_flashsafe structure and the flashsafe found in flash.

CYG_FLASHSAFE_ERR_FLASH_PROG

This is returned when the flash driver reports a programming error.

CYG_FLASHSAFE_ERR_FLASH_ERASE

This is returned when the flash driver reports a block erase error.

CYG_FLASHSAFE_ERR_NO_DATA

This is returned when there is no current valid block in the flashsafe. This will usually only happen when the flashsafe is new.

CYG_FLASHSAFE_ERR_BAD_KEY

This is returned when a given key cannot be found in the flashsafe.

CYG_FLASHSAFE_ERR_NOT_OPEN

This is returned when a `cyg_flashsafe_write()` or `cyg_flashsafe_commit()` are called before making a call to `cyg_flashsafe_open()`.

CYG_FLASHSAFE_ERR_NO_SPACE

This is returned when the size of data is too large for either the buffer or the flashsafe block as a whole.

The function `cyg_flashsafe_errmsg()`, if given one of these error codes, will return a string describing the error.

Part XXI. PCI Library

Table of Contents

Chapter 71. The eCos PCI Library

The PCI library is an optional part of eCos, and is only applicable to some platforms.

PCI Library

The eCos PCI library provides the following functionality:

1. Scan the PCI bus for specific devices or devices of a certain class.
2. Read and change generic PCI information.
3. Read and change device-specific PCI information.
4. Allocate PCI memory and IO space to devices.
5. Translate a device's PCI interrupts to equivalent HAL vectors.

Example code fragments are from the pci1 test (see `io/pci/<release>/tests/pci1.c`).

All of the functions described below are declared in the header file `<cyg/io/pci.h>` which all clients of the PCI library should include.

PCI Overview

The PCI bus supports several address spaces: memory, IO, and configuration. All PCI devices must support mandatory configuration space registers. Some devices may also present IO mapped and/or memory mapped resources. Before devices on the bus can be used, they must be configured. Basically, configuration will assign PCI IO and/or memory address ranges to each device and then enable that device. All PCI devices have a unique address in configuration space. This address is comprised of a bus number, a device number, and a function number. Special devices called bridges are used to connect two PCI busses together. The PCI standard supports up to 255 busses with each bus having up to 32 devices and each device having up to 8 functions.

The environment in which a platform operates will dictate if and how eCos should configure devices on the PCI bus. If the platform acts as a host on a single PCI bus, then devices may be configured individually from the relevant device driver. If the platform is not the primary host, such as a PCI card plugged into a PC, configuration of PCI devices may be left to the PC BIOS. If PCI-PCI bridges are involved, configuration of all devices is best done all at once early in the boot process. This is because all devices on the secondary side of a bridge must be evaluated for their IO and memory space requirements before the bridge can be configured.

Initializing the bus

The PCI bus needs to be initialized before it can be used. This only needs to be done once - some HALs may do it as part of the platform initialization procedure, other HALs may leave it to the application or device drivers to do it. The following function will do the initialization only once, so it's safe to call from multiple drivers:

```
void cyg_pci_init( void );
```

Scanning for devices

After the bus has been initialized, it is possible to scan it for devices. This is done using the function:

```
cyg_bool cyg_pci_find_next(  cyg_pci_device_id cur_devid,
                             cyg_pci_device_id *next_devid );
```

It will scan the bus for devices starting at *`cur_devid`*. If a device is found, its devid is stored in *`next_devid`* and the function returns `true`.

The pci1 test's outer loop looks like:

```
cyg_pci_init();
if (cyg_pci_find_next(CYG_PCI_NULL_DEVID, &devid)) {
    do {
         <use devid>
    } while (cyg_pci_find_next(devid, &devid));
}
```

What happens is that the bus gets initialized and a scan is started. CYG_PCI_NULL_DEVID causes cyg_pci_find_next() to restart its scan. If the bus does not contain any devices, the first call to cyg_pci_find_next() will return false.

If the call returns true, a loop is entered where the found devid is used. After devid processing has completed, the next device on the bus is searched for; cyg_pci_find_next() continues its scan from the current devid. The loop terminates when no more devices are found on the bus.

This is the generic way of scanning the bus, enumerating all the devices on the bus. But if the application is looking for a device of a given device class (e.g., a SCSI controller), or a specific vendor device, these functions simplify the task a bit:

```
cyg_bool cyg_pci_find_class(  cyg_uint32 dev_class,
                              cyg_pci_device_id *devid );

cyg_bool cyg_pci_find_device( cyg_uint16 vendor, cyg_uint16 device,
                              cyg_pci_device_id *devid );
```

They work just like cyg_pci_find_next(), but only return true when the dev_class or vendor/device qualifiers match those of a device on the bus. The devid serves as both an input and an output operand: the scan starts at the given device, and if a device is found devid is updated with the value for the found device.

The <cyg/io/pci_cfg.h> header file (included by pci.h) contains definitions for PCI class, vendor and device codes which can be used as arguments to the find functions. The list of vendor and device codes is not complete: add new codes as necessary. If possible also register the codes at the PCI Database (http://www.pcidatabase.com) which is where the eCos definitions are generated from.

Generic config information

When a valid device ID (devid) is found using one of the above functions, the associated device can be queried and controlled using the functions:

```
void cyg_pci_get_device_info (  cyg_pci_device_id devid,
                                cyg_pci_device *dev_info );

void cyg_pci_set_device_info (  cyg_pci_device_id devid,
                                cyg_pci_device *dev_info );
```

The cyg_pci_device structure (defined in pci.h) primarily holds information as described by the PCI specification [1]. The pci1 test prints out some of this information:

```
// Get device info
cyg_pci_get_device_info(devid, &dev_info);
diag_printf("\n Command   0x%04x, Status 0x%04x\n",
            dev_info.command, dev_info.status);
```

The command register can also be written to, controlling (among other things) whether the device responds to IO and memory access from the bus.

Specific config information

The above functions only allow access to generic PCI config registers. A device can have extra config registers not specified by the PCI specification. These can be accessed with these functions:

```
void cyg_pci_read_config_uint8(   cyg_pci_device_id devid,
                                  cyg_uint8 offset, cyg_uint8 *val);

void cyg_pci_read_config_uint16(  cyg_pci_device_id devid,
                                  cyg_uint8 offset, cyg_uint16 *val);

void cyg_pci_read_config_uint32(  cyg_pci_device_id devid,
                                  cyg_uint8 offset, cyg_uint32 *val);

void cyg_pci_write_config_uint8(  cyg_pci_device_id devid,
                                  cyg_uint8 offset, cyg_uint8 val);

void cyg_pci_write_config_uint16( cyg_pci_device_id devid,
                                  cyg_uint8 offset, cyg_uint16 val);

void cyg_pci_write_config_uint32( cyg_pci_device_id devid,
                                  cyg_uint8 offset, cyg_uint32 val);
```

The write functions should only be used for device-specific config registers since using them on generic registers may invalidate the contents of a previously fetched cyg_pci_device structure.

Allocating memory

A PCI device ignores all IO and memory access from the PCI bus until it has been activated. Activation cannot happen until after device configuration. Configuration means telling the device where it should map its IO and memory resources. This is done with one of the following functions::

```
cyg_bool cyg_pci_configure_device( cyg_pci_device *dev_info );

cyg_bool cyg_pci_configure_bus( cyg_uint8 bus, cyg_uint8 *next_bus );
```

The `cyg_pci_configure_device` handles all IO and memory regions that need configuration on non-bridge devices. On platforms with multiple busses connected by bridges, the `cyg_pci_configure_bus` function should be used. It will recursively configure all devices on the given *`bus`* and all subordinate busses. `cyg_pci_configure_bus` will use `cyg_pci_configure_device` to configure individual non-bridge devices.

Each region is represented in the PCI device's config space by BARs (Base Address Registers) and is handled individually according to type using these functions:

```
cyg_bool cyg_pci_allocate_memory( cyg_pci_device *dev_info,
                                  cyg_uint32 bar,
                                  CYG_PCI_ADDRESS64 *base );

cyg_bool cyg_pci_allocate_io(     cyg_pci_device *dev_info,
                                  cyg_uint32 bar,
                                  CYG_PCI_ADDRESS32 *base );
```

The memory bases (in two distinct address spaces) are increased as memory regions are allocated to devices. Allocation will fail (the function returns false) if the base exceeds the limits of the address space (IO is 1MB, memory is 2^32 or 2^64 bytes).

These functions can also be called directly by the application/driver if necessary, but this should not be necessary.

The bases are initialized with default values provided by the HAL. It is possible for an application to override these using the following functions:

```
void cyg_pci_set_memory_base(  CYG_PCI_ADDRESS64 base );

void cyg_pci_set_io_base( CYG_PCI_ADDRESS32 base );
```

When a device has been configured, the cyg_pci_device structure will contain the physical address in the CPU's address space where the device's memory regions can be accessed.

This information is provided in `base_map[]` - there is a 32 bit word for each of the device's BARs. For 32 bit PCI memory regions, each 32 bit word will be an actual pointer that can be used immediately by the driver: the memory space will normally be linearly addressable by the CPU.

However, for 64 bit PCI memory regions, some (or all) of the region may be outside of the CPUs address space. In this case the driver will need to know how to access the region in segments. This functionality may be adopted by the eCos HAL if deemed useful in the future. The 2GB available on many systems should suffice though.

Interrupts

A device may generate interrupts. The HAL vector associated with a given device on the bus is platform specific. This function allows a driver to find the actual interrupt vector for a given device:

```
cyg_bool cyg_pci_translate_interrupt(  cyg_pci_device *dev_info,
                                       CYG_ADDRWORD *vec );
```

If the function returns false, no interrupts will be generated by the device. If it returns true, the CYG_ADDRWORD pointed to by vec is updated with the HAL interrupt vector the device will be using. This is how the function is used in the `pci1` test:

```
if (cyg_pci_translate_interrupt(&dev_info, &irq))
    diag_printf(" Wired to HAL vector %d\n", irq);
else
    diag_printf(" Does not generate interrupts.\n");
```

The application/drive should attach an interrupt handler to a device's interrupt before activating the device.

Activating a device

When the device has been allocated memory space it can be activated. This is not done by the library since a driver may have to initialize more state on the device before it can be safely activated.

Activating the device is done by enabling flags in its command word. As an example, see the `pci1` test which can be configured to enable the devices it finds. This allows these to be accessed from GDB (if a breakpoint is set on `cyg_test_exit`):

```
#ifdef ENABLE_PCI_DEVICES
    {
        cyg_uint16 cmd;

        // Don't use cyg_pci_set_device_info since it clears
        // some of the fields we want to print out below.
        cyg_pci_read_config_uint16(dev_info.devid,
                                   CYG_PCI_CFG_COMMAND, &cmd);
        cmd |= CYG_PCI_CFG_COMMAND_IO|CYG_PCI_CFG_COMMAND_MEMORY;
        cyg_pci_write_config_uint16(dev_info.devid,
                                    CYG_PCI_CFG_COMMAND, cmd);
    }
    diag_printf(" **** Device IO and MEM access enabled\n");
#endif
```

Note

The best way to activate a device is actually through `cyg_pci_set_device_info()`, but in this particular case the cyg_pci_device structure contents from before the activation is required for printout further down in the code.

Links

See these links for more information about PCI:

1. http://www.pcisig.com/ - information on the PCI specifications

2. http://www.yourvote.com/pci/ - list of vendor and device IDs

3. http://www.picmg.org/ - PCI Industrial Computer Manufacturers Group

PCI Library reference

This document defines the PCI Support Library for eCos.

The PCI support library provides a set of routines for accessing the PCI bus configuration space in a portable manner. This is provided by two APIs. The high level API is used by device drivers, or other code, to access the PCI configuration space portably. The low level API is used by the PCI library itself to access the hardware in a platform-specific manner, and may also be used by device drivers to access the PCI configuration space directly.

Underlying the low-level API is HAL support for the basic configuration space operations. These should not generally be used by any code other than the PCI library, and are present in the HAL to allow low level initialization of the PCI bus and devices to take place if necessary.

PCI Library API

The PCI library provides the following routines and types for accessing the PCI configuration space.

The API for the PCI library is found in the header file `<cyg/io/pci.h>`.

Definitions

The header file contains definitions for the common configuration structure offsets and specimen values for device, vendor and class code.

Types and data structures

The following types are defined:

```
typedef CYG_WORD32 cyg_pci_device_id;
```

This is comprised of the bus number, device number and functional unit numbers packed into a single word. The macro `CYG_PCI_DEV_MAKE_ID()`, in conjunction with the `CYG_PCI_DEV_MAKE_DEVFN()` macro, may be used to construct a device id from the bus, device and functional unit numbers. Similarly the macros `CYG_PCI_DEV_GET_BUS()`, `CYG_PCI_DEV_GET_DEVFN()`, `CYG_PCI_DEV_GET_DEV()`, and `CYG_PCI_DEV_GET_FN()` may be used to extract the constituent parts of a device id. It should not be necessary to use these macros under normal circumstances. The following code fragment demonstrates how these macros may be used:

```
// Create a packed representation of device 1, function 0
cyg_uint8 devfn = CYG_PCI_DEV_MAKE_DEVFN(1,0);

// Create a packed devid for that device on bus 2
cyg_pci_device_id devid = CYG_PCI_DEV_MAKE_ID(2, devfn);

diag_printf("bus %d, dev %d, func %d\n",
            CYG_PCI_DEV_GET_BUS(devid),
            CYG_PCI_DEV_GET_DEV(CYG_PCI_DEV_GET_DEVFN(devid)),
            CYG_PCI_DEV_GET_FN(CYG_PCI_DEV_GET_DEVFN(devid));
```

```
typedef struct cyg_pci_device;
```

This structure is used to contain data read from a PCI device's configuration header by `cyg_pci_get_device_info()`. It is also used to record the resource allocations made to the device.

```
typedef CYG_WORD64 CYG_PCI_ADDRESS64;
typedef CYG_WORD32 CYG_PCI_ADDRESS32;
```

Pointers in the PCI address space are 32 bit (IO space) or 32/64 bit (memory space). In most platform and device configurations all of PCI memory will be linearly addressable using only 32 bit pointers as read from `base_map[]`.

The 64 bit type is used to allow handling 64 bit devices in the future, should it be necessary, without changing the library's API.

Functions

```
void cyg_pci_init(void);
```

Initialize the PCI library and establish contact with the hardware. This function is idempotent and can be called either by all drivers in the system, or just from an application initialization function.

```
cyg_bool cyg_pci_find_device( cyg_uint16        vendor,
                              cyg_uint16        device,
                              cyg_pci_device_id *devid );
```

Searches the PCI bus configuration space for a device with the given *`vendor`* and *`device`* ids. The search starts at the device pointed to by *`devid`*, or at the first slot if it contains `CYG_PCI_NULL_DEVID`. *`*devid`* will be updated with the ID of the next device found. Returns `true` if one is found and `false` if not.

```
cyg_bool cyg_pci_find_class(  cyg_uint32        dev_class,
                              cyg_pci_device_id *devid );
```

Searches the PCI bus configuration space for a device with the given *`dev_class`* class code. The search starts at the device pointed to by *`devid`*, or at the first slot if it contains `CYG_PCI_NULL_DEVID`.

`*devid` will be updated with the ID of the next device found. Returns `true` if one is found and `false` if not.

```
cyg_bool cyg_pci_find_next(  cyg_pci_device_id cur_devid,
                             cyg_pci_device_id *next_devid );
```

Searches the PCI configuration space for the next valid device after *`cur_devid`*. If *`cur_devid`* is given the value `CYG_PCI_NULL_DEVID`, then the search starts at the first slot. It is permitted for *`next_devid`* to point to *`cur_devid`*. Returns `true` if another device is found and `false` if not.

```
cyg_bool cyg_pci_find_matching( cyg_pci_match_func  *matchp,
                                void                *match_callback_data,
                                cyg_pci_device_id   *devid );
```

Searches the PCI bus configuration space for a device whose properties match those required by the caller supplied *`cyg_pci_match_func`*. The search starts at the device pointed to by *`devid`*, or at the first slot if it contains `CYG_PCI_NULL_DEVID`. The *`devid`* will be updated with the ID of the next device found. This function returns `true` if a matching device is found and `false` if not.

The match_func has a type declared as:

```
typedef cyg_bool (cyg_pci_match_func)( cyg_uint16 vendor,
                   cyg_uint16 device,
                   cyg_uint32 class,
                   void *     user_data);
```

The *`vendor`*, *`device`*, and *`class`* are from the device configuration space. The *`user_data`* is the callback data passed to `cyg_pci_find_matching`.

```
void cyg_pci_get_device_info ( cyg_pci_device_id devid,
                               cyg_pci_device *dev_info );
```

This function gets the PCI configuration information for the device indicated in *`devid`*. The common fields of the cyg_pci_device structure, and the appropriate fields of the relevant header union member are filled in from the device's configuration space. If the device has not been enabled, then this function will also fetch the size and type information from the base address registers and place it in the `base_size[]` array.

```
void cyg_pci_set_device_info ( cyg_pci_device_id  devid,
                               cyg_pci_device     *dev_info );
```

This function sets the PCI configuration information for the device indicated in *devid*. Only the configuration space registers that are writable are actually written. Once all the fields have been written, the device info will be read back into **dev_info*, so that it reflects the true state of the hardware.

```
void cyg_pci_read_config_uint8(  cyg_pci_device_id  devid,
                                 cyg_uint8          offset,
                                 cyg_uint8          *val );

void cyg_pci_read_config_uint16( cyg_pci_device_id  devid,
                                 cyg_uint8          offset,
                                 cyg_uint16         *val );

void cyg_pci_read_config_uint32( cyg_pci_device_id  devid,
                                 cyg_uint8 offset,  cyg_uint32 *val );
```

These functions read registers of the appropriate size from the configuration space of the given device. They should mainly be used to access registers that are device specific. General PCI registers are best accessed through `cyg_pci_get_device_info()`.

```
void cyg_pci_write_config_uint8(  cyg_pci_device_id devid,
                                  cyg_uint8 offset, cyg_uint8 val );

void cyg_pci_write_config_uint16( cyg_pci_device_id devid,
                                  cyg_uint8 offset, cyg_uint16 val );

void cyg_pci_write_config_uint32( cyg_pci_device_id devid,
                                  cyg_uint8 offset, cyg_uint32 val );
```

These functions write registers of the appropriate size to the configuration space of the given device. They should mainly be used to access registers that are device specific. General PCI registers are best accessed through `cyg_pci_get_device_info()`. Writing the general registers this way may render the contents of a cyg_pci_device structure invalid.

Resource allocation

These routines allocate memory and I/O space to PCI devices.

```
cyg_bool cyg_pci_configure_device( cyg_pci_device *dev_info )
```

Allocate memory and IO space to all base address registers using the current memory and IO base addresses in the library. The allocated base addresses, translated into directly usable values, will be put into the matching `base_map[]` entries in **dev_info*. If **dev_info* does not contain valid `base_size[]` entries, then the result is `false`. This function will also call `cyg_pci_translate_interrupt()` to put the interrupt vector into the HAL vector entry.

```
cyg_bool cyg_pci_configure_bus( cyg_uint8 bus, cyg_uint8 *next_bus )
```

Allocate memory and IO space to all base address registers on all devices on the given bus and all subordinate busses. If a PCI-PCI bridge is found on *bus*, this function will call itself recursively in order to configure the bus on the other side of the bridge. Because of the nature of bridge devices, all devices on the secondary side of a bridge must be allocated memory and IO space before the memory and IO windows on the bridge device can be properly configured. The *next_bus* argument points to the bus number to assign to the next subordinate bus found. The number will be incremented as new busses are discovered. If successful, `true` is returned. Otherwise, `false` is returned.

```
cyg_bool cyg_pci_translate_interrupt( cyg_pci_device  *dev_info,
                                      CYG_ADDRWORD    *vec );
```

Translate the device's PCI interrupt (INTA#-INTD#) to the associated HAL vector. This may also depend on which slot the device occupies. If the device may generate interrupts, the translated vector number will be stored in *vec* and the result is `true`. Otherwise the result is `false`.

```
cyg_bool cyg_pci_allocate_memory( cyg_pci_device    *dev_info,
                                  cyg_uint32        bar,
                                  CYG_PCI_ADDRESS64 *base );

cyg_bool cyg_pci_allocate_io( cyg_pci_device    *dev_info,
                              cyg_uint32        bar,
                              CYG_PCI_ADDRESS32 *base );
```

These routines allocate memory or I/O space to the base address register indicated by *`bar`*. The base address in *`*base`* will be correctly aligned and the address of the next free location will be written back into it if the allocation succeeds. If the base address register is of the wrong type for this allocation, or *`dev_info`* does not contain valid `base_size[]` entries, the result is `false`. These functions allow a device driver to set up its own mappings if it wants. Most devices should probably use `cyg_pci_configure_device()`.

```
void cyg_pci_set_memory_base( CYG_PCI_ADDRESS64 base );

void cyg_pci_set_io_base( CYG_PCI_ADDRESS32 base );
```

These routines set the base addresses for memory and I/O mappings to be used by the memory allocation routines. Normally these base addresses will be set to default values based on the platform. These routines allow these to be changed by application code if necessary.

PCI Library Hardware API

This API is used by the PCI library to access the PCI bus configuration space. Although it should not normally be necessary, this API may also be used by device driver or application code to perform PCI bus operations not supported by the PCI library.

```
void cyg_pcihw_init(void);
```

Initialize the PCI hardware so that the configuration space may be accessed.

```
void cyg_pcihw_read_config_uint8(  cyg_uint8 bus,
                                   cyg_uint8 devfn,
                                   cyg_uint8 offset,
                                   cyg_uint8 *val);

void cyg_pcihw_read_config_uint16( cyg_uint8 bus,
                                   cyg_uint8 devfn,
                                   cyg_uint8 offset,
                                   cyg_uint16 *val);

void cyg_pcihw_read_config_uint32( cyg_uint8 bus,
                                   cyg_uint8 devfn,
                                   cyg_uint8 offset,
                                   cyg_uint32 *val);
```

These functions read a register of the appropriate size from the PCI configuration space at an address composed from the *`bus`*, *`devfn`* and *`offset`* arguments.

```
void cyg_pcihw_write_config_uint8(  cyg_uint8 bus,
                                    cyg_uint8 devfn,
                                    cyg_uint8 offset,
                                    cyg_uint8 val);

void cyg_pcihw_write_config_uint16( cyg_uint8 bus,
                                    cyg_uint8 devfn,
                                    cyg_uint8 offset,
                                    cyg_uint16 val);

void cyg_pcihw_write_config_uint32( cyg_uint8 bus,
                                    cyg_uint8 devfn,
                                    cyg_uint8 offset,
                                    cyg_uint32 val);
```

These functions write a register of the appropriate size to the PCI configuration space at an address composed from the *bus*, *devfn* and *offset* arguments.

```
cyg_bool cyg_pcihw_translate_interrupt( cyg_uint8     bus,
                                        cyg_uint8     devfn,
                                        CYG_ADDRWORD  *vec);
```

This function interrogates the device and determines which HAL interrupt vector it is connected to.

HAL PCI support

HAL support consists of a set of C macros that provide the implementation of the low level PCI API.

```
HAL_PCI_INIT()
```

Initialize the PCI bus.

```
HAL_PCI_READ_UINT8( bus, devfn, offset, val )
HAL_PCI_READ_UINT16( bus, devfn, offset, val )
HAL_PCI_READ_UINT32( bus, devfn, offset, val )
```

Read a value from the PCI configuration space of the appropriate size at an address composed from the *bus*, *devfn* and *offset*.

```
HAL_PCI_WRITE_UINT8( bus, devfn, offset, val )
HAL_PCI_WRITE_UINT16( bus, devfn, offset, val )
HAL_PCI_WRITE_UINT32( bus, devfn, offset, val )
```

Write a value to the PCI configuration space of the appropriate size at an address composed from the *bus*, *devfn* and *offset*.

```
HAL_PCI_TRANSLATE_INTERRUPT( bus, devfn, *vec, valid )
```

Translate the device's interrupt line into a HAL interrupt vector.

```
HAL_PCI_ALLOC_BASE_MEMORY
HAL_PCI_ALLOC_BASE_IO
```

These macros define the default base addresses used to initialize the memory and I/O allocation pointers.

```
HAL_PCI_PHYSICAL_MEMORY_BASE
HAL_PCI_PHYSICAL_IO_BASE
```

PCI memory and IO range do not always correspond directly to physical memory or IO addresses. Frequently the PCI address spaces are windowed into the processor's address range at some offset. These macros define offsets to be added to the PCI base addresses to translate PCI bus addresses into physical memory addresses that can be used to access the allocated memory or IO space.

Note

The chunk of PCI memory space directly addressable though the window by the CPU may be smaller than the amount of PCI memory actually provided. In that case drivers will have to access PCI memory space in segments. Doing this will be platform specific and is currently beyond the scope of the HAL.

```
HAL_PCI_IGNORE_DEVICE( bus, dev, fn )
```

This macro, if defined, may be used to limit the devices which are found by the bus scanning functions. This is sometimes necessary for devices which need special handling. If this macro evaluates to `true`, the given device will not be found by `cyg_pci_find_next` or other bus scanning functions.

```
HAL_PCI_IGNORE_BAR( dev_info, bar_num )
```

This macro, if defined, may be used to limit which BARs are discovered and configured. This is sometimes necessary for platforms with limited PCI windows. If this macro evaluates to `true`, the given BAR will not be discovered by `cyg_pci_get_device_info` and therefore not configured by `cyg_pci_configure_device`.

Part XXII. SPI Support

Documentation for drivers of this type is often integrated into the eCos board support documentation. You should review the documentation for your target board for details. Standalone and more generic drivers are documented in the following sections.

Table of Contents

Chapter 72. SPI Support

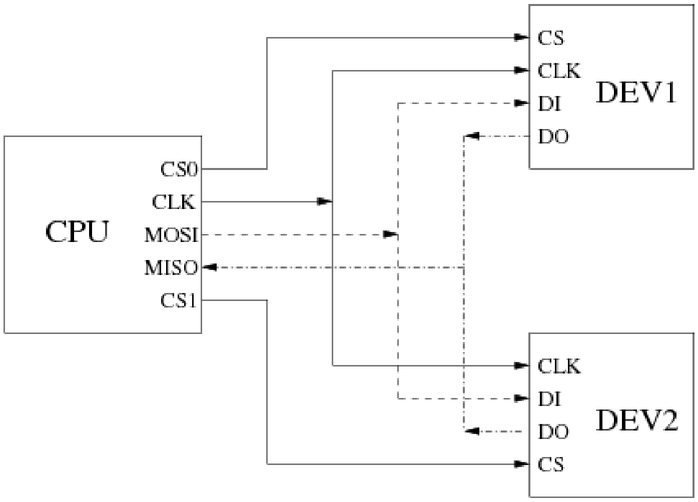
Name

Overview — eCos Support for SPI, the Serial Peripheral Interface

Description

The Serial Peripheral Interface (SPI) is one of a number of serial bus technologies. It can be used to connect a processor to one or more peripheral chips, for example analog-to-digital convertors or real time clocks, using only a small number of pins and PCB tracks. The technology was originally developed by Motorola but is now also supported by other vendors.

A typical SPI system might look like this:



At the start of a data transfer the master cpu asserts one of the chip select signals and then generates a clock signal. During each clock tick the cpu will output one bit on its master-out-slave-in line and read one bit on the master-in-slave-out line. Each device is connected to the clock line, the two data lines, and has its own chip select. If a device's chip select is not asserted then it will ignore any incoming data and will tristate its output. If a device's chip select is asserted then during each clock tick it will read one bit of data on its input pin and output one bit on its output pin.

The net effect is that the cpu can write an arbitrary amount of data to one of the attached devices at a time, and simultaneously read the same amount of data. Some devices are inherently uni-directional. For example an LED unit would only accept data from the cpu: it will not send anything meaningful back; the cpu will still sample its input every clock tick, but this should be discarded.

A useful feature of SPI is that there is no flow control from the device back to the cpu. If the cpu tries to communicate with a device that is not currently present, for example an MMC socket which does not contain a card, then the I/O will still proceed. However the cpu will read random data. Typically software-level CRC checksums or similar techniques will be used to allow the cpu to detect this.

SPI communication is not fully standardized. Variations between devices include the following:

1. Many devices involve byte transfers, where the unit of data is 8 bits. Others use larger units, up to 16 bits.
2. Chip selects may be active-high or active-low. If the attached devices use a mixture of polarities then this can complicate things.
3. Clock rates can vary from 128KHz to 20MHz or greater. With some devices it is necessary to interrogate the device using a slow clock, then use the obtained information to select a faster clock for subsequent transfers.
4. The clock is inactive between data transfers. When inactive the clock's polarity can be high or low.
5. Devices depend on the phase of the clock. Data may be sampled on either the rising edge or the falling edge of the clock.

6. A device may need additional delays, for example between asserting the chip select and the first clock tick.

7. Some devices involve complicated transactions: perhaps a command from cpu to device; then an initial status response from the device; a data transfer; and a final status response. From the cpu's perspective these are separate stages and it may be necessary to abort the operation after the initial status response. However the device may require that the chip select remain asserted for the whole transaction. A side effect of this is that it is not possible to do a quick transfer with another device in the middle of the transaction.

8. Certain devices, for example MMC cards, depend on a clock signal after a transfer has completed and the chip select has dropped. This clock is used to finish some processing within the device.

Inside the cpu the clock and data signals are usually managed by dedicated hardware. Alternatively SPI can be implemented using bit-banging, but that approach is normally used for other serial bus technologies such as I²C. The chip selects may also be implemented by the dedicated SPI hardware, but often GPIO pins are used instead.

eCos Support for SPI

The eCos SPI support for any given platform is spread over a number of different packages:

- This package, `CYGPKG_IO_SPI`, exports an API for accessing devices attached to an SPI bus. This API handles issues such as locking between threads. The package does not contain any hardware-specific code, instead it will call into an SPI bus driver package.

  In future this package may be extended with a bit-banging implementation of an SPI bus driver. This would depend on lower-level code for manipulating the GPIO pins used for the clock, data and chip select signals, but timing and framing could be handled by generic code.

- There will be a bus driver package for the specific SPI hardware on the target hardware, for example `CYGPKG_DEVS_SPI_MCFxxxx_QSPI`. This is responsible for the actual I/O. A bus driver may be used on many different boards, all with the same SPI bus but with different devices attached to that bus. Details of the actual devices should be supplied by other code.

- The generic API depends on cyg_spi_device data structures. These contain the information needed by a bus driver, for example the appropriate clock rate and the chip select to use. Usually the data structures are provided by the platform HAL since it is that package which knows about all the devices on the board.

  On some development boards the SPI pins are brought out to expansion connectors, allowing end users to add extra devices. In such cases the platform HAL may not know about all the devices on the board. Data structures for the additional devices can instead be supplied by application code.

- Some types of SPI devices may have their own driver package. For example the `CYGPKG_DEVS_FLASH_SPI_COMMON` package provides an abstraction layer to the standard eCos I/O Flash API package (`CYGPKG_IO_FLASH`) for SFDP compliant flash devices. Another common use for SPI buses is to provide MultiMediaCard (MMC/SD) support. The `CYGPKG_DEVS_DISK_MMC` package provides a block device interface for higher-level code such as file systems. Other SPI devices such as analog-to-digital converters are much simpler and come in many varieties. There are no dedicated packages to support each such device: the chances are low that another board would use the exact same device, so there are no opportunities for code reuse. Instead the devices may be accessed directly by application code or by extra functions in the platform HAL.

Typically all appropriate packages will be loaded automatically when you configure eCos for a given target. If the application does not use any of the SPI I/O facilities, directly or indirectly, then linker garbage collection should eliminate all unnecessary code and data. All necessary initialization should happen automatically. However the exact details may depend on the target, so the platform HAL documentation should be checked for further details.

There is one important exception to this: if the SPI devices are attached to an expansion connector then the platform HAL will not know about these devices. Instead more work will have to be done by application code.

Name

SPI Functions — allow applications and other packages to access SPI devices

Synopsis

#include <cyg/io/spi.h>

CYG_SPI_OP_RETURN_TYPE **cyg_spi_transfer**(*device*, *polled*, *count*, *tx_data*, *rx_data*);

CYG_SPI_OP_RETURN_TYPE **cyg_spi_tick**(*device*, *polled*, *count*);

int **cyg_spi_get_config**(*device*, *key*, *buf*, *len*);

int **cyg_spi_set_config**(*device*, *key*, *buf*, *len*);

CYG_SPI_OP_RETURN_TYPE **cyg_spi_transaction_begin**(*device*);

cyg_bool **cyg_spi_transaction_begin_nb**(*device*);

CYG_SPI_OP_RETURN_TYPE **cyg_spi_transaction_transfer**(*device*, *polled*, *count*, *tx_data*, *rx_data*, *drop_cs*);

CYG_SPI_OP_RETURN_TYPE **cyg_spi_transaction_tick**(*device*, *polled*, *count*);

CYG_SPI_OP_RETURN_TYPE **cyg_spi_transaction_end**(*device*);

Description

All SPI functions take a pointer to a cyg_spi_device structure as their first argument. This is an opaque data structure, usually provided by the platform HAL. It contains the information needed by the SPI bus driver to interact with the device, for example the required clock rate and polarity.

An SPI transfer involves the following stages:

1. Perform thread-level locking on the bus. Only one thread at a time is allowed to access an SPI bus. This eliminates the need to worry about locking at the bus driver level. If a platform involves multiple SPI buses then each one will have its own lock. Prepare the bus for transfers to the specified device, for example by making sure it will tick at the right clock rate.
2. Assert the chip select on the specified device, then transfer data to and from the device. There may be a single data transfer or a sequence. It may or may not be necessary to keep the chip select asserted throughout a sequence.
3. Optionally generate some number of clock ticks without asserting a chip select, for those devices which need this to complete an operation.
4. Return the bus to a quiescent state. Then unlock the bus, allowing other threads to perform SPI operations on devices attached to this bus.

The simple functions `cyg_spi_transfer` and `cyg_spi_tick` perform all these steps in a single call. These are suitable for simple I/O operations. The alternative transaction-oriented functions each perform just one of these steps. This makes it possible to perform multiple transfers while only locking and unlocking the bus once, as required for more complicated devices.

With the exception of `cyg_spi_transaction_begin_nb` all the functions will block until completion. The type of the return values depends on whether the underlying hardware driver can return error conditions or not. If it can, this will be indicated with the C preprocessor define `CYGINT_IO_SPI_DRV_REPORTS_ERRORS`, and the return type `CYG_SPI_OP_RETURN_TYPE` will be a standard error code as defined in the C/POSIX header `<errno.h>`. Alternatively for drivers which do not return errors, the return value's type will be `void`, or in other words, no return value. If there are any errors, they would be generated by an SPI bus peripheral, and not by the attached SPI device. The SPI bus does not receive any feedback from a device about possible errors, instead those have to be handled by software at a higher level.

If a driver is capable of indicating errors, and an error value is returned, this may imply the SPI operation requested may not have started, may not have completed, or may only have partially completed. The exact interpretation will depend on both hardware properties and low-level driver implementation. SPI API users are free to ignore all return values, in which case the usage of the SPI API will be identical, irrespective of the setting of `CYGINT_IO_SPI_DRV_REPORTS_ERRORS`.

An SPI transfer will always take a predictable amount of time, depending on the transfer size and the clock rate. If a thread cannot afford the time it will take to perform a complete large transfer then a number of smaller transfers can be used instead.

SPI operations should always be performed at thread-level or during system initialization, and not inside an ISR or DSR. This greatly simplifies locking. Also a typical ISR or DSR should not perform a blocking operation such as an SPI transfer.

SPI transfers can happen in either polled or interrupt-driven mode. Typically polled mode should be used during system initialization, before the scheduler has been started and interrupts have been enabled. Polled mode should also be used in single-threaded applications such as RedBoot. A typical multi-threaded application should normally use interrupt-driven mode because this allows for more efficient use of cpu cycles. Polled mode may be used in a multi-threaded application but this is generally undesirable: the cpu will spin while waiting for a transfer to complete, wasting cycles; also the current thread may get preempted or timesliced, making the timing of an SPI transfer much less predictable. On some hardware interrupt-driven mode is impossible or would be very inefficient. In such cases the bus drivers will only support polled mode and will ignore the `polled` argument.

Simple Transfers

`cyg_spi_transfer` can be used for SPI operations to simple devices. It takes the following arguments:

cyg_spi_device* `device`
: This identifies the SPI device that should be used.

cyg_bool `polled`
: Polled mode should be used during system initialization and in a single-threaded application. Interrupt-driven mode should normally be used in a multi-threaded application.

cyg_uint32 `count`
: This identifies the number of data items to be transferred. Usually each data item is a single byte, but some devices use a larger size up to 16 bits.

const cyg_uint8* `tx_data`
: The data to be transferred to the device. If the device will only output data and ignore its input then a null pointer can be used. Otherwise the array should contain `count` data items, usually bytes. For devices where each data item is larger than one byte the argument will be interpreted as an array of shorts instead, and should be aligned to a 2-byte boundary. The bottom n bits of each short will be sent to the device. The buffer need not be aligned to a cache-line boundary, even for SPI devices which use DMA transfers, but some bus drivers may provide better performance if the buffer is suitably aligned. The buffer will not be modified by the transfer.

cyg_uint8* `rx_data`
: A buffer for the data to be received from the device. If the device does not generate any output then a null pointer can be used. The same size and alignment rules apply as for `tx_data`.

`cyg_spi_transfer` performs all the stages of an SPI transfer: locking the bus; setting it up correctly for the specified device; asserting the chip select and transferring the data; dropping the chip select at the end of the transfer; returning the bus to a quiescent state; and unlocking the bus.

Additional Clock Ticks

Some devices require a number of clock ticks on the SPI bus between transfers so that they can complete some internal processing. These ticks must happen at the appropriate clock rate but no chip select should be asserted and no data transfer will happen.

cyg_spi_tick provides this functionality. The device argument identifies the SPI bus, the required clock rate and the size of each data item. The polled argument has the usual meaning. The count argument specifies the number of data items that would be transferred, which in conjunction with the size of each data item determines the number of clock ticks.

Transactions

A transaction-oriented API is available for interacting with more complicated devices. This provides separate functions for each of the steps in an SPI transfer.

cyg_spi_transaction_begin must be used at the start of a transaction. This performs thread-level locking on the bus, blocking if it is currently in use by another thread. Then it prepares the bus for transfers to the specified device, for example by making sure it will tick at the right clock rate.

cyg_spi_transaction_begin_nb is a non-blocking variant, useful for threads which cannot afford to block for an indefinite period. If the bus is currently locked the function returns false immediately. If the bus is not locked then it acts as cyg_spi_transaction_begin and returns true.

Once the bus has been locked it is possible to perform one or more data transfers by calling cyg_spi_transaction_transfer. This takes the same arguments as cyg_spi_transfer, plus an additional one drop_cs. A non-zero value specifies that the device's chip select should be dropped at the end of the transfer, otherwise the chip select remains asserted. It is essential that the chip select be dropped in the final transfer of a transaction. If the protocol makes this difficult then cyg_spi_transaction_tick can be used to generate dummy ticks with all chip selects dropped.

If the device requires additional clock ticks in the middle of a transaction without being selected, cyg_spi_transaction_tick can be used. This will drop the device's chip select if necessary, then generate the appropriate number of ticks. The arguments are the same as for cyg_spi_tick.

cyg_spi_transaction_end should be called at the end of a transaction. It returns the SPI bus to a quiescent state, then unlocks it so that other threads can perform I/O.

A typical transaction might involve the following. First a command should be sent to the device, consisting of four bytes. The device will then respond with a single status byte, zero for failure, non-zero for success. If successful then the device can accept another n bytes of data, and will generate a 2-byte response including a checksum. The device's chip select should remain asserted throughout. The code for this would look something like:

```
#include <cyg/io/spi.h>
#include <cyg/hal/hal_io.h>    // Defines the SPI devices
…
    cyg_spi_transaction_begin(&hal_spi_eprom);
    // Interrupt-driven transfer, four bytes of command
    cyg_spi_transaction_transfer(&hal_spi_eprom, 0, 4, command, NULL, 0);
    // Read back the status
    cyg_spi_transaction_transfer(&hal_spi_eprom, 0, 1, NULL, status, 0);
    if (!status[0]) {
        // Command failed, generate some extra ticks to drop the chip select
        cyg_spi_transaction_tick(&hal_spi_eprom, 0, 1);
    } else {
        // Transfer the data, then read back the final status. The
        // chip select should be dropped at the end of this.
        cyg_spi_transaction_transfer(&hal_spi_eprom, 0, n, data, NULL, 0);
        cyg_spi_transaction_transfer(&hal_spi_eprom, 0, 2, NULL, status, 1);
        // Code for checking the final status should go here
    }
    // Transaction complete so clean up
    cyg_spi_transaction_end(&hal_spi_eprom);
```

A number of variations are possible. For example the command and status could be packed into the beginning and end of two 5-byte arrays, allowing a single transfer.

Device Configuration

The functions `cyg_spi_get_config` and `cyg_spi_set_config` can be used to examine and change parameters associated with SPI transfers. The only keys that are defined for all devices are `CYG_IO_GET_CONFIG_SPI_CLOCKRATE` and `CYG_IO_SET_CONFIG_SPI_CLOCKRATE`. Some types of device, for example MMC cards, support a range of clock rates. The cyg_spi_device structure will be initialized with a low clock rate. During system initialization the device will be queried for the optimal clock rate, and the cyg_spi_device should then be updated. The argument should be a clock rate in Hertz. For example the following code switches communication to 1Mbit/s:

```
cyg_uint32    new_clock_rate = 1000000;
cyg_uint32    len            = sizeof(cyg_uint32);
if (cyg_spi_set_config(&hal_mmc_device,
                       CYG_IO_SET_CONFIG_SPI_CLOCKRATE,
                       (const void *)&new_clock_rate, &len)) {
    // Error recovery code
}
```

If an SPI bus driver does not support the exact clock rate specified it will normally use the nearest valid one. SPI bus drivers may define additional keys appropriate for specific hardware. This means that the valid keys are not known by the generic code, and theoretically it is possible to use a key that is not valid for the SPI bus to which the device is attached. It is also possible that the argument used with one of these keys is invalid. Hence both `cyg_spi_get_config` and `cyg_spi_set_config` can return error codes. The return value will be 0 for success, non-zero for failure. The SPI bus driver's documentation should be consulted for further details.

Both configuration functions will lock the bus, in the same way as `cyg_spi_transfer`. Changing the clock rate in the middle of a transfer or manipulating other parameters would have unexpected consequences.

Name

Porting — Adding SPI support to new hardware

Description

Adding SPI support to an eCos port can take two forms. If there is already an SPI bus driver for the target hardware then both that driver and this generic SPI package `CYGPKG_IO_SPI` should be included in the ecos.db target entry. Typically the platform HAL will need to supply some platform-specific information needed by the bus driver. In addition the platform HAL should provide cyg_spi_device structures for every device attached to the bus. The exact details of this depend on the bus driver so its documentation should be consulted for further details. If there is no suitable SPI bus driver yet then a new driver package will have to be written.

Adding a Device

The generic SPI package `CYGPKG_IO_SPI` defines a data structure cyg_spi_device. This contains the information needed by the generic package, but not the additional information needed by a bus driver to interact with the device. Each bus driver will define a larger data structure, for example cyg_mcf52xx_qspi_device, which contains a cyg_spi_device as its first field. This is analogous to C++ base and derived classes, but without any use of virtual functions. The bus driver package should be consulted for the details.

During initialization an SPI bus driver may need to know about all the devices attached to that bus. For example it may need to know which cpu pins should be configured as chip selects rather than GPIO pins. To achieve this all device definitions should specify the particular bus to which they are attached, for example:

```
struct cyg_mcf52xx_qspi_device hal_spi_atod CYG_SPI_DEVICE_ON_BUS(0) =
{
    .spi_common.spi_bus = &cyg_mcf52xx_qspi_bus,
    …
};
```

The `CYG_SPI_DEVICE_ON_BUS` macro adds information to the structure which causes the linker to group all such structures in a single table. The bus driver's initialization code can then iterate over this table.

Adding Bus Support

An SPI bus driver usually involves a new hardware package. This needs to perform the following:

1. Define a device structure which contains a cyg_spi_device as its first element. This should contain all the information needed by the bus driver to interact with a device on that bus.

2. Provide functions for the following operations:

   ```
   spi_transaction_begin
   spi_transaction_transfer
   spi_transaction_tick
   spi_transaction_end
   spi_get_config
   spi_set_config
   ```

   These correspond to the main API functions, but can assume that the bus is already locked so no other thread will be manipulating the bus or any of the attached devices. Some of these operations may be no-ops.

3. Define a bus structure which contains a cyg_spi_bus as its first element. This should contain any additional information needed by the bus driver.

4. Optionally, instantiate the bus structure. The instance should have a well-known name since it needs to be referenced by the device structure initializers. For some drivers it may be best to create the bus inside the driver package. For other drivers it

may be better to leave this to the platform HAL or the application. It depends on how much platform-specific knowledge is needed to fill in the bus structure.

5. Create a HAL table for the devices attached to this bus.

6. Arrange for the bus to be initialized early on during system initialization. Typically this will happen via a prioritized static constructor with priority `CYG_INIT_BUS_SPI`. As part of this initialization the bus driver should invoke the `CYG_SPI_BUS_COMMON_INIT` macro on its cyg_spi_bus field.

7. Provide the appropriate documentation, including details of how the SPI device structures should be initialized.

There are no standard SPI testcases. It is not possible to write SPI code without knowing about the devices attached to the bus, and those are inherently hardware-specific.

Chapter 73. Freescale MCFxxxx ColdFire QSPI Bus Driver

Name

CYGPKG_DEVS_SPI_MCFxxxx_QSPI — eCos Support for the Freescale Coldfire QSPI Bus

Description

Several processors in the Freescale ColdFire family come with an on-chip SPI device, also known as QSPI. This package provides an eCos bus driver for that device. It implements the functionality defined by the generic SPI package CYGPKG_IO_SPI. The driver supports both polled and interrupt-driven transfers. Typical supported transfer rates range from 128KHz to 33MHz, although the exact details depend on the specific ColdFire processor being used and on the processor's clock speed. The hardware does not support DMA so large transfers at high transfer rates will consume much of the available cpu time.

This bus driver package does not instantiate any cyg_spi_bus structures. It is possible for a processor to have more than one SPI bus, so it is better to leave it to the processor HAL to define the bus or buses. Instead the bus driver package just provides functions and utility macros for use by the processor HAL. Similarly the bus driver package does not provide any cyg_spi_device structures. Exactly which devices are attached to the SPI bus is a characteristic of the platform so usually it is the platform HAL which provides the device instances.

Configuration Options

This SPI bus driver package should be loaded automatically when selecting a target containing a ColdFire processor with QSPI hardware, and it should never be necessary to load the package explicitly. If the application does not use any of the SPI functionality then all the SPI support code should be removed at link-time and the application does not suffer any overheads.

The package contains a single configuration option CYGHWR_DEVS_SPI_MCFxxxx_QSPI_MULTIPLE_BUSES. Usually this option should not be manipulated by application developers, instead it is set by the processor HAL. When the option is disabled the driver will optimize for the common case of a single bus.

The only other configuration options provided by this package relate to compiler flags.

Defining Buses

The header file cyg/io/mcfxxxx_qspi.h provides a utility macro CYG_MCFxxxx_QSPI_BUS to allow processor HALs to instantiate a bus. Existing HALs such as the MCF521x's will show how to use this macro.

Defining Devices

For most boards the platform HAL will create cyg_spi_device instances for all attached SPI devices, and will initialize the system so that the SPI-related processor pins are connected appropriately. Some development boards may not have any SPI devices but instead export the relevant signals to expansion connectors. In those cases it will be the responsibility of application code to create the device instances and manipulate the GPIO pins appropriately.

Device instances should take the form of a cyg_mcfxxxx_qspi_device structure, which contains a cyg_spi_device as its first field.

```
#include <cyg/io/mcfxxxx_qspi.h>
…
cyg_mcfxxxx_qspi_device hal_spi_atod CYG_SPI_DEVICE_ON_BUS(mcfxxxx_qspi) = {
    .qspi_common.spi_bus    = &cyg_mcfxxxx_qspi_bus,
    …
};
```

This defines a variable hal_spi_atod which can be used by other packages or by application code as an argument to the I/O functions provided by the generic SPI package CYGPKG_IO_SPI. A gcc extension, designated initializers, is used to fill in the *qspi_common.spi_bus* structure field. The structure contains a further seven fields which define exactly how to interact with the SPI device. Most of these fields are simply hardware register values, and the appropriate ColdFire User Manual should be consulted for full details of these registers. The header file `cyg/hal/hal_io.h` will provide #define's for the various bits, for example `HAL_MCFxxxx_QSPIx_QMR_MSTR` for the master mode bit of the QMR register.

qspi_qmr

When performing a transfer to this SPI device the bus driver will use the *qspi_qmr* field for the QSPI hardware's QMR register. The main fields in this register are:

MSTR This bit specifies that the QSPI hardware should operate in master mode. It must always be set.

BITS The data items transferred can range from 8 to 16 bits. For example, to specify 12-bit data items the *qspi_qmr* field should include `HAL_MCFxxxx_QSPIx_QMR_BITS_12`.

CPOL Clock polarity. The default is inactive-low, active-high. If the device requires the opposite polarity then `HAL_MCFxxxx_QSPIx_QMR_CPOL` should be specified.

CPHA Clock phase. The default is to capture data on the leading clock edge. If the device captures data on the trailing edge instead then `HAL_MCFxxxx_QSPIx_QMR_CPHA` should be specified.

BAUD Baud rate divider. This should be a small number, usually between 1 and 255, which controls the clock rate. The value to be used depends on the device's maximum clock rate, the specific processor used, and the processor's clock speed.

qspi_qdlyr

This field is used to set the QSPI delay register QDLYR when performing transfers to this device. It contains two delay fields, QCD and DTL, which can be used in conjunction with *qspi_qcr* for fine control over bus timing. Most devices do not have any special requirements here so a value of 0 can be used. The register also contains an SPE bit to start a transfer, but that bit is used by the bus driver and should not be set in the device structure.

qspi_qwr

This field is used to set the QWR register. Only one bit, CSIV, in this register may be defined. The other fields in the register are manipulated by the bus driver. Usually if the device has an active-low chip select then the CSIV bit should be set, otherwise the structure field should be 0. If a custom chip select control function is used then that may require different CSIV behaviour.

qspi_qcr

This is used to fill in the command RAM registers during a data transfer. It contains five fields. The CONT bit is not normally required but can provide additional control over the chip select. Note that some versions of the various ColdFire User Manuals give an incomplete description of this bit and the errata sheets should be consulted as well. The BITSE bit should be set if transfers involve data items which are not 8 bits. The DT and DSCK bits can be used to enable one or both delays in the QDLYR register. The QSPI_CS field consists of four bits for the four QSPI chip select pins. If all the devices connected to the SPI bus are active-high and each is connected directly to a chip select, then only of these bits should be set. If all the devices are active-low then only one of the bits should be clear.

With some hardware the QSPI_CS bits can be more complicated. For example consider an SPI bus with an active-high device attached to QSPI chip selects 0 and 1, and active-low devices attached to the other two chip selects. The device definition for the CS0 device should have the QWR CSIV bit clear. The QCR QSPI_CS bits should have bits 0, 2 and 3 set. Between transfers all chip select pins will be low. This will activate the devices on CS2 and CS3, but since there is no clock signal this is harmless. When a transfer happens CS0, CS2 and CS3 will all be high, and CS1 will remain low. This will activate the device on CS0, but leave the other three devices inactive. Hence only the specified device is active during a transfer.

If the hardware requires further control over the chip selects then the device definition can include a custom *qspi_cs_control* function.

There is no support for using different QCR values for different parts of a transfer, for example the first data item versus the rest of the transfer. Such functionality is rarely useful and would require extra complexity in the bus driver, including performance-critical parts.

qspi_qcr_tick

This is used to fill in the command RAM registers during a tick operation, when none of the devices should be active. Some devices need to see a certain number of clock signals even when their chip select is not active, or they will not operate correctly. The hardware fields are the same as for *qspi_qcr*. Usually the QSPI_CS bits will be all 0 or all 1, but some hardware may require a more complicated value.

qspi_tick_data

When performing a tick operation this field will be used as the data to be transferred. Usually the value will not matter because, by the definition of an SPI tick, none of the SPI devices will be selected.

qspi_cs_control

Some hardware may have chip select requirements which cannot be satisfied simply by setting the QWR CSIV and the QCR QSPI_CS bits. For example if there are more than four SPI devices then the surplus may have their chip selects connected to GPIO pins. Also some devices may require that the chip select remain asserted for the duration of a multi-transfer transaction, and that is not supported directly by the QSPI hardware. To cope with such cases it is possible to define a custom chip select control function.

Consider a simple SPI device on a board with a 64MHz MCF5282 processor. The device uses 8-bit data, default clock polarity and phase, can be driven at up to 10 MHz, does not require any special delays, has an active-high chip select, and is connected to the processor's QSPI CS0 pin. There are no other devices on the bus.

```
#include <cyg/io/mcfxxxx_qspi.h>
…
cyg_mcfxxxx_qspi_device hal_spi_dev0 CYG_SPI_DEVICE_ON_BUS(mcfxxxx_qspi) = {
    .qspi_common.spi_bus    = &cyg_mcfxxxx_qspi_bus,
    .qspi_qmr               = HAL_MCFxxxx_QSPIx_QMR_MSTR |
                              HAL_MCFxxxx_QSPIx_QMR_BITS_8 |
                              0x04,
    .qspi_qdlyr             = 0,
    .qspi_qwr               = 0,
    .qspi_qcr               = HAL_MCFxxxx_QSPIx_QCRn_QSPI_CS_CS0,
    .qspi_qcr_tick          = 0,
    .qspi_tick_data         = 0xFF,
    .qspi_cs_control        = (void (*)(cyg_mcfxxxx_qspi_device*, int)) 0
};
```

For a more complicated example, consider a board with an MCF5272 processor and an SPI device that involves 12-bit data items, uses inverted clock polarity and phase, can only be driven at the slowest clock rate, does not require any special delays or chip select logic, has an active-low chip select, and is connected to the processor's QSPI CS2 pin:

```
#include <cyg/io/mcfxxxx_qspi.h>
…
cyg_mcfxxxx_qspi_device hal_spi_dev2 CYG_SPI_DEVICE_ON_BUS(mcfxxxx_qspi) = {
    .qspi_common.spi_bus    = &cyg_mcfxxxx_qspi_bus,
    .qspi_qmr               = HAL_MCFxxxx_QSPIx_QMR_MSTR |
                              HAL_MCFxxxx_QSPIx_QMR_BITS_12 |
                              HAL_MCFxxxx_QSPIx_QMR_CPOL |
                              HAL_MCFxxxx_QSPIx_QMR_CPHA |
                              0xFF,
    .qspi_qdlyr             = 0,
    .qspi_qwr               = HAL_MCFxxxx_QSPIx_QWR_CSIV,
    .qspi_qcr               = HAL_MCFxxxx_QSPIx_QCRn_BITSE |
                              HAL_MCFxxxx_QSPIx_QCRn_QSPI_CS_CS0 |
                              HAL_MCFxxxx_QSPIx_QCRn_QSPI_CS_CS1 |
                              HAL_MCFxxxx_QSPIx_QCRn_QSPI_CS_CS3,
    .qspi_qcr_tick          = HAL_MCFxxxx_QSPIx_QCRn_BITSE |
                              HAL_MCFxxxx_QSPIx_QCRn_QSPI_CS_CS0 |
                              HAL_MCFxxxx_QSPIx_QCRn_QSPI_CS_CS1 |
                              HAL_MCFxxxx_QSPIx_QCRn_QSPI_CS_CS2 |
                              HAL_MCFxxxx_QSPIx_QCRn_QSPI_CS_CS3,
    .qspi_tick_data         = 0xFF,
    .qspi_cs_control        = (void (*)(cyg_mcfxxxx_qspi_device*, int)) 0
};
```

This definition assumes that there are no attached SPI devices with an active-high chip select. If there are such devices then the *qspi_qcr* and *qspi_qcr_tick* fields should be modified so that these devices are not activated at the wrong time.

The header file cyg/io/mcfxxxx_qspi.h provides a utility macro CYG_MCFxxxx_QSPI_DEVICE which can be used to instantiate a device. Essentially the macro just expands to a structure definition as above.

Advanced Chip Select Control

The ColdFire QSPI hardware provides support for controlling the chip select signals of up to four SPI devices. In many situations this support is adequate, but there are exceptions:

1. The QSPI chip select outputs may share processor pins with other on-chip ColdFire devices. For example on the mcf5272 the QSPI CS2 signal uses the same pin as the uart1 CTS signal, so if the application needs uart1 and hardware flow-control then that QSPI CS2 pin is no longer available.
2. If the hardware has more than four SPI devices then additional chip selects are needed.
3. With most SPI devices the chip select signal only needs to be asserted while I/O is taking place, but there are exceptions. For example interactions with an MMC card involve a sequence of transfers, and the chip select must remain asserted in between these transfers. Depending on thread priorities and other factors there may be a considerable delay between these transfers and the QSPI hardware does not provide any way of keeping a chip select asserted indefinitely.

The issue of insufficient chip selects can usually be handled by adding extra hardware, for example an external decoder chip possibly complemented by inverters if there is a mixture of active-high and active-low devices. This approach can be supported simply by programming the right values for *qspi_qcr* and *qspi_qcr_tick*, but the cost of the extra hardware may be unacceptable. An alternative approach is to use one or more of the processor's GPIO pins to control the extra devices.

The issue of persistent chip selects can be handled in one of two main ways. A GPIO pin can be used to control the chip select, bypassing the QSPI support. Alternatively the QWR CSIV bit can be used in an inverted sense, to activate an SPI device rather than to define the inactive state.

To support these variations an arbitrary chip select control function can be specified for a device. Such a function takes two arguments. The first is a pointer to the SPI device, possibly allowing the function to be shared between multiple devices. The second is one of the following:

CYG_MCFxxxx_QSPI_CS_INIT

> During system initialization the QSPI bus driver will iterate over all the attached SPI devices. If a device has a *qspi_cs_control* function then this will be invoked. A typical action would be to configure a GPIO pin. Note that these calls happen quite early during system initialization so other subsystems like standard I/O may not be set up yet.

CYG_MCFxxxx_QSPI_CS_ASSERT

> This is used to assert the chip select, in other words to set the chip select to low for an active-low device or high for an active-high device. It will be called at the start of any transfer, unless the previous transfer has left the chip select asserted.

CYG_MCFxxxx_QSPI_CS_DROP

> This is used to deassert the chip select. It will be called at the end of any transfer that specifies drop_cs. It will also be called at the start of a tick operation.

To support persistent chip selects via the CSIV signal the bus driver package provides two chip control functions cyg_mcfxxxx_qspi_csiv_cs_control_active_high and cyg_mcfxxxx_qspi_csiv_cs_control_active_low. To use these with say an active-low device:

1. The *qspi_qwr* field should be set to `HAL_MCFxxxx_QSPIx_QWR_CSIV`, so the chip select is high when there is no I/O taking place.

2. The *qspi_cs_control* field should be set to `&cyg_mcfxxxx_qspi_csiv_cs_control_active_low`. This function will be invoked by the bus driver to assert or drop the signal (initialization is a no-op).

3. The QSPI_CS bits in the *qspi_qcr* field still have the usual meaning.

4. At the start of a transfer `cyg_mcfxxxx_qspi_csiv_cs_control_active_low` will clear the QWR CSIV bit. There is no I/O taking place yet so all chip select outputs will switch to low, activating all active-low devices. This is generally harmless since there is no clock signal.

5. When the I/O actually starts the *qspi_qcr* field will be used, deactivating all devices except the current one.

6. At the end of each individual transfer the chip selects will revert to their inactive state, which because of the CSIV setting means low. Again this will activate all active-low devices, but there is no clock signal so no I/O takes place.

7. For the last transfer of a transaction or for a tick operation `cyg_mcfxxxx_qspi_cs_control_active_low` will be invoked again with a DROP argument. It will reset the QWR CSIV bit to 1, deactivating all devices.

The overall effect is a persistent chip select with the desired polarity, using just the QSPI hardware facilities rather than a GPIO pin.

Chapter 74. Microchip (Atmel) USART-as-SPI Bus Driver

Name

CYGPKG_DEVS_SPI_ATMEL_USPI — eCos Support for the Microchip (Atmel) USART-as-SPI Bus

Description

The Microchip (previously Atmel) SAM E70, S70, V70 and V71 processors do come with on-chip SPI controllers, but also with the ability to configure the on-chip USART controllers as SPI masters. This package provides an eCos bus driver for those USART-as-SPI interfaces. The CYGPKG_DEVS_SPI_ARM_AT91 package provides the bus driver support for the standard SPI controllers.

This package implements the functionality defined by the generic SPI package CYGPKG_IO_SPI. The driver supports both polled and DMA-driven transfers. Typical supported transfer rates range from 3KHz to 25MHz, although the exact details depend on the specific processor configuration.

This bus driver package does not instantiate any cyg_spi_device structures. Exactly which devices are attached to the SPI bus is a characteristic of the platform so usually it is the platform HAL which provides the device instances.

Configuration Options

This SPI bus driver package should be loaded automatically when selecting a target containing a suitable SAM processor with USART-as-SPI hardware, and it should never be necessary to load the package explicitly. If the application does not use any of the SPI functionality then all the SPI support code will be removed at link-time and the application does not suffer any overheads.

The package contains a single configuration option CYGNUM_DEVS_SPI_ATMEL_USPI_BAUD_RATE_MAX. Usually this option will not need to be manipulated by application developers, since it purely sets an upper bound on the acceptable device baudrate that will be accepted, though may not be achievable depending on the CPU configuration.

The only other configuration options provided by this package relate to compiler flags.

Instantiating Buses

In the platform CDL, when this CYGPKG_DEVS_SPI_ATMEL_USPI is configured, an implements entry should be provided for the USART-as-SPI bus to instantiate.

Note

Currently the CYGPKG_HAL_CORTEXM_SAM HAL allows for the dedicated SPI buses numbered 0 and 1, and the USART-as-SPI buses numbered 2..4.

For example, the samx70_ek platform CDL provides for SPI bus#2 by declaring:

```
implements CYGINT_HAL_CORTEXM_SAM_SPI2
```

When a bus is implemented the Chip-Select GPIOs associated with the bus (at **least** one) should be provided by the platform CDL defining a corresponding CYGHWR_HAL_CORTEXM_SAM_SPI*x*_CS_GPIOS definition for the bus in question.

For the SPI bus#2 example above the platform CDL would also define CYGHWR_HAL_CORTEXM_SAM_SPI2_CS_GPIOS option and provide the list of chip-select pins, e.g.:

```
default_value { "SPI_CS(B,3)" }
```

Defining Devices

For most boards the platform HAL will create cyg_spi_device instances for all attached SPI devices, and will initialize the system so that the SPI-related processor pins are connected appropriately.

Device instances should take the form of a cyg_spi_atmel_device_t structure, which contains a cyg_spi_device as its first field. For example, for a device on bus#2:

```
#include <cyg/io/spi_atmel_uspi.h>
…
cyg_spi_atmel_device_t hal_spi_example CYG_SPI_DEVICE_ON_BUS(2) = {
    .spi_device.spi_bus = &cyg_spi_atmel_uspi_bus2,
    …
};
```

This defines a variable `hal_spi_example` which can be used by other packages or by application code as an argument to the I/O functions provided by the generic SPI package `CYGPKG_IO_SPI`. A gcc extension, designated initializers, is used to fill in the *`spi_device.spi_bus`* structure field. The structure contains a further **seven** fields which define exactly how to interact with the specific SPI device.

`dev_num`

This is the index into the `CYGHWR_HAL_CORTEXM_SAM_SPIx_CS_GPIOS` list of GPIOs for the specific Chip-Select GPIO used to select access to the device on the relevant SPI bus.

`cl_pol`

The clock polarity (0 or 1).

`cl_pha`

The clock phase (0 or 1).

`cl_brate`

The `SCK` baudrate used when communicating with the device.

`cs_up_udly`

Required microsecond delay between CS active and the transfer starting.

`cs_dw_udly`

Required microsecond delay between transfer ending and CS going inactive.

`tr_bt_udly`

Minimum microsecond delay between two transfers (between CS inactive to active again).

For example, the following instantaties a AT25080 serial EEPROM memory device on SPI bus#2:

```
#include <cyg/io/spi_atmel_uspi.h>
…
cyg_spi_atmel_device_t cyg_aardvark_at25080 CYG_SPI_DEVICE_ON_BUS(2) = {
    .spi_device.spi_bus = &cyg_spi_atmel_uspi_bus2,
    .dev_num     = 0,       // CS#0
    .cl_pol      = 0,       // Clock polarity
    .cl_pha      = 1,       // Clock phase
    .cl_brate    = 2000000, // Clock baud rate 2MHz. At 3.3v 2.1MHz is allowed for this part.
    .cs_up_udly  = 1,       // Tcss (CS setup time) for this part at 3.3V is 250ns.
    .cs_dw_udly  = 1,       // Tcsh (CS hold time) for this part at 3.3V is 250ns.
    .tr_bt_udly  = 1        // Tcs (CS high time) for this part at 3.3V is 250ns.
};
```

Part XXIII. I^{2}C Support

Documentation for drivers of this type is often integrated into the eCos board support documentation. You should review the documentation for your target board for details. Standalone and more generic drivers are documented in the following sections.

Table of Contents

Chapter 75. I²C Support

Name

Overview — eCos Support for I²C, the Inter IC Bus

Description

The Inter IC Bus (I²C) is one of a number of serial bus technologies. It can be used to connect a processor to one or more peripheral chips, for example analog-to-digital convertors or real time clocks, using only a small number of pins and PCB tracks. The technology was originally developed by Philips Semiconductors but is supported by many other vendors. The bus specification is freely available.

In a typical I²C system the processor acts as the I²C bus master. The peripheral chips act as slaves. The bus consists of just two wires: SCL carries a clock signal generated by the master, and SDA is a bi-directional data line. The normal clock frequency is 100KHz. Each slave has a 7-bit address. With some chips the address is hard-wired, and it is impossible to have two of these chips on the same bus. With other chips it is possible to choose between one of a small number of addresses by connecting spare pins to either VDD or GND.

An I²C data transfer involves a number of stages:

1. The bus master generates a start condition, a high-to-low transition on the SDA line while SCL is kept high. This signalling cannot occur during data transfer.

2. The bus master clocks the 7-bit slave address onto the SDA line, followed by a direction bit to distinguish between reads and writes.

3. The addressed device acknowledges. If the master does not see an acknowledgement then this suggests it is using the wrong address for the slave device.

4. If the master is transmitting data to the slave then it will send this data one byte at a time. The slave acknowledges each byte. If the slave is unable to accept more data, for example because it has run out of buffer space, then it will generate a nack and the master should stop sending.

5. If the master is receiving data from the slave then the slave will send this data one byte at a time. The master should acknowledge each byte, until the last one. When the master has received all the data it wants it should generate a nack and the slave will stop sending. This nack is essential because it causes the slave to stop driving the SDA line, releasing it back to the master.

6. It is possible to switch direction in a single transfer, using what is known as a repeated start. This involves generating another start condition, sending the 7-bit address again, followed by a new direction bit.

7. At the end of a transfer the master should generate a stop condition, a low-to-high transition on the SDA line while SCL is kept high. Again this signalling does not occur at other times.

There are a number of extensions. The I²C bus supports multiple bus masters and there is an arbitration procedure to allow a master to claim the bus. Some devices can have 10-bit addresses rather than 7-bit addresses. There is a fast mode operating at 400KHz instead of the usual 100KHz, and a high-speed mode operating at 3.4MHz. Currently most I²C-based systems do not involve any of these extensions.

At the hardware level I²C bus master support can be implemented in one of two ways. Some processors provide a dedicated I²C device, with the hardware performing much of the work. On other processors the I²C device is implemented in software, by bit-banging some GPIO pins. The latter approach can consume a significant number of cpu cycles, but is often acceptable because only occasional access to the I²C devices is needed.

eCos Support for I²C

The eCos I²C support for any given platform is spread over a number of different packages:

- This package, `CYGPKG_IO_I2C`, exports a generic API for accessing devices attached to an I²C bus. This API handles issues such as locking between threads. The package does not contain any hardware-specific code. Instead it will use a separate I²C bus driver to handle the hardware, and it defines the interface that such bus drivers should provide. The package only provides support for a bus master, not for acting as a slave device.

  `CYGPKG_IO_I2C` also provides the hardware-independent portion of a bit-banged bus implementation. This needs to be complemented by a hardware-specific function that actually manipulates the SDA and SCL lines.

- If the processor has a dedicated I²C device then there will be a bus driver package for that hardware. The processor may be used on many different platforms and the same bus driver can be used on each one. The actual I²C devices attached to the bus will vary from one platform to the next.

- The generic API depends on cyg_i2c_device data structures. These contain the information needed by a bus driver, for example the device address. Usually the data structures are provided by the platform HAL since it is that package which knows about all the devices on the platform.

  On some development boards the I²C lines are brought out to expansion connectors, allowing end users to add extra devices. In such cases the platform HAL may not know about all the devices on the board. Data structures for the additional devices can instead be supplied by application code.

- If the board uses a bit-banged bus then typically the platform HAL will also instantiate the bus instance, providing the function that handles the low-level SDA and SCL manipulation. Usually this code cannot be shared because each board may use different GPIO pins for driving SCL and SDA, so the code belongs in the platform HAL rather than in a separate package.

- Some types of I²C devices may have their own driver package. For example a common type of I²C device is a battery-backed wallclock, and eCos defines how these devices should be supported. Such an I²C device will have its own wallclock device driver and the device will not be accessed directly by application code. For other types of device eCos does not define an API and there will not be separate device driver packages. Instead application code is expected to use the cyg_i2c_device structures directly to access the hardware.

Typically all appropriate packages will be loaded automatically when you configure eCos for a given platform. If the application does not use any of the I²C I/O facilities, directly or indirectly, then linker garbage collection should eliminate all unnecessary code and data. All necessary initialization should happen automatically. However the exact details may depend on the platform, so the platform HAL documentation should be checked for further details.

There is one important exception to this: if the I²C devices are attached to an expansion connector then the platform HAL will not know about these devices. Instead more work will have to be done by application code.

Name

I²C Functions — allow applications and other packages to access I²C devices

Synopsis

#include <cyg/io/i2c.h>

`cyg_uint32` **`cyg_i2c_tx`**(*`device`*, *`tx_data`*, *`count`*);

`cyg_uint32` **`cyg_i2c_rx`**(*`device`*, *`rx_data`*, *`count`*);

`void` **`cyg_i2c_transaction_begin`**(*`device`*);

`cyg_bool` **`cyg_i2c_transaction_begin_nb`**(*`device`*);

`cyg_uint32` **`cyg_i2c_transaction_tx`**(*`device`*, *`send_start`*, *`tx_data`*, *`count`*, *`send_stop`*);

`cyg_uint32` **`cyg_i2c_transaction_rx`**(*`device`*, *`send_start`*, *`rx_data`*, *`count`*, *`send_nack`*, *`send_stop`*);

`void` **`cyg_i2c_transaction_stop`**(*`device`*);

`void` **`cyg_i2c_transaction_end`**(*`device`*);

Description

All I²C functions take a pointer to a cyg_i2c_device structure as their first argument. These structures are usually provided by the platform HAL. They contain the information needed by the I²C bus driver to interact with the device, for example the device address.

An I²C transaction involves the following stages:

1. Perform thread-level locking on the bus. Only one thread at a time is allowed to access an I²C bus. This eliminates the need to worry about locking at the bus driver level. If a platform involves multiple I²C buses then each one will have its own lock.
2. Generate a start condition, send the address and direction bit, and wait for an acknowledgement from the addressed device.
3. Either transmit data to or receive data from the addressed device.
4. The previous two steps may be repeated several times, allowing data to move in both directions during a single transfer.
5. Generate a stop condition, ending the current data transfer. It is now possible to start another data transfer while the bus is still locked, if desired.
6. End the transaction by unlocking the bus, allowing other threads to access other devices on the bus.

The simple functions `cyg_i2c_tx` and `cyg_i2c_rx` perform all these steps in a single call, making them suitable for many I/O operations. The alternative transaction-oriented functions provide greater control when appropriate, for example if a repeated start is necessary for a bi-directional data transfer.

With the exception of `cyg_i2c_transaction_begin_nb` all the functions will block until completion. The tx routines will return 0 if the specified device does not respond to its address, or the number of bytes actually transferred. This may be less than the number requested if the device sends an early nack, for example because it has run out of buffer space. The rx routines will return 0 or the number of bytes received. Usually this will be the same as the `count` parameter. A slave device has no way of indicating to the master that no more data is available, so the rx operation cannot complete early.

I²C operations should always be performed at thread-level or during system initialization, and not inside an ISR or DSR. This greatly simplifies locking. Also a typical ISR or DSR should not perform a blocking operation such as an I²C transfer.

Simple Transfers

`cyg_i2c_tx` and `cyg_i2c_rx` can be used for simple data transfers. They both go through the following steps: lock the bus, generate the start condition, send the device address and the direction bit, either send or receive the data, generate the stop condition, and unlock the bus. At the end of a transfer the bus is back in its idle state, ready for the next transfer.

`cyg_i2c_tx` returns the number of bytes actually transmitted. This may be 0 if the device does not respond when its address is sent out. It may be less than the number of bytes requested if the device generates an early nack, typically because it has run out of buffer space.

`cyg_i2c_rx` returns 0 if the device does not respond when its address is sent out, or the number of bytes actually received. Usually this will be the number of bytes requested because an I²C slave device has no way of aborting an rx operation early.

Transactions

To allow multiple threads to access devices on the I²C some locking is required. This is encapsulated inside transactions. The `cyg_i2c_tx` and `cyg_i2c_rx` functions implicitly use such transactions, but the functionality is also available directly to application code. Amongst other things transactions can be used for more complicated interactions with I²C devices, in particular ones involving repeated starts.

`cyg_i2c_transaction_begin` must be used at the start of a transaction. This performs thread-level locking on the bus, blocking if it is currently in use by another thread.

`cyg_i2c_transaction_begin_nb` is a non-blocking variant, useful for threads which cannot afford to block for an indefinite period. If the bus is currently locked the function returns false immediately. If the bus is not locked then it acts as `cyg_i2c_transaction_begin` and returns true.

Once the bus has been locked it is possible to perform one or more data transfers by calling `cyg_i2c_transaction_tx`, `cyg_i2c_transaction_rx` and `cyg_i2c_transaction_stop`. Code should ensure that a stop condition has been generated by the end of a transaction.

Once the transaction is complete `cyg_i2c_transaction_end` should be called. This unlocks the bus, allowing other threads to perform I²C I/O to devices on the same bus.

As an example consider reading the registers in an FS6377 programmable clock generator. The first step is to write a byte 0 to the device, setting the current register to 0. Then a repeated start condition should be generated and it is possible to read the 16 byte-wide registers, starting with the current one. Typical code for this might look like:

```
cyg_uint8  tx_data[1];
cyg_uint8  rx_data[16];

cyg_i2c_transaction_begin(&hal_alaia_i2c_fs6377);
tx_data[0] = 0x00;
cyg_i2c_transaction_tx(&hal_alaia_i2c_fs6377,
                       true, tx_data, 1, false);
cyg_i2c_transaction_rx(&hal_alaia_i2c_fs6377,
                       true, rx_data, 16, true, true);
cyg_i2c_transaction_end(&hal_alaia_i2c_fs6377);
```

Here `hal_alaia_i2c_fs6377` is a cyg_i2c_device structure provided by the platform HAL. A transaction is begun, locking the bus. Then there is a transmit for a single byte. This transmit involves generating a start condition and sending the address and direction bit, but not a stop condition. Next there is a receive for 16 bytes. This also involves a start condition, which the device will interpret as a repeated start because it has not yet seen a stop. The start condition will be followed by the address and direction bit, and then the device will start transmitting the register contents. Once all 16 bytes have been received the rx routine will send a nack rather than an ack, halting the transfer, and then a stop condition is generated. Finally the transaction is ended, unlocking the bus.

The arguments to `cyg_i2c_transaction_tx` are as follows:

const cyg_i2c_device* `device`

This identifies the I²C device that should be used.

cyg_bool `send_start`

If true, generate a start condition and send the address and direction bit. If false, skip those steps and go straight to transmitting the actual data. The latter can be useful if the data to be transmitted is spread over several buffers. The first tx call will involve generating the start condition but subsequent tx calls can skip this and just continue from the previous one.

`send_start` must be true if the tx call is the first operation in a transaction, or if the previous call was an rx or stop.

const cyg_uint8* `tx_data`
cyg_uint32 `count`

These arguments specify the data to be transmitted to the device.

cyg_bool `send_stop`

If true, generate a stop condition at the end of the transmit. Usually this is done only if the transmit is the last operation in a transaction.

The arguments to `cyg_i2c_transaction_rx` are as follows:

const cyg_i2c_device* `device`

This identifies the I²C device that should be used.

cyg_bool `send_start`

If true, generate a start condition and send the address and direction bit. If false, skip those steps and go straight to receiving the actual data. The latter can be useful if the incoming data should be spread over several buffers. The first rx call will involve generating the start condition but subsequent rx calls can skip this and just continue from the previous one. Another use is for devices which can send variable length data, consisting of an initial length and then the actual data. The first rx will involve generating the start condition and reading the length, a subsequent rx will then just read the data.

`send_start` must be true if the rx call is the first operation in a transaction, if the previous call was a tx or stop, or if the previous call was an an rx and the `send_nack` flag was set.

cyg_uint8* `rx_data`
cyg_uint32 `count`

These arguments specify how much data should be received and where it should be placed.

cyg_bool `send_nack`

If true generate a nack instead of an ack for the last byte received. This causes the slave to end its transmit. The next operation should either involve a repeated start or a stop. `send_nack` should be set to false only if `send_stop` is also false, the next operation will be another rx, and that rx does not specify `send_start`.

cyg_bool `send_stop`

If true, generate a stop condition at the end of the transmit. Usually this is done only if the transmit is the last operation in a transaction.

The final transaction-oriented function is `cyg_i2c_transaction_stop`. This just generates a stop condition. It should be used if the previous operation was a tx or rx that, for some reason, did not set the `send_stop` flag. A stop condition must be generated before the transaction is ended.

Initialization

The generic package `CYGPKG_IO_I2C` arranges for all I²C bus devices to be initialized via a single prioritized C++ static constructor. This constructor will run early on during system startup, before any application code, with priority `CYG_INIT_BUS_I2C`. Other code should not try to access any of the I²C devices until after the buses have been initialized.

Name

Porting — Adding I²C support to new hardware

Description

Adding I²C support to an eCos port involves a number of steps. The generic I²C package `CYGPKG_IO_I2C` should be included in the appropriate ecos.db target entry or entries. Next cyg_i2c_device structures should be provided for every device on the bus. Usually this is the responsibility of the platform HAL. In the case of development boards where the I²C SDA and SCL lines are accessible via an expansion connector, more devices may have been added and it will be the application's responsibility to provide the structures. Finally there is a need for one or more cyg_i2c_bus structures. Amongst other things these structures provide functions for actually driving the bus. If the processor has dedicated I²C hardware then this structure will usually be provided by a device driver package. If the bus is implemented by bit-banging then the bus structure will usually be provided by the platform HAL.

Adding a Device

The eCos I²C API works in terms of cyg_i2c_device structures, and these provide the information needed to access the hardware. A cyg_i2c_device structure contains the following fields:

cyg_i2c_bus* `i2c_bus`
: This specifies the bus which the slave device is connected to. Most boards will only have a single I²C bus, but multiple buses are possible.

cyg_uint16 `i2c_address`
: For most devices this will be the 7-bit I²C address the device will respond to. There is room for future expansion, for example to support 10-bit addresses.

cyg_uint16 `i2c_flags`
: This field is not used at present. It exists for future expansion, for example to allow for fast mode or high-speed mode, and incidentally pads the structure to a 32-bit boundary.

cyg_uint32 `i2c_delay`
: This holds the clock period which should be used when interacting with the device, in nanoseconds. Usually this will be 10000 ns, corresponding to a 100KHz clock, and the header `cyg/io/i2c.h` provides a `#define` `CYG_I2C_DEFAULT_DELAY` for this. Sometimes it may be desirable to use a slower clock, for example to reduce noise problems.

The normal way to instantiate a `cyg_i2c_device` structure uses the `CYG_I2C_DEVICE` macro, also provided by `cyg/io/i2c.h`:

```
#include <cyg/io/i2c.h>

CYG_I2C_DEVICE(cyg_i2c_wallclock_ds1307,
               &hal_alaia_i2c_bus,
               0x68,
               0x00,
               CYG_I2C_DEFAULT_DELAY);

CYG_I2C_DEVICE(hal_alaia_i2c_fs6377,
               &hal_alaia_i2c_bus,
               0x58,
               0x00,
               CYG_I2C_DEFAULT_DELAY);
```

The arguments to the macro are the variable name, an I²C bus pointer, the device address, the flags field, and the delay field. The above code fragment defines two I²C device variables, `cyg_i2c_wallclock_ds1307` and `hal_alaia_i2c_fs6377`, which can be used for the first argument to the `cyg_i2c_` functions. Both devices are on the same bus. The device addresses are 0x68 and 0x58 respectively, and the devices do not have any special requirements.

When the platform HAL provides these structures it should also export them for use by the application and other packages. Usually this involves an entry in `cyg/hal/plf_io.h`, which gets included automatically via one of the main exported HAL header files cyg/hal/hal_io.h. Unfortunately exporting the structures directly can be problematical because of circular dependencies between the I²C header and the HAL headers. Instead the platform HAL should define a macro HAL_I2C_EXPORTED_DEVICES:

```
# define HAL_I2C_EXPORTED_DEVICES                                   \
    extern cyg_i2c_bus                 hal_alaia_i2c_bus;           \
    extern cyg_i2c_device              cyg_i2c_wallclock_ds1307;    \
    extern cyg_i2c_device              hal_alaia_i2c_fs6377;
```

This macro gets expanded automatically by cyg/io/i2c.h once the data structures themselves have been defined, so application code can just include that header and all the buses and devices will be properly exported and usable.

There is no single convention for naming the I²C devices. If the device will be used by some other package then typically that specifies the name that should be used. For example the DS1307 wallclock driver expects the I²C device to be called cyg_i2c_wallclock_ds1307, so failing to observe that convention will lead to compile-time and link-time errors. If the device will not be used by any other package then it is up to the platform HAL to select the name, and as long as reasonable care is taken to avoid name space pollution the exact name does not matter.

Bit-banged Bus

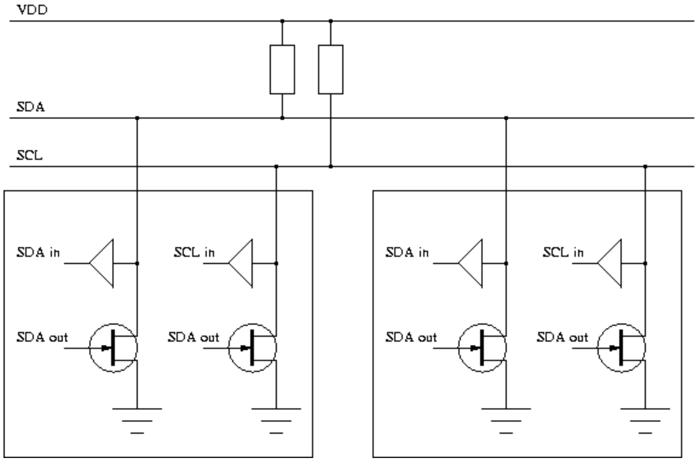
Some processors come with dedicated I²C hardware. On other hardware the I²C bus involves simply connecting some GPIO pins to the SCL and SDA lines and then using software to implement the I²C protocol. This is usually referred to as bit-banging the bus. The generic I²C package CYGPKG_IO_I2C provides the main code for a bit-banged implementation, requiring one platform-specific function that does the actual GPIO pin manipulation. This function is usually hardware-specific because different boards will use different pins for the I²C bus, so typically it is left to the platform HAL to provide this function and instantiate the I²C bus object. There is no point in creating a separate package for this because the code cannot be re-used for other platforms.

Instantiating a bit-banged I²C bus requires the following:

```
#include <cyg/io/i2c.h>

static cyg_bool
hal_alaia_i2c_bitbang(cyg_i2c_bus* bus, cyg_i2c_bitbang_op op)
{
    cyg_bool result    = 0;
    switch(op) {
        …
    }
    return result;
}

CYG_I2C_BITBANG_BUS(hal_alaia_i2c_bus, &hal_alaia_i2c_bitbang);
```

This gives a structure hal_alaia_i2c_bus which can be used when defining the cyg_i2c_device structures. The second argument specifies the function which will do the actual bit-banging. It takes two arguments. The first identifies the bus, which can be useful if the hardware has multiple I²C buses. The second specifies the bit-bang operation that should be performed. To understand these operations consider how I²C devices should be wired up according to the specification:

Figure 75.1. I²C wiring specification



Master and slave devices are interfaced to the bus in exactly the same way. The default state of the bus is to have both lines high via the pull-up resistors. Any device on the bus can lower either line, when allowed to do so by the protocol. Usually the SDA line only changes while SCL is low, but the start and stop conditions involve SDA changing while SCL is high. All devices have the ability to both read and write both lines. In reality not all bit-banged hardware works quite like this. Instead just two GPIO pins are used, and these are switched between input and output mode as required.

The bitbang function should support the following operations:

`CYG_I2C_BITBANG_INIT`

This will be called during system initialization, as a side effect of a prioritized C++ static constructor. The bitbang function should ensure that both SCL and SDA are driven high.

`CYG_I2C_BITBANG_SCL_HIGH`
`CYG_I2C_BITBANG_SCL_LOW`
`CYG_I2C_BITBANG_SDA_HIGH`
`CYG_I2C_BITBANG_SDA_LOW`

These operations simply set the appropriate lines high or low.

`CYG_I2C_BITBANG_SCL_HIGH_CLOCKSTRETCH`

In its simplest form this operation should simply set the SCL line high, indicating that the data on the SDA line is stable. However there is a complication: if a device is not ready yet then it can throttle back the master by keeping the SCL line low. This is known as clock-stretching. Hence for this operation the bitbang function should allow the SCL line to float high, then poll it until it really has become high. If a single pin is used for the SCL line then this pin should be turned back into a high output at the end of the call.

`CYG_I2C_BITBANG_SCL_LOW_SDA_INPUT`

This is used when there is a change of direction and the slave device is about to start driving the SDA line. This can be significant if a single pin is used to handle both input and output of SDA, to avoid a situation where both the master and the slave are driving the SDA line for an extended period of time. The operation combines dropping the SCL line and switching SDA to an input in an atomic or near-atomic operation.

`CYG_I2C_BITBANG_SDA_READ`

The SDA line is currently set as an input and the bitbang function should sample and return the current state.

The bitbang function returns a boolean. For most operations this return value is ignored. For `CYG_I2C_BITBANG_SDA_READ` it should be the current level of the SDA line.

Depending on the hardware some care may have to be taken when manipulating the GPIO pins. Although the I²C subsystem performs the required locking at the bus level, the device registers controlling the GPIO pins may get used by other subsystems or by the application. It is the responsibility of the bitbang function to perform appropriate locking, whether via a mutex or by briefly disabling interrupts around the register accesses.

Full Bus Driver

If the processor has dedicated I²C hardware then usually this will involve a separate device driver package in the `devs/i2c` hierarchy of the eCos component repository. That package should also be included in the appropriate ecos.db target entry or entries. The device driver may exist already, or it may have to be written from scratch.

A new I²C driver basically involves creating an cyg_i2c_bus structure. The device driver should supply the following fields:

`i2c_init_fn`
: This function will be called during system initialization to set up the I²C hardware. The generic I²C code creates a static object with a prioritized constructor, and this constructor will invoke the init functions for the various I²C buses in the system.

`i2c_tx_fn`
`i2c_rx_fn`
`i2c_stop_fn`
: These functions implement the core I²C functionality. The arguments and results are the same as for the transaction functions `cyg_i2c_transaction_tx`, `cyg_i2c_transaction_rx` and `cyg_i2c_transaction_stop`.

void* `i2c_extra`
: This field holds any extra information that may be needed by the device driver. Typically it will be a pointer to some driver-specific data structure.

To assist with instantiating a cyg_i2c_bus object the header file `cyg/io/i2c.h` provides a macro. Typical usage would be:

```
struct xyzzy_data {
    …
} xyzzy_object;

static void
xyzzy_i2c_init(struct cyg_i2c_bus* bus)
{
    …
}

static cyg_uint32
xyzzy_i2c_tx(const cyg_i2c_device* dev,
             cyg_bool send_start,
             const cyg_uint8* tx_data, cyg_uint32 count,
             cyg_bool send_stop)
{
    …
}

static cyg_uint32
xyzzy_i2c_rx(const cyg_i2c_device* dev,
             cyg_bool send_start,
             cyg_uint8* rx_data, cyg_uint32 count,
             cyg_bool send_nack, cyg_bool send_stop)
{
    …
}

static void
xyzzy_i2c_stop(const cyg_i2c_device* dev)
{
    …
}

CYG_I2C_BUS(cyg_i2c_xyzzy_bus,
            &xyzzy_i2c_init,
            &xyzzy_i2c_tx,
            &xyzzy_i2c_rx,
            &xyzzy_i2c_stop,
            (void*) &xyzzy_object);
```

The generic I^{2}C code contains these functions for a bit-banged I^{2}C bus device. It can be used as a starting point for new drivers. Note that the bit-bang code uses the `i2c_extra` field to hold the hardware-specific bitbang function rather than a pointer to some data structure.

Chapter 76. Freescale MCFxxxx ColdFire I^2C Bus Driver

Name

`CYGPKG_DEVS_I2C_MCFxxxx` — eCos Support for the Freescale Coldfire I^{2}C Bus

Description

Several processors in the Freescale ColdFire family come with one or more on-chip I^{2}C bus devices. This package provides an eCos I^{2}C bus driver. It was originally developed on an MCF5280 but should work with any ColdFire processor that uses a compatible bus device. The driver implements the functionality defined by the generic I^{2}C package `CYGPKG_IO_I2C`.

Caution

The hardware does not support DMA or fifos, so usually a transfer will involve an interrupt for every byte transferred. Since the I^{2}C bus typically runs at 100KHz large transfers will consume much of the available cpu time.

This package does not provide any cyg_i2c_bus structures. The number of I^{2}C buses varies between ColdFire processors. If multiple buses are available then exactly which one(s) are in use on a given hardware platform depends entirely on that platform. The desired I^{2}C bus speed also depends on the platform, and there may be other issues such as how the processor pins should be set up. Hence it is left to other code, usually the processor HAL, to instantiate the bus structure(s). This driver package supplies the necessary functions and utility macros. Similarly this package does not provide any cyg_i2c_device structures. Which I^{2}C devices are hooked up to which I^{2}C bus is entirely a characteristic of the hardware platform, so it is up to the platform HAL to instantiate the necessary structures.

The driver will operate in interrupt-driven mode if interrupts are enabled when a transfer is initiated. Otherwise it will operate in polled mode. This allows the driver to be used in a variety of configurations including inside RedBoot.

Configuration Options

The I^{2}C bus driver package should be loaded automatically when selecting a target containing a suitable ColdFire processor, and it should never be necessary to load the package explicitly. If the application does not use any of the I^{2}C functionality, directly or indirectly, then all the I^{2}C code should be removed at link-time and the application does not suffer any overheads.

By default the driver assumes a single I^{2}C bus and optimizes for that case. For example options like the ISR vector and priority are handled by compile-time `#define`'s in the platform HAL's exported header files rather than by per-bus structure fields. This helps to reduce both code and data overheads. If the driver should support multiple I^{2}C buses then `CYGHWR_DEVS_I2C_MCFxxxx_MULTIPLE_BUSES` should be enabled. Typically this will be done by the processor HAL using a CDL requires property. If bus instantiation happens outside the processor HAL and hence the HAL's header files do not provide the appropriate definitions, then this configuration option should also be defined.

The only other configuration options in this package provide control over the compiler flags used to build the driver code.

Defining the Bus and Devices

For most hardware targets the processor HAL will instantiate the cyg_i2c_bus and the platform HAL will instantiate the cyg_i2c_device structures. Between them they will also initialize the hardware so that the I^{2}C-related pins are connected appropriately. Some development boards have no I^{2}C devices, but the I^{2}C bus signals are accessible via an expansion connector and I^{2}C devices can be put on a daughter board. In such cases it may be necessary for the application to instantiate the device structures.

To facilitate bus instantiation the header file `cyg/io/i2c_mcfxxxx.h` provides a utility macro `CYG_MCFxxxx_I2C_BUS`. This takes six parameters:

1. The name of the bus, for example `hal_dnp5280_i2c_bus`. This name will be used when instantiating the I^{2}C devices.

2. An initialization function. If no platform-specific initialization is needed then this can be the `cyg_mcfxxxx_i2c_init` function exported by this driver. Otherwise it can be a platform-specific function which, for example, sets up the relevant pins appropriately and then chains into `cyg_mcfxxxx_i2c_init`.

3. The base address of the I^{2}C bus. For example on an MCF5282 with the IPSBAR set to its usual value of 0x40000000, the I^{2}C bus is at location 0x40000300.

4. The interrupt vector, for example `CYGNUM_HAL_ISR_I2C_IIF` on an MCF5282.

5. The interrupt priority. Typically this will be a configurable option within the platform HAL.

6. A value for the I^{2}C bus's I2FDR register. That register controls the bus speed. Typical bus speeds are 100KHz and 400KHz, depending on the capabilities of the attached devices. There is no simple relationship between the system clock speed, the desired bus speed, and the FDR register. Although the driver could determine the FDR setting using a lookup table and appropriate code, it is better to determine the correct value once during the porting process and avoid unnecessary run-time overheads.

For the common case where only a single I^{2}C bus should be supported (`CYGHWR_DEVS_I2C_MCFxxxx_MULTIPLE_BUSES` is disabled), the last four parameters should be provided by preprocessor `#define`'s, typically in `cyg/hal/plf_io.h` which gets `#include`'d automatically via `cyg/hal/hal_io.h`. This header can also define the `HAL_I2C_EXPORTED_DEVICES` macro as per the generic I^{2}C package:

```
#include <pkgconf/hal_m68k_dnp5280.h>
…
#ifdef CYGHWR_HAL_M68K_DNP5280_I2C
#define HAL_MCFxxxx_I2C_SINGLETON_BASE   (HAL_MCFxxxx_MBAR+HAL_MCF5282_I2C0_BASE)
#define HAL_MCFxxxx_I2C_SINGLETON_ISRVEC CYGNUM_HAL_ISR_I2C_IIF
#define HAL_MCFxxxx_I2C_SINGLETON_ISRPRI CYGNUM_HAL_M68K_DNP5280_I2C_ISRPRI
#define HAL_MCFxxxx_I2C_SINGLETON_FDR    CYGNUM_HAL_M68K_DNP5280_I2C_FDR

#define HAL_I2C_EXPORTED_DEVICES \
    extern cyg_i2c_bus hal_dnp5280_i2c_bus;
#endif
```

On this particular platform the I^{2}C bus is only accessible on an expansion connector so the support is conditional on a configuration option `CYGHWR_HAL_M68K_DNP5280_I2C`. The interrupt priority and I2FDR values are also controlled by configuration options. On other platforms the I^{2}C support may not be conditional and the priority and/or FDR values may be hard-wired.

The I^{2}C bus instantiation should happen in an ordinary C or C++ file, typically in the platform HAL. The corresponding object file should go into `libtarget.a` and the file should only contain I^{2}C-related code to get the maximum benefit of linker garbage collection.

```
#include <cyg/infra/cyg_type.h>
#include <cyg/hal/hal_io.h>
#include <cyg/io/i2c.h>
#include <cyg/io/i2c_mcfxxxx.h>

CYG_MCFxxxx_I2C_BUS(hal_dnp5280_i2c_bus,
                    &cyg_mcfxxxx_i2c_init,
                    HAL_MCFxxxx_I2C_SINGLETON_BASE,
                    HAL_MCFxxxx_I2C_SINGLETON_ISRVEC,
                    HAL_MCFxxxx_I2C_SINGLETON_ISRPRI,
                    HAL_MCFxxxx_I2C_SINGLETON_FDR);
```

Obviously if `CYGHWR_DEVS_I2C_MCFxxxx_MULTIPLE_BUSES` is enabled then the singleton macros may not be defined and the appropriate numbers should be used directly. This example assumes no special initialization is needed. If there are special initialization requirements then a custom function can be used instead of `cyg_mcfxxxx_i2c_init`, and the custom function should chain to the latter.

I^{2}C device structures can be instantiated in the usual way, for example:

```
CYG_I2C_DEVICE(cyg_i2c_wallclock_ds1307,
               &hal_dnp5280_i2c_bus,
               0x68,
               0x00,
               CYG_I2C_DEFAULT_DELAY);
```

Part XXIV. ADC Support

Documentation for drivers of this type is often integrated into the eCos board support documentation. You should review the documentation for your target board for details. Standalone and more generic drivers are documented in the following sections.

Table of Contents

Chapter 77. ADC Support

Name

eCos Support for Analog/Digital Converters — Overview

Introduction

ADC support in eCos is based around the standard character device interface. Hence all device IO function, or file IO functions may be used to access ADC devices.

ADC devices are presented as read-only serial channels that generate samples at a given rate. The size of each sample is hardware specific and is defined by the cyg_adc_sample_t type. The sample rate may be set at runtime by the application. Most ADC devices support several channels which are all sampled at the same rate. Therefore setting the rate for one channel will usually change the rate for all channels on that device.

Examples

The use of the ADC devices is best shown by example. The following is a simple example of using the eCos device interface to access the ADC:

```
int res;
cyg_io_handle_t handle;

// Get a handle for ADC device 0 channel 0
res = cyg_io_lookup( "/dev/adc00", &handle );

if( res != ENOERR )
    handle_error(err);

for(;;)
{
    cyg_adc_sample_t sample;
    cyg_uint32 len = sizeof(sample);

    // read a sample from the channel
    res = cyg_io_read( handle, &sample, &len );

    if( res != ENOERR )
        handle_error(err);

    use_sample( sample );
}
```

In this example, the required channel is looked up and a handle on it acquired. Conventionally ADC devices are named "/dev/adcXY" where X is the device number and Y the channel within that device. Following this, samples are read from the device sequentially.

ADC devices may also be accessed using FILEIO operations. These allow more sophisticated usage. The following example shows `select()` being used to gather samples from several devices.

```
int fd1, fd2;

// open channels, non-blocking
fd1 = open( "/dev/adc01", O_RDONLY|O_NONBLOCK );
fd2 = open( "/dev/adc02", O_RDONLY|O_NONBLOCK );

if( fd1 < 0 || fd2 < 0 )
    handle_error( errno );

for(;;)
{
    fd_set rd;
    int maxfd = 0;
    int err;
    cyg_adc_sample_t samples[128];
    int len;

    FD_ZERO( &rd );

    FD_SET( fd1, &rd );
    FD_SET( fd2, &rd );
    maxfd = max(fd1,fd2);

    // select on available data on each channel.
    err = select( maxfd+1, &rd, NULL, NULL, NULL );

    if( err < 0 )
        handle_error(errno);

    // If channel 1 has data, handle it
    if( FD_ISSET( fd1, &rd ) )
    {
        len = read( fd1, &samples, sizeof(samples) );

        if( len > 0 )
            handle_samples_chan1( &samples, len/sizeof(sample[0]) );
    }

    // If channel 2 has data, handle it
    if( FD_ISSET( fd2, &rd ) )
    {
        len = read( fd2, &samples, sizeof(samples) );

        if( len > 0 )
            handle_samples_chan2( &samples, len/sizeof(sample[0]) );
    }

}
```

This test uses FILEIO operations to access ADC channels. It starts by opening two channels for reading only and with blocking disabled. It then falls into a loop using select to wake up whenever either channel has samples available.

Details

As indicated, the main interface to ADC devices is via the standard character device interface. However, there are a number of aspects that are ADC specific.

Sample Type

Samples can vary in size depending on the underlying hardware and is often a non-standard number of bits. The actual number of bits is defined by the hardware driver package, and the generic ADC package uses this to define a type cyg_adc_sample_t which can contain at least the required number of bits. All reads from an ADC channel should be expressed in multiples of this type, and actual bytes read will also always be a multiple.

Sample Rate

The sample rate of an ADC device can be varied by calling a `set_config` function, either at the device IO API level or at the FILEIO level. The following two functions show how this is done at each:

```
int set_rate_io( cyg_io_handle_t handle, int rate )
{
    cyg_adc_info_t info;
    cyg_uint32 len = sizeof(info);

    info.rate = rate;
    return cyg_io_set_config( handle,
                              CYG_IO_SET_CONFIG_ADC_RATE,
                              &info,
                              &len);
}

int set_rate_fileio( int fd, int rate )
{
    cyg_adc_info_t info;

    info.rate = rate;

    return cyg_fs_fsetinfo( fd,
                            CYG_IO_SET_CONFIG_ADC_RATE,
                            &info,
                            sizeof(info) );
}
```

Enabling a Channel

Channels are initialized in a disabled state and generate no samples. When a channel is first looked up or opened, then it is automatically enabled and samples start to accumulate. A channel may then be disable or re-enabled via a `set_config` function:

```
int disable_io( cyg_io_handle_t handle )
{
    return cyg_io_set_config( handle,
                              CYG_IO_SET_CONFIG_ADC_DISABLE,
                              NULL,
                              NULL);
}

int enable_io( cyg_io_handle_t handle )
{
    return cyg_io_set_config( handle,
                              CYG_IO_SET_CONFIG_ADC_DISABLE,
                              NULL,
                              NULL);
}
```

Flushing a channel

After any run-time channel re-configuration it may desirable (depending on the application) to flush any buffered data that may be present from prior to the channel config update. For example, this can be done at the IO API level by using the `CYG_IO_SET_CONFIG_ADC_FLUSH` config key:

```
int flush_io( cyg_io_handle_t handle )
{
    return cyg_io_set_config( handle,
                              CYG_IO_SET_CONFIG_ADC_FLUSH,
                              NULL,
                              NULL);
}
```

Configuration

The ADC package defines a number of generic configuration options that apply to all ADC implementations:

cdl_component CYGPKG_IO_ADC_DEVICES

> This option enables the hardware device drivers for the current platform. ADC devices will only be enabled if this option is itself enabled.

cdl_option CYGNUM_IO_ADC_SAMPLE_SIZE

> This option defines the sample size for the ADC devices. Given in bits, it will be rounded up to 8, 16 or 32 to define the cyg_adc_sample_t type. This option is usually set by the hardware device driver.

cdl_option CYGPKG_IO_ADC_SELECT_SUPPORT

> This option enables support for the `select()` API function on all ADC devices. This option can be disabled if the `select()` is not used, saving some code and data space.

cdl_component CYGIMP_IO_ADC_INSTRUMENTATION

> If the system instrumentation support is enabled for the active configuration then this option can be used to enable the ADC I/O specific instrumentation support.
>
> Instrumentation records will only be generated if this option is itself enabled, and if the relevant individual event code sub-options are enabled. The default state is for all the instrumentation to be disabled. Some options will generate a lot of instrumentation records in a heavily loaded system and so care may need to be taken regarding the instrumentation enabled vs the instrumentation recording mechanism. Depending on why the ADC I/O driver instrumentation is being enabled (debugging, timing validation, etc.) the user can choose which events they wish to record by enabling the specific CDL option.

In addition to the generic options, each hardware device driver defines some parameters for each device and channel. The exact names of the following option depends on the hardware device driver, but options of this form should be available in all drivers.

cdl_option CYGDAT_IO_ADC_EXAMPLE_CHANNELN_NAME

> This option specifies the name of the device for an ADC channel. Channel names should be of the form "/dev/adcXY" where X is the device number and Y the channel within that device.

cdl_option CYGNUM_IO_ADC_EXAMPLE_CHANNELN_BUFSIZE

> This option specifies the buffer size for an ADC channel. The value is expressed in multiples of cyg_adc_sample_t rather than bytes. The default value is 128.

cdl_option CYGNUM_IO_ADC_EXAMPLE_DEFAULT_RATE

> This option defines the initial default sample rate for all channels. The hardware driver may place constraints on the range of values this option may take.

Name

Overview — ADC Device Drivers

Introduction

This section describes how to write an ADC hardware device. While users of ADC devices do not need to read it, it may provide added insight into how the devices work.

Data Structures

An ADC hardware driver is represented by a number of data structures. These are generic `device` and `channel` data structures, a driver private device data structure, a generic character device table entry and a driver function table. Most of these structures are instantiated using macros, which will be described here.

The data structure instantiation for a typical single device, four channel ADC would look like this:

```
//==========================================================================
// Instantiate data structures

// -------------------------------------------------------------------------
// Driver functions:

CYG_ADC_FUNCTIONS( example_adc_funs,
                   example_adc_enable,
                   example_adc_disable,
                   example_adc_set_rate );

// -------------------------------------------------------------------------
// Device instance:

static example_adc_info example_adc_info0 =
{
    .base               = CYGARC_HAL_EXAMPLE_ADC_BASE,
    .vector             = CYGNUM_HAL_INTERRUPT_ADC
};

CYG_ADC_DEVICE( example_adc_device,
                &example_adc_funs,
                &example_adc_info0,
                CYGNUM_IO_ADC_EXAMPLE_DEFAULT_RATE );

// -------------------------------------------------------------------------
// Channel instances:

#define EXAMPLE_ADC_CHANNEL( __chan )                                    \
CYG_ADC_CHANNEL( example_adc_channel##__chan,                            \
                 __chan,                                                 \
                 CYGNUM_IO_ADC_EXAMPLE_CHANNEL##__chan##_BUFSIZE,        \
                 &example_adc_device );                                  \
                                                                         \
DEVTAB_ENTRY( example_adc_channel##__chan##_device,                      \
              CYGDAT_IO_ADC_EXAMPLE_CHANNEL##__chan##_NAME,              \
              0,                                                         \
              &cyg_io_adc_devio,                                         \
              example_adc_init,                                          \
              example_adc_lookup,                                        \
              &example_adc_channel##__chan );

EXAMPLE_ADC_CHANNEL( 0 );
EXAMPLE_ADC_CHANNEL( 1 );
EXAMPLE_ADC_CHANNEL( 2 );
EXAMPLE_ADC_CHANNEL( 3 );
```

The macro `CYG_ADC_FUNCTIONS()` instantiates a function table called `example_adc_funs` and populates it with the ADC driver functions (see later for details).

Then an instance of the driver private device data structure is instantiated. In addition to the device base address and interrupt vector shown here, this structure should contain the interrupt object and handle for attaching to the vector. It may also contain any other variables needed to manage the device.

The macro `CYG_ADC_DEVICE()` instantiates a cyg_adc_device structure, named `example_adc_device` which will contain pointers to the function table and private data structure. The initial sample rate is also supplied here.

For each channel, an ADC channel structure and a device table entry must be created. The macro `EXAMPLE_ADC_CHANNEL()` is defined to simplify this process. The macro `CYG_ADC_CHANNEL` defines a cyg_adc_channel structure, which contains the channel number, the buffer size, and a pointer to the device object defined earlier. The call to `DEVTAB_ENTRY()` generates a device table entry containing the configured channel name, a pointer to a device function table defined in the generic ADC driver, pointers to init and lookup functions implemented here, and a pointer to the channel data structure just defined.

Finally, four channels, numbered 0 to 3 are created.

Functions

There are several classes of function that need to be defined in an ADC driver. These are those function that go into the channel's device table, those that go into the ADC device's function table, calls that the driver makes into the generic ADC package, and interrupt handling functions.

Device Table Functions

These functions are placed in the standard device table entry for each channel and handle initialization and location of the device within the generic driver infrastructure.

`static bool example_adc_init(struct cyg_devtab_entry *tab)` This function is called from the device IO infrastructure to initialize the device. It should perform any work needed to start up the device, short of actually starting the generation of samples. This function will be called for each channel, so if there is initialization that only needs to be done once, such as creating an interrupt object, then care should be taken to do this. This function should also call `cyg_adc_device_init()` to initialize the generic parts of the driver.

`static Cyg_ErrNo example_adc_lookup(struct cyg_devtab_entry **tab, struct cyg_devtab_entry *sub_tab, const char *name)` This function is called when a client looks up or opens a channel. It should call `cyg_adc_channel_init()` to initialize the generic part of the channel. It should also perform any operations needed to start the channel generating samples.

Driver Functions

These are the functions installed into the driver function table by the `CYG_ADC_FUNCTIONS()` macro.

`static void example_adc_enable( cyg_adc_channel *chan )` This function is called from the generic ADC package to enable the channel in response to a `CYG_IO_SET_CONFIG_ADC_ENABLE` config operation. It should take any steps needed to start the channel generating samples.

`static void example_adc_disable( cyg_adc_channel *chan )` This function is called from the generic ADC package to enable the channel in response to a `CYG_IO_SET_CONFIG_ADC_DISABLE` config operation. It should take any steps needed to stop the channel generating samples.

`static void example_adc_set_rate( cyg_adc_channel *chan, cyg_uint32 rate )` This function is called from the generic ADC package to enable the channel in response to a `CYG_IO_SET_CONFIG_ADC_RATE` config operation. It should take any steps needed to change the sample rate of the channel, or of the entire device.

Generic Package Functions

These functions are called by a hardware ADC device driver to perform operations in the generic ADC package.

`__externC void cyg_adc_device_init( cyg_adc_device *device )` This function is called from the driver's init function and is used to initialize the cyg_adc_device object.

`__externC void cyg_adc_channel_init(cyg_adc_channel *chan)` This function is called from the driver's lookup function and is used to initialize the cyg_adc_channel object.

`__externC cyg_uint32 cyg_adc_receive_sample(cyg_adc_channel *chan, cyg_adc_sample_t sample)` This function is called from the driver's ISR to add a new sample to the buffer. The return value will be either zero, or `CYG_ISR_CALL_DSR` and should be ORed with the return value of the ISR.

`__externC void cyg_adc_wakeup(cyg_adc_channel *chan )` This function is called from the driver's DSR to cause any threads waiting for data to wake up when a new sample is available. It should only be called if the *`wakeup`* field of the channel object is `true`.

Interrupt Functions

These functions are internal to the driver, but make calls on generic package functions. Typically an ADC device will have a single interrupt vector with which it signals available samples on the channels and any error conditions such as overruns.

`static cyg_uint32 example_adc_isr(cyg_vector_t vector, cyg_addrword_t data)` This function is the ISR attached to the ADC device's interrupt vector. It is responsible for reading samples from the channels and passing them on to the generic layer. It needs to check each channel for data, and call `cyg_adc_receive_sample()` for each new sample available, and then ready the device for the next interrupt. It's activities are best explained by example:

```
static cyg_uint32 example_adc_isr(cyg_vector_t vector, cyg_addrword_t data)
{
    cyg_adc_device *example_device = (cyg_adc_device *) data;
    example_adc_info *example_info = example_device->dev_priv;
    cyg_uint32 res = 0;
    int i;

    // Deal with errors if necessary
    DEVICE_CHECK_ERRORS( example_info );

    // Look for all channels with data available
    for( i = 0; i < CHANNEL_COUNT; i++ )
    {
        if( CHANNEL_SAMPLE_AVALIABLE(i) )
        {
            // Fetch data from this channel and pass up to higher
            // level.

            cyg_adc_sample_t data = CHANNEL_GET_SAMPLE(i);

            res |= CYG_ISR_HANDLED | cyg_adc_receive_sample( example_info->channel[i], data );
        }
    }

   // Clear any interrupt conditions
    DEVICE_CLEAR_INTERRUPTS( example_info );

    cyg_drv_interrupt_acknowledge(example_info->vector);

    return res;
}
```

`static void example_adc_dsr(cyg_vector_t vector, cyg_ucount32 count, cyg_addrword_t data)` This function is the DSR attached to the ADC device's interrupt vector. It is called by the kernel if the ISR return value contains the `CYG_ISR_HANDLED` bit. It needs to call `cyg_adc_wakeup()` for each channel that has its *`wakeup`* field set. Again, and example should make it all clear:

```
static void example_adc_dsr(cyg_vector_t vector, cyg_ucount32 count, cyg_addrword_t data)
{
    cyg_adc_device *example_device = (cyg_adc_device *) data;
    example_adc_info *example_info = example_device->dev_priv;
    int i;

    // Look for all channels with pending wakeups
    for( i = 0; i < CHANNEL_COUNT; i++ )
    {
        if( example_info->channel[i]->wakeup )
            cyg_adc_wakeup( example_info->channel[i] );
    }
}
```

Chapter 78. STM32 ADC Driver

Name

STM32 — ADC Driver

Description

This driver supports the ADC devices available in some variants of the ST STM32 family of microprocessors.

Sample Size

The STM32 ADC on F1 series devices produces 12-bit samples. The F2/F4/F7/L4 series devices can produce 12-, 10-, 8- or 6-bit samples and the H7 family can additionally produce 14- and 16-bit samples. The `CYGNUM_DEVS_ADC_CORTEXM_STM32_WIDTH` CDL configuration option allows the sample width to be set for suitable variants. The default is for 12-bit samples, which will cause the generic layer to define cyg_adc_sample_t as a 16-bit value.

Sample Rates

The ADC hardware is limited to a maximum of 2K samples per channel. Since up to 16 channels are sampled on a round-robin basis, this means that the total sample rate can be 16K samples per second.

Configuration

The option `CYGNUM_DEVS_ADC_CORTEXM_STM32_CLOCK_DIV` specifies the divider used to control the ADC system clock supplied by the RCC.

If system instrumentation is enabled then the `CYGIMP_DEVS_ADC_CORTEXM_STM32_INSTRUMENTATION` option is made available, and can be enabled to allow the ADC device driver to generate instrumentation. When enabed the sub-options `CYGDBG_DEVS_ADC_CORTEXM_STM32_INSTRUMENT_CONTROL` and `CYGDBG_DEVS_ADC_CORTEXM_STM32_INSTRUMENT_DMA` control which events are generated.

Each ADC device is controlled by a CDL component, `CYGHWR_DEVS_ADC_CORTEXM_STM32_ADCX` for each device *X*, which must be enabled to initialize the device. For STM32F1 devices only ADC devices 1 and 3 are available, since ADC2 shares GPIO lines and an interrupt with ADC1. For STM32F2/F4/F7/H7 devices ADC devices 1, 2 and 3 are available.

For STM32F2/F4/F7/L4/H7 devices the selection of the timer `TRGO` event to be used for the specific ADC device is controlled by a CDL component, `CYGHWR_DEVS_ADC_CORTEXM_STM32_ADC`*`X`*`_TIM` for each device *X*.

For STM32F2/F4/F7/L4/H7 devices the selection of the DMA stream interrupt to be used for the specific ADC device is controlled by a CDL component, `CYGHWR_DEVS_ADC_CORTEXM_STM32_ADCX_DMA` for each device *X*, which must be configured to avoid clashes with other peripherals that may share the same DMA stream.

The option `CYGNUM_DEVS_ADC_CORTEXM_STM32_ADCX_SAMPLE_TIME` defines the duration over which each sample is taken for each channel, in microseconds.

The option `CYGNUM_DEVS_ADC_CORTEXM_STM32_ADCX_DEFAULT_RATE` defines the default sample rate for all channels attached to device *X*.

The option `CYGNUM_DEVS_ADC_CORTEXM_STM32_ADCX_DMA_INT_PRI` defines the DMA interrupt prioirity for this device. The default of 0x80 sets it in the middle of the prioirity range.

For each channel *X* in ADC device *Y* the CDL script provides the following configuration options:

`cdl_component CYGHWR_DEVS_ADC_CORTEXM_STM32_ADCY_CHANNELX`

> If the application needs to access the on-chip ADC channel X via an eCos ADC driver then this option should be enabled.

`cdl_option CYGDAT_DEVS_ADC_CORTEXM_STM32_ADCT_CHANNELX_NAME`

This option controls the name that an eCos application should use to access this device via cyg_io_lookup(), open(), or similar calls.

`cdl_option CYGDAT_DEVS_ADC_CORTEXM_STM32_ADCY_CHANNELX_BUFSIZE`

This defines the size of the channel's sample buffer, in samples.

Chapter 79. STR7XX ADC Driver

Name

STR7XX — ADC Driver

Description

This driver supports the ADC devices available in some variants of the ST STR7XX family of microprocessors.

Sample Size

The STR7XX ADC produces 12 bit samples. Therefore this driver sets `CYGNUM_IO_ADC_SAMPLE_SIZE` to 12. This will cause the generic layer to define cyg_adc_sample_t as a 16 bit value.

Sample Rates

The ADC hardware is limited to a maximum of 1K samples per channel. Since channels are sampled on a round-robin basis at 4 times this rate, this means that the total sample rate is 4K samples per second.

The option `CYGNUM_DEVS_ADC_ARM_STR7XX_DEFAULT_RATE` defines a default sample rate and is initially set to 500.

Configuration

For each channel *X* supported the CDL script provides the following configuration options:

cdl_component CYGPKG_DEVS_ADC_ARM_STR7XX_CHANNELX

> This defines whether the channel is included.

cdl_option CYGDAT_DEVS_ADC_ARM_STR7XX_CHANNELN_NAME

> This defines the name of the channel.

cdl_option CYGNUM_DEVS_ADC_ARM_STR7XX_CHANNELX_BUFSIZE

> This defines the size of the channel's sample buffer, in samples.

Chapter 80. TSC ADC Driver

Name

TSC — ADC Driver

Description

This driver supports the Touch Screen Controller device available in some variants of the Freescale i.MXxx family of microprocessors.

In addition to the TSC device itself, this driver uses GPT1 to provide the data rate clock.

Sample Size

The TSC ADC produces 12 bit samples. Therefore this driver sets `CYGNUM_IO_ADC_SAMPLE_SIZE` to 12. This will cause the generic layer to define cyg_adc_sample_t as a 16 bit value.

Sample Rates

The ADC collects samples from the inputs sequentially. For the three inputs this driver currently supports (INAUX0, INAUX1 and INAUX2) it takes about 40us to collect all samples. Therefore the hardware is limited to a maximum rate of about 25KHz.

The option `CYGNUM_DEVS_ADC_ARM_TSC_DEFAULT_RATE` defines a default sample rate and is initially set to 500.

Configuration

For each channel *X* supported the CDL script provides the following configuration options:

cdl_component CYGPKG_DEVS_ADC_ARM_TSC_CHANNELX

> This defines whether the channel *N* is included.

cdl_option CYGDAT_DEVS_ADC_ARM_TSC_CHANNELN_NAME

> This defines the name of the channel *N*. By default the channels are named `"/dev/inauxN"`.

cdl_option CYGNUM_DEVS_ADC_ARM_TSC_CHANNELX_BUFSIZE

> This defines the size of the channel's sample buffer, in samples.

Chapter 81. Atmel AFEC (ADC) Driver

Name

Atmel — ADC Driver

Description

This driver supports the AFE (Analog Front-End) Controller available in some Atmel microprocessor variants, e.g. the SAM4E family.

Sample Size

The Atmel AFEC natively supports 12-bit samples, but via digital averaging 13-, 14-, 15- and 16-bit samples are supported. The controller also supports "low-resolution" 10-bit samples, but at the same sampling rate as the native 12-bit samples. When higher resolution, averaged, samples are used the maximum achievable sample rate will be lower than the native resolution. The per-controller `CYGNUM_DEVS_ADC_ATMEL_AFECx_RESOLUTION` configuration option specifies the sample resolution to be used.

This driver left-aligns results so that the full-range of the ADC I/O generic layer defined cyg_adc_sample_t type is used regardless of the resolution configured for this driver. The default generic layer `CYGNUM_IO_ADC_SAMPLE_SIZE` definition of 16 should be sufficient for most applications.

Configuration

If system instrumentation is enabled then the `CYGIMP_DEVS_ADC_ATMEL_AFEC_INSTRUMENTATION` option is made available, and can be enabled to allow the ADC device driver to generate instrumentation. When enabed the sub-options `CYGDBG_DEVS_ADC_ATMEL_AFEC_INSTRUMENT_CONTROL` and `CYGDBG_DEVS_ADC_ATMEL_AFEC_INSTRUMENT_DMA` control which events are generated.

Each ADC device is controlled by a CDL component, `CYGHWR_DEVS_ADC_ATMEL_AFECx` for each device *x*, which must be enabled to initialize the device. The number of channels available to a controller instance is defined by the relevant variant or platform HAL as required.

The option `CYGNUM_DEVS_ADC_ATMEL_AFECx_RESOLUTION` defines the internal sample size.

The option `CYGNUM_DEVS_ADC_ATMEL_AFECx_OFFSET` defines the channel offset compensation applied to all channels..

The option `CYGNUM_DEVS_ADC_ATMEL_AFECx_DEFAULT_RATE` defines the default samples-per-second rate for all channels attached to device *x*. This default can be over-ridden at run-time using the generic ADC I/O layer provided support as required.

For each channel *x* in an AFEC device *y* the CDL script provides the following configuration options:

`CYGHWR_DEVS_ADC_ATMEL_AFECy_CHANNELx`

> If the application needs to access the on-chip ADC channel *x* via an eCos ADC driver then this option should be enabled.

`CYGDAT_DEVS_ADC_ATMEL_AFECy_CHANNELx_NAME`

> This option controls the name that an eCos application should use to access this device via cyg_io_lookup(), open(), or similar calls. This allows meaningful names to be assigned if required, rather than the default "/dev/adc*yx*" style.

`CYGDAT_DEVS_ADC_ATMEL_AFECy_CHANNELx_INPUT`

> By default input channels are configured in single-ended mode, which is the normal acquisition mode. This option allows individual channels to be configured in `Differential` mode, where the input is generated against an adjacent "paired" channel.

`CYGDAT_DEVS_ADC_ATMEL_AFEC`*`y`*`_CHANNEL`*`x`*`_BUFSIZE`

This defines the size of the channel's sample buffer, in cyg_adc_sample_t samples.

Chapter 82. NXP i.MX RT ADC Driver

Name

NXP — ADC Driver

Description

This driver supports the ADC (Analog-to-Digital Converter) Controller of the NXP i.MX RT microprocessor variants, e.g. the i.MX RT105x family.

Sample Size

The i.MX ADC natively supports 12-bit samples. The controller also supports “low-resolution” 10-bit and 8-bit samples. When higher resolution, averaged, samples are used the maximum achievable sample rate will be lower than the native resolution. The per-controller `CYGNUM_DEVS_ADC_NXP_IMX_ADCx_RESOLUTION` configuration option specifies the hardware sample resolution to be used.

This driver left-aligns results so that the full-range of the ADC I/O generic layer defined cyg_adc_sample_t type is used regardless of the resolution configured for this driver. The default generic layer `CYGNUM_IO_ADC_SAMPLE_SIZE` definition of 16 should be sufficient for most applications. However, it should be noted that the underlying cyg_adc_sample_t type is signed, whereas the returned data should be interpreted as an unsigned value (0..MAX) for this controller.

Configuration

Each ADC device is controlled by a CDL component, `CYGHWR_DEVS_ADC_NXP_IMX_ADCx` for each controller device *x*, which must be enabled to initialize the device. The number of channels is currently fixed to 16 for each controller instance, to support the external signals 0..15 as defined by the Processor Reference Manual.

The option `CYGNUM_DEVS_ADC_NXP_IMX_ADCx_RESOLUTION` defines the internal hardware sample size.

The option `CYGNUM_DEVS_ADC_NXP_IMX_ADCx_CLOCK` allows the selection of the conversion clock as either the internal asynchronous `ADACK` clock (default) or the system `IPG` clock. The Processor Reference Manual (PRM) ADC section should be consulted for the pro's and con's of the clock selection, frequency, averaging, etc.

The option `CYGNUM_DEVS_ADC_NXP_IMX_ADCx_SPEED` allows the selection of whether the `Normal` (default) or `High-Speed` internal ADACK clock is used.

The option `CYGNUM_DEVS_ADC_NXP_IMX_ADCx_SAMPLE_PERIOD` defines the sample period for a conversion as the number of ADC clock cycles.

The option `CYGNUM_DEVS_ADC_NXP_IMX_ADCx_AVG` controls whether the hardware averaging is enabled. When enabled the option`CYGNUM_DEVS_ADC_NXP_IMX_ADCx_AVG_SAMPLES` specifies the number of samples that are averaged to provide a single reading.

The option `CYGNUM_DEVS_ADC_NXP_IMX_ADCx_OFFSET` defines the channel offset compensation applied to all channels..

The option `CYGNUM_DEVS_ADC_NXP_IMX_ADCx_DEFAULT_RATE` defines the default samples-per-second rate for all channels attached to device *x*. This default can be over-ridden at run-time using the generic ADC I/O layer provided support as required.

If system instrumentation is enabled then the `CYGIMP_DEVS_ADC_NXP_IMC_ADC_INSTRUMENTATION` option is made available, and can be enabled to allow the ADC device driver to generate instrumentation. When enabed there are sub-options available to further control which events are recorded.

For each channel *x* in an ADC device *y* the CDL script provides the following configuration options:

`CYGHWR_DEVS_ADC_NXP_IMX_ADCy_CHANNELx`

> If the application needs to access the on-chip ADC channel *x* via an eCos ADC driver then this option should be enabled.

`CYGDAT_DEVS_ADC_NXP_IMX_ADCy_CHANNELx_NAME`

This option controls the name that an eCos application should use to access this device via cyg_io_lookup(), open(), or similar calls. This allows meaningful names to be assigned if required, rather than the default “/dev/adc*y*c*x*” style.

`CYGDAT_DEVS_ADC_NXP_IMX_ADCy_CHANNELx_BUFSIZE`

This defines the size of the channel's sample buffer, in multiples of cyg_adc_sample_t samples.

Part XXV. Pulse Width Modulation (PWM) Support

Documentation for drivers of this type is often integrated into the eCos board support documentation. You should review the documentation for your target board for details. Standalone and more generic drivers are documented in the following sections.

Table of Contents

Chapter 83. PWM Support

Name

Overview — eCos Support for PWM, the Inter IC Bus

Synopsis

#include <cyg/io/pwm.h>

```
cyg_pwm_device *cyg_pwm_find(char *name);

int cyg_pwm_set(cyg_pwm_device *device, cyg_uint32 channel, cyg_pwm_period period,
cyg_pwm_period width);

int cyg_pwm_get(cyg_pwm_device *device, cyg_uint32 channel, cyg_pwm_period *period,
cyg_pwm_period *width);

int cyg_pwm_polarity(cyg_pwm_device *device, cyg_uint32 channel, cyg_uint32 polarity);

int cyg_pwm_pin_enable(cyg_pwm_device *device, cyg_uint32 channel, cyg_uint32 pin);

int cyg_pwm_pin_disable(cyg_pwm_device *device, cyg_uint32 channel, cyg_uint32 pin);

int cyg_pwm_start(cyg_pwm_device *device, cyg_uint32 channel);

int cyg_pwm_stop(cyg_pwm_device *device, cyg_uint32 channel);
```

Description

This API provided access to very basic PWM (Pulse Width Modulation) functionality where supported by underlying hardware. The functionality simply allows one or more external pins to be programmed to generate a square wave of a fixed period with a fixed duty cycle. Features of more advanced PWM devices are not supported.

The model for PWM devices allows a single device to contain several independent channels which may be programmed separately. Each channel may have several output pins which may be enabled separately but which all generate the same signal programmed into the channel. Channels within a device, and pins within a channel are all numbered from zero.

The function `cyg_pwm_find()` looks for a PWM device by name and returns a pointer to it. The PWM device is typically defined in the platform HAL and may be given a platform specific name. Typical names may be `"pwm0"` or `"pwm1"`.

The function `cyg_pwm_set()` sets the square wave parameters for the selected channel. Both *`period`* and *`width`* are specified in nanoseconds. For example setting period to 100000 and width to 25000 sets up a square wave with a 100us (10KHz) period and a 25% duty cycle. The function `cyg_pwm_get()` returns the current settings.

The function `cyg_pwm_polarity()` sets the polarity of the signal. By default the width defines the high portion of the wave. Setting polarity to `CYG_PWM_POLARITY_INVERTED` means it defines the low part.

The function `cyg_pwm_pin_enable()` enables the selected channel output pin and `cyg_pwm_pin_disable()` disables it. While is is unusual, it is permitted to enable any number of channel pins.

The function `cyg_pwm_pin_start()` starts the PWM channel generating the square wave and `cyg_pwm_pin_stop()` stops it. The channel configuration functions may all be called before starting the channel, they may also be called after it started and will immediately affect it. If none are called before starting the channel then default values, defined in the platform HAL will be used. There is no need to stop a channel to reconfigure it, unless this is required by the application or attached hardware.

Part XXVI. Framebuffer Support

Table of Contents

Chapter 84. Framebuffer Support

Name

Overview — eCos Support for Framebuffer Devices

Description

Framebuffer devices are the most common way for a computer system to display graphical output to users. There are immense variations in the implementations of such devices. CYGPKG_IO_FRAMEBUF provides an abstraction layer for use by application code and other packages. It defines an API for manipulating framebuffers, mapping this API on to functionality provided by the appropriate device driver. It also defines the interface which such device drivers should implement. For simple hardware it provides default implementations of much of this interface, greatly reducing the effort needed to write a device driver.

This package does not constitute a graphics library. It does not implement functionality like drawing text or arbitrary lines, let alone any kind of windowing system. Instead it operates at the lower level of individual pixels and blocks of pixels, in addition to control operations such as hardware initialization. Some applications may use the framebuffer API directly. Others will instead use a higher-level graphics library, and it is that library which uses the framebuffer API.

It is assumed that users are already familiar with the fundamentals of computer graphics, and no attempt is made here to explain terms like display depth, palette or pixel.

Note

This package is work-in-progress. The support for 1bpp, 2bpp and 4bpp display depths is incomplete. For double-buffered displays the code does not yet maintain a bounding box of the updated parts of the display. The package has also been designed to allow for expansion with new functionality.

Configuration

CYGPKG_IO_FRAMEBUF only contains hardware-independent code. It should be complemented by one or more framebuffer device drivers appropriate for the target platform. These drivers may be specific to the platform, or they may be more generic with platform-specific details such as the framebuffer memory base address provided by the platform HAL. When creating a configuration for a given target the device driver(s) will always be included automatically (assuming one has been written or ported). However by default this driver will be inactive and will not get built, so does not add any unnecessary size overhead for applications which do not require graphics. To activate the device driver CYGPKG_IO_FRAMEBUF must be added explicitly to the configuration, for example using **ecosconfig add framebuf**. After this the full framebuffer API will be available to other packages and to application code.

This package contains very few configuration options. Instead it is left to device drivers or higher-level code to provide appropriate configurability. One option, CYGFUN_IO_FRAMEBUF_INSTALL_DEFAULT_PALETTE, relates to the initialization of paletted displays.

There are a number of calculated and inferred configuration options and a number of interfaces. These provide information such as whether or not there is a backlight. The most important one is CYGDAT_IO_FRAMEBUF_DEVICES, which holds a list of framebuffer identifiers for use with the macro-based API. If there is a single framebuffer device driver which supports one display in either landscape or portrait mode, the configuration option may hold a value like 240x320x8 320x240x8r90.

Application Programmer Interfaces

Framebuffer devices require a difficult choice between flexibility and performance. On the one hand the API should be able to support multiple devices driving separate displays, or a single device operating in different modes at different times. On the other hand graphics tends to involve very large amounts of I/O: even something as simple as drawing a background image can involve setting many thousands of pixels. Efficiency requires avoiding all possible overheads including function calls. Instead the API should make extensive use of macros or inline functions. Ideally details of the framebuffer device such as the stride would be known constants at compile-time, giving the compiler as much opportunity as possible to optimize the code. Clearly this is difficult if multiple framebuffer devices are in use or if the device mode may get changed at run-time.

To meet the conflicting requirements the generic framebuffer package provides two APIs: a fast macro API which requires selecting a single framebuffer device at compile or configure time; and a slower function API without this limitation. The two are very similar, for example:

```
#include <cyg/io/framebuf.h>

void
clear_screen(cyg_fb* fb, cyg_fb_colour colour)
{
    cyg_fb_fill_block(fb, 0, 0,
                      fb->fb_width, fb->fb_height,
                      colour);
}
```

or the equivalent macro version:

```
#include <cyg/io/framebuf.h>

#define FRAMEBUF 240x320x8

void
clear_screen(cyg_fb_colour colour)
{
    CYG_FB_FILL_BLOCK(FRAMEBUF, 0, 0,
                      CYG_FB_WIDTH(FRAMEBUF), CYG_FB_HEIGHT(FRAMEBUF),
                      colour);
}
```

The function-based API works in terms of cyg_fb structures, containing all the information needed to manipulate the device. Each framebuffer device driver will export one or more of these structures, for example `cyg_alaia_fb_240x320x8`, and the driver documentation should list the variable names. The macro API works in terms of identifiers such as `240x320x8`, and by a series of substitutions the main macro gets expanded to the appropriate device-specific code, usually inline. Again the device driver documentation should list the supported identifiers. In addition the configuration option `CYGDAT_IO_FRAMEBUF_DEVICES` will contain the full list. By convention the identifier will be specified by a `#define`'d symbol such as `FRAMEBUF`, or in the case of graphics libraries by a configuration option.

If a platform has multiple framebuffer devices connected to different displays then there will be separate cyg_fb structures and macro identifiers for each one. In addition some devices can operate in multiple modes. For example a PC VGA card can operate in a monochome 640x480 mode, an 8bpp 320x200 mode, and many other modes, but only one of these can be active at a time. The different modes are also represented by different cyg_fb structures and identifiers, effectively treating the modes as separate devices. It is the responsibility of higher-level code to ensure that only one mode is in use at a time.

It is possible to use the macro API with more than one device, basically by compiling the code twice with different values of `FRAMEBUF`, taking appropriate care to avoid identifier name clashes. This gives the higher performance of the macros at the cost of increased code size.

All of the framebuffer API, including exports of the device-specific cyg_fb structures, is available through a single header file `<cyg/io/framebuf.h>`. The API follows a number of conventions. Coordinates (0,0) correspond to the top-left corner of the display. All functions and macros which take a pair of coordinates have x first, y second. For block operations these coordinates are followed by width, then height. Coordinates and dimensions use cyg_ucount16 variables, which for any processor should be the most efficient unsigned data type with at least 16 bits - usually plain unsigned integers. Colours are identified by cyg_fb_colour variables, again usually unsigned integers.

To allow for the different variants of the English language, the API allows for a number of alternate spellings. Colour and color can be used interchangeably, so there are data types cyg_fb_colour and cyg_fb_color, and functions `cyg_fb_make_colour` and `cyg_fb_make_color`. Similarly gray is accepted as a variant of grey so the predefined colours `CYG_FB_DEFAULT_PALETTE_LIGHTGREY` and `CYG_FB_DEFAULT_PALETTE_LIGHTGRAY` are equivalent.

The API is split into the following categories:

parameters
getting information about a given framebuffer device such as width, height and depth. Colours management is complicated so has its own category.

control
operations such as switching the display on and off, and more device-specific ones such as manipulating the backlight.

colours
determining the colour format (monochrome, paletted, …), manipulating the palette, or constructing true colours.

drawing
primitives for manipulating pixels and blocks of pixels.

iteration
efficiently iterating over blocks of pixels.

Thread Safety

The framebuffer API never performs any locking so is not thread-safe. Instead it assumes that higher-level code such as a graphics library performs any locking that may be needed. Adding a mutex lock and unlock around every drawing primitive, including pixel writes, would be prohibitively expensive.

It is also assumed that the framebuffer will only be updated from thread context. With most hardware it will also be possible to access a framebuffer from DSR or ISR context, but this should be avoided in portable code.

Name

Parameters — determining framebuffer capabilities

Synopsis

#include <cyg/io/framebuf.h> typedef struct cyg_fb { cyg_ucount16 fb_depth; cyg_ucount16 fb_format; cyg_ucount16 fb_width; cyg_ucount16 fb_height; #ifdef CYGHWR_IO_FRAMEBUF_FUNCTIONALITY_VIEWPORT cyg_ucount16 fb_viewport_width; cyg_ucount16 fb_viewport_height; #endif void* fb_base; cyg_ucount16 fb_stride; cyg_uint32 fb_flags0; … } cyg_fb;

`cyg_fb*` **`CYG_FB_STRUCT`**`(`*`FRAMEBUF`*`);`

`cyg_ucount16` **`CYG_FB_DEPTH`**`(`*`FRAMEBUF`*`);`

`cyg_ucount16` **`CYG_FB_FORMAT`**`(`*`FRAMEBUF`*`);`

`cyg_ucount16` **`CYG_FB_WIDTH`**`(`*`FRAMEBUF`*`);`

`cyg_ucount16` **`CYG_FB_HEIGHT`**`(`*`FRAMEBUF`*`);`

`cyg_ucount16` **`CYG_FB_VIEWPORT_WIDTH`**`(`*`FRAMEBUF`*`);`

`cyg_ucount16` **`CYG_FB_VIEWPORT_HEIGHT`**`(`*`FRAMEBUF`*`);`

`void*` **`CYG_FB_BASE`**`(`*`FRAMEBUF`*`);`

`cyg_ucount16` **`CYG_FB_STRIDE`**`(`*`FRAMEBUF`*`);`

`cyg_uint32` **`CYG_FB_FLAGS0`**`(`*`FRAMEBUF`*`);`

Description

When developing an application for a specific platform the various framebuffer parameters such as width and height are known, and the code can be written accordingly. However when writing code that should work on many platforms with different framebuffer devices, for example a graphics library, the code must be able to get these parameters and adapt.

Code using the function API can extract the parameters from the cyg_fb structures at run-time. The macro API provides dedicated macros for each parameter. These do not follow the usual eCos convention where the result is provided via an extra argument. Instead the result is returned as normal, and is guaranteed to be a compile-time constant. This allows code like the following:

```
#if CYG_FB_DEPTH(FRAMEBUF) < 8
    …
#else
    …
#endif
```

or alternatively:

```
    if (CYG_FB_DEPTH(FRAMEBUF) < 8) {
        …
    } else {
        …
    }
```

or:

```
    switch (CYG_FB_DEPTH(FRAMEBUF)) {
        case  1 : … break;
        case  2 : … break;
        case  4 : … break;
        case  8 : … break;
        case 16 : … break;
        case 32 : … break;
    }
```

In terms of the code actually generated by the compiler these approaches have much the same effect. The macros expand to a compile-time constant so unnecessary code can be easily eliminated.

The available parameters are as follows:

depth

The number of bits per pixel or bpp. The common depths are 1, 2, 4, 8, 16 and 32.

format

How the pixel values are mapped on to visible colours, for example true colour or paletted or greyscale.

width
height

The number of framebuffer pixels horizontally and vertically.

viewport width
viewport height

With some devices the framebuffer height and/or width are greater than what the display can actually show. The display is said to offer a viewport into the larger framebuffer. The number of visible pixels is determined from the viewport width and height. The position of the viewport is controlled via an `ioctl`. Within a cyg_fb structure these fields are only present if `CYGHWR_IO_FRAMEBUF_FUNCTIONALITY_VIEWPORT` is defined, to avoid wasting data space on fields that are unnecessary for the current platform. For the macro API the viewport macros should only be used if `CYG_FB_FLAGS0_VIEWPORT` is set for the framebuffer:

```
#if (CYG_FB_FLAGS0(FRAMEBUF) & CYG_FB_FLAGS0_VIEWPORT)
    …
#endif
```

base
stride

For linear framebuffers these parameters provide the information needed to access framebuffer memory. The stride is in bytes.

flags0

This gives further information about the hardware capabilities. Some of this overlaps with other parameters, especially when it comes to colour, because it is often easier to test for a single flag than for a range of colour modes. The current flags are:

`CYG_FB_FLAGS0_LINEAR_FRAMEBUFFER`

Framebuffer memory is organized in a conventional fashion and can be accessed directly by higher-level code using the base and stride parameters.

`CYG_FB_FLAGS0_LE`

This flag is only relevant for 1bpp, 2bpp and 4bpp devices and controls how the pixels are organized within each byte. If the flag is set then the layout is little-endian: for a 1bpp device pixel (0,0) occupies bit 0 of the first byte of framebuffer memory. The more common layout is big-endian where pixel (0,0) occupies bit 7 of the first byte.

`CYG_FB_FLAGS0_TRUE_COLOUR`

The framebuffer uses a true colour format where the value of each pixel directly encodes the red, green and blue intensities. This is common for 16bpp and 32bpp devices, and is occasionally used for 8bpp devices.

`CYG_FB_FLAGS0_PALETTE`

The framebuffer uses a palette. A pixel value does not directly encode the colours, but instead acts as an index into a separate table of colour values. That table may be read-only or read-write. Paletted displays are common for 8bpp and some 4bpp displays.

`CYG_FB_FLAGS0_WRITEABLE_PALETTE`

The palette is read-write.

`CYG_FB_FLAGS0_DELAYED_PALETTE_UPDATE`

Palette updates can be synchronized to a vertical blank, in other words a brief time period when the display is not being updated, by using `CYG_FB_UPDATE_VERTICAL_RETRACE` as the last argument to `cyg_fb_write_palette` or `CYG_FB_WRITE_PALETTE`. With some hardware updating the palette in the middle of a screen update may result in visual noise.

`CYG_FB_FLAGS0_VIEWPORT`

The framebuffer contains more pixels than can be shown on the display. Instead the display provides a viewport into the framebuffer. An `ioctl` can be used to move the viewport.

`CYG_FB_FLAGS0_DOUBLE_BUFFER`

The display does not show the current contents of the framebuffer, so the results of drawing into the framebuffer are not immediately visible. Instead higher-level code needs to perform an explicit synch operation to update the display.

`CYG_FB_FLAGS0_PAGE_FLIPPING`

The hardware supports two or more pages, each of width*height pixels, only one of which is visible on the display. This allows higher-level code to update one page without disturbing what is currently visible. An `ioctl` is used to switch the visible page.

`CYG_FB_FLAGS0_BLANK`

The display can be blanked without affecting the framebuffer contents or settings.

`CYG_FB_FLAGS0_BACKLIGHT`

There is a backlight which can be switched on or off. Some hardware provides finer-grained control over the backlight intensity.

`CYG_FB_FLAGS0_MUST_BE_ON`

Often it is desirable to perform some initialization such as clearing the screen or setting the palette before the display is switched on, to avoid visual noise. However not all hardware allows this. If this flag is set then it is possible to access framebuffer memory and the palette before the `cyg_fb_on` or `CYG_FB_ON` operation. It may also be possible to perform some other operations such as activating the backlight, but that is implementation-defined.

To allow for future expansion there are also flags1, flags2, and flags3 fields. These may get used for encoding additional `ioctl` functionality, support for hardware acceleration, and similar features.

Linear Framebuffers

There are drawing primitives for writing and reading individual pixels. However these involve a certain amount of arithmetic each time to get from a position to an address within the frame buffer, plus function call overhead if the function API is used, and this will slow down graphics operations.

When the framebuffer device is known at compile-time and the macro API is used then there are additional macros specifically for iterating over parts of the frame buffer. These should prove very efficient for many graphics operations. However if the device is selected at run-time then the macros are not appropriate and code may want to manipulate framebuffer memory directly. This is possible if two conditions are satisfied:

1. The `CYG_FB_FLAGS0_LINEAR_FRAMEBUFFER` flag must be set. Otherwise framebuffer memory is either not directly accessible or has a non-linear layout.
2. The `CYG_FB_FLAGS0_DOUBLE_BUFFER` flag must be clear. An efficient double buffer synch operation requires knowing what part of the framebuffer have been updated, and the various drawing primitives will keep track of this. If higher-level code then starts manipulating the framebuffer directly the synch operation may perform only a partial update.

The base, stride, depth, width and height parameters, plus the `CYG_FB_FLAGS0_LE` flag for 1bpp, 2bpp and 4bpp devices, provide all the information needed to access framebuffer memory. A linear framebuffer has pixel (0,0) at the base address. Incrementing y means adding stride bytes to the pointer.

The base and stride parameters may be set even if `CYG_FB_FLAGS0_LINEAR_FRAMEBUFFER` is clear. This can be useful if for example the display is rotated in software from landscape to portrait mode. However the meaning of these parameters for non-linear framebuffers is implementation-defined.

Name

Control Operations — managing a framebuffer

Synopsis

#include <cyg/io/framebuf.h>

int **cyg_fb_on**(*fbdev*);

int **cyg_fb_off**(*fbdev*);

int **cyg_fb_ioctl**(*fbdev*, *key*, *data*, *len*);

int **CYG_FB_ON**(*FRAMEBUF*);

int **CYG_FB_OFF**(*FRAMEBUF*);

int **CYG_FB_IOCTL**(*FRAMEBUF*, *key*, *data*, *len*);

Description

The main operations on a framebuffer are drawing and colour management. However on most hardware it is also necessary to switch the display on before the user can see anything, and application code should be able to control when this happens. There are also miscellaneous operations such as manipulating the backlight or moving the viewpoint. These do not warrant dedicated functions, especially since the functionality will only be available on some hardware, so an ioctl interface is used.

Switching the Display On or Off

With most hardware nothing will be visible until there is a call to cyg_fb_on or an invocation of the CYG_FB_ON macro. This will initialize the framebuffer control circuitry, start sending the data signals to the display unit, and switch on the display if necessary. The exact initialization semantics are left to the framebuffer device driver. In some cases the hardware may already be partially or fully initialized by a static constructor or by boot code that ran before eCos.

There are some circumstances in which initialization can fail, and this is indicated by a POSIX error code such as ENODEV. An example would be plug and play hardware where the framebuffer device is not detected at run-time. Another example is hardware which can operate in several modes, with separate cyg_fb structures for each mode, if the hardware is already in use for a different mode. A return value of 0 indicates success.

Some but not all hardware allows the framebuffer memory and, if present, the palette to be manipulated before the device is switched on. That way the user does not see random noise on the screen during system startup. The flag CYG_FB_FLAGS0_MUST_BE_ON should be checked:

```
static void
init_screen(cyg_fb_colour background)
{
    int result;

#if (! (CYG_FB_FLAGS0(FRAMEBUF) & CYG_FB_FLAGS0_MUST_BE_ON))
    CYG_FB_FILL_BLOCK(FRAMEBUF, 0, 0,
                      CYG_FB_WIDTH(FRAMEBUF), CYG_FB_HEIGHT(FRAMEBUF),
                      background);
#endif

    result = CYG_FB_ON(FRAMEBUF);
    if (0 != result) {
        <handle unusual error condition>
    }
#if (CYG_FB_FLAGS0(FRAMEBUF) & CYG_FB_FLAGS0_MUST_BE_ON)
    CYG_FB_FILL_BLOCK(FRAMEBUF, 0, 0,
                      CYG_FB_WIDTH(FRAMEBUF), CYG_FB_HEIGHT(FRAMEBUF),
                      background);
#endif
}
```

Obviously if the application has already manipulated framebuffer memory or the palette but then the `cyg_fb_on` operation fails, the system is left in an undefined state.

It is also possible to switch a framebuffer device off, using the function `cyg_fb_off` or the macro `CYG_FB_OFF`, although this functionality is rarely used in embedded systems. The exact semantics of switching a device off are implementation-defined, but typically it involves shutting down the display, stopping the data signals to the display, and halting the control circuitry. The framebuffer memory and the palette are left in an undefined state, and application code should assume that both need full reinitializing when the device is switched back on. Some hardware may also provide a blank operation which typically just manipulates the display, not the whole framebuffer device. Normally `cyg_fb_on` returns 0. The API allows for a POSIX error code as with `cyg_fb_on`, but switching a device off is not an operation that is likely to fail.

If a framebuffer device can operate in several modes, represented by several cyg_fb structures and macro identifiers, then switching modes requires turning the current device off before turning the next one one.

Miscellaneous Control Operations

Some hardware functionality such as an LCD panel backlight is common but not universal. Supporting these does not warrant dedicated functions. Instead a catch-all `ioctl` interface is provided, with the arguments just passed straight to the device driver. This approach also allows for future expansion and for device-specific operations. `cyg_fb_ioctl` and `CYG_FB_IOCTL` take four arguments: a cyg_fb structure or framebuffer identifier; a key that specifies the operation to be performed; an arbitrary pointer, which should usually be a pointer to a data structure specific to the key; and a length field. Key values from 0 to 0x7fff are generic. Key values from 0x8000 onwards are reserved for the individual framebuffer device drivers, for device-specific functionality. The length field should be set to the size of the data structure, and may get updated by the device driver.

With most ioctl operations the device can indicate whether or not it supports the functionality by one of the flags, for example:

```
void
backlight_off(cyg_fb* fb)
{
    if (fb->fb_flags0 & CYG_FB_FLAGS0_BACKLIGHT) {
        cyg_fb_ioctl_backlight  new_setting;
        size_t                  len = sizeof(cyg_fb_ioctl_backlight);
        int                     result;

        new_setting.fbbl_current = 0;
        result = cyg_fb_ioctl(fb, CYG_FB_IOCTL_BACKLIGHT_SET,
                              &new_setting, &len);
        if (0 != result) {
            …
        }
    }
}
```

The operation returns zero for success or a POSIX error code on failure, for example `ENOSYS` if the device driver does not implement the requested functionality.

Viewport

```
# define CYG_FB_IOCTL_VIEWPORT_GET_POSITION     0x0100
# define CYG_FB_IOCTL_VIEWPORT_SET_POSITION     0x0101

typedef struct cyg_fb_ioctl_viewport {
    cyg_ucount16    fbvp_x;     // position of top-left corner of the viewport within
    cyg_ucount16    fbvp_y;     // the framebuffer
    cyg_ucount16    fbvp_when;  // set-only, now or vert retrace
} cyg_fb_ioctl_viewport;
```

On some targets the framebuffer device has a higher resolution than the display. Only a subset of the pixels, the viewport, is currently visible. Application code can exploit this functionality to achieve certain effects, for example smooth scrolling. Framebuffers which support this functionality will have the `CYG_FB_FLAGS0_VIEWPORT` flag set. The viewport dimensions are available as additional parameters to the normal framebuffer width and height.

The current position of the viewport can be obtained using an `CYG_FB_IOCTL_VIEWPORT_GET_POSITION` ioctl operation. The data argument should be a pointer to a cyg_fb_ioctl_viewport structure. On return the *`fbvp_x`* and *`fbvp_y`* fields will be filled in. To move the viewport use `CYG_FB_IOCTL_VIEWPORT_SET_POSITION` with *`fbvp_x`* and *`fbvp_y`* set to the top left corner of the new viewport within the framebuffer, and *`fbvp_when`* set to either `CYG_FB_UPDATE_NOW` or `CYG_FB_UPDATE_VERTICAL_RETRACE`. If the device driver cannot easily synchronize to a vertical retrace period then this last field is ignored.

```
void
move_viewport(cyg_fb* fb, int dx, int dy)
{
#ifdef CYGHWR_IO_FRAMEBUF_FUNCTIONALITY_VIEWPORT
    cyg_fb_ioctl_viewport viewport;
    int len = sizeof(cyg_fb_ioctl_viewport);
    int result;

    result = cyg_fb_ioctl(fb, CYG_FB_IOCTL_VIEWPORT_GET_POSITION,
                          &viewport, &len);
    if (result != 0) {
        …
    }
    if (((int)viewport.fbvp_x + dx) < 0) {
        viewport.fbvp_x = 0;
    } else if ((viewport.fbvp_x + dx + fb->fb_viewport_width) > fb->fb_width) {
        viewport.fbvp_x = fb->fb_width - fb->fb_viewport_width;
    } else {
        viewport.fbvp_x += dx;
    }
    if (((int)viewport.fbvp_y + dy) < 0) {
        viewport.fbvp_y = 0;
    } else if ((viewport.fbvp_y + dy + fb->fb_viewport_height) > fb->fb_height) {
        viewport.fbvp_y = fb->fb_height - fb->fb_viewport_height;
    } else {
        viewport.fbvp_y += dy;
    }
    result = cyg_fb_ioctl(fb, CYG_FB_IOCTL_VIEWPORT_SET_POSITION,
                            &viewport, &len);
    if (result != 0) {
        …
    }
#else
    CYG_UNUSED_PARAM(cyg_fb*, fb);
    CYG_UNUSED_PARAM(int, dx);
    CYG_UNUSED_PARAM(int, dy);
#endif
}
```

If an attempt is made to move the viewport beyond the boundaries of the framebuffer then the resulting behaviour is undefined. Some hardware may behave reasonably, wrapping around as appropriate, but portable code cannot assume this. The above code fragment is careful to clip the viewport to the framebuffer dimensions.

Page Flipping

```
# define CYG_FB_IOCTL_PAGE_FLIPPING_GET_PAGES   0x0200
# define CYG_FB_IOCTL_PAGE_FLIPPING_SET_PAGES   0x0201
```

```
typedef struct cyg_fb_ioctl_page_flip {
    cyg_uint32      fbpf_number_pages;
    cyg_uint32      fbpf_visible_page;
    cyg_uint32      fbpf_drawable_page;
    cyg_ucount16    fbpf_when;  // set-only, now or vert retrace
} cyg_fb_ioctl_page_flip;
```

On some targets the framebuffer has enough memory for several pages, only one of which is visible at a time. This allows the application to draw into one page while displaying another. Once drawing is complete the display is flipped to the newly drawn page, and the previously displayed page is now available for updating. This technique is used for smooth animation, especially in games. The flag `CYG_FB_FLAGS0_PAGE_FLIPPING` indicates support for this functionality.

`CYG_FB_IOCTL_PAGE_FLIPPING_GET_PAGES` can be used to get the current settings of the page flipping support. The data argument should be a pointer to a cyg_fb_ioctl_page_flip structure. The resulting *`fbpf_number_pages`* field indicates the total number of pages available: 2 is common, but more pages are possible. *`fbpf_visible_page`* gives the page that is currently visible to the user, and will be between 0 and (*`fbpf_number_pages`* - 1). Similarly *`fbpf_drawable_page`* gives the page that is currently visible. It is implementation-defined whether or not the visible and drawable page can be the same one.

`CYG_FB_IOCTL_PAGE_FLIPPING_SET_PAGES` can be used to change the visible and drawable page. The *`fbpf_number_pages`* field is ignored. *`fbpf_visible_page`* and *`fbpf_drawable_page`* give the new settings. *`fbpf_when`* should be one of `CYG_FB_UPDATE_NOW` or `CYG_FB_UPDATE_VERTICAL_RETRACE`, but may be ignored by some device drivers.

```
#if !(CYG_FB_FLAGS0(FRAMEBUF) & CYG_FB_FLAGS0_PAGE_FLIPPING)
# error Current framebuffer device does not support page flipping
#endif

static cyg_uint32 current_visible = 0;

static void
page_flip_init(cyg_fb_colour background)
{
    cyg_fb_ioctl_page_flip flip;
    size_t len = sizeof(cyg_fb_ioctl_page_flip);

    flip.fbpf_visible_page  = current_visible;
    flip.fbpf_drawable_page = 1 - current_visible;
    flip.fbpf_when          = CYG_FB_UPDATE_NOW;
    CYG_FB_IOCTL(FRAMEBUF, CYG_FB_IOCTL_PAGE_FLIPPING_SET_PAGES,
                 &flip, &len);
    CYG_FB_FILL_BLOCK(FRAMEBUF, 0, 0,
                      CYG_FB_WIDTH(FRAMEBUF), CYG_FB_HEIGHT(FRAMEBUF),
                      background);
    flip.fbpf_visible_page  = 1 - current_visible;
    flip.fbpf_drawable_page = current_visible;
    CYG_FB_IOCTL(FRAMEBUF, CYG_FB_IOCTL_PAGE_FLIPPING_SET_PAGES,
                 &flip, &len);
    CYG_FB_FILL_BLOCK(FRAMEBUF, 0, 0,
                      CYG_FB_WIDTH(FRAMEBUF), CYG_FB_HEIGHT(FRAMEBUF),
                      background);
    current_visible = 1 - current_visible;
}

static void
page_flip_toggle(void)
{
    cyg_fb_ioctl_page_flip flip;
    size_t len = sizeof(cyg_fb_ioctl_page_flip);

    flip.fbpf_visible_page  = 1 - current_visible;
    flip.fbpf_drawable_page = current_visible;
    CYG_FB_IOCTL(FRAMEBUF, CYG_FB_IOCTL_PAGE_FLIPPING_SET_PAGES,
                 &flip, &len);
    current_visible = 1 - current_visible;
}
```

A page flip typically just changes a couple of pointers within the hardware and device driver. No attempt is made to synchronize the contents of the pages, that is left to higher-level code.

Blanking the Screen

```
# define CYG_FB_IOCTL_BLANK_GET                   0x0300
# define CYG_FB_IOCTL_BLANK_SET                   0x0301

typedef struct cyg_fb_ioctl_blank {
    cyg_bool        fbbl_on;
} cyg_fb_ioctl_blank;
```

Some hardware allows the display to be switched off or blanked without shutting down the entire framebuffer device, greatly reducing power consumption. The current blanking state can be obtained using `CYG_FB_IOCTL_BLANK_GET` and the state can be updated using `CYG_FB_IOCTL_BLANK_SET`. The data argument should be a pointer to a cyg_fb_ioctl_blank structure. Support for this functionality is indicated by the `CYG_FB_FLAGS0_BLANK` flag.

```
static cyg_bool
display_blanked(cyg_fb_* fb)
{
    cyg_fb_ioctl_blank blank;
    size_t len = sizeof(cyg_fb_ioctl_blank);

    if (! (fb->fb_flags0 & CYG_FB_FLAGS0_BLANK)) {
        return false;
    }
    (void) cyg_fb_ioctl(fb, CYG_FB_IOCTL_BLANK_GET, &blank, &len);
    return !blank.fbbl_on;
}
```

Controlling the Backlight

```
# define CYG_FB_IOCTL_BACKLIGHT_GET               0x0400
# define CYG_FB_IOCTL_BACKLIGHT_SET               0x0401

typedef struct cyg_fb_ioctl_backlight {
    cyg_ucount32    fbbl_current;
    cyg_ucount32    fbbl_max;
} cyg_fb_ioctl_backlight;
```

Many LCD panels provide some sort of backlight, making the display easier to read at the cost of increased power consumption. Support for this is indicated by the `CYG_FB_FLAGS0_BACKLIGHT` flag. `CYG_FB_IOCTL_BACKLIGHT_GET` can be used to get both the current setting and the maximum value. If the maximum is 1 then the backlight can only be switched on or off. Otherwise it is possible to control the intensity.

```
static void
set_backlight_50_percent(void)
{
#if (CYG_FB_FLAGS0(FRAMEBUF) & CYG_FB_FLAGS0_BACKLIGHT)
    cyg_fb_ioctl_backlight backlight;
    size_t len = sizeof(cyg_fb_ioctl_backlight);

    CYG_FB_IOCTL(FRAMEBUF, CYG_FB_IOCTL_BACKLIGHT_GET, &backlight, &len);
    backlight.fbbl_current = (backlight.fbbl_max + 1) >> 1;
    CYG_FB_IOCTL(FRAMEBUF, CYG_FB_IOCTL_BACKLIGHT_SET, &backlight, &len);
#endif
}
```

Name

Colours — formats and palette management

Synopsis

```
#include <cyg/io/framebuf.h>

typedef struct cyg_fb {
    cyg_ucount16    fb_depth;
    cyg_ucount16    fb_format;
    cyg_uint32      fb_flags0;
    …
} cyg_fb;

extern const cyg_uint8  cyg_fb_palette_ega[16 * 3];
extern const cyg_uint8  cyg_fb_palette_vga[256 * 3];

#define CYG_FB_DEFAULT_PALETTE_BLACK        0x00
#define CYG_FB_DEFAULT_PALETTE_BLUE         0x01
#define CYG_FB_DEFAULT_PALETTE_GREEN        0x02
#define CYG_FB_DEFAULT_PALETTE_CYAN         0x03
#define CYG_FB_DEFAULT_PALETTE_RED          0x04
#define CYG_FB_DEFAULT_PALETTE_MAGENTA      0x05
#define CYG_FB_DEFAULT_PALETTE_BROWN        0x06
#define CYG_FB_DEFAULT_PALETTE_LIGHTGREY    0x07
#define CYG_FB_DEFAULT_PALETTE_LIGHTGRAY    0x07
#define CYG_FB_DEFAULT_PALETTE_DARKGREY     0x08
#define CYG_FB_DEFAULT_PALETTE_DARKGRAY     0x08
#define CYG_FB_DEFAULT_PALETTE_LIGHTBLUE    0x09
#define CYG_FB_DEFAULT_PALETTE_LIGHTGREEN   0x0A
#define CYG_FB_DEFAULT_PALETTE_LIGHTCYAN    0x0B
#define CYG_FB_DEFAULT_PALETTE_LIGHTRED     0x0C
#define CYG_FB_DEFAULT_PALETTE_LIGHTMAGENTA 0x0D
#define CYG_FB_DEFAULT_PALETTE_YELLOW       0x0E
#define CYG_FB_DEFAULT_PALETTE_WHITE        0x0F
```

cyg_ucount16 **CYG_FB_FORMAT**(*framebuf*);

void **cyg_fb_read_palette**(*fb*, *first*, *count*, *data*);

void **cyg_fb_write_palette**(*fb*, *first*, *count*, *data*, *when*);

cyg_fb_colour **cyg_fb_make_colour**(*fb*, *r*, *g*, *b*);

void **cyg_fb_break_colour**(*fb*, *colour*, *r*, *g*, *b*);

void **CYG_FB_READ_PALETTE**(*FRAMEBUF*, *first*, *count*, *data*);

void **CYG_FB_WRITE_PALETTE**(*FRAMEBUF*, *first*, *count*, *data*, *when*);

cyg_fb_colour **CYG_FB_MAKE_COLOUR**(*FRAMEBUF*, *r*, *g*, *b*);

void **CYG_FB_BREAK_COLOUR**(*FRAMEBUF*, *colour*, *r*, *g*, *b*);

Description

Managing colours can be one of the most difficult aspects of writing graphics code, especially if that code is intended to be portable to many different platforms. Displays can vary from 1bpp monochrome, via 2bpp and 4bpp greyscale, through 4bpp and 8bpp paletted, and up to 16bpp and 32bpp true colour - and those are just the more common scenarios. The various drawing primitives like `cyg_fb_write_pixel` work in terms of cyg_fb_colour values, usually an unsigned integer. Exactly how the hardware interprets a cyg_fb_colour depends on the format.

Colour Formats

There are a number of ways of finding out how these values will be interpreted by the hardware:

1. The `CYG_FB_FLAGS0_TRUE_COLOUR` flag is set for all true colour displays. The format parameter can be examined for more details but this is not usually necessary. Instead code can use `cyg_fb_make_colour` or `CYG_FB_MAKE_COLOUR` to construct a cyg_fb_colour value from red, green and blue components.

2. If the `CYG_FB_FLAGS0_WRITEABLE_PALETTE` flag is set then a cyg_fb_colour value is an index into a lookup table known as the palette, and this table contains red, green and blue components. The size of the palette is determined by the display depth, so 16 entries for a 4bpp display and 256 entries for an 8bpp display. Application code or a graphics library can install its own palette so can control exactly what colour each cyg_fb_colour value corresponds to. Alternatively there is support for installing a default palette.

3. If `CYG_FB_FLAGS0_PALETTE` is set but `CYG_FB_FLAGS0_WRITEABLE_PALETTE` is clear then the hardware uses a fixed palette. There is no easy way for portable software to handle this case. The palette can be read at run-time, allowing the application's desired colours to be mapped to whichever palette entry provides the best match. However normally it will be necessary to write code specifically for the fixed palette.

4. Otherwise the display is monochrome or greyscale, depending on the depth. There are still variations, for example on a monochrome display colour 0 can be either white or black.

As an alternative or to provide additional information, the exact colour format is provided by the *`fb_format`* field of the cyg_fb structure or by the `CYG_FB_FORMAT` macro. It can be one of the following (more entries may be added in future):

`CYG_FB_FORMAT_1BPP_MONO_0_BLACK`

simple 1bpp monochrome display, with 0 as black or the darker of the two colours, and 1 as white or the ligher colour.

`CYG_FB_FORMAT_1BPP_MONO_0_WHITE`

simple 1bpp monochrome display, with 0 as white or the lighter of the two colours, and 1 as black or the darker colour.

`CYG_FB_FORMAT_1BPP_PAL888`

a 1bpp display which cannot easily be described as monochrome. This is unusual and not readily supported by portable code. It can happen if the framebuffer normally runs at a higher depth, for example 4bpp or 8bpp paletted, but is run at only 1bpp to save memory. Hence only two of the palette entries are used, but can be set to arbitrary colours. The palette may be read-only or read-write.

`CYG_FB_FORMAT_2BPP_GREYSCALE_0_BLACK`

a 2bpp display offering four shades of grey, with 0 as black or the darkest of the four shades, and 3 as white or the lightest.

`CYG_FB_FORMAT_2BPP_GREYSCALE_0_WHITE`

a 2bpp display offering four shades of grey, with 0 as white or the lightest of the four shades, and 3 as black or the darkest.

`CYG_FB_FORMAT_2BPP_PAL888`

a 2bpp display which cannot easily be described as greyscale, for example providing black, red, blue and white as the four colours. This is unusual and not readily supported by portable code. It can happen if the framebuffer normally runs at a higher depth, for example 4bpp or 8bpp paletted, but is run at only 2bpp to save memory. Hence only four of the palette entries are used, but can be set to arbitrary colours. The palette may be read-only or read-write.

`CYG_FB_FORMAT_4BPP_GREYSCALE_0_BLACK`

a 4bpp display offering sixteen shades of grey, with 0 as black or the darkest of the 16 shades, and 15 as white or the lighest.

`CYG_FB_FORMAT_4BPP_GREYSCALE_0_WHITE`

a 4bpp display offering sixteen shades of grey, with 0 as white or the lightest of the 16 shades, and 15 as black or the darkest.

`CYG_FB_FORMAT_4BPP_PAL888`

a 4bpp paletted display, allowing for 16 different colours on screen at the same time. The palette may be read-only or read-write.

`CYG_FB_FORMAT_8BPP_PAL888`

an 8bpp paletted display, allowing for 256 different colours on screen at the same time. The palette may be read-only or read-write.

`CYG_FB_FORMAT_8BPP_TRUE_332`

an 8bpp true colour display, with three bits (eight levels) of red and green intensity and two bits (four levels) of blue intensity.

`CYG_FB_FORMAT_16BPP_TRUE_565`

a 16bpp true colour display with 5 bits each for red and blue and 6 bits for green.

`CYG_FB_FORMAT_16BPP_TRUE_555`

a 16bpp true colour display with five bits each for red, green and blue, and one unused bit.

`CYG_FB_FORMAT_32BPP_TRUE_0888`

a 32bpp true colour display with eight bits each for red, green and blue and eight bits unused.

For the true colour formats the format does not define exactly which bits in the pixel are used for which colour. Instead the `cyg_fb_make_colour` and `cyg_fb_break_colour` functions or the equivalent macros should be used to construct or decompose pixel values.

Paletted Displays

Palettes are the common way of implementing low-end colour displays. There are two variants. A read-only palette provides a fixed set of colours and it is up to application code to use these colours appropriately. A read-write palette allows the application to select its own set of colours. Displays providing a read-write palette will have the `CYG_FB_FLAGS0_WRITEABLE_PALETTE` flag set in addition to `CYG_FB_FLAGS0_PALETTE`.

Even if application code can install its own palette, many applications do not exploit this functionality and instead stick with a default. There are two standard palettes: the 16-entry PC EGA for 4bpp displays; and the 256-entry PC VGA, a superset of the EGA one, for 8bpp displays. This package provides the data for both, in the form of arrays `cyg_fb_palette_ega` and `cyg_fb_palette_vga`, and 16 `#define`'s such as `CYG_FB_DEFAULT_PALETTE_BLACK` for the EGA colours and the first 16 VGA colours. By default device drivers for read-write paletted displays will install the appropriate default palette, but this can be suppressed using configuration option `CYGFUN_IO_FRAMEBUF_INSTALL_DEFAULT_PALETTE`. If a custom palette will be used then installing the default palette involves wasting 48 or 768 bytes of memory.

It should be emphasized that displays vary widely. A colour such as `CYG_FB_DEFAULT_PALETTE_YELLOW` may appear rather differently on two different displays, although it should always be recognizable as yellow. Developers may wish to fine-tune the palette for specific hardware.

The current palette can be retrieved using `cyg_fb_read_palette` or `CYG_FB_READ_PALETTE`. The *`first`* and *`count`* arguments control which palette entries should be retrieved. For example, to retrieve just palette entry 12 *`first`* should be set to 12 and *`count`* should be set to 1. To retrieve all 256 entries for an 8bpp display, *`first`* should be set to 0 and *`count`* should be set to 256. The *`data`* argument should point at an array of bytes, allowing three bytes for every entry. Byte 0 will contain the red intensity for the first entry, byte 1 green and byte 2 blue.

For read-write palettes the palette can be updated using `cyg_fb_write_palette` or `CYG_FB_WRITE_PALETTE`. The *`first`* and *`count`* arguments are the same as for `cyg_fb_read_palette`, and the *`data`* argument should point at a suitable byte array packed in the same way. The *`when`* argument should be one of `CYG_FB_UPDATE_NOW` or `CYG_FB_UPDATE_VERTICAL_RETRACE`. With some displays updating the palette in the middle of an update may result in visual noise, so synchronizing to the vertical retrace avoids this. However not all device drivers will support this.

There is an assumption that palette entries use 8 bits for each of the red, green and blue colour intensities. This is not always the case, but the device drivers will perform appropriate adjustments. Some hardware may use only 6 bits per colour, and the device driver will ignore the bottom two bits of the supplied intensity values. Occasionally hardware may use more than 8 bits, in which case the supplied 8 bits are shifted left appropriately and zero-padded. Device drivers for such hardware may also provide device-specific routines to manipulate the palette in a non-portable fashion.

True Colour displays

True colour displays are often easier to manage than paletted displays. However this comes at the cost of extra memory. A 16bpp true colour display requires twice as much memory as an 8bpp paletted display, yet can offer only 32 or 64 levels of intensity for each colour as opposed to the 256 levels provided by a palette. It also requires twice as much video memory bandwidth to send all the pixel data to the display for every refresh, which may impact the performance of the rest of the system. A 32bpp true colour display offers the same colour intensities but requires four times the memory and four times the bandwidth.

Exactly how the colour bits are organized in a cyg_fb_colour pixel value is not defined by the colour format. Instead code should use the `cyg_fb_make_colour` or `CYG_FB_MAKE_COLOUR` primitives. These take 8-bit intensity levels for red, green and blue, and return the corresponding cyg_fb_colour. When using the macro interface the arithmetic happens at compile-time, for example:

```
#define BLACK        CYG_FB_MAKE_COLOUR(FRAMEBUF,   0,   0,   0)
#define WHITE        CYG_FB_MAKE_COLOUR(FRAMEBUF, 255, 255, 255)
#define RED          CYG_FB_MAKE_COLOUR(FRAMEBUF, 255,   0,   0)
#define GREEN        CYG_FB_MAKE_COLOUR(FRAMEBUF,   0, 255,   0)
#define BLUE         CYG_FB_MAKE_COLOUR(FRAMEBUF,   0,   0, 255)
#define YELLOW       CYG_FB_MAKE_COLOUR(FRAMEBUF, 255, 255,  80)
```

Displays vary widely so the numbers may need to be adjusted to give the exact desired colours.

For symmetry there are also `cyg_fb_break_colour` and `CYG_FB_BREAK_COLOUR` primitives. These take a cyg_fb_colour value and decompose it into its red, green and blue components.

Name

Drawing Primitives — updating the display

Synopsis

#include <cyg/io/framebuf.h>

void **cyg_fb_write_pixel**(*fbdev*, *x*, *y*, *colour*);

cyg_fb_colour **cyg_fb_read_pixel**(*fbdev*, *x*, *y*);

void **cyg_fb_write_hline**(*fbdev*, *x*, *y*, *len*, *colour*);

void **cyg_fb_write_vline**(*fbdev*, *x*, *y*, *len*, *colour*);

void **cyg_fb_fill_block**(*fbdev*, *x*, *y*, *width*, *height*, *colour*);

void **cyg_fb_write_block**(*fbdev*, *x*, *y*, *width*, *height*, *data*, *offset*, *stride*);

void **cyg_fb_read_block**(*fbdev*, *x*, *y*, *width*, *height*, *data*, *offset*, *stride*);

void **cyg_fb_move_block**(*fbdev*, *x*, *y*, *width*, *height*, *new_x*, *new_y*);

void **cyg_fb_synch**(*fbdev*, *when*);

void **CYG_FB_WRITE_PIXEL**(*FRAMEBUF*, *x*, *y*, *colour*);

cyg_fb_colour **CYG_FB_READ_PIXEL**(*FRAMEBUF*, *x*, *y*);

void **CYG_FB_WRITE_HLINE**(*FRAMEBUF*, *x*, *y*, *len*, *colour*);

void **CYG_FB_WRITE_VLINE**(*FRAMEBUF*, *x*, *y*, *len*, *colour*);

void **CYG_FB_FILL_BLOCK**(*FRAMEBUF*, *x*, *y*, *width*, *height*, *colour*);

void **CYG_FB_WRITE_BLOCK**(*FRAMEBUF*, *x*, *y*, *width*, *height*, *data*, *offset*, *stride*);

void **CYG_FB_READ_BLOCK**(*FRAMEBUF*, *x*, *y*, *width*, *height*, *data*, *offset*, *stride*);

void **CYG_FB_MOVE_BLOCK**(*FRAMEBUF*, *x*, *y*, *width*, *height*, *new_x*, *new_y*);

void **CYG_FB_SYNCH**(*FRAMEBUF*, *when*);

Description

The eCos framebuffer infrastructure defines a small number of drawing primitives. These are not intended to provide full graphical functionality like multiple windows, drawing text in arbitrary fonts, or anything like that. Instead they provide building blocks for higher-level graphical toolkits. The available primitives are:

1. Manipulating individual pixels.
2. Drawing horizontal and vertical lines.
3. Block fills.
4. Moving blocks between the framebuffer and main memory.

5. Moving blocks within the framebuffer.

6. For double-buffered devices, synchronizing the framebuffer contents with the actual display.

There are two versions for each primitive: a macro and a function. The macro can be used if the desired framebuffer device is known at compile-time. Its first argument should be a framebuffer identifier, for example `320x240x16`, and must be one of the entries in the configuration option `CYGDAT_IO_FRAMEBUF_DEVICES`. In the examples below it is assumed that `FRAMEBUF` has been `#define`'d to a suitable identifier. The function can be used if the desired framebuffer device is selected at run-time. Its first argument should be a pointer to the appropriate cyg_fb structure.

The pixel, line, and block fill primitives take a cyg_fb_colour argument. For details of colour handling see Framebuffer Colours. This argument should have no more bits set than are appropriate for the display depth. For example on a 4bpp only the bottom four bits of the colour may be set, otherwise the behaviour is undefined.

None of the primitives will perform any run-time error checking, except possibly for some assertions in a debug build. If higher-level code provides invalid arguments, for example trying to write a block which extends past the right hand side of the screen, then the system's behaviour is undefined. It is the responsibility of higher-level code to perform clipping to the screen boundaries.

Manipulating Individual Pixels

The primitives for manipulating individual pixels are very simple: a pixel can be written or read back. The following example shows one way of drawing a diagonal line:

```
void
draw_diagonal(cyg_fb* fb,
              cyg_ucount16 x, cyg_ucount16 y, cyg_ucount16 len,
              cyg_fb_colour colour)
{
    while ( len-- ) {
        cyg_fb_write_pixel(fb, x++, y++, colour);
    }
}
```

The next example shows how to draw a horizontal XOR line on a 1bpp display.

```
void
draw_horz_xor(cyg_ucount16 x, cyg_ucount16 y, cyg_ucount16 len)
{
    cyg_fb_colour colour;
    while ( len--) {
        colour = CYG_FB_READ_PIXEL(FRAMEBUF, x, y);
        CYG_FB_WRITE_PIXEL(FRAMEBUF, x++, y, colour ^ 0x01);
    }
}
```

The pixel macros should normally be avoided. Determining the correct location within framebuffer memory corresponding to a set of coordinates for each pixel is a comparatively expensive operation. Instead there is direct support for iterating over parts of the display, avoiding unnecessary overheads.

Drawing Simple Lines

Higher-level graphics code often needs to draw single-pixel horizontal and vertical lines. If the application involves multiple windows then these will usually have thin borders around them. Widgets such as buttons and scrollbars also often have thin borders.

`cyg_fb_draw_hline` and `CYG_FB_DRAW_HLINE` draw a horizontal line of the specified *`colour`*, starting at the *`x`* and *`y`* coordinates and extending to the right (increasing x) for a total of *`len`* pixels. A 50 pixel line starting at (100,100) will end at (149,100).

`cyg_fb_draw_vline` and `CYG_FB_DRAW_VLINE` take the same arguments, but the line extends down (increasing y).

These primitives do not directly support drawing lines more than one pixel thick, but block fills can be used to achieve those. There is no generic support for drawing arbitrary lines, instead that is left to higher-level graphics toolkits.

Block Fills

Filling a rectangular part of the screen with a solid colour is another common requirement for higher-level code. The simplest example is during initialization, to set the display's whole background to a known value. Block fills are also often used when creating new windows or drawing the bulk of a simple button or scrollbar widget. `cyg_fb_fill_block` and `CYG_FB_FILL_BLOCK` provide this functionality.

The *`x`* and *`y`* arguments specify the top-left corner of the block to be filled. The *`width`* and *`height`* arguments specify the number of pixels affected, a total of `width * height`. The following example illustrates part of the process for initializing a framebuffer, assumed here to have a writeable palette with default settings.

```
int
display_init(void)
{
    int result = CYG_FB_ON(FRAMEBUF);
    if ( result ) {
        return result;
    }
    CYG_FB_FILL_BLOCK(FRAMEBUF, 0, 0,
                      CYG_FB_WIDTH(FRAMEBUF), CYG_FB_HEIGHT(FRAMEBUF),
                      CYG_FB_DEFAULT_PALETTE_WHITE);
    …
}
```

Copying Blocks between the Framebuffer and Main Memory

The block transfer primitives serve two main purposes: drawing images. and saving parts of the current display to be restored later. For simple linear framebuffers the primitives just implement copy operations, with no data conversion of any sort. For non-linear ones the primitives act as if the framebuffer memory was linear. For example, consider a 2bpp display where the two bits for a single pixel are split over two separate bytes in framebuffer memory, or two planes. For a block write operation the source data should still be organized with four full pixels per byte, as for a linear framebuffer of the same depth. and the block write primitive will distribute the bits over the framebuffer memory as required. Similarly a block read will combine the appropriate bits from different locations in framebuffer memory and the resulting memory block will have four full pixels per byte.

Because the block transfer primitives perform no data conversion, if they are to be used for rendering images then those images should be pre-formatted appropriately for the framebuffer device. For small images this would normally happen on the host-side as part of the application build process. For larger images it will usually be better to store them in a compressed format and decompress them at run-time, trading off memory for cpu cycles.

The *`x`* and *`y`* arguments specify the top-left corner of the block to be transferred, and the *`width`* and *`height`* arguments determine the size. The *`data`*, *`offset`* and *`stride`* arguments determine the location and layout of the block in main memory:

`data`

The source or destination for the transfer. For 1bpp, 2bpp and 4bpp devices the data will be packed in accordance with the framebuffer device's endianness as per the `CYG_FB_FLAGS0_LE` flag. Each row starts in a new byte so there may be some padding on the right. For 16bpp and 32bpp the data should be aligned to the appropriate boundary.

`offset`

Sometimes only part of an image should be written to the screen. A vertical offset can be achieved simply by adjusting *`data`* to point at the appropriate row within the image instead of the top row. For 8bpp, 16bpp and 32bpp displays an additional horizontal offset can also be achieved by adjusting *`data`*. However for 1bpp, 2bpp and 4bpp displays the starting position within the image may be in the middle of a byte. Hence the horizontal pixel offset can instead be specified with the *`offset`* argument.

stride This indicates the number of bytes between rows. Usually it will be related to the *width*, but there are exceptions such as when drawing only part of an image.

The following example fills a 4bpp display with an image held in memory and already in the right format. If the image is smaller than the display it will be centered. If the image is larger then the center portion will fill the entire display.

```
void
draw_image(const void* data, int width, int height)
{
    cyg_ucount16 stride;
    cyg_ucount16 x, y, offset;

#if (4 != CYG_FB_DEPTH(FRAMEBUF))
# error This code assumes a 4bpp display
#endif

    stride = (width + 1) >> 1;  // 4bpp to byte stride

    if (width < CYG_FB_WIDTH(FRAMEBUF)) {
        x      = (CYG_FB_WIDTH(FRAMEBUF) - width) >> 1;
        offset = 0;
    } else {
        x      = 0;
        offset = (width - CYG_FB_WIDTH(FRAMEBUF)) >> 1;
        width  = CYG_FB_WIDTH(FRAMEBUF);
    }
    if (height < CYG_FB_HEIGHT(FRAMEBUF)) {
        y      = (CYG_FB_HEIGHT(FRAMEBUF) - height) >> 1;
    } else {
        y      = 0;
        data   = (const void*)((const cyg_uint8*)data +
                 (stride * ((height - CYG_FB_HEIGHT(FRAMEBUF)) >> 1));
        height = CYG_FB_HEIGHT(FRAMEBUF);
    }
    CYG_FB_WRITE_BLOCK(FRAMEBUF, x, y, width, height, data, offset, stride);
}
```

Moving Blocks with the Framebuffer

Sometimes it is necessary to move a block of data around the screen, especially when using a higher-level graphics toolkit that supports multiple windows. Block moves can be implemented by a read into main memory followed by a write block, but this is expensive and imposes an additional memory requirement. Instead the framebuffer infrastructure provides a generic block move primitive. It will handle all cases where the source and destination positions overlap. The *x* and *y* arguments specify the top-left corner of the block to be moved, and *width* and *height* determine the block size. *new_x* and *new_y* specify the destination. The source data will remain unchanged except in areas where it overlaps the destination.

Synchronizing Double-Buffered Displays

Some framebuffer devices are double-buffered: the framebuffer memory that gets manipulated by the drawing primitives is separate from what is actually displayed, and a synch operation is needed to update the display. In some cases this may be because the actual display memory is not directly accessible by the processor, for example it may instead be attached via an SPI bus. Instead drawing happens in a buffer in main memory, and then this gets transferred over the SPI bus to the actual display hardware during a synch. In other cases it may be a software artefact. Some drawing operations, especially ones involving complex curves, can take a very long time and it may be considered undesirable to have the user see this happening a few pixels at a time. Instead the drawing happens in a separate buffer in main memory and then a double buffer synch just involves a block move to framebuffer memory. Typically that block move is much faster than the drawing operation. Obviously there is a cost: an extra area of memory, and the synch operation itself can consume many cycles and much of the available memory bandwidth.

It is the responsibility of the framebuffer device driver to provide the extra main memory. As far as higher-level code is concerned the only difference between an ordinary and a double-buffered display is that with the latter changes do not become visible until a synch operation has been performed. The framebuffer infrastructure provides support for a bounding box, keeping track of what has been updated since the last synch. This means only the updated part of the screen has to be transferred to the display hardware.

The synch primitives take two arguments. The first identifies the framebuffer device. The second should be one of `CYG_FB_UPDATE_NOW` for an immediate update, or `CYG_FB_UPDATE_VERTICAL_RETRACE`. Some display hardware involves a lengthy vertical retrace period every 10-20 milliseconds during which nothing gets drawn to the screen, and performing the synch during this time means that the end user is unaware of the operation (assuming the synch can be completed in the time available). When the hardware supports it, specifying `CYG_FB_UPDATE_VERTICAL_RETRACE` means that the synch operation will block until the next vertical retrace takes place and then perform the update. This may be an expensive operation, for example it may involve polling a bit in a register. In a multi-threaded environment it may also be unreliable because the thread performing the synch may get interrupted or rescheduled in the middle of the operation. When the hardware does not involve vertical retraces, or when there is no easy way to detect them, the second argument to the synch operation will just be ignored and the update will always happen immediately.

It is up to higher level code to determine when a synch operation is appropriate. One approach for typical event-driven code is to perform the synch at the start of the event loop, just before waiting for an input or timer event. This may not be optimal. For example if there two small updates to opposite corners of the screen then it would be better to make two synch calls with small bounding boxes, rather than a single synch call with a a large bounding box that requires most of the framebuffer memory to be updated.

Leaving out the synch operations leads to portability problems. On hardware which does not involve double-buffering the synch operation is a no-op, usually eliminated at compile-time, so invoking synch does not add any code size or cpu cycle overhead. On double-buffered hardware, leaving out the synch means the user cannot see what has been drawn into the framebuffer.

Name

Pixel Manipulation — iterating over the display

Synopsis

#include <cyg/io/framebuf.h>

CYG_FB_PIXEL0_VAR (*FRAMEBUF*);

void **CYG_FB_PIXEL0_SET**(*FRAMEBUF*, *x*, *y*);

void **CYG_FB_PIXEL0_GET**(*FRAMEBUF*, *x*, *y*);

void **CYG_FB_PIXEL0_ADDX**(*FRAMEBUF*, *incr*);

void **CYG_FB_PIXEL0_ADDY**(*FRAMEBUF*, *incr*);

void **CYG_FB_PIXEL0_WRITE**(*FRAMEBUF*, *colour*);

cyg_fb_colour **CYG_FB_PIXEL0_READ**(*FRAMEBUF*);

void **CYG_FB_PIXEL0_FLUSHABS**(*FRAMEBUF*, *x0*, *y0*, *width*, *height*);

void **CYG_FB_PIXEL0_FLUSHREL**(*FRAMEBUF*, *x0*, *y0*, *dx*, *dy*);

Description

A common requirement for graphics code is to iterate over parts of the framebuffer. Drawing text typically involves iterating over a block of pixels for each character, say 8 by 8, setting each pixel to either a foreground or background colour. Drawing arbitrary lines typically involves moving to the start position and then adjusting the x and y coordinates until the end position is reached, setting a single pixel each time around the loop. Drawing images which are not in the frame buffer's native format typically involves iterating over a block of pixels, from top to bottom and left to right, setting pixels as the image is decoded.

Functionality like this can be implemented in several ways. One approach is to use the pixel write primitive. Typically this involves some arithmetic to get from the x and y coordinates to a location within framebuffer memory so it is fairly expensive compared with a loop which just increments a pointer. Another approach is to write the data first to a separate buffer in memory and then use a block write primitive to move it to the framebuffer, but again this involves overhead. The eCos framebuffer support provides a third approach: a set of macros specifically for iterating over the frame buffer. Depending on the operation being performed and the details of the framebuffer implementation, these macros may be optimal or near-optimal. Obviously there are limitations. Most importantly the framebuffer device must be known at compile-time: the compiler can do a better job optimizing the code if information such as the frame buffer width are constant. Also each iteration must be performed within a single variable scope: it is not possible to do some of the iteration in one function, some in another.

The Pixel Macros

All the pixel macros take a framebuffer identifier as their first argument. This is the same identifier that can be used with the other macros like `CYG_FB_WRITE_HLINE` and `CYG_FB_ON`, one of the entries in the configuration option `CYGDAT_IO_FRAMEBUF_DEVICES`. Using an invalid identifier will result in numerous compile-time error messages which may bear little resemblance to the original code. In the examples below it is assumed that `FRAMEBUF` has been `#define`'d to a suitable identifier.

Typical use of the pixel macros will look like this:

```
    CYG_FB_PIXEL0_VAR(FRAMEBUF);
    …
    CYG_FB_PIXEL0_FLUSHABS(FRAMEBUF, x, y, width, height);
```

The VAR macro will define one or more local variables to keep track of the current pixel position, as appropriate to the framebuffer device. The other pixel macros will then use these variables. For a simple 8bpp linear framebuffer there will be just a byte pointer. For a 1bpp display there may be several variables: a byte pointer, a bit index within that byte, and possibly a cached byte; using a cached value means that the framebuffer may only get read and written once for every 8 pixels, and the compiler may well allocate a register for the cached value; on some platforms framebuffer access will bypass the processor's main cache, so reading from or writing to framebuffer memory will be slow; reducing the number of framebuffer accesses may greatly improve performance.

Because the VAR macro defines one or more local variables it is normally placed at the start of a function or block, alongside other local variable definitions.

One the iteration has been completed there should be a FLUSHABS or FLUSHREL macro. This serves two purposes. First, if the local variables involve a dirty cached value or similar state then this will be written back. Second, for double-buffered displays the macro sets a bounding box for the part of the screen that has been updated. This allows the double buffer synch operation to update only the part of the display that has been modified, without having to keep track of the current bounding box for every updated pixel. For FLUSHABS the *x0* and *y0* arguments specify the top-left corner of the bounding box, which extends for *width* by *height* pixels. For FLUSHREL *x0* and *y0* still specify the top-left corner, but the bottom-right corner is now determined from the current pixel position offset by *dx* and *dy*. More specifically, *dx* should move the current horizontal position one pixel to the right of the right-most pixel modified, such that (x + dx) - x0 gives the width of the bounding box. Similarly *dy* should move the current vertical position one pixel below the bottom-most pixel modified. In typical code the current pixel position will already correspond in part or in whole to the bounding box corner, as a consequence of iterating over the block of memory.

If a pixel variable has been used only for reading framebuffer memory, not for modifying it, then it should still be flushed. A FLUSHABS with a width and height of 0 can be used to indicate that the bounding box is empty. If it is known that the framebuffer device being used does not support double-buffering then again it is possible to specify an empty bounding box. Otherwise portable code should specify a correct bounding box. If the framebuffer device that ends up being used does not support double buffering then the relevant macro arguments are eliminated at compile-time and do not result in any unnecessary code. In addition if there is no cached value or other state then the whole flush operation will be a no-op and no code will be generated.

Failure to perform the flush may result in strange drawing artefacts on some displays which can be very hard to debug. A FLUSHABS or FLUSHREL macro only needs to be invoked once, at the end of the iteration.

The SET macro sets the current position within the framebuffer. It can be used many times within an iteration. However it tends to be somewhat more expensive than ADDX or ADDY, so usually SET is only executed once at the start of an iteration.

```
    CYG_FB_PIXEL0_VAR(FRAMEBUF);
    CYG_FB_PIXEL0_SET(FRAMEBUF, x, y);
    …
    CYG_FB_PIXEL0_FLUSHREL(FRAMEBUF, x, y, 0, 0);
```

The GET macro retrieves the x and y coordinates corresponding to the current position. It is provided mainly for symmetry, but can prove useful for debugging.

```
    CYG_FB_PIXEL0_VAR(FRAMEBUF);
    CYG_FB_PIXEL0_SET(FRAMEBUF, x, y);
    …
#ifdef DEBUG
    CYG_FB_PIXEL0_GET(FRAMEBUF, new_x, new_y);
    diag_printf("Halfway through: x now %d, y now %d\n", new_x, new_y);
#endif
    …
    CYG_FB_PIXEL0_FLUSHREL(FRAMEBUF, x, y, 0, 0);
```

The ADDX and ADDY macros adjust the current position. The most common increments are 1 and -1, moving to the next or previous pixel horizontally or vertically, but any increment can be used.

```
    CYG_FB_PIXEL0_VAR(FRAMEBUF);
    CYG_FB_PIXEL0_SET(FRAMEBUF, x, y);
    for (rows = height; rows; rows--) {
        for (columns = width; columns; columns--) {
            <perform operation>
            CYG_FB_PIXEL0_ADDX(FRAMEBUF, 1);
        }
        CYG_FB_PIXEL0_ADDX(FRAMEBUF, -1 * width);
        CYG_FB_PIXEL0_ADDY(FRAMEBUF, 1);
    }
    CYG_FB_PIXEL0_FLUSHREL(FRAMEBUF, x, y, width, 0);
```

Here the current position is moved one pixel to the right each time around the inner loop. In the outer loop the position is first moved back to the start of the current row, then moved one pixel down. For the final flush the current x position is off by `width`, but the current y position is already correct.

The final two macros `READ` and `WRITE` can be used to examine or update the current pixel value.

```
    CYG_FB_PIXEL0_VAR(FRAMEBUF);
    CYG_FB_PIXEL0_SET(FRAMEBUF, x, y);
    for (rows = height; rows; rows--) {
        for (columns = width; columns; columns--) {
            cyg_fb_colour colour = CYG_FB_PIXEL0_READ(FRAMEBUF);
            if (colour == colour_to_replace) {
                CYG_FB_PIXEL0_WRITE(FRAMEBUF, replacement);
            }
            CYG_FB_PIXEL0_ADDX(FRAMEBUF, 1);
        }
        CYG_FB_PIXEL0_ADDX(FRAMEBUF, -1 * width);
        CYG_FB_PIXEL0_ADDY(FRAMEBUF, 1);
    }
    CYG_FB_PIXEL0_FLUSHREL(FRAMEBUF, x, y, width, 0);
```

Concurrent Iterations

Although uncommon, in some cases application code may need to iterate over two or more blocks. An example might be an advanced block move where each copied pixel requires some processing. To support this there are `PIXEL1`, `PIXEL2` and `PIXEL3` variants of all the `PIXEL0` macros. For example:

```
    CYG_FB_PIXEL0_VAR(FRAMEBUF);
    CYG_FB_PIXEL1_VAR(FRAMEBUF);

    CYG_FB_PIXEL0_SET(FRAMEBUF, dest_x, dest_y_);
    CYG_FB_PIXEL1_SET(FRAMEBUF, source_x, source_y);
    for (rows = height; rows; rows--) {
        for (columns = width; columns; columns--) {
            colour = CYG_FB_PIXEL1_READ(FRAMEBUF);
            <do some processing on colour>
            CYG_FB_PIXEL0_WRITE(FRAMEBUF, colour);
            CYG_FB_PIXEL0_ADDX(FRAMEBUF, 1);
            CYG_FB_PIXEL1_ADDX(FRAMEBUF, 1);
        }
        CYG_FB_PIXEL0_ADDX(FRAMEBUF, -100);
        CYG_FB_PIXEL0_ADDY(FRAMEBUF, 1);
        CYG_FB_PIXEL1_ADDX(FRAMEBUF, -100);
        CYG_FB_PIXEL1_ADDY(FRAMEBUF, 1);
    }

    CYG_FB_PIXEL0_FLUSHABS(FRAMEBUF, source_x, source_y, width, height);
    CYG_FB_PIXEL1_FLUSHABS(FRAMEBUF, 0, 0, 0, 0);  // Only used for reading
```

The `PIXEL0`, `PIXEL1`, `PIXEL2` and `PIXEL3` macros all use different local variables so there are no conflicts. The variable names also depend on the framebuffer device. If the target has two displays and two active framebuffer devices then the pixel macros can be used with the two devices without conflict:

```
    CYG_FB_PIXEL0_VAR(FRAMEBUF0);
    CYG_FB_PIXEL0_VAR(FRAMEBUF1);
    …
```

Name

Porting — writing a new framebuffer device driver

Description

As with most device drivers, the easiest way to write a new framebuffer package is to start with an existing one. Suitable ones include the PC VGA mode13 driver, an 8bpp paletted display, and the ARM iPAQ driver, a 16bpp true colour display. This document only outlines the process.

Before writing any code it is necessary to decide how many framebuffer devices should be provided by the device driver. Each such device requires a cyg_fb structure and appropriate functions, and an identifier for use with the macro API plus associated macros. There are no hard rules here. Some device drivers may support just a single device, others may support many devices which drive the hardware in different modes or orientations. Optional functionality such as viewports and page flipping may be supported by having different cyg_fb devices, or by a number of configuration options which affect a single cyg_fb device. Usually providing multiple cyg_fb structures is harmless because the unused ones will get eliminated at link-time.

Configuration

The CDL for a framebuffer package is usually straightforward. A framebuffer package should be a hardware package and reside in the `devs/framebuf` hierarchy, further organized by architecture. Generic framebuffer packages, if any, can go into a `generic` subdirectory, and will normally rely on the platform HAL to provide some platform-specific information such as base addresses. The package should be part of the target definition and hence loaded automatically, but should be `active_if CYGPKG_IO_FRAMEBUF` so that the driver only gets built if the generic framebuffer support is explicitly added to the configuration.

The configuration option `CYGDAT_IO_FRAMEBUF_DEVICES` should hold all the valid identifiers which can be used as the first argument for the macro API. This helps application developers to select the appropriate identifier, and allows higher-level graphics library packages to check that they have been configured correctly. This is achieved using something like the following, where `mode13_320x200x8` is a valid identifier for the PC VGA driver:

```
  requires { is_substr(CYGDAT_IO_FRAMEBUF_DEVICES, " mode13_320x200x8 ") }
```

The spaces ensure that the CDL inference engine keeps the identifiers separate.

`CYGPKG_IO_FRAMEBUF` contains a number of interfaces which should be implemented by individual device drivers when appropriate. This is used to eliminate some code or data structure fields at compile-time, keeping down memory requirements. The interfaces are `CYGHWR_IO_FRAMEBUF_FUNCTIONALITY_32BPP`, `CYGHWR_IO_FRAMEBUF_FUNCTIONALITY_TRUE_COLOUR`, `CYGHWR_IO_FRAMEBUF_FUNCTIONALITY_PALETTE`, `CYGHWR_IO_FRAMEBUF_FUNCTIONALITY_WRITEABLE_PALETTE`, `CYGHWR_IO_FRAMEBUF_FUNCTIONALITY_DOUBLE_BUFFER`, and `CYGHWR_IO_FRAMEBUF_FUNCTIONALITY_VIEWPORT`. For example if a device driver provides a true colour display but fails to implement the relevant interface then functions like `cyg_fb_make_colour` will be no-ops.

Device drivers for paletted displays should observe the generic configuration option `CYGFUN_IO_FRAMEBUF_INSTALL_DEFAULT_PALETTE` and install either `cyg_fb_palette_ega` or `cyg_fb_palette_vga` as part of their `cyg_fb_on` implementation.

Exported Header File(s)

Each framebuffer device driver should export one or more header files to `cyg/io/framebufs`. A custom build step in `CYGPKG_IO_FRAMEBUF` ensures that application code can just `#include cyg/io/framebuf.h` and this will automatically include the device-specific headers. Drivers may export one header per cyg_fb device or a single header for all devices, without affecting any code outside the device driver.

Each exported header serves two purposes. First it defines the parameters, drawing primitive macros, and iteration macros for each device. Second it declares the cyg_fb structure.

Parameters

The parameter section should resemble the following:

```
#define CYG_FB_320x240x16_STRUCT            cyg_ipaq_fb_320x240x16
#define CYG_FB_320x240x16_DEPTH             16
#define CYG_FB_320x240x16_FORMAT            CYG_FB_FORMAT_16BPP_TRUE_565
#define CYG_FB_320x240x16_WIDTH             320
#define CYG_FB_320x240x16_HEIGHT            240
#define CYG_FB_320x240x16_VIEWPORT_WIDTH    320
#define CYG_FB_320x240x16_VIEWPORT_HEIGHT   240
#define CYG_FB_320x240x16_FLAGS0            (CYG_FB_FLAGS0_LINEAR_FRAMEBUFFER | \
                                             CYG_FB_FLAGS0_TRUE_COLOUR        | \
                                             CYG_FB_FLAGS0_BLANK              | \
                                             CYG_FB_FLAGS0_BACKLIGHT)
#define CYG_FB_320x240x16_FLAGS1            0
#define CYG_FB_320x240x16_FLAGS2            0
#define CYG_FB_320x240x16_FLAGS3            0
#define CYG_FB_320x240x16_BASE              ((void*)0x01FC0020)
#define CYG_FB_320x240x16_STRIDE            640
```

Here `320x240x16` is the framebuffer identifier for use with the macro API. Application code like:

```
#define FRAMEBUF 320x240x16
cyg_ucount16 width = CYG_FB_WIDTH(FRAMEBUF);
```

will end up using the `CYG_FB_320x240x16_WIDTH` definition. To allow for efficient portable code all parameters must be compile-time constants. If the hardware may allow some of the parameters to be varied, for example different resolutions, then this should be handled either by defining separate devices for each resolution or by configuration options.

The viewport width and height should always be defined. If the device driver does not support a viewport then these will be the same as the standard width and height.

To allow for future expansion there are `FLAGS1`, `FLAGS2` and `FLAGS3` parameters. No flags are defined for these at present, but device drivers should still define the parameters.

Drawing Primitives

For each device the exported header file should define macros for the drawing primitives, using the same naming convention as for parameters. In the case of true colour displays there should also be macros for the make-colour and break-colour primitives:

```
#define CYG_FB_320x240x16_WRITE_PIXEL(_x_, _y_, _colour_) …
#define CYG_FB_320x240x16_READ_PIXEL(_x_, _y_) …
#define CYG_FB_320x240x16_WRITE_HLINE(_x_, _y_, _len_, _colour_) …
#define CYG_FB_320x240x16_WRITE_VLINE(_x_, _y_, _len_, _colour_) …
#define CYG_FB_320x240x16_FILL_BLOCK(_x_, _y_, _w_, _h_, _colour_) …
#define CYG_FB_320x240x16_WRITE_BLOCK(_x_, _y_, _w_, _h_, _data_, _off_, _s_) …
#define CYG_FB_320x240x16_READ_BLOCK(_x_, _y_, _w_, _h_, _data_, _off_, _s_) …
#define CYG_FB_320x240x16_MOVE_BLOCK(_x_, _y_, _w_, _h_, _new_x_, _new_y_) …
#define CYG_FB_320x240x16_MAKE_COLOUR(_r_, _g_, _b_) …
#define CYG_FB_320x240x16_BREAK_COLOUR(_colour_, _r_, _g_, _b_) …
```

For typical linear framebuffers there are default implementations of all of these primitives in the generic framebuffer package, held in the exported header `cyg/io/framebuf.inl`. Hence the definitions will typically look something like:

```
#include <cyg/io/framebuf.inl>
…
#define CYG_FB_320x240x16_WRITE_PIXEL(_x_, _y_, _colour_)         \
    CYG_MACRO_START                                               \
    cyg_fb_linear_write_pixel_16_inl(CYG_FB_320x240x16_BASE,      \
                                     CYG_FB_320x240x16_STRIDE,    \
                                     _x_, _y_, _colour_);         \
    CYG_MACRO_END
```

```
#define CYG_FB_320x240x16_READ_PIXEL(_x_, _y_)                    \
    ({ cyg_fb_linear_read_pixel_16_inl(CYG_FB_320x240x16_BASE,    \
                                       CYG_FB_320x240x16_STRIDE,  \
                                       _x_, _y_); })
…
```

All of the drawing primitives have variants for the common display depths and layouts: 1le, 1be, 2le, 2be, 4le, 4be, 8, 16 and 32. The inlines take the framebuffer memory base address as the first argument, and the stride in bytes as the second. Similarly there are default definitions of the true colour primitives for `8BPP_TRUE_332`, `16BPP_TRUE_565`, `16BPP_TRUE_555`, and `32BPP_TRUE_0888`:

```
#define CYG_FB_320x240x16_MAKE_COLOUR(_r_, _g_, _b_)             \
    ({  CYG_FB_MAKE_COLOUR_16BPP_TRUE_565(_r_, _g_, _b_); })
#define CYG_FB_320x240x16_BREAK_COLOUR(_colour_, _r_, _g_, _b_)  \
    CYG_MACRO_START                                              \
    CYG_FB_BREAK_COLOUR_16BPP_TRUE_565(_colour_, _r_, _g_, _b_); \
    CYG_MACRO_END
```

These default definitions assume the most common layout of colours within a pixel value, so for example `CYG_FB_MAKE_COLOUR_16BPP_TRUE_565` assumes bits 0 to 4 hold the blue intensity, bits 5 to 10 the green, and bits 11 to 15 the red.

If the hardware does not implement a linear framebuffer then obviously writing the device driver will be significantly more work. The macros will have to perform the operations themselves instead of relying on generic implementations. The required functionality should be obvious, and the generic implementations can still be consulted as a reference. For complicated hardware it may be appropriate to map the macros onto function calls, rather than try to implement everything inline.

Note

At the time of writing the support for linear framebuffers is incomplete. Only 8bpp, 16bpp and 32bpp depths have full support. There may also be future extensions, for example `r90`, `r180` and `r270` variants to support rotation in software, and `db` variants to support double-buffered displays.

Iteration Macros

In addition to the drawing primitives the exported header file should define iteration macros:

```
#define CYG_FB_320x240x16_PIXELx_VAR(  _fb_, _id_) …
#define CYG_FB_320x240x16_PIXELx_SET(  _fb_, _id_, _x_, _y_) …
#define CYG_FB_320x240x16_PIXELx_GET(  _fb_, _id_, _x_, _y_) …
#define CYG_FB_320x240x16_PIXELx_ADDX( _fb_, _id_, _incr_) …
#define CYG_FB_320x240x16_PIXELx_ADDY( _fb_, _id_, _incr_) …
#define CYG_FB_320x240x16_PIXELx_WRITE(_fb_, _id_, _colour_) …
#define CYG_FB_320x240x16_PIXELx_READ( _fb_, _id_)…
#define CYG_FB_320x240x16_PIXELx_FLUSHABS( _fb_, _id_, _x0_, _y0_, _w_, _h_) …
#define CYG_FB_320x240x16_PIXELx_FLUSHREL( _fb_, _id_, _x0_, _y0_, _dx_, _dy_) …
```

The *`_fb_`* argument will be the identifier, in this case `320x240x16`, and the *`_id_`* will be a small number, 0 for a `PIXEL0` iteration, 1 for `PIXEL1`, and so on. Together these two should allow unique local variable names to be constructed. Again there are default definitions of the macros in `cyg/io/framebuf.inl` for linear framebuffers:

```
#define CYG_FB_320x240x16_PIXELx_VAR(  _fb_, _id_) \
    CYG_FB_PIXELx_VAR_16(  _fb_, _id_)
#define CYG_FB_320x240x16_PIXELx_SET(  _fb_, _id_, _x_, _y_)    \
    CYG_MACRO_START                                             \
    CYG_FB_PIXELx_SET_16( _fb_, _id_,                           \
                          CYG_FB_320x240x16_BASE,               \
                          320, _x_, _y_);                       \
    CYG_MACRO_END
```

The linear `SET` and `GET` macros take base and stride information. The `ADDX` and `ADDY` macros only need the stride. By convention most of the macros are wrapped in `CYG_MACRO_START`/`CYG_MACRO_END` or `({`/`})` pairs, allowing debug code to be inserted if necessary. However the _VAR macro must not be wrapped in this way: its purpose is to define one or more local variables; wrapping the macro would declare the variables in a new scope, inaccessible to the other macros.

Again for non-linear framebuffers it will be necessary to implement these macros fully rather than rely on generic implementations, but the generic versions can be consulted as a reference.

The cyg_fb declaration

Finally there should be an export of the cyg_fb structure or structures. Typically this uses the _STRUCT parameter, reducing the possibility of an accidental mismatch between the macro and function APIs:

```
extern cyg_fb   CYG_FB_320x240x16_STRUCT;
```

Driver-Specific Source Code

Exporting parameters and macros in a header file is not enough. It is also necessary to actually define the cyg_fb structure or structures, and to provide hardware-specific versions of the control operations. For non-linear framebuffers it will also be necessary to provide the drawing functions. There is a utility macro CYG_FB_FRAMEBUFFER for instantiating a cyg_fb structure. Drivers may ignore this macro and do the work themselves, but at an increased risk of compatibility problems with future versions of the generic code.

```
CYG_FB_FRAMEBUFFER(CYG_FB_320x240x16_STRUCT,
                   CYG_FB_320x240x16_DEPTH,
                   CYG_FB_320x240x16_FORMAT,
                   CYG_FB_320x240x16_WIDTH,
                   CYG_FB_320x240x16_HEIGHT,
                   CYG_FB_320x240x16_VIEWPORT_WIDTH,
                   CYG_FB_320x240x16_VIEWPORT_HEIGHT,
                   CYG_FB_320x240x16_BASE,
                   CYG_FB_320x240x16_STRIDE,
                   CYG_FB_320x240x16_FLAGS0,
                   CYG_FB_320x240x16_FLAGS1,
                   CYG_FB_320x240x16_FLAGS2,
                   CYG_FB_320x240x16_FLAGS3,
                   0, 0, 0, 0,   // fb_driver0 -> fb_driver3
                   &cyg_ipaq_fb_on,
                   &cyg_ipaq_fb_off,
                   &cyg_ipaq_fb_ioctl,
                   &cyg_fb_nop_synch,
                   &cyg_fb_nop_read_palette,
                   &cyg_fb_nop_write_palette,
                   &cyg_fb_dev_make_colour_16bpp_true_565,
                   &cyg_fb_dev_break_colour_16bpp_true_565,
                   &cyg_fb_linear_write_pixel_16,
                   &cyg_fb_linear_read_pixel_16,
                   &cyg_fb_linear_write_hline_16,
                   &cyg_fb_linear_write_vline_16,
                   &cyg_fb_linear_fill_block_16,
                   &cyg_fb_linear_write_block_16,
                   &cyg_fb_linear_read_block_16,
                   &cyg_fb_linear_move_block_16,
                   0, 0, 0, 0 // fb_spare0 -> fb_spare3
);
```

The first 13 arguments to the macro correspond to the device parameters. The next four are arbitrary CYG_ADDRWORD values for use by the device driver. Typically these are used to share on/off/ioctl functions between multiple cyg_fb structure. They are followed by function pointers: on/off/ioctl control; double buffer synch; palette management; true colour support; and the drawing primitives. nop versions of the on, off, ioctl, synch, palette management and true colour functions are provided by the generic framebuffer package, and often these arguments to the CYG_FB_FRAMEBUFFER macro will be discarded at compile-time because the relevant CDL interface is not implemented. The final four arguments are currently unused and should be 0. They are intended for future expansion, with a value of 0 indicating that a device driver does not implement non-core functionality.

As with the macros there are default implementations of the true colour primitives for `8bpp_true_332`, `16bpp_true_565`, `16bpp_true_555` and `32bpp_true_0888`, assuming the most common layout for these colour modes. There are also default implementations of the drawing primitives for linear framebuffers, with variants for the common display depths and layouts. Obviously non-linear framebuffers will need rather more work.

Typically a true colour or grey scale framebuffer device driver will have to implement just three hardware-specific functions:

```
int
cyg_ipaq_fb_on(cyg_fb* fb)
{
    …
}

int
cyg_ipaq_fb_off(cyg_fb* fb)
{
    …
}

int
cyg_ipaq_fb_ioctl(cyg_fb* fb, cyg_ucount16 key, void* data, size_t* len)
{
    int result;

    switch(key) {
        case CYG_FB_IOCTL_BLANK_GET: …
        …
        default: result = ENOSYS; break;
    }
    return result;
}
```

These control operations are entirely hardware-specific and cannot be implemented by generic code. Paletted displays will need two more functions, again hardware-specific:

```
void
cyg_pcvga_fb_read_palette(cyg_fb* fb, cyg_ucount32 first, cyg_ucount32 len,
                          void* data)
{
    …
}

void
cyg_pcvga_fb_write_palette(cyg_fb* fb, cyg_ucount32 first,  cyg_ucount32 len,
                           const void* data, cyg_ucount16 when)
{
    …
}
```

Future Expansion

As has been mentioned before framebuffer hardware varies widely. The design of a generic framebuffer API requires complicated trade-offs between efficiency, ease of use, ease of porting, and still supporting a very wide range of hardware. To some extent this requires a lowest common denominator approach, but the design allows for some future expansion and optional support for more advanced features like hardware acceleration.

The most obvious route for expansion is the `ioctl` interface. Device drivers can define their own keys, values `0x8000` and higher, for any operation. Alternatively a device driver does not have to implement just the interface provided by the generic framebuffer package: additional functions and macros can be exported as required.

Currently there are only a small number of `ioctl` operations. Additional ones may get added in future, for example to support a hardware mouse cursor, but only in cases where the functionality is likely to be provided by a significant number of framebuffer devices. Adding new generic functionality adds to the maintenance overhead of both code and documentation. When a new generic ioctl operation is added there will usually also be one or more new flags, so that device drivers can indicate they support the functionality. At the time of writing only 12 of the 32 FLAGS0 flags are used, and a further 96 are available in FLAGS1, FLAGS2 and FLAGS3.

Another route for future expansion is the four spare arguments to the CYG_FB_FRAMEBUFFER macro. As an example of how these may get used in future, consider support for 3d hardware acceleration. One of the spare fields would become another table of function pointers to the various accelerators, or possibly a structure. A FLAGS0 flag would indicate that the device driver implements such functionality.

Other forms of expansion such as defining a new standard drawing primitive would be more difficult, since this would normally involve changing the CYG_FB_FRAMEBUFFER macro. Such expansion should not be necessary because the existing primitives provide all reasonable core functionality. Instead other packages such as graphics libraries can work on top of the existing primitives.

Chapter 85. CSB337/900 Framebuffer Device Driver

Name

CYGPKG_DEVS_FRAMEBUF_ARM_CSB337900 — eCos framebuffer support for a CSB337/900

Description

This package provides an eCos framebuffer device driver for a Cogent CSB337 board with a CSB900 add-on to provide the LCD panel. It has dependencies on both pieces of hardware so cannot be used with any other combination. The driver is a hardware package and is loaded automatically when configuring eCos for a csb337900 target, but not when configuring for a vanilla csb337 target. By default it is inactive and does not add any code size or data overheads. To activate the driver the generic framebuffer package CYGPKG_IO_FRAMEBUF should be added to the configuration. The driver's functionality is only accessible via the API defined by the generic package.

There are a number of design issues with this hardware combination. The framebuffer memory cannot be accessed in the conventional linear way, Instead the driver contains custom drawing code which will be significantly slower than the equivalent linear framebuffer routines. There are also problems with the colour handling: only three bits of control are available for each of red, green and blue intensity,

The driver supports four cyg_fb structures: cyg_csb337900_fb_240x320x8, cyg_csb337900_fb_320x240x8r90, cyg_csb337900_fb_240x320x8r180, and cyg_csb337900_fb_320x240x8r270. These all run the hardware in the same resolution but in the four different orientations, using hardware to perform the rotation. The corresponding identifiers for the macro API are 240x320x8, 320x240x8r90, 240x320x8r180 and 320x240x8r270. Obviously only one of these framebuffer devices can be used at a time. All the devices implement 8bpp non-linear displays with a writeable palette. The supported ioctl operations are CYG_FB_IOCTL_BLANK_GET, CYG_FB_IOCTL_BLANK_SET, CYG_FB_IOCTL_BACKLIGHT_GET and CYG_FB_IOCTL_BACKLIGHT_SET. The backlight can only be switched on or off, there is no support for different levels of intensity.

Chapter 86. i.MXxx Framebuffer Device Driver

Name

CYGPKG_DEVS_FRAMEBUF_ARM_IMX — eCos Support for the i.MXxx framebuffer

Description

CYGPKG_DEVS_FRAMEBUF_ARM_IMX provides an eCos framebuffer device driver for the LCD panel on the i.MXxx family of processors. The driver is a hardware package and is loaded automatically when configuring eCos for an i.MXxx target. By default it is inactive and does not add any code size or data overheads. To activate the driver the generic framebuffer package CYGPKG_IO_FRAMEBUF should be added to the configuration. The driver's functionality is only accessible via the API defined by the generic package.

The driver supports a single framebuffer device, driving the LCD panel in 565 true colour mode with a resolution of 640x480 pixels at 16bpp. The cyg_fb structure for this is cyg_imx_fb_640x480x16, and the identifier for use with the framebuffer macro API is 640x480x16. The ioctl operations supported are CYG_FB_IOCTL_BLANK_GET, CYG_FB_IOCTL_BLANK_SET, CYG_FB_IOCTL_BACKLIGHT_GET and CYG_FB_IOCTL_BACKLIGHT_SET. The backlight supports 256 levels of intensity.

The driver must be configured for the LCD panel attached using the CYGPKG_DEVS_FRAMEBUF_ARM_IMX_LCD option. Normally this would be set by the platform HAL package using a requires statement. At present only one panel type is supported: the Chunghwa CLAA057VA01CT.

Chapter 87. iPAQ Framebuffer Device Driver

Name

CYGPKG_DEVS_FRAMEBUF_ARM_IPAQ — eCos Support for the iPAQ framebuffer

Description

CYGPKG_DEVS_FRAMEBUF_ARM_IPAQ provides an eCos framebuffer device driver for the LCD panel on an iPAQ. The driver is a hardware package and is loaded automatically when configuring eCos for an iPAQ target. By default it is inactive and does not add any code size or data overheads. To activate the driver the generic framebuffer package CYGPKG_IO_FRAMEBUF should be added to the configuration. The driver's functionality is only accessible via the API defined by the generic package.

The driver supports a single framebuffer device, driving the LCD panel in 565 true colour mode with a resolution of 320x240 pixels at 16bpp. The cyg_fb structure for this is cyg_ipaq_fb_320x240x16, and the identifier for use with the framebuffer macro API is 320x240x16. The ioctl operations supported are CYG_FB_IOCTL_BLANK_GET, CYG_FB_IOCTL_BLANK_SET, CYG_FB_IOCTL_BACKLIGHT_GET and CYG_FB_IOCTL_BACKLIGHT_SET. The backlight supports 32 levels of intensity.

Chapter 88. PC VGA Framebuffer Device Driver

Name

CYGPKG_DEVS_FRAMEBUF_I386_PCVGA — eCos Support for PC VGA Cards

Description

This package provides an eCos framebuffer device driver for a VGA graphics card in a standard PC. Although VGA is now very old technology, more recent PC graphics hardware usually still provides compatibility. The device driver works by interacting directly with the hardware, not via the video BIOS, so the level of compatibility must be very high. Hence the driver cannot be guaranteed to work with all PC graphics cards and should be used with care.

CYGPKG_DEVS_FRAMEBUF_I386_PCVGA is a hardware package and is loaded automatically when configuring eCos for a PC target. By default it is inactive and does not add any code size or data overheads. To activate the driver the generic framebuffer package CYGPKG_IO_FRAMEBUF should be added to the configuration. The driver's functionality is only accessible via the API defined by the generic package.

The driver supports just one of the VGA graphics modes, mode 13. This provides an 8bpp linear framebuffer with a resolution of 320x200 pixels, and uses a writeable palette for colour management. The cyg_fb structure for this is cyg_pcvga_fb_mode13_320x200x8, and the identifier for use with the framebuffer macro API is mode13_320x200x8. The only ioctl operations supported are CYG_FB_IOCTL_BLANK_GET and CYG_FB_IOCTL_BLANK_SET.

There is one configuration option specific to the PC VGA hardware. Some of the documentation for these cards recommends a delay after every VGA register access because not all graphics cards can keep up with full-speed I/O. By default the driver implements such a delay, but obviously this slows down certain operations. If the graphics card actually being used does not require this delay then the configuration option CYGHWR_DEVS_FRAMEBUF_I386_PCVGA_REGISTER_ACCESS_DELAY can be disabled.

Chapter 89. Synthetic Target Framebuffer Device

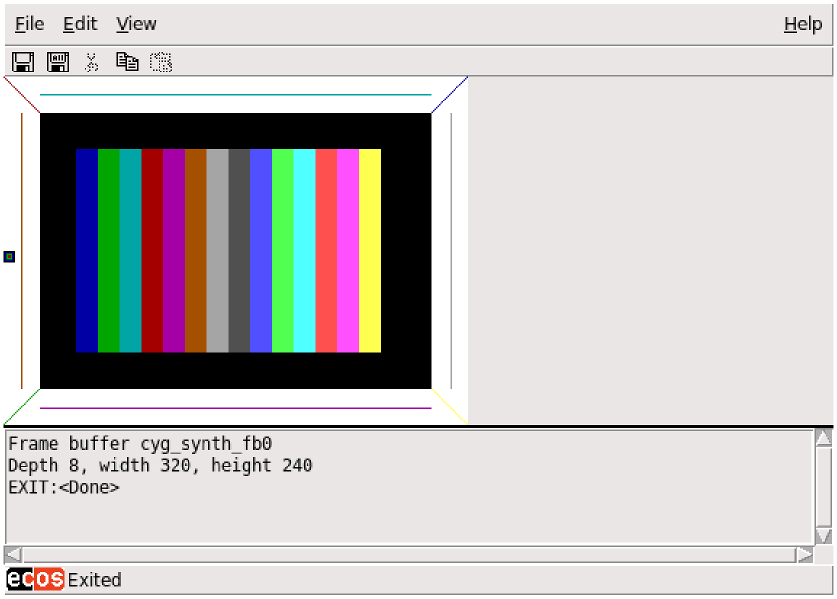
Name

Synthetic Target Framebuffer Device — Emulate framebuffer hardware in the synthetic target

Overview

This package CYGPKG_DEVS_FRAMEBUF_SYNTH provides a framebuffer device driver for the eCos synthetic target.

Figure 89.1. Synthetic Target Framebuffer X Window



```
Frame buffer cyg_synth_fb0
Depth 8, width 320, height 240
EXIT:<Done>
```

The driver supports up to four framebuffer devices fb0, fb1, fb2 and fb3. The width, height, depth, and display format of each framebuffer can be controlled via configuration options. It is also possible to set a viewport for each device and to enable page flipping.

To use the framebuffer support the eCos application must run inside an X session, not from the console, and it must be started with *--io* to enable the I/O auxiliary. The I/O auxiliary will start a separate instance of a host-side utility framebuf for each target-side framebuffer device. The framebuf utility can access the eCos framebuffer data via a shared memory region and draw it to the screen using X library calls. It needs the X server to run with a TrueColor visual and a display of depth of 24 or 32 bits per pixel.

Installation

The synthetic target framebuffer driver depends on host-side support which must be built and installed. The relevant code resides in the host subdirectory of the synthetic target framebuffer package, and building it involves the standard **configure**, **make** and **make install** steps. This will build and install a utility program framebuf that does the actual drawing of the eCos framebuffer contents to the host-side X display. It will also install a Tcl script and some support files. framebuf is an X11 application so can only be built on Linux systems with the appropriate X11 development package or packages.

There are two main ways of building the host-side software. It is possible to build both the generic host-side software and all package-specific host-side software, including the framebuffer support, in a single build tree. This involves using the **configure** script at the toplevel of the eCos repository. For more information on this, see the README.host file at the top of the repository. Note that if you have an existing build tree which does not include the synthetic target framebuffer support then it will be necessary to rerun the toplevel configure script: the search for appropriate packages happens at configure time.

The alternative is to build just the host-side for this package. This requires a separate build directory, building directly in the source tree is disallowed. The **configure** options are much the same as for a build from the toplevel, and the README.host file can be consulted for more details. It is essential that the framebuffer support be configured with the same `--prefix` option as other eCos host-side software, especially the I/O auxiliary provided by the architectural synthetic target HAL package, otherwise the I/O auxiliary will be unable to locate the framebuffer support.

Configuration

The package is loaded automatically when creating a configuration for the synthetic target. However it is inactive unless the generic framebuffer support `CYGPKG_IO_FRAMEBUF` is also added to the configuration, for example by `ecosconfig add framebuf`.

By default the package enables a single framebuffer device `fb0` with a corresponding cyg_fb data structure `cyg_synth_fb0`. The default settings for this device are 320 by 240 pixels, a depth of 8 bits per pixel, a paletted display, no viewport support, and no page flipping. All of these settings can be changed by configuration options inside the CDL component `CYGPKG_DEVS_FRAMEBUF_SYNTH_FB0`. The supported display formats are: 8 bpp paletted; 8bpp true colour 332; 16bpp true 565; 16bpp true 555; and 32bpp 0888. This allows the synthetic target to match the actual display capabilities of the hardware that is being emulated. If the actual hardware has more than one framebuffer device then this can be emulated by enabling additional components `CYGPKG_DEVS_FRAMEBUF_SYNTH_FB1` …, and setting the appropriate options.

Customization

In addition to the target-side configurability it is possible to customize the host-side behaviour. For example, the default behaviour is for `fb0` to be drawn inside the I/O auxiliary's main window, if it is not too large. `fb1`, `fb2` and `fb3` will be drawn inside separate toplevel windows, as will `fb0` if that has been configured too large for embedding in the main window. This behaviour can be changed by providing a custom Tcl/Tk procedure that creates the containing frame for the framebuffer device.

Customization involves adding a synth_device `framebuf` section to the `.tdf` target definition file, usually `default.tdf` or `~/.ecos/synth/default.tdf`.

```
proc my_framebuf_create_frame { … } {
    …
}

synth_device framebuf {
    fb2_magnification   2
    create_frame_proc   my_framebuf_create_frame
}
```

The pixel size on the host display may be rather smaller than on the final hardware, causing a serious mismatch between the application's appearance when using synthetic target emulation and when using real hardware. To reduce this problem the host-side can magnify the target-side framebuffer devices. In the example above each target-side pixel in device `fb2` will be drawn using 2*2 pixels on the host side. Valid magnifications are 1, 2, 3 and 4. With a magnification of 4 an eCos framebuffer device of 320*240 pixels will be drawn in an X window of 1280*960 pixels.

The *`create_frame_proc`* entry can be used to specify a custom Tcl/Tk procedure that will create the containing Tk frames for the host-side displays. This procedure can be written for a specific configuration, but it is supplied with all the parameters associated with the framebuffer device so can be more generic. An example is supplied in the package's `misc` subdirectory:

1. Create a configuration for the synthetic target with the default template.
2. Import the `example.ecm` configuration fragment from the `misc` subdirectory. This will add the generic framebuffer support package, enable all four framebuffer devices, and configure each device. Build the resulting configuration.
3. Compile the `example.c` program and link it against the eCos configuration.
4. Incorporate the `example.tdf` fragment into the appropriate target definition file, typically `default.tdf` or `~/.ecos/synth/default.tdf`.

5. Run the example executable. The four framebuffer devices should get instantiated in a separate window in a single column. `FB0` just contains a static display. `FB1` supports two pages, one with vertical stripes and one with horizontal stripes, and the two pages are flipped at regular intervals. `FB2` has a static display similar to `FB0`, but is drawn in a viewport of only 160x120 pixels. However `example.tdf` magnifies this by 2 so it appears the same size as the other devices. The application moves the viewport around the underlying framebuffer device. `FB3` is also a static display, a simple set of vertical stripes. However this framebuffer is paletted and the palette is changed at regular intervals, causing apparent movement.

Part XXVII. CAN Support

Documentation for drivers of this type is often integrated into the eCos board support documentation. You should review the documentation for your target board for details. Standalone and more generic drivers are documented in the following sections.

Important

The eCosPro-CAN package and its associated device drivers are STRICTLY LICENSED FOR INTERNAL EVALUATION AND TESTING PURPOSES ONLY for a maximum period of THREE months from the initial delivery of your eCosPro release. It may not be used for production purposes nor redistributed in full or in part in any format, including source code, binary code and object code format. Shipment of prototypes, hardware or products containing the package in any format is STRICTLY PROHIBITED.

A separate COMMERCIAL LICENSE for this package from eCosCentric is required to receive technical support for the package as well as permit distribution of binary forms of this package.

Some releases of eCosPro may not include evaluation copies of this package. In this case, please contact eCosCentric for licensing and availability. You must obtain written permission from eCosCentric to exceed the evaluation period.

Table of Contents

Chapter 90. CAN Support

⚠

Important

This eCosPro-CAN Middleware package is STRICTLY LICENSED FOR NON-COMMERCIAL PURPOSES ONLY. It may not be used for Commercial purposes in full or in part in any format, including source code, binary code and object code format.

A Commercial eCosPro License version 3 (or above) which explicity includes this Middleware Package is required for Commercial use.

Name

Overview — eCos Support for CAN, the Controller Area Network

Description

The Controller Area Network, CAN, was originally designed for use in automotive systems where many small sensors report small values frequently. Consequently, CAN uses small messages of up to 8 bytes payload, which permits many messages to be passed each second and also reduces packet transmission latency.

CAN is a simple broadcast network carried on a twisted pair. It uses CSMA/CD, like Ethernet, to access the medium, however the signal levels and messages format are designed to resolve collisions in favour of the highest priority packet, rather than the jam and random retry of Ethernet. This yields a priority-based approach to contention that is better suited to real time applications.

CAN messages contain an 11 bit message ID, up to 8 bytes of data, a 15 data, a 15 bit CRC and assorted control bits. The message ID is also the priority used to resolve collisions, and has to be unique. Since the network is broadcast, and there are no fixed node addresses, receivers need to have hardware acceptance filters to avoid having to process every packet.

There are two very distinct operational modes for CAN devices. The first of these is BasicCAN. This is the approach familiar from Ethernet controllers. Packets to be transmitted are fed one at a time into an output FIFO and are sent on the wire when it appears to be idle. Received packets are pulled off the wire and are compared against one or more acceptance filters. If they match they are placed into an input FIFO ready for the CPU to collect them.

The second approach is FullCAN. Here, the controller contains a number of packet buffers which are used for transmission or reception. To transmit, the packet is put into a buffer and the buffer state changed to "transmit" and the packet goes out when the controller and the line are ready. Received packets are compared against an acceptance filter on each buffer that is marked "receive" and copied into whichever buffer matches first. The clear intention here is that specific buffers are statically dedicated to particular communication channels.

The eCos CAN subsystem currently supports the BasicCAN mode only. FullCAN devices are driven in such a way as to provide BasicCAN functionality. FullCAN functionality can always be emulated using software.

You can find more information at the CAN in Automation website.

eCos Support for CAN

The eCos CAN support for any given platform is spread over a number of different packages:

- This package, `CYGPKG_IO_CAN`, exports a generic API for accessing devices attached to a CAN network. This API handles issues such as locking between threads. The package does not contain any hardware-specific code. Instead it uses a separate CAN device driver to handle the hardware and defines the interface that such network drivers should provide.
- Each CAN device has its own device driver, which is implemented as a separate package. For devices that may be attached to a variety of different boards, the device driver will be generic and a second platform specific package will be used to customize it to each platform. For devices that are associated with a specific chipset, only a single package may be present.

Typically all appropriate packages will be loaded automatically when you configure eCos for a given platform. If the application does not use any of the CAN I/O facilities, directly or indirectly, then linker garbage collection should eliminate all unnecessary code and data. All necessary initialization should happen automatically. However the exact details may depend on the platform, so the platform HAL documentation should be checked for further details.

There is an important exception to this: if the CAN devices are attached to an expansion connector, such as PCI, then the platform HAL will not know about these devices. Instead the necessary packages will need to be added explicitly during configuration.

Support for CAN-FD

The eCos CAN subsystem also provides support for CAN-FD messages. CAN-FD messages mainly looks similar to standard CAN messages except that the length field can specify packet sizes up to 64 bytes, and there is an option to transmit the data part of a message at a higher baud rate than the rest.

CAN-FD support is only enabled if a driver capable of supporting the protocol is present. Otherwise CAN-FD is not present.

CAN-FD support mainly consist of some extra fields in the CAN message structure and some additional API functions. Both standard CAN and CAN-FD messages may be sent and received using the common structure.

Name

CAN Functions — allow applications and other packages to access CAN devices

Synopsis

#include <cyg/io/can.h>

```
int cyg_can_init();

int cyg_can_open(devname, dev);

int cyg_can_close(dev);

cyg_can_msg* cyg_can_msg_alloc();

void cyg_can_msg_free(msg);

int cyg_can_send(dev, msg);

int cyg_can_send_nowait(dev, msg);

int cyg_can_recv(dev, msg);

int cyg_can_recv_poll(dev, msg);

int cyg_can_recv_timeout(dev, msg, timeout);

void cyg_can_poll();

int cyg_can_filter_set(dev, ide, match, mask);

int cyg_can_filter_get(dev, ide, match, mask);

int cyg_can_filter_ext_set(dev, *filters, len);

int cyg_can_filter_ext_get(dev, *filters, *len);

int cyg_can_baud_set(dev, baud);

int cyg_can_baud_get(dev, baud);

int cyg_can_baud_fd_set(dev, baud);

int cyg_can_baud_fd_get(dev, baud);

int cyg_can_autobaud(dev);

const char cyg_can_error_string(code);
```

Initialization and Device Access

Before performing any CAN system operations, the application must call `cyg_can_init()`. This function initializes the CAN subsystem and causes all the configured devices to initialize themselves. Only the first call to this function will initialize the subsystem, subsequent calls will do nothing, so libraries and independent systems may call it in their initialization routines without needing to ensure it is called only once.

To gain access to a specific CAN channel, the application must call `cyg_can_open()`. Channel names are defined in the configuration and are typically "can0" "can1" and so on. If the channel is not found this function will return `CYG_CAN_NOTFOUND`; it may also return errors generated by the device driver. If the channel is found the call will return `CYG_CAN_NOERROR` and the location pointed to by the *`dev`* parameter will be initialized with a handle on the channel. This handle must be used in all subsequent calls to access this channel.

When the application has finished with a channel it must call `cyg_can_close()` on the handle.

Buffer Management

The CAN subsystem uses buffers to pass messages between the application and the CAN subsystem. These buffers are allocated and managed by the CAN subsystem. The exact number of buffers is controllable in the configuration.

Each buffer contains the following fields:

cyg_can_msg * `next`

This field is used within the CAN subsystem to link this message buffer into lists. When the buffer is in the possession of the user (`state` is `CYG_CAN_MSG_STATE_USER`) then this may be used for application purposes.

unsigned int `rtr`

Remote Transmission Request. If this field is set in a transmitted message buffer, then the RTR bit on the message will be set and the `data` field will be ignored. On reception this field reflects the state of the RTR bit in the received message.

unsigned int `ide`

Extended ID. If this field is set then the `id` field contains a 29 bit extended ID. If it is clear then the ID is 11 bits.

unsigned int `state`

This field is used within the CAN subsystem to track the current state of the buffer. The following states are supported:

CYG_CAN_MSG_STATE_FREE

The message buffer is not currently being used and is on the CAN subsystem's free list.

CYG_CAN_MSG_STATE_USER

The message buffer is currently in the possession of the application code and is outside the control of the CAN subsystem.

CYG_CAN_MSG_STATE_TX

The message buffer is either currently being transmitted, or is in a queue of buffers awaiting transmission.

CYG_CAN_MSG_STATE_RX

The message buffer is the current pending receive buffer for a CAN channel. The next message from that channel will be received into this buffer.

CYG_CAN_MSG_STATE_RXQ

The message buffer is currently on a channel's receive queue. A message has been received into it but not yet been passed on to the user.

unsigned int `len`

The length of the data carried in the message. This can range from zero to 8. In a message with the RTR field set, this indicates the size of data being requested.

int `result`

An error code. For normal successful receptions of messages this will be `CYG_CAN_NOERROR`. Message buffers are also used to report special events on the channel such as transitions to passive error and bus off states. In these cases, the event will be reported using a message buffer with this field set to the appropriate error code.

cyg_uint32 `timestamp`
: Some CAN channels contain a timer that can be used to timestamp received packets. If that is the case, then the timestamp will be stored in this field. If the channel does not have any hardware timing facility, this field will not be used. This field is used for internal purposes during message transmission.

cyg_uint32 `id`
: Message ID. This is the ID to be transmitted with the message, or the ID received. If the `ide` field is set, then this will contain a 29 bit ID, otherwise it will contain an 11 bit ID.

cyg_uint8 `data[8]`
: Message data. Only the first `len` bytes of data are valid. If the `rtr` field is set, then the contents of this field are ignored.

A message buffer may be allocated by calling `cyg_can_msg_alloc()` and freed by calling `cyg_can_msg_free()`.

CAN-FD Buffers

When CAN-FD is enabled the message buffer contains some extra and modified fields:

unsigned int `fdf`
: FD Format. This marks a message as being in FD format. If this field is set in a transmitted message then it will be sent in CAN-FD format. A message with a `len` value of more than 8 will also be considered to be in FD format regardless of the state of this field. On reception this field reflects the state of the FDF bit in the received message.

unsigned int `brs`
: Bit Rate Switch. This indicates whether the message is transmitted with a higher bit rate for the data portion. This field is only valid for FD format messages. If this field is set in a transmitted message then its data bytes will be transmitted at the FD bit rate. On reception this field reflects the state of the BRS bit in the received message.

unsigned int `esi`
: Error State Indicator. This field indicates the error state of the transmitting node, 0 for error active and 1 for error passive. This field is only valid for FD format messages. If this field is set in a transmitted message then the ESI bit on the message will be set. On reception this field reflects the state of the ESI bit in the received message.

unsigned int `len`
: This field is expanded for FD messages to permit any value between 0 and 64. The CAN-FD protocol only permits a limited selection of message sizes: 0 to 8, 12, 16, 20, 24, 32, 48 and 64, encoded into a 4 bit field. This encoding is not used in this field, instead the actual number of bytes are used, with the translation being done in the device driver.

: On transmission if this value is greater than 8 then the message is sent in FD format, regardless of the value of `fdf`. If the supplied value is not one of the protocol-permitted sizes it will be increased to the next greater size.

: On reception this field will be translated from the protocol encoding into the correct number of bytes.

cyg_uint8 `data[64]`
: When CAN-FD is enabled, this field is increased to 64 bytes. Only the first `len` bytes will be valid. If, on transmission, the length has been increased to the next supported size, some extra bytes at the end of the supplied data in the buffer may be sent. On reception, bytes beyond the end of the specified length may have been written by the driver.

Transmit and Receive

To transmit a message an application must acquire a message buffer from the CAN subsystem, fill it in with the message to be sent and call `cyg_can_send()`. Following a successful call the buffer becomes the property of the CAN subsystem and will be returned to the free pool when the message has been transmitted. If an error is detected then the call will return an error code and the message buffer will be returned to the user for reuse or retransmission.

To send a message without waiting for it to complete, the application can call `cyg_can_send_nowait()`.

To receive a message the application calls `cyg_can_recv()`. If there is a message waiting, then a pointer to the message buffer will be installed in the location pointed to by the *`msg`* argument and `CYG_CAN_NOERROR` is returned.

If the application does no want to wait for a message to arrive, it can call `cyg_can_recv_poll()` which will just test for a message and return. If a message is present then `CYG_CAN_NOERROR` is returned and the *`msg`* filled in with a pointer to a message buffer. If no message is present then the function will return `CYG_CAN_AGAIN`.

The application can also wait for a defined length of time for a message to arrive by calling `cyg_can_recv_timeout()`. The additional *`timeout`* argument supplies an *absolute* timeout in system ticks. If a message is present then `CYG_CAN_NOERROR` is returned and the *`msg`* filled in with a pointer to a message buffer. If no message arrives before the timeout expires then the function will return `CYG_CAN_TIMEOUT`.

Regardless of which receive function is used, a successful return results in a message buffer being passed back to the caller. The `result` field of this message buffer will either contain `CYG_CAN_NOERROR` for a normal message, or it will contain an error code indicating an event that has occurred on the channel. When the application has finished with the buffer it must return it to the CAN subsystem by calling `cyg_can_msg_free()`.

The function `cyg_can_poll()` may be called to force all channels to check for transmission completion or pending receptions. When using interrupt driven devices it is unnecessary to call this. However, if there are any polled devices, this is the only way to ensure timely processing of received messages. This function should therefore be called from the application main loop, or from a separate timer driven thread, or by any other appropriate means to ensure communication proceeds in a timely fashion.

Basic Filtering

The functions `cyg_can_filter_set()` and `cyg_can_filter_get()` allow the basic hardware filter to be set and queried. The basic filter model consists of a match value, such that if an ID when bitwise ANDed with the mask equals mask equals the match value ANDed with the mask, then the message is accepted. If the hardware does not support a filter that conforms to this model then no hardware filtering is done, but the filter will still be applied by the CAN subsystem to all incoming packets.

In `cyg_can_filter_set()`, the *`match`* and *`mask`* arguments define the filter. The *`ide`* indicates whether the filter is for normal or extended IDs. It is hardware dependent what happens when the filter ID size does not match the ID size being used on the network.

Extended Filtering

The functions `cyg_can_filter_ext_set()` and `cyg_can_filter_ext_get()` implement an extended filtering mechanism. In this case the application can submit an array of filters which will accept a message if any one of them matches the received ID. If it is possible, extended filtering will be implemented in the CAN controller hardware. Otherwise it will be implemented in software. Setting the extended filters will invalidate the basic filter and vice versa.

The filters are an array of cyg_can_filter structures. Each filter consists of a mask and a match field. For each filter, if the received ID bitwise ANDed with the mask equals the match field ANDed with the mask, then the message is accepted. In addition to the 11 or 29 bits of the ID, the mask and match fields can contain two extra bits: CYG_CAN_FILTER_IDE matches the message IDE bit for 29 bit addressing, and CYG_CAN_FILTER_RTR matches the message RTR bit.

Some care should be taken in setting the IDE and RTR bits in the filters. In general, if the intention is to match on either bit, then it should be set in both fields. Setting the bit only in the mask field will match packets that have the bit clear, which is unlikely to be what it wanted. Incoming message IDs are only matched against similarly sized filters: a message with a 29 bit ID is only matched against filters that have the IDE bits set, and 11 bit ID are only matched against filters with IDE clear. This approach is to ensure consistency between software and hardware filters, and between different hardware filters.

In `cyg_can_filter_ext_set()` the *`filters`* argument is the address of the filter array, and *`len`* defines the number of elements. An error will be returned if the filters are invalid or the list is too long. In `cyg_can_filter_ext_get()`, the *`*len`* argument is a pointer to the length; it should be set to the size of the *filters* array before the call and will be updated with the number of actual filters returned. If the *filters* argument is NULL, the number of filters set will be returned in **len*. If the number of filters set is larger than the value of **len*, or if no extended filters are set, then an error will be returned.

The configuration option CYGNUM_IO_CAN_FILTER_MAX describes the maximum number of extended filters than can be stored. Usually controller drivers will set this value according to the amount of hardware resource available in the filter system.

Baud Rate

The functions cyg_can_baud_set() and cyg_can_baud_get() allow the channel baud rate to be set and queried. Baud rates from 10kb/s to 1Mb/s may be set, although not all device drivers will necessarily support all rates, many will only support a subset. Also, due to interactions between the input clock to the device and the divider granularity, it may not be possible to set some baud rates accurately at some system clock rates; it may be necessary to alter the system clock speed to enable communication.

The functions cyg_can_baud_fd_set() and cyg_can_baud_fd_get() allow the channel FD baud rate to be set and queried. When an FD format message is sent or received, the data bytes will be transmitted at this rate rather than the standard rate.

Autobaud Support

The function cyg_can_autobaud() supports automatic baud rate detection. This will only be present if the controller driver supports autobaud or listen-only mode. It indicates this by implementing the CYGINT_IO_CAN_AUTOBAUD interface. If the driver does no support this feature then this function will not be defined.

The approach for baud detection is to switch the controller to listen-only mode, where it cannot affect the bus state. Each of a set of candidate baud rates are set and the function waits for a period of time for a valid packet to arrive. If no packet is seen, then attention moves to the next baud rate. If a packet is received, then that baud rate is selected and set in the controller in non-listen-mode. If no packets are seen at any baud rate, the original rate is restored to the controller.

There are two configuration options that control the behaviour of the autobaud mechanism:

CYGPKG_IO_CAN_AUTOBAUD_RATES

> This is the set of baud rates tested when autobaud is enabled. It is a comma separated list of rates used to initialize an array in the CAN subsystem. The default value contains all the standard CANOpen rates, but for specific applications only the subset of rates that might be used should be listed.

CYGNUM_IO_CAN_AUTOBAUD_TIMEOUT

> Maximum time in ticks that the autobaud code will wait for bus activity at each baud rate. If no packet is received or an error reported in this time, it will move on to the next baud rate. The default 50 ticks equals half a second at the default 100Hz clock frequency. If this option is set to zero, then the autobaud code will wait indefinitely at each baud rate for either a packet or an error. This is useful during testing when traffic is initiated by hand.

Autobaud support comes with a number of caveats. Baud detection depends on traffic being present on the CAN bus and being frequent enough for packets to be seen during the timeout period for each candidate baud rate. Some baud rates may not be supportable by the controller. If there is only one other node in the network sending packets, the lack of acknowledgements may cause it to move into Error Passive or Bus Off mode and stop transmitting. Consequently autobaud detection cannot be considered to be a fully reliable operation and it is quite possible for cyg_can_autobaud() to terminate without detecting the baud rate. The length of time taken to detect the baud rate may be as long as the number of candidate rates multiplied by the timeout period.

Errors

Many of the CAN API calls return error codes. The result field of the message buffer structure may also contain an error code from this set. The following error codes may be returned by API calls:

CYG_CAN_NOERROR

No error, the operation completed successfully.

CYG_CAN_NOTFOUND

When returned from `cyg_can_open()` this error code means that the named CAN channel could not be found.

CYG_CAN_INVALID

When returned from `cyg_can_filter_set()` this error code means that the filter is invalid. When returned from `cyg_can_baud_get()` this error code means that the baud rate is invalid, or the hardware cannot support it with sufficient accuracy in the current system configuration.

CYG_CAN_TIMEOUT

When returned from `cyg_can_recv_timeout()` this error code means that the timeout has expired with no message being received.

The following error codes may be passed back in the result field of a message buffer acquired from one of the receive functions. Refer to the CAN specification for details of what these events actually mean.

CYG_CAN_NOERROR

The message buffer contains a CAN message that was received from the channel.

CYG_CAN_WARN_TX

This error code indicates that the CAN channel's transmit error counter has exceeded its warning limit, which is usually 96.

CYG_CAN_WARN_RX

This error code indicates that the CAN channel's receive error counter has exceeded its warning limit, which is usually 96.

CYG_CAN_PASSIVE

This error code indicates that the CAN channel has gone into "error passive" mode.

CYG_CAN_BUSOFF

This error code indicates that the CAN channel had gone into "bus off" mode.

CYG_CAN_OVERRUN

This error code indicates that the CAN channel has lost one or more CAN messages due to all the hardware buffers being full.

CYG_CAN_RXERROR

This error code indicates that the controller has detected one or more low level error conditions. Support for detecting these errors may only be enabled when driver autobaud support is enabled. It may not be generated during normal operation.

The `cyg_can_error_string()` function translates these error codes into strings for diagnostic purposes.

Name

Configuration — CAN subsystem configuration

Description

The CAN subsystem has a number of configuration options:

cdl_interface CYGINT_IO_CAN_DRIVER

> This CDL interface counts the number of CAN channels present in the system. Each driver must have an **implements** command for this interface for each channel it supports.

cdl_option CYGNUM_IO_CAN_MSG_COUNT_BASE

> This CDL option defines the base number of message buffers allocated regardless of the number of channels available.
>
> Default value: 10

cdl_option CYGNUM_IO_CAN_MSG_COUNT_DRIVER

> This CDL option defines the number of message buffers allocated per CAN channel. The total number of message buffers allocated is `CYGNUM_IO_CAN_MSG_COUNT_BASE` plus `CYGNUM_IO_CAN_MSG_COUNT_DRIVER` times `CYGINT_IO_CAN_DRIVER`.
>
> Default value: 10

cdl_option CYGNUM_IO_CAN_FILTER_MAX

> This option defines the number of extended filters that can be stored. Drivers may use a requires statement to increase this number to reflect the availability of hardware filters. If hardware filtering is not supported, then this defines the number of software filters that can be stored.
>
> Default value: 10

cdl_option CYGDBG_IO_CAN_DEBUG

> This CDL option enables diagnostic output to be generated by the CAN subsystem. This is mostly useful in debugging problems with the CAN subsystem or in the development of device drivers.
>
> Default value: 0

Name

Device Drivers — Writing new CAN device drivers

Description

Adding CAN support for a new device to eCos involves a number of steps. First a new package for the driver must be created and added to ecos.db. This package must contain CDL, headers and sources for implementing the driver. If the driver can apply to devices on different platforms, then a further package that configures the generic device to each platform will also be needed.

Device Driver Data Structures

Each CAN device is represented by a cyg_can_device structure. Most of the fields of this structure are private to the CAN subsystem and it needs to be defined in such a way that it is included in a table of all the CAN devices. To make this simple for the driver writer a macro has been defined to create this structure for each channel.

```
CYG_CAN_DEVICE( tag, chan, name, priv );
```

The arguments to this macro are:

tag
: This is a general name for the device driver as a whole, it should usually be the name of the device chip or interface type. Typical values might be `sja1000` or `flexcan`. This tag is used to ensure that the data structures declared by this macro are unique to this driver.

chan
: This distinguishes between separate channels supported by the driver. Typical values might be `can0` or `can1`.

name
: This is the name of the channel as used in the `cyg_can_open()` function. It is a string constant and is usually defined by the driver's CDL. Typical CAN channel names are "can0" and "can1". However this argument will usually be a CDL option such as `CYGPKG_DEVS_CAN_CHANNEL0_NAME` or `CYGPKG_DEVS_CAN_CHANNEL1_NAME`.

priv
: This is a pointer to a private data structure that contains device specific information such as its base address and interrupt vector. The CAN subsystem does not interpret the contents of this in any way. Typical values for this might be `cyg_can_flexcan_drv_0` or `cyg_can_sja1000_drv[1]`.

The interface to each driver is via a table of function calls that is pointed to by the cyg_can_device structure. This structure has the following definition:

```
struct cyg_can_device_calls {
       int (*init)  (cyg_can_device *dev);
       int (*open)  (cyg_can_device *dev);
       int (*close) (cyg_can_device *dev);
       int (*send)  (cyg_can_device *dev, cyg_can_msg *msg);
       int (*poll)  (cyg_can_device *dev);
       int (*filter)(cyg_can_device *dev, cyg_bool ide, cyg_uint32 match, cyg_uint32 mask);
       int (*baud)  (cyg_can_device *dev, cyg_uint32 baud);
       int (*filter_ext)(cyg_can_device *dev, cyg_can_filter *filters, int len);
};
```

`init()`
: This is called to initialize the channel when `cyg_can_init()` is called. It should locate the channel and initialize it ready for communication. It should also install any interrupts and initialize the fields of the private data structure.

`open()`
: This is called if the name of this channel matches the device name passed to `cyg_can_open()`. There is no requirement for the driver to do anything here, but possible things it might do is to allocate per-client resources in the hardware, or just keep count of the number of users.

`close()`
: This is called when `cyg_can_close()` is called. As with the `open()` function, there is no required behaviour here, but if `open()` allocated resources, then this is where they should be released.

`send()`
: This function is called to transmit a message by `cyg_can_send()` and is passed a pointer to a message buffer. This function should return one of several return codes depending on the state of the channel:

  CYG_CAN_BUSY

  > The channel is busy and cannot transmit the message at this point. The CAN subsystem will add the message to its pending queue until the channel is available, at which point the buffer will be passed back to the driver by the `cyg_can_tx_done()` function.

  CYG_CAN_WAIT

  > The channel has started the transmission of the message, but it is not yet complete. The CAN subsystem will cause the sending thread to wait until the driver calls `cyg_can_tx_done()`. This is usually used by interrupt driven drivers to make the sender wait for the transmit completion interrupt.

  CYG_CAN_NOERROR

  > The channel has transmitted the message immediately, and has also called `cyg_can_tx_done()` to complete the process. This is mainly used by polled drivers, which don't use interrupts.

  CYG_CAN_*

  > Any other error code indicates a hardware error of some sort and will be passed back to the caller by the CAN subsystem.

`poll()`
: This is usually called from the driver's DSR routine to handle any events that have occurred on the channel. It may also be called from the CAN subsystem at various times, and may be called from application code calling `cyg_can_poll()`.

`filter()`
: This function is called to set the hardware ID filter. Not all hardware supports a filter mechanism that matches the model implemented by the CAN subsystem. This function should return `CYG_CAN_NOERROR` whether it can set the hardware filter or not. The CAN subsystem will always apply the filter in software to all received messages, so setting the hardware filter is an optimization to reduce the number of messages received. There is no function to return the filter since the CAN subsystem records the filter itself.

`baud()`
: This function is called to set the network baud rate. The driver is at liberty to support only a subset of the possible baud rates, and return `CYG_CAN_INVALID` for those it cannot. It is also possible for certain baud rates not be supported for certain system clock speeds due to limitations in the clock divider. It may not be possible for the driver to detect this.

`filter_ext()`
: This function is called to set the extended hardware ID filters. Not all hardware supports the extended filter mechanism and this function pointer may be NULL, in which case the CAN subsystem will apply the filters in software. If not NULL then the driver should return `CYG_CAN_NOERROR` if all the extended filters can be handled, and `CYG_CAN_INVALID` if not. Partial implementation of the filters is not supported; either all the filters are installed into the hardware or none are.

The CAN subsystem defines a macro to create this function table:

```
CYG_CAN_DEVICE_CALLS( tag );
```

That macro creates a function table with `filter_ext()` set to NULL. If extended filters are supported, the driver should use the following macro:

```
CYG_CAN_DEVICE_CALLS_EXT( tag );
```

The *`tag`* argument should match the *`tag`* argument of the `CYG_CAN_DEVICE()` macro. This macro does two things. First it declares static function prototypes for all the driver functions of the form `cyg_can_<tag>_<function>` (e.g. `cyg_can_sja1000_init()` or `cyg_can_flexcan_send()`). Second, it defines a function table called cyg_can_<tag>_calls. The `CYG_CAN_DEVICE()` macro assumes that a function table of this name is defined.

Device Driver API Calls

In addition to the standard device driver API calls defined by the kernel, there are a number of additional CAN specific API calls that a device driver must use to interact with the CAN subsystem. These functions may only be called from thread level with the DSR lock claimed, or from a DSR. They cannot be called from an ISR.

```
cyg_can_msg *cyg_can_tx_done( cyg_can_device *can_dev, cyg_can_msg *msg );
```

This must be called when a transmission is completed. For polled drivers it should be called from the `send()` function while for interrupt driven drivers it should be called from the *`poll()`* routine invoked from the driver's DSR. The function is called with a pointer to the transmitted message buffer and following this call the message buffer will become the property of the CAN subsystem and may be returned to the free pool. The return value from this function will a pointer be another message buffer to transmit, or NULL. The driver should use the resources recently freed by the completion of the previous transmission to start transmission of this message.

```
cyg_can_msg *cyg_can_rx_buffer( void );
```

This is called by the driver to acquire a message buffer in to which a message will be received. The normal approach is to allocate a buffer during driver initialization and to keep at least one pending buffer available at all times. New buffers will usually be passed back to the running driver by the `cyg_can_rx_done()` function.

```
cyg_can_msg *cyg_can_rx_done( cyg_can_device *can_dev, cyg_can_msg *msg );
```

This must be called when a driver has a completed message buffer to return to the user. This buffer may either contain a received message, or may be reporting a channel event. This call is made with a pointer to the message buffer to be returned. Following this call the message buffer becomes the property of the CAN subsystem. The return value of this function will be a pointer to a message buffer to replace the one just passed back. Thus with one call the driver both gives up an old buffer and gets a new one to use in its place.

It is possible for `cyg_can_rx_done()` to return a NULL pointer if there are currently no more buffers available. The driver must therefore be able to handle this. The usual approach is to check, just before it is needed, for a current pending buffer. If no buffer is present then call `cyg_can_rx_buffer()` and if this returns NULL then take action to, for example, throw the message away.

```
typedef struct
{
    cyg_uint8           tseg1_min;      // Time segment 1 = prop+phase1
    cyg_uint8           tseg1_max;
    cyg_uint8           tseg2_min;      // Time segment 2 = phase2
    cyg_uint8           tseg2_max;
    cyg_uint16          divider_min;    // Clock quantum divider
    cyg_uint16          divider_max;
} cyg_can_bitrate_param;

typedef struct
{
    // Input parameters
    cyg_uint32          clock;          // System input clock in Hz
    // Input/Output parameters
    cyg_uint32          bitrate;        // Target bit rate in Hz
                                        // Must be set on input
                                        // Updated with actual rate set

    cyg_uint16          sample;         // Sample point in tenths of a percent
                                        // If zero, CIA recommended value used
                                        // Updated with actual sample point

    // Output calculated values
    cyg_uint8           prop;           // Propogation segment in quanta
    cyg_uint8           phase1;         // Phase segment 1 in quanta
    cyg_uint8           phase2;         // Phase segment 2 in quanta
    cyg_uint16          divider;        // Clock divider

} cyg_can_bitrate;

int cyg_can_calculate_bitrate(  const cyg_can_bitrate_param *param,
                                cyg_can_bitrate             *bitrate );
```

The function `cyg_can_calculate_bitrate` may be called from the driver to calculate the timing values for a given bit rate.

The *`param`* argument contains details of the bit timing hardware in the device, mainly derived from the field widths in the timing register(s). These details comprise minimum and maximum values for *`tseg1`* (propogation segment plus phase segment one), *`tseg2`* (phase segment two) and the quantum clock *`divider`*.

The *`bitrate`* argument must have the input *`clock`* and target *`bitrate`* set. The *`sample`* point must either be set, or zeroed for a CIA recommended value to be chosen. On return the fields *`prop`*, *`phase1`* and *`phase2`* will be set to the quantum counts calculated for each segment. The *`divider`* field will set to the calculated divider. The *`bitrate`* and *`sample`* fields will be updated with the actual bit rate and sample point.

This function will either return `CYG_CAN_NOERROR` if a valid set of timing values have been calculated, or `CYG_CAN_INVALID` if no values could be found. The routine will attempt to find values that give the closest match to the bitrate and sample point requested. If an exact match is found for both, that setting is returned. If no exact match is found, success is only reported if the calculated bitrate is within 5% of the requested rate.

Most drivers should be able to use the values returned by this routine directly. Most CAN devices have one or two registers that contain fields that are directly analogous to these values. All that is needed is for them to be shifted in to position. Quanta are divided more or less equally between *`prop`* and *`phase1`*. However, some hardware may combine these values into a single field or have different sized fields for each. In these cases then the driver may need to add them together, or move some quanta from one to another to match the hardware.

Configuration

The only direct configuration requirement on device drivers is that for each channel supported, the driver should have an "implements CYGINT_IO_CAN_DRIVER" statement to ensure that the correct number of message buffers are available. The name of the channels should also be defined in the CDL. A minimal CDL file for the XYZZY driver would be as follows:

```
cdl_package CYGPKG_DEVS_CAN_XYZZY {
    display       "XYZZY CAN driver"
    description   "XYZZY CAN driver."

    parent        CYGPKG_IO_CAN
    active_if     CYGPKG_IO_CAN

    include_dir   cyg/devs/can

    compile -library=libextras.a xyzzy.c

    cdl_component CYGPKG_DEVS_CAN_CHANNEL0 {
        display   "CAN channel 0 configuration"
        default_value 1
        implements    CYGINT_IO_CAN_DRIVER

       cdl_option CYGPKG_DEVS_CAN_CHANNEL0_NAME {
            display       "CAN channel 0 name"
            flavor        data
            default_value { "\"can0\"" }
            description   "Name of CAN channel 0"
        }
    }

    cdl_component CYGPKG_DEVS_CAN_CHANNEL1 {
        display   "CAN channel 1 configuration"
        default_value 1
        implements    CYGINT_IO_CAN_DRIVER

        cdl_option CYGPKG_DEVS_CAN_CHANNEL1_NAME {
            display       "CAN channel 1 name"
            flavor        data
            default_value { "\"can1\"" }
            description   "Name of CAN channel 1"
        }

    }

    # Further entries for extra channels would go here

}
```

If the driver is multi-platform, then the channel configurations should go into the second platform specific package which may also need to define suitable configuration options to customize the generic driver.

Driver Template

The following example show the general structure of a CAN device driver for a fictional XYZZY device.

The first thing we need to do is to define the data structures that interface the device to the CAN subsystem:

```
#include <pkgconf/hal.h>
#include <pkgconf/io_can.h>
#include <pkgconf/devs_can_xyzzy.h>

#include <cyg/io/can_dev.h>

//=============================================================================
// Define private data structure. At the very least this needs to
// contain the base address of the device and the interrupt vector.
// If this is an interrupt driven device, then it will also need to
// contain the data structures to manage the interrupt.

struct cyg_can_xyzzy_priv
{
    cyg_uint32                  devno;          // device number
    CYG_ADDRESS                 base;           // base address
    cyg_vector_t                vector;         // vector number

    cyg_can_msg                 *tx_msg;        // current tx message buffer
    cyg_can_msg                 *rx_msg;        // pending rx message buffer

    cyg_handle_t                interrupt_handle;
    cyg_interrupt               interrupt_object;

    // Further device fields here
};
```

```
//=============================================================================
// Define device function call table. This should be done before the
// CYG_CAN_DEVICE() macro is called.

CYG_CAN_DEVICE_CALLS( xyzzy );

//=============================================================================
// Define driver-private structures for each of the channels. For this
// example we define just two.

#ifdef CYGPKG_DEVS_CAN_CHANNEL0
struct cyg_can_xyzzy_priv cyg_can_xyzzy_drv_0 =
    { 0, CYGARC_HAL_XYZZY_BASE_CAN_0, CYGNUM_HAL_INTERRUPT_CAN_0 };
#endif
#ifdef CYGPKG_DEVS_CAN_CHANNEL1
struct cyg_can_xyzzy_priv cyg_can_xyzzy_drv_1 =
    { 1, CYGARC_HAL_XYZZY_BASE_CAN_1, CYGNUM_HAL_INTERRUPT_CAN_1 };
#endif


//=============================================================================
// Define CAN device table entries.

#ifdef CYGPKG_DEVS_CAN_CHANNEL0
CYG_CAN_DEVICE( xyzzy, can0, CYGPKG_DEVS_CAN_CHANNEL0_NAME, cyg_can_xyzzy_drv_0 );
#endif
#ifdef CYGPKG_DEVS_CAN_CHANNEL1
CYG_CAN_DEVICE( xyzzy, can1, CYGPKG_DEVS_CAN_CHANNEL1_NAME, cyg_can_xyzzy_drv_1 );
#endif
```

The first thing that needs writing is the initialization routine:

```
static int cyg_can_xyzzy_init(cyg_can_device *dev)
{
    int result = CYG_CAN_NOERROR;
    struct cyg_can_xyzzy_priv *priv = (struct cyg_can_xyzzy_priv *)dev->private;


    // Locate, validate and initialize the channel hardware. This
    // may include setting up the acceptance filter to accept all IDs
    // and setting the baud rate to a default (100kHz say).

    // Install interrupt handlers

    cyg_drv_interrupt_create( priv->vector,
                              0,
                              (CYG_ADDRWORD)dev,
                              cyg_can_xyzzy_isr,
                              cyg_can_xyzzy_dsr,
                              &priv->interrupt_handle,
                              &priv->interrupt_object );
    cyg_drv_interrupt_attach( priv->interrupt_handle );
    cyg_drv_interrupt_unmask( priv->vector );

    // Perform any final initialization, for example clearing and then
    // enabling interrupts in the channel.

    // Allocate a pending buffer for message receive.
    priv->rx_msg = cyg_can_rx_buffer();

    return result;
}
```

The open and close routines come next. Most drivers don't need to do much here so these examples are the minimum necessary:

```
static int cyg_can_xyzzy_open(cyg_can_device *dev)
{
    return CYG_CAN_NOERROR;
}

static int cyg_can_xyzzy_close(cyg_can_device *dev)
{
    return CYG_CAN_NOERROR;
}
```

The send routine is responsible for actually transmitting a message:

```
static int cyg_can_xyzzy_send(cyg_can_device *dev, cyg_can_msg *msg)
{
    struct cyg_can_xyzzy_priv *priv = (struct cyg_can_xyzzy_priv *)dev->private;

    // If there is still a current tx message or the transmit hardware
    // is still busy, return busy so that the upper levels will
    // queue this request.
    if( priv->tx_msg != NULL || xyzzy_tx_busy( priv ) )
        return CYG_CAN_BUSY;

    // Record current transmit packet
    priv->tx_msg = msg;

    // Write the message header and ID to be sent into the channel
    // transmit buffer, ensuring that the length and the IDE bit is
    // set correctly, and the ID is correct.

    // If the RTR flag is not set, install the data in the transmit
    // buffer. If the RTR flag is set, do not install the data and set
    // the RTR bit in the frame info.

    // Start the transmission.

    // Return CYG_CAN_WAIT to cause the sending thread to wait for completion.

    return CYG_CAN_WAIT;
}
```

The following routine is internal to the driver, it is called from the `poll()` routine to actually receive a message into a buffer:

```
static int cyg_can_xyzzy_recv(cyg_can_device *dev, cyg_can_msg *msg)
{
    int result = CYG_CAN_NOERROR;
    struct cyg_can_xyzzy_priv *priv = (struct cyg_can_xyzzy_priv *)dev->private;
    cyg_ucount8 ide, rtr, len;
    cyg_uint32 id = 0;

    // Get the message frame header and decode it into some locals:
    // ide, rtr, len and id.

    // If we have a message buffer, move the message out into it.

    if( msg != NULL )
    {
        msg->ide = ide;
        msg->rtr = rtr;
        msg->len = len;
        msg->id  = id;

        if( !rtr )
        {
            // Copy data from receive frame into message buffer.
        }
    }
    else
    {
            // Do whatever is needed to throw the message away since
            // there is no buffer available.
    }

    // Do whatever is needed to release the receive buffer and ready
    // it for a new message and/or cancel the interrupt.

    return result;
}
```

The `poll()` handles most of the asynchronous events:

```
static int cyg_can_xyzzy_poll(cyg_can_device *dev)
{
    int result = CYG_CAN_NOERROR;
    struct cyg_can_xyzzy_priv *priv = (struct cyg_can_xyzzy_priv *)dev->private;

    // If there is a pending transmission and the hardware channel
    // indicates that it is finished, then call cyg_can_tx_done().

    if( priv->tx_msg != NULL && xyzzy_tx_done( priv ) )
    {
        cyg_can_msg *msg = priv->tx_msg;

        priv->tx_msg = NULL;
        msg = cyg_can_tx_done( dev, msg );

        // If we have been passed a new message to transmit, send it.
        if( msg != NULL )
            cyg_can_xyzzy_send( dev, msg );
    }

    // While there are messages available in the receive buffer or
    // FIFO, pull them out and pass them back to the CAN subsystem.

    while( xyzzy_rx_done( priv ) )
    {
        // If there is no current rx buffer, try to allocate one here.
        if( priv->rx_msg == NULL )
            priv->rx_msg = cyg_can_rx_buffer();

        // Either receive the message, or clear the channel.
        cyg_can_xyzzy_recv( dev, priv->rx_msg );

        // If we have a buffer, pass it back.
        if( priv->rx_msg != NULL )
            priv->rx_msg = cyg_can_rx_done( dev, priv->rx_msg );
    }

    // See if any other CAN channel events have occurred.

    if( xyzzy_event( priv ) )
    {
        // Decode the event and set result to an appropriate error
        // code.

        // Return a message buffer recording this event. As above, we
        // may need to allocate a fresh buffer if none is available.

        if( result != CYG_CAN_NOERROR )
        {
            if( priv->rx_msg == NULL )
                priv->rx_msg = cyg_can_rx_buffer();

            if( priv->rx_msg != NULL )
            {
                priv->rx_msg->result = result;
                priv->rx_msg = cyg_can_rx_done( dev, priv->rx_msg );
            }
            result = CYG_CAN_NOERROR;
        }
    }

    return result;
}
```

The simplest form for the ISR is for it to just mask the channel's interrupt vector and cause the DSR to run. The DSR can then simply call `cyg_can_xyzzy_poll()` to handle the channel events. Alternatively, the ISR could handle the hardware, but the DSR still needs to be run to call `cyg_can_tx_done()`, `cyg_can_rx_done()` and `cyg_can_rx_buffer()`.

```
static cyg_uint32 cyg_can_xyzzy_isr(cyg_vector_t    vector,
                                    cyg_addrword_t  data)
{
    cyg_can_device *dev = (cyg_can_device *)data;

    // Block interrupts from this device until the DSR is run
    cyg_drv_interrupt_mask( vector );

    // Ack the interrupt in the system interrupt controller
    cyg_drv_interrupt_acknowledge( vector );

    // Pass handling on to DSR
    return (CYG_ISR_HANDLED|CYG_ISR_CALL_DSR);
}

static void cyg_can_xyzzy_dsr(cyg_vector_t    vector,
                              cyg_ucount32    count,
                              cyg_addrword_t  data)
{
    cyg_can_device *dev = (cyg_can_device *)data;

    // Poll hardware for pending events
    cyg_can_xyzzy_poll( dev );

    // Re-allow device interrupts
    cyg_drv_interrupt_unmask( vector );
}
```

Finally, the filter and baud rate functions are very simple:

```
static int cyg_can_xyzzy_filter(cyg_can_device  *dev,
                                cyg_bool        ide,
                                cyg_uint32      match,
                                cyg_uint32 mask)
{
    int result = CYG_CAN_NOERROR;
    struct cyg_can_xyzzy_priv *priv = (struct cyg_can_xyzzy_priv *)dev->private;

    // Set the hardware filter to match the parameters. If the
    // hardware filter cannot be used, return CYG_CAN_NOERROR anyway,
    // since filtering will also be done in the CAN subsystem.

    return result;
}

static int cyg_can_xyzzy_filter_ext(cyg_can_device  *dev,
                                    cyg_can_filter  *filter,
                                    int             len)
{
    int result = CYG_CAN_NOERROR;
    struct cyg_can_xyzzy_priv *priv = (struct cyg_can_xyzzy_priv *)dev->private;

    // Set the extended filters to match the parameters. If the
    // hardware filter cannot be used, return CYG_CAN_INVALID, only
    // return CYG_CAN_NOERROR if all filters can be installed.
    return result;
}

static int cyg_can_xyzzy_baud(cyg_can_device  *dev,
                              cyg_uint32      baud)
{
    int result = CYG_CAN_NOERROR;
    struct cyg_can_xyzzy_priv *priv = (struct cyg_can_xyzzy_priv *)dev->private;

    // Set the baud rate, which may involve checking that the
    // requested rate is supported. If not the return
    // CYG_CAN_INVALID.

    result = cyg_can_xyzzy_set_baudrate( priv, baud );

    return result;
}
```

Chapter 91. NXP FlexCAN CAN Driver

Name

FlexCAN — CAN Driver

Description

This driver supports the FlexCAN CAN devices available in some NXP i.MXRT microprocessors.

Filter

The device provides filtering support that is compatible with the filter model defined by the CAN subsystem, and therefore hardware filtering is employed.

Baud Rates

The following baud rates are supported: 1Mb/s, 500kb/s, 250kb/s, 125kb/s, 100kb/s, 50kb/s, and 20kb/s.

Configuration

This driver requires an external package to configure it for the particular microcontroller, normally the platform (`PLF`) or variant (`VAR`) HAL. Each CAN channel is then implemented by a controller instance as specified.

For each channel # supported the CDL script in this package will provide the following configuration options:

cdl_component CYGPKG_DEVS_CAN_CHANNEL#

> This defines whether the channel is included. It depends on the associated `FLEXCAN` controller being implemented for the platform.
>
> ### Note
>
> The CAN channels are numbered logically from 0, whereas the physical FLEXCAN controllers are numbered from 1. So if the platform supports the `FLEXCAN2` controller the correspoding CAN channel device name will be *`"can1"`* by default.

cdl_option CYGPKG_DEVS_CAN_CHANNEL#_NAME

> This defines the name of the channel. This is the name that an eCos application should use to access the device via the I/O API.

cdl_option CYGPKG_DEVS_CAN_CHANNEL#_INTERRUPT_PRI

> This option defines the interrupt priority for the `FLEXCAN` controller interrupts.

cdl_option CYGPKG_DEVS_CAN_CHANNEL#_MB_RX

> This option defines the number of MailBox (MB) slots available for use as RX buffers. The minimum acceptable setting allowed being 2 dedicated RX buffers. However, for normal application configurations it is expected that more RX buffers will be required than TX buffers, so the default reflects such “normal” ratio.
>
> The number of available TX mailboxes (CYGPKG_DEVS_CAN_CHANNEL#_MB_TX) is calculated as the number of remaining mailboxes after the RX allocation.

cdl_option CYGPKG_DEVS_CAN_CHANNEL#_BITRATE

> This option defines the start-of-day (default) bitrate for the CAN channel.

Chapter 92. FlexCAN CAN Driver

Name

FlexCAN — CAN Driver

Description

This driver supports the FlexCAN CAN devices available in some Freescale Coldfire and PowerPC microprocessors.

Filter

The device provides filtering support that is compatible with the filter model defined by the CAN subsystem, and therefore hardware filtering is employed.

Baud Rates

The following baud rates are supported: 1Mb/s, 800kb/s, 500kb/s, 250kb/s, 125kb/s, 100kb/s, 50kb/s, 20kb/s, 10kb/s.

Configuration

This driver requires an additional package to configure it to the particular microcontroller. For each channel *X* supported the CDL script in this package must provide the following configuration options:

cdl_component CYGPKG_DEVS_CAN_CHANNELX

> This defines whether the channel is included.

cdl_option CYGPKG_DEVS_CAN_CHANNELX_NAME

> This defines the name of the channel.

cdl_option CYGNUM_DEVS_CAN_CHANNELX_BASE

> This defines the base address of the CAN channel controller in the memory map. It will typically be an expression involving HAL-defined constants which resolves to the address of the the device.

cdl_option CYGNUM_DEVS_CAN_CHANNELX_VECTOR

> This defines the interrupt vector associated with buffer 0. It is assumed that the vectors for buffers 1..15 follow this, and that the error and bus off interrupt vectors follow these. This will typically be a HAL-defined constant.

Chapter 93. MSCAN CAN Driver

Name

Mscan — CAN Driver

Description

This driver supports the MSCAN CAN devices available in some Freescale Coldfire and PowerPC microprocessors.

Filter

The device provides filtering support that is compatible with the filter model defined by the CAN subsystem, and therefore hardware filtering is employed.

Baud Rates

The following baud rates are supported: 500kb/s, 250kb/s, 125kb/s, 100kb/s, 50kb/s, 20kb/s, 10kb/s.

Configuration

This driver requires an additional package to configure it to the particular microcontroller. For each channel *X* supported the CDL script in this package must provide the following configuration options:

cdl_component CYGPKG_DEVS_CAN_CHANNELX

> This defines whether the channel is included.

cdl_option CYGPKG_DEVS_CAN_CHANNELX_NAME

> This defines the name of the channel.

cdl_option CYGNUM_DEVS_CAN_CHANNELX_BASE

> This defines the base address of the CAN channel controller in the memory map. It will typically be an expression involving HAL-defined constants which resolves to the address of the the device.

cdl_option CYGNUM_DEVS_CAN_CHANNELX_VECTOR

> This defines the interrupt vector associated with buffer 0. It is assumed that the vectors for buffers 1..15 follow this, and that the error and bus off interrupt vectors follow these. This will typically be a HAL-defined constant.

Chapter 94. LPC2XXXX CAN Driver

Name

LPC2XXX — CAN Driver

Description

This driver supports the CAN devices available in some variants of the Philips LPC2XXX family of microprocessors. The device itself is similar to the the SJA1000, but is sufficiently different that the drivers cannot be shared.

Filter

The filter mechanism present in the LPC2XXX CAN devices is not compatible with the filter model adopted by the CAN subsystem. Consequently the hardware filter is not used and only software filtering is applied.

Baud Rates

The following baud rates are supported: 1Mb/s, 800kb/s, 500kb/s, 250kb/s, 125kb/s, 100kb/s, 50kb/s, 20kb/s, 10kb/s.

Configuration

For each channel *X* supported the CDL script provides the following configuration options:

cdl_component CYGPKG_DEVS_CAN_CHANNELX

> This defines whether the channel is included. For channels 2 and 3 the default value is conditional on the LPC2XXX variant.

cdl_option CYGPKG_DEVS_CAN_CHANNELX_NAME

> This defines the name of the channel.

Chapter 95. Atmel SAM CAN Driver

Name

Atmel SAM — CAN Driver

Description

This driver supports the CAN devices available in some variants of Atmel's SAM4, SAM7, SAM9 and SAMA5 microprocessor families.

Filter

The filter mechanism present in the SAM4/7/9 and SAMA5 CAN devices is not compatible with the filter model adopted by the CAN subsystem. Consequently the hardware filter is not used and only software filtering is applied.

Baud Rates

The following baud rates are supported: 1Mb/s, 800kb/s, 500kb/s, 250kb/s, 125kb/s, 100kb/s, 50kb/s, 20kb/s and 10kb/s. However, the accuracy of baud rates that can be supported at any particular MCLK frequency depend on the resolution of the baud rate divisor.

Configuration

For each channel *X* supported the CDL script provides the following configuration options:

cdl_interface CYGINT_DEVS_CAN_CHANNELX

> This determines whether the given channel is available on the board. It is usually implemented by the platform HAL.

cdl_component CYGPKG_DEVS_CAN_CHANNELX

> This defines whether the channel is active. It will default to active for all channels whose corresponding interface is implemented, but may be disabled by the user.

cdl_option CYGPKG_DEVS_CAN_CHANNELX_NAME

> This defines the name of the channel.

cdl_option CYGPKG_DEVS_CAN_CHANNELX_MBOX_COUNT

> This defines the number of mailboxes supported by the device. This is usually eight or sixteen. The HAL for the microprocessor variant will usually set this value during configuration.

cdl_option CYGPKG_DEVS_CAN_CHANNELX_EXT

> If set this option configures this channel to receive extended, 29 bit, identifiers as well as standard, 11 bit, identifiers. Otherwise it receives only 11 bit identifiers. The default is to receive both message types.

At present there is support for up to two channels.

Chapter 96. Atmel MCAN CAN Driver

Name

Atmel MCAN — CAN Driver

Description

This driver supports the CAN devices available in some variants of Atmel's SAMX70 microprocessor families.

In addition to standard CAN messages, this device is capable of handling CAN-FD messages.

Filter

At present the hardware filter is not used and only software filtering is applied.

Baud Rates

All standard baud rates are supported. However, the accuracy of baud rates that can be supported at any particular MCLK frequency depend on the resolution of the baud rate divisor.

At present autobaud is not supported.

Configuration

For each channel *X* supported the CDL script provides the following configuration options:

cdl_interface CYGINT_DEVS_CAN_CHANNELX

> This determines whether the given channel is available on the board. It is usually implemented by the platform HAL.

cdl_component CYGPKG_DEVS_CAN_CHANNELX

> This defines whether the channel is active. It will default to active for all channels whose corresponding interface is implemented, but may be disabled by the user.

cdl_option CYGPKG_DEVS_CAN_CHANNELX_NAME

> This defines the name of the channel.

cdl_option CYGPKG_DEVS_CAN_CHANNELX_BITRATE

> This defines the default bitrate at which the channel will start.

cdl_option CYGPKG_DEVS_CAN_CHANNELX_BITRATE_FD

> This defines the default fast data bitrate at which the channel will start.

cdl_option CYGPKG_DEVS_CAN_CHANNELX_RX_FIFO_SIZE

> This defines the number of receive buffers in RX FIFO 0. Only FIFO 0 is currently used.

cdl_option CYGPKG_DEVS_CAN_CHANNELX_TX_BUF_COUNT

> This defines the number of transmit buffers and the number of entries in the transmit event FIFO.

cdl_option CYGPKG_DEVS_CAN_CHANNELX_

cdl_option CYGPKG_DEVS_CAN_CHANNELX_EXT

> If set this option configures this channel to receive extended, 29 bit, identifiers as well as standard, 11 bit, identifiers. Otherwise it receives only 11 bit identifiers. The default is to receive both message types.

At present there is support for up to two channels.

Chapter 97. SJA1000 CAN Driver

Name

SJA1000 — CAN Driver

Description

This driver supports the Philips SJA1000 CAN device. As a stand-alone device, the SJA1000 may be connected to the target system by a variety of mechanisms, including direct connection, PCI bus, ISA bus or USB. This driver is structured to support any of these methods, although not all are currently implemented.

Access Methods

At present only Peak PCAN-PCI boards are supported by this driver. As a consequence only the PCI access method is currently implemented and within that, only PCAN-PCI boards are recognised and initialized. New access methods and boards will be added as they become available.

Filter

The device provides filtering support that is compatible with the filter model defined by the CAN subsystem, and therefore hardware filtering is employed.

Baud Rates

The following baud rates are supported: 1Mb/s, 500kb/s, 250kb/s, 125kb/s, 100kb/s, 50kb/s, 20kb/s, 10kb/s.

Configuration

This driver requires an additional package to configure it to the particular microcontroller. For each channel *X* supported the CDL script in this package must provide the following configuration options:

cdl_component CYGPKG_DEVS_CAN_CHANNELX

> This defines whether the channel is included. For channels 2 and 3 the default value is conditional on the SJA1000 variant.

cdl_option CYGPKG_DEVS_CAN_CHANNELX_NAME

> This defines the name of the channel.

In addition to the above options, each channel should contain the following **implements** or **requires** commands, as appropriate:

implements CYGINT_DEVS_CAN_SJA1000_REQUIRED

> This adds to the count of SJA1000 devices implemented by the driver.

implements CYGINT_DEVS_CAN_SJA1000_PCI

> This adds to the count of SJA1000 PCI-based devices implemented by the driver. If the SJA1000 is accessed through a PCI device then this interface should be implemented.

Chapter 98. BXCAN CAN Driver

Name

BXCAN — CAN Driver

Description

This driver supports the BXCAN devices available in some variants of the ST STM32 family of microprocessors.

Filter

Hardware filtering is used to provide the standard CAN subsystem filtering semantics.

Baud Rates

The following baud rates are supported: 1Mb/s, 500kb/s, 250kb/s, 125kb/s, 100kb/s, 50kb/s. However, the accuracy of baud rates that can be supported at any particular PCLK1 frequency depend on the resolution of the baud rate divisor.

Configuration

For each channel *X* supported the CDL script provides the following configuration options:

cdl_component CYGINT_DEVS_CAN_CHANNELX

> This interface should be implemented by the platform HAL for each channel that is connected to a CAN bus or socket on the board.

cdl_component CYGPKG_DEVS_CAN_CHANNELX

> This defines whether the channel is initialized. By default this depends on `CYGINT_DEVS_CAN_CHANNELX`.

cdl_option CYGPKG_DEVS_CAN_CHANNELX_NAME

> This defines the name of the channel.

Chapter 99. STR7XX CAN Driver

Name

STR7XX — CAN Driver

Description

This driver supports the CAN devices available in some variants of the ST STR7XX family of microprocessors.

Filter

The filter mechanism present in the STR7XX CAN devices is not compatible with the filter model adopted by the CAN subsystem. Consequently the hardware filter is not used and only software filtering is applied.

Baud Rates

The following baud rates are supported: 1Mb/s, 500kb/s, 250kb/s, 125kb/s, 100kb/s. However, the accuracy of baud rates that can be supported at any particular PCLK1 frequency depend on the resolution of the baud rate divisor.

Configuration

For each channel *X* supported the CDL script provides the following configuration options:

cdl_component CYGPKG_DEVS_CAN_CHANNELX

> This defines whether the channel is included. For channels 2 and 3 the default value is conditional on the STR7XX variant.

cdl_option CYGPKG_DEVS_CAN_CHANNELX_NAME

> This defines the name of the channel.

At present, only one channel, 0, is supported.

Part XXVIII. Coherent Connection Bus

Table of Contents

Chapter 100. Coherent Connection Bus overview

Introduction

The CYGPKG_COHERENT_CCB package implements support for Coherent Connection Bus (CCB) communications. This enables eCos-based systems to communicate with and control compatible members of Coherent's range of laser products.

The package provides an API for applications to send and receive messages from an eCos-based master to slave devices over the CCB. Familiarity with CCB and the relevant specifications is assumed in the following documentation.

The implementation is based on the relevant sections of the Coherent's “Integrator's Guide Coherent OBIS“ document (Part No. 1215508 Rev. AB - CoherentOBIS_IntegratorsGuide_1215508RevAB.pdf).

The CCB package features:

- CCB packet driver interface support
- Master device support
- Example applications

The eCos CCB backend interface is transport agnostic, though CCB is normally routed over an RS-485 connection. The configured target must provide a suitable platform/variant hardware package implementing the necessary low-level hardware I/O support. The CCB support relies on the platform specific CDL forcing any per-device configuration to ensure a 921600 baud 8N1 connection for the underlying serial (RS-485) communications.

Chapter 101. Configuration

This chapter shows how to incorporate the CCB support into an eCos configuration, and how to configure it once included.

Configuration Overview

The CCB support is contained in a single eCos package `CYGPKG_IO_CCB`. However, it depends on the services of a collection of other packages for complete functionality. For example, the `CYGPKG_KERNEL`, `CYGPKG_ERROR` and `CYGPKG_IO_FILEIO` packages.

Quick Start

Incorporating the CCB support into your application is straightforward. The essential starting point is to incorporate the CCB eCos package (`CYGPKG_COHERENT_CCB`) into your configuration.

This may be achieved directly using **ecosconfig add** on the command line, or the **Build->Packages…** menu item within the eCos Configuration Tool.

Depending on the CCB package configuration other packages may be required (e.g. UART device driver support). The package requires that the `CYGPKG_KERNEL` and `CYGPKG_ERROR` packages are included in the eCos application configuration.

It is recommended to include the `CYGPKG_IO_FILEIO` package for `select()` functionality to allow client-applications to avoid having to use a busy message read loop.

Configuring the CCB memory footprint

CYGNUM_IO_CCB_SIZE_MSG

This option defines the maximum size of message that can be received, and should include space for the standard message header. The default value caters for the maximum possible message size. For low-memory targets or where the maximum message sizes are fixed and known for all bus nodes, then this option can be tuned to minimise the memory footprint. Any messages that exceed this configured size will be dropped.

The value of this option is used to define the buffer sizes required by the low-level I/O driver to accept packets.

CYGNUM_IO_CCB_RECV_PKTS

This option defines the maximum number of pending received messages that can be queued for the client application. As per the Coherent Connection Bus Protocol “Only complete messages are buffered; if there is not enough buffer space remaining for a new message, the message will be discarded”. This option can be tuned to reflect the ability of the client application to read and process messages, against the memory footprint required for each descriptor.

Configuring the CCB control thread

CYGNUM_COHERENT_CCB_CONTROL_STACK_SIZE

The CCB subsystem uses an internal thread to handle asynchronous actions and to process packet reception. This option defines the *ADDITIONAL* size that is added to the variant `CYGNUM_HAL_STACK_SIZE_TYPICAL` setting to define the stack size for this thread.

CYGNUM_COHERENT_CCB_CONTROL_PRIORITY

The CCB subsystem uses an internal thread to handle asynchronous actions. This option defines the priority at which this thread is scheduled. To avoid resource starvation issues it should normally be a higher priority than the thread implementing the CCB client-application.

Configuring the CCB master server

The common CCB server support is enabled by the CYGPKG_IO_CCB_MASTER option. When enabled it provides access to the following relevant options.

CYGNUM_IO_CCB_MASTER_NUM_SLAVES

> This option specifies the maximum number of slave devices supported across all of the physical CCB ports present. This is used to avoid the need for dynamic allocation, but does mean that the value may need to be tuned to reflect the application and variant/platform usage. The maximum of 253 reflects the total unit bus limit. The master will ignore address acquisition requests when the pool of slave descriptors is exhausted.
>
> The default value of 1 reflects the expected use case of a single physical interface (port) with only a single slave device connected to that RS-485 bus. If the implementation provides multiple hardware ports, or expects multiple slaves per hardware port then this configuration option should be updated accordingly.

CYGNUM_IO_CCB_MASTER_MAX_SLAVEID_LEN

> This option specifies the maximum length of the per-slave unique identifiers supported. The default is to accept all slaves by using the maximum possible length (this option includes space for the NUL-terminator). For targets with limited memory resources, or where the application is limited to a known set of device identifier lengths, this value can be tuned to save on the run-time memory footprint. Slave devices with identifiers longer than this option will be ignored.

CYGNUM_IO_CCB_MASTER_SLAVE_POLL

> If a message has not been received from an assigned address for the period of time (in seconds) specified by this option then the master CCB stack will issue a PING request to the slave. After the fixed number of retries (CYGNUM_IO_CCB_MASTER_SLAVE_POLL_ATTEMPTS) without a response a slave is considered disconnected.

Note

An eCos CCB package extension disables the slave polling if this option is set to zero (0). Such a configuration would not be conformant with the CCB protocol definition.

CYGNUM_IO_CCB_MASTER_SLAVE_POLL_WAIT

> This option defines the period (in seconds) that the master CCB stack will wait between PING requests. A slave is declared disconnected if no response is received after CYGNUM_IO_CCB_MASTER_SLAVE_POLL_ATTEMPTS.

Chapter 102. API Overview

The main client-application access to CCB functionality is via the common named device /dev/ccb. It is the responsibility of the client-application to open a handle onto this common device, and then to read and write messages as appropriate.

The common layer will automatically start the protocol control thread and use the standard eCos cyg_io interface connection to the specific CCB interface (hardware port) device drivers.

For CYGPKG_IO_FILEIO configurations the client-application can use the open(), write(), read() and select() functions. If CYGPKG_IO_FILEIO is not configured then the standard cyg_io_lookup(), cyg_io_read() and cyg_io_write() function interface is used.

Note

The client-application writes to the CCB are *always* blocking. The read operations can be configured as non-blocking if required, and if CYGPKG_FILEIO select() support is configured will default to non-blocking operation.

The common and driver-specific CCB packages also provide access to some GET/SET operations that can be accessed via the relevant cyg_fs_fgetinfo/cyg_fs_fsetinfo or cyg_io_get_config/cyg_io_set_config functions. The “*key*” values supported are:

CYG_IO_GET_CONFIG_PENDING_MESSAGES

This *key* allows a snapshot view of the number of pending client-application RX messages.

Note

As packet reception might occur during calls to this config operation, the information returned may already be stale. If the thread reading this count is the only thread reading messages then this value can be viewed as the minimum number of messages pending. i.e. the client-application should always be able to read the returned number of messages.

A client-application using select() to do a non-busy wait for a message should not normally need to worry about the number of pending messages. This config *key* support is provided to correspond with the functionality described by the CCB protocol documentation.

CYG_IO_GET_CONFIG_CCB_STATISTICS

This option allows, where provided by the underlying hardware interface port driver, sets of statistics to be monitored. The <cyg/io/ccb_port.h> header file defines the cyg_ccb_devio_stats_t structure used to hold RX and TX information for a specific device driver. This config *key* functionality uses the <cyg/io/ccb.h> header file defined cyg_ccb_io_stats_t structure to step through the available interface statistics.

The initial cyg_ccb_io_stats_t.priv_ctx field should be initialised to NULL, and then this *key* config call repeated until the common CCB support returns cyg_ccb_io_stats_t.priv_ctx == NULL (indicating no more data). As per the following pseudocode example:

```
cyg_ccb_io_stats_t stats;
cyg_bool moredata = true;

stats.priv_ctx = NULL; // start

do {
  int res = cyg_fs_fgetinfo(pcapp->fd,CYG_IO_GET_CONFIG_CCB_STATISTICS,&stats,sizeof(stats));
  if (res < 0) {
    diag_printf("FAIL:<Get statistics error %d \"%s\">\n",res,strerror(errno));
    moredata = false;
  }
  if (moredata) {
    moredata = (stats.priv_ctx != NULL);
    if (moredata) {
      diag_printf("Device \"%s\" : RX pkts=%u (bytes=%llu) dropped=%u : TX pkts=%u (bytes=%llu) retrans=%u failed=
                  stats.dname,
                  stats.dstats.rx_pkts,
                  stats.dstats.rx_bytes,
                  stats.dstats.rx_pkts_dropped,
                  stats.dstats.tx_pkts,
                  stats.dstats.tx_bytes,
                  stats.dstats.tx_pkts_retrans,
                  stats.dstats.tx_pkts_failed);
    }
  }
} while (moredata);
```

When using a `cyg_io_handle_t` reference to the common CCB layer then the example above would use the corresponding `cyg_io` interface:

```
  Cyg_ErrNo res = cyg_io_get_config(pcapp->handle,CYG_IO_GET_CONFIG_CCB_STATISTICS,&stats,&slen);
```

CYG_IO_SET_CONFIG_CCB_BUS_RESET

Normally the client-application should not need to force a global bus reset. The common CCB layer performs a bus reset as part of the normal application startup. This config “key” may be useful during testing/development, or if an unrecoverable error is detected.

There is also a minimal function call API providing some “stateless“ message helper routines.

Application support API

These functions are available for the client-application to aid in message processing.

See the Testing section's `ccb_master` test client-application for an example of access to the `/dev/ccb` and use of these functions.

Name

cyg_ccb_build_message — Construct message

Synopsis

#include <cyg/io/ccb.h>

cyg_uint32 **cyg_ccb_build_message**(*msg*, *slave_addr*, *tag*, *cmdreq*);

Description

This function allows a standard client-application message to be constructed. The *msg* parameter is used to reference a message object to be filled. The *slave_addr* and *tag* parameters describe the destination address and client-application “message tracking number” respectively. The final parameter *cmdreq* is the NUL-terminated ASCII command or request. The passed *cmdreq* string should *NOT* contain a terminating CR (Carriage-Return). Any command/request terminators needed are added by this function.

The following pseudocode example shows the use of the function to construct a message and then send this message via the write function to a previously opened CCB handle.

```
cyg_uin32 mlen = cyg_ccb_build_message(msg,slave_addr,tag,"*IDN0?");
if (mlen) {
  write(fd,msg,mlen);
} else {
  raise_error();
}
```

Return value

The function returns the total number of bytes written to the passed *msg* buffer, or zero (`0`) if an error is detected.

Name

cyg_ccb_check_response — Check response

Synopsis

#include <cyg/io/ccb.h>

`cyg_bool` **`cyg_ccb_check_response`**`(`*`response`*`, `*`rlen`*`, `*`ecode`*`);`

Description

This helper function allows standard CCB client-application message responses to be parsed. The *`response`* is a pointer to the response data to be processed, with the *`rlen`* specifying the number of bytes (characters) of data valid in the supplied *`response`* buffer.

Responses are expected to be either the non-error “OK\r” or an error number indicated “ERR#\r” form (where # is either a positive or negative decimal number).

If non-NULL then the *`ecode`* parameter references the location updated with the “ERR” response error number, or the value `0` if an “OK”: response is given.

Return value

The boolean `true` result indicates that a valid response string was supplied. The return value of `false` indicates that the string was malformed (possibly a data reply and not a response string after all).

I/O Device Driver Interface

The header file `<cyg/io/ccb_devio.h>` defines the interface between the common CCB support and the target specific device drivers. The device drivers provide the physical CCB packet communication support.

Physical connections are supplied to the common layer via the relevant target/platform defining a `cyg_ccb_port_instance_t` structure (via the `CCB_PORT()` macro), which provides the mapping to the relevant low-level hardware I/O driver via the supplied named device.

The `cyg_ccb_devio_port_t` structure defines a hardware port instance (i.e. a physical RS-485 hardware interface) driver in conjunction with a standard I/O driver device descriptor `DEVTAB_ENTRY()` definition. A driver normally instantiates itself via the `CCB_DEVIO_PORT()` macro to populate a `cyg_ccb_devio_port_t` structure.

The device drivers interface with the CCB common layer via the `cyg_ccb_devio_funs_t` and `cyg_ccb_callbacks_t` structures implemented by the `src/ccb_devio.c` support.

The device drivers fundamentally provide a per-port blocking transmit function, and asynchronous packet reception. The driver calls back into the common CCB layer via the (DSR context) `cyg_ccb_callbacks_t` functions `tx_done()` and `rx_pkt()`.

Chapter 103. Internals

The main CCB protocol control thread is provided by the package's `src/ccb_protocol.c` source file. The control thread loops waiting for event flags indicating that some CCB processing is required.

The main tasks of the control loop are monitoring slave packet reception timing and polling slaves as configured, to ensure they are still connected, and processing asynchronous RX packets from any of the available hardware ports. The CCB bus management messages are handled by this control loop; and validated client-application messages are forwarded to the client-application via a RX message queue for the client-application to use via the relevant read interface. The bus management code can also inject locally generated slave connect/disconnect messages into this RX queue for processing by the client-application (as per the CCB design).

The `src/ccb_protocol.c` source also implements the I/O driver layer used to present the “/dev/ccb” named device, used by the client-application to allow messages to be sent and received. This works in conjunction with the `src/ccb_devio.c` source that provides the basic message<->packet framing/deframing support.

Chapter 104. Debug and Test

Debugging

Asserts

If the target platform resources allow, then the first step in debugging should be to enable ASSERTs. The inclusion of assert checking will increase the code footprint and lower the performance, but does allow the code to catch internal errors from unexpected data values. e.g. when the application/client is not able to guarantee the validity of data passed into the CCB code.

The CCB asserts are controlled via the standard eCos Infrastructure CYGPKG_INFRA package CYGDBG_USE_ASSERTS option. If enabled then run-time assertion checks are performed by the CCB package.

If assertions are enabled and a debugger is being used, it is normally worthwhile setting a breakpoint at start-up on the cyg_assert_fail symbol. The debugger will then stop prior to entering the default busy-loop assert processing.

Diagnostic Output

In conjuction with the CYGDBG_COHERENT_CCB_DEBUG CDL configuration setting and its sub-options, the header-file src/ccb_diag.h implements the CCB I/O package specific debug control.

When CYGDBG_COHERENT_CCB_DEBUG is enabled a set of individually selectable sub-systems are available to control the diagnostic output generated.

However, when developing or debugging the CCB implementation it may be simpler (with less build side-effects) to control the debugging output via direct uncommenting of the necessary manifests at the head of the src/ccb_diag.h source file, than re-configuring the complete eCos configuration via the CDL. This approach will limit rebuilding to just the CCB package.

Note

When enabled, some diagnostic output may adversely affect the operation of the CCB support as seen by 3rd-party code. For example, “slow” serial diagnostic output of the packet parsing and response generation could mean that a significant amount of time passes, such that the CCB support no longer adheres to the timeout limits imposed by external code.

Testing

The configuration option CYGPKG_IO_CCB_TESTS defines a set of tests that are built.

By default the package will only build the deterministic, automatic, tests. However, the option CYGPKG_IO_CCB_TESTS_MANUAL can be defined to build extra tests that may require manual user-intervention, or are more realistic real-world example applications.

ccb_ut

The ccb_ut test application performs some unit-testing of the CCB implementation. The test assumes specific features of the tests/ccb_master.c example client-application to perform a variety of fixed tests.

The ccb_ut implements a software driver which can be used to simulate a bus, without the requirement for the configuration to have access to a physical RS-485 bus. This is used to test core MASTER CCB client-application functionality.

Note

It is recommended that before executing this test that you disconnect any physical hardware.

Currently not all aspects of the CCB protocol are exercised by the test.

ccb_master

The ccb_master is a simple example client-application that could form the basis of an actual customer implementation. The test does not automatically exit (and hence is unsuitable for the eCosCentric automated test farm).

The application currently just waits for a slave to connect, and then issues a fixed set of queries to the slave.

Part XXIX. STM32 Coherent Connection Bus Driver

Table of Contents

Chapter 105. STM32 Coherent Connection Bus Driver overview

Introduction

The CYGPKG_DEVS_CCB_CORTEXM_STM32 package provides a low-level bus driver implementation for use by the Coherent Connection Bus (CCB) CYGPKG_IO_CCB communication package. It provides the device-level API to implement the actual hardware interface support as required. Familiarity with CCB and the relevant specifications is assumed in the following documentation.

The implementation is based on the relevant sections of the Coherent's “Integrator's Guide Coherent OBIS“ document (Part No. 1215508 Rev. AB - CoherentOBIS_IntegratorsGuide_1215508RevAB.pdf).

This driver package provides the I/O connection via the STM32 U(S)ART interfaces. Normally CCB is routed over a half-duplex RS-485 connection. This driver provides software collision detection, with the corresponding TX auto-retry (up to to an optional configuration limit if required).

Chapter 106. Configuration

This chapter shows how to incorporate the CCB support into an eCos configuration, and how to configure it once included.

Configuration Overview

The common CCB support is contained in the eCos package `CYGPKG_IO_CCB`. However, it depends on the services of a collection of other packages for complete functionality, with this `CYGPKG_DEVS_CCB_CORTEXM_STM32` package providing specific I/O functionality.

Normally this package should not need to be manually added, since it will be automatically provided as part of suitable target configurations.

The functionality of this driver package itself relies on some platform/variant provided configuration. For platforms that require software control of the transceiver Device Enable (DE) signal for any of the CCB ports they will need to implement the feature `CYGINT_DEVS_CCB_CORTEXM_STM32_TXCTRL`.

Configuring the STM32 CCB driver

CCB use of the underlying STM32 U(S)ART interfaces is only enabled if the serial driver support for the specific hardware interface is disabled.

Common configuration options applicable to all configured U(S)ART interfaces:

CYGPKG_DEVS_CCB_CORTEXM_STM32_TIMER

> This option is used to select the STM32 timer block assigned to this CCB I/O driver. It is used to ensure correct bus operation timing. The sub-option `CYGNUM_DEVS_CCB_CORTEXM_STM32_TIMER_INTR_PRI` is used to configure the relative interrupt priority for the timer handler.

CYGNUM_DEVS_CCB_CORTEXM_STM32_RETRIES

> This option enables support for abandoning a transmission after the configured number of retry attempts. The default of 0 disables the retry limit check, with a colliding transmission being retried infinitely. NOTE: The Coherent OBIS Integrators Guide Part# 11215508 Rev. AB (dated 4/2012) has conflicting descriptions re. transmission. The “Random Delay” section is explicit in stating “… there is no provision for discarding a message after many collisions; message transmission will retry until it succeeds”. This however is counter to the Figure 5-6 “Outbound Message Transmission Flow” diagram, which implements a retry counter and terminates the transmission attempt after a number of retries. This configuration option allows the developer to choose the model required by their application. A value of 0 will disable the retry support and a colliding TX will be retried until it is successful (no collision detected). A non-zero value will be treated as a count of the number of attempts to be made before abandoning the transmission and indicating a TX BUSY error.

CYGIMP_DEVS_CCB_CORTEXM_STM32_STATISTICS

> This option is normally disabled by default since it has a (minor) memory footprint and performance hit. If enabled then the driver will track counts of packet transfers and errors. This may be useful to client applications to ascertain bus performance and “quality”.

For the following CDL option names the *#* character in the option names indicates the port number for a specific hardware interface.

If the underlying platform/variant provides access to a STM32 U(S)ART interface, via implementing `CYGINT_DEVS_CCB_CORTEXM_STM32_UART#`, then the following per-interface configuration options are available:

CYGPKG_DEVS_CCB_CORTEXM_STM32_UART#

This is the main option to control use of an interface as a CCB port. Access to the options described below are dependant on this option being enabled.

CYGDAT_DEVS_CCB_CORTEXM_STM32_UART#_NAME

This option specifies the name of the CCB port for the corresponding STM32 U(S)ART. This is the name that an eCos application should use to access this device via cyg_io_lookup(), open(), or similar calls.

Note

Normally for CCB use the client-application should *NOT* need to directly interact with this named device driver, since the common `CYGPKG_IO_CCB` support will automatically access the target platform configured devices declared via `CCB_PORT()` definitions.

CYGNUM_DEVS_CCB_CORTEXM_STM32_UART#_INT_PRI

Interrupt handler priority for U(S)ART events.

CYGNUM_DEVS_CCB_CORTEXM_STM32_UART#_TXINTR_PRI

TX DMA interrupt handler priority.

CYGNUM_DEVS_CCB_CORTEXM_STM32_UART#_RXINTR_PRI

RX DMA interrupt handler priority.

CYGHWR_DEVS_CCB_CORTEXM_STM32_UART#_ONEBIT

This option controls the configuration of the STM32 hardware serial bit sampling. The (default) NOISY selection is suited to off-board interfaces where noise/glitches may occur, but is less tolerant of clock differences. The CLOCK selection is more tolerant of clock deviation between the transmitted and receiver. Use of CLOCK may be more suited to on-board high bitrate connections.

Chapter 107. Debug and Test

Debugging

Asserts

If the target platform resources allow, then the first step in debugging should be to enable ASSERTs. The inclusion of assert checking will increase the code footprint and lower the performance, but does allow the code to catch internal errors from unexpected data values. e.g. when the application/client is not able to guarantee the validity of data passed into the CCB code.

The CCB driver asserts are controlled via the standard eCos Infrastructure CYGPKG_INFRA package CYGDBG_USE_ASSERTS option. If enabled then run-time assertion checks are performed by the CCB driver package.

If assertions are enabled and a debugger is being used, it is normally worthwhile setting a breakpoint at start-up on the cyg_assert_fail symbol. The debugger will then stop prior to entering the default busy-loop assert processing.

Diagnostic Output

The STM32 CCB driver provides the ability for some diagnostic output to be manually enabled by editing the src/stm32_ccb.c directly. However, in normal application development the low-level diagnostics for this driver should not be required.

Note

The diagnostics within this package are to aid driver development and debug, and will adversely affect the operation of the CCB support as seen by 3rd-party code. For example, “slow” serial diagnostic output of the packet parsing and response generation could mean that a significant amount of time passes, such that the CCB support no longer adheres to the timeout limits imposed by external code.

Similar to the diagnostic output, the source can be edited to manually enable support for software driven signals that can be sampled by a suitable Logic State Analyser (LSA) setup. This can be used to track time critical events.

Warning

Any STM32 I/O pins chosen for LSA signalling should be carefully chosen to avoid adverse operation of the target platform in use.

Part XXX. MODBUS

Important

This eCosPro-MODBUS package is STRICTLY LICENSED FOR INTERNAL EVALUATION AND TESTING PURPOSES ONLY for a maximum period of THREE months from the initial delivery of your eCosPro release. It may not be used for production purposes nor redistributed in full or in part in any format, including source code, binary code and object code format. Shipment of prototypes, hardware or products containing the package in any format is STRICTLY PROHIBITED.

A separate COMMERCIAL LICENSE for this package from eCosCentric is required to receive technical support for the package as well as permit distribution of binary forms of this package.

Some releases of eCosPro may not include evaluation copies of this package. In this case, please contact eCosCentric for licensing and availability. You must obtain written permission from eCosCentric to exceed the evaluation period.

Table of Contents

Chapter 108. MODBUS overview

Introduction

eCosPro-MODBUS is eCosCentric's commercial name for the `CYGPKG_MODBUS` package. The `CYGPKG_MODBUS` package implements a MODBUS server and a ModbusTCP transport layer, and provides an API to enable applications to implement the actual hardware interaction as required.

The implementation is based on the following documents available from the MODBUS www.modbus.org website:

- “MODBUS Application Protocol Specification V1.1b3”
- “MODBUS Messaging on TCP/IP Implementation Guide V1.0b”

Familiarity with MODBUS, and ModbusTCP, and the relevant specifications is assumed in the following eCosPro-MODBUS specific documentation.

The MODBUS package features:

- MODBUS server

  Generic MODBUS request processing. This parses MODBUS function code requests, passing validated requests to the relevant backend handler function as appropriate.

- ModbusTCP transport

  TCP/IP (Ethernet) transaction support. This accepts ModbusTCP client requests, passing valid requests to the MODBUS server for processing.

- Example applications

For MODBUS server applications the user-supplied code registers a backend driver, which will implement the actual hardware operations as defined by the supported MODBUS function codes.

The backend interface is transport agnostic. MODBUS over serial or TCP share the same PDU format structure within requests, with the transport layer wrapping requests and responses as appropriate for the medium being used:

```
ModbusTCP       | MBAP Header | FuncCode | Data…. |
MODBUS Serial       | Address | FuncCode | Data…. | Checksum |
                              \                     /
                               \_____identical_____/
```

The `CYGPKG_MODBUS` package currently only implements a ModbusTCP transport layer. There is no MODBUS client API, or serial (MODBUS-RTU, MODBUS-ASCII) transport support.

Chapter 109. Configuration

This chapter shows how to incorporate the MODBUS support into an eCos configuration, and how to configure it once included.

Configuration Overview

The MODBUS support is contained in a single eCos package `CYGPKG_MODBUS`. However, it depends on the services of a collection of other packages for complete functionality. For example, the ModbusTCP server implementation is tightly bound with the eCos networking stack support.

Quick Start

Incorporating the MODBUS support into your application is straightforward. The essential starting point is to incorporate the MODBUS eCos package (`CYGPKG_MODBUS`) into your configuration.

This may be achieved directly using **ecosconfig add** on the command line, or the **Build->Packages…** menu item within the eCos Configuration Tool.

Depending on the MODBUS package configuration other packages may be required (e.g. network stack support). The package requires that the `CYGPKG_KERNEL`, `CYGPKG_ERROR` and `CYGPKG_MEMALLOC` packages are included in the eCos application configuration.

Configuring the MODBUS server

The common MODBUS server support is enabled by the `CYGFUN_MODBUS_SERVER` option. When enabled it provides access to the following relevant options.

CYGNUM_MODBUS_SERVER_MAXSERVER

> This option defines the maximum number of client connections supported by the server. This is the number of individual, active, concurrent, socket connections allowed.

CYGNUM_MODBUS_SERVER_TRANSACTIONS

> This option specifies the maximum number of active, simultaneous, MODBUS transactions per client connection.

CYGNUM_MODBUS_SERVER_TRANSPORTS

> This option defines the maximum number of active transports the MODBUS server will support. If the end application configuration provides more than this number then not all transports may be instantiated. For most applications only a single transport will ever be provided and needed. For example, ModbusTCP for Ethernet based systems.

CYGNUM_MODBUS_SERVER_MESSAGES_API

> Depending on the other configuration options the package will define a base number of message descriptors needed in `CYGNUM_MODBUS_SERVER_MESSAGES`. This option extends the size of the message pool over and above the explicit active MODBUS transaction message requirement. This can be tuned to reflect the requirements of the user application implementing the backend.

CYGNUM_MODBUS_SERVER_THREAD_STACK_SIZE

> This option is used to increase the size of the main MODBUS control thread. This value is added to the platform defined `CYGNUM_HAL_STACK_SIZE_TYPICAL`.

CYGNUM_MODBUS_SERVER_THREAD_PRIORITY

This option defines the scheduler priority of the main MODBUS control thread.

Configuring the ModbusTCP Server

The `CYGPKG_MODBUS_TRANSPORT_TCP` provides the transport implementation for ModbusTCP. When enabled the configuration must include a suitable network stack (BSD or lwIP).

CYGNUM_MODBUS_SERVER_PORT

This option defines the port that the ModbusTCP server will listen to for client requests. The default of `502` is the standard ModbusTCP port, and normally would not need to be re-configured.

CYGIMP_MODBUS_TRANSPORT_TCP_ACM

Enabling this option allows a set of client network addresses to be configured at run-time via the ACM API. When addresses have been configured only connections from those registered hosts will be accepted.

- The `Disabled` mode configures the ModbusTCP transport to ignore ACM API operations, and to accept connections from any host.
- The `Insecure` mode will accept any connection UNTIL at least one explicit address has been registered via the ACM API, and then host addresses will be checked for acceptance.
- The `Secure` mode will NOT accept any connections unless the address has been explicitly registered via the ACM API.

The setting of this option will depend on how the server device will be deployed and accessed in the field. For testing it may be acceptable to have any client host interact with the device, in which case `Disabled` or `Insecure` should be selected. Selecting `Insecure` allows for start-of-day acceptance of any client host address, but for some application control, configuration, to subsequently limit the acceptable addresses. The `Secure` mode ensures that only application configured client host addresses are ever supported, which requires suitable run-time application configuration to setup the required ACM pools to allow access.

Warning

Using `Secure` mode means that if the application does NOT call the `cyg_modbus_acm_add()` to register an address then *NO* connections can be established to the MODBUS server.

CYGNUM_MODBUS_TRANSPORT_TCP_IDLE_TIMEOUT

This option specifies the number of seconds before idle connections to the ModbusTCP server are closed. The default setting of 0 disables the feature. Normally connections from clients are held open until the client explicitly closes them. This option allows for an “idle” timeout to be specified that will close the connection at the server end if no MODBUS requests are received from a client connection within this configured timeout.

Note

If ACM support is configured then only non-priority connections will be closed by the ModbusTCP server.

CYGNUM_MODBUS_TRANSPORT_TCP_THREAD_STACK_SIZE

This option is used to increase the size of the ModbusTCP internal thread. This value is added to the platform defined `CYGNUM_HAL_STACK_SIZE_TYPICAL`.

CYGNUM_MODBUS_TRANSPORT_TCP_THREAD_PRIORITY

This option defines the scheduler priority of the ModbusTCP internal thread.

Chapter 110. API Overview

The main MODBUS API provides a serialisation layer between the low-level MODBUS operations and user-application threads.

For MODBUS server configurations the main server control loop thread interacts with a transport layer, for example ModbusTCP, and an application specific backend layer.

The transport layer provides the physical communication support for the selected medium. For ModbusTCP this will normally be Ethernet, and for MODBUS ASCII/RTU a RS232 or RS485 connection. The transport medium layer wraps the common MODBUS PDU request and response messages for transmission.

The server control thread processes the MODBUS PDU encapsulated requests, calling the provided backend routines as appropriate.

The application supplied backend *`descriptor`* provides the support for the specific hardware I/O present on the device being implemented.

Application API

These functions are used by the user application to configure the server operation.

Name

cyg_modbus_server_start — Start MODBUS server

Synopsis

#include <cyg/modbus.h>

```
cyg_handle_t cyg_modbus_server_start(descriptor, backend_private);
```

Description

Initialise the server from the CDL configured state, and start the transport layers listening for client requests. The supplied *descriptor* specifies the backend driver to be used to handle requests. The *backend_private* parameter can be used to reference any context required by the specific backend *descriptor*.

The call will block until all the configured transports are ready to accept connections from remote clients.

Return value

A handle onto the server instance, or 0 on failure.

Name

cyg_modbus_server_stop — Stop MODBUS server

Synopsis

#include <cyg/modbus.h>

```
Cyg_ErrNo cyg_modbus_server_stop(server);
```

Description

Stop the referenced *server* from accepting any new requests. The call may block until any "pending" transactions are completed by the backend-handlers/user-application, or such pending transactions have been timed-out by the main MODBUS server control loop. This means the function may block for a maximum of the active transaction timeout (as specified by the backend supplied `timeout` field).

Return value

The function will return `ENOERR` on success. If the stop event cannot be delivered to the MODBUS control thread due to the communication mailbox being full, then `EAGAIN` is returned. If the *server* parameter is invalid then `EINVAL` will be returned.

Backend API

These functions are provided for the user application attached backend handler routines. They are used to provide responses back to the remote client, with the necessary control thread and transport layer information referenced by the opaque `cyg_handle_t ctx` parameter passed to the backend handler function.

For handlers that can provide immediate responses the functions can be called directly from the handler function (in the main MODBUS control thread context). If the handler cannot provide an immediate response, then the processing should be deferred to another thread which can then subsequently use the same API to deliver a response.

Note

For valid responses it is the responsibility of the handler function to return a correctly formatted PDU object to the client.

Name

cyg_modbus_response — Provide MODBUS response

Synopsis

#include <cyg/modbus.h>

```
Cyg_ErrNo cyg_modbus_response(ctx, len, data);
```

Description

Provide the MODBUS server with a valid PDU response for a request handed to the relevant backend handler. The *data* allows the backend application to reference its own data buffer, which may be in read-only memory.

Return value

Currently only `ENOERR` is returned.

Name

cyg_modbus_raw_pdu — Access raw PDU buffer

Synopsis

#include <cyg/modbus.h>

`void` **`cyg_modbus_raw_pdu`**`(`*`ctx`*`, `*`len`*`, `*`data`*`);`

Description

This function provides low-level access to the request PDU buffer for the backend handler code. Before calling the backend handler function the MODBUS server will have already parsed the request and will pass the relevant request data as parameters to the handler function. However, for some backend implementations it may be useful to re-use the request memory space as the holding location for the response. This can avoid the backend having to manage its own dynamic memory allocations.

The example `packages/services/modbus/current/tests/backend_dummy.c` source file provided with the package provides examples of using this `cyg_modbus_raw_pdu()` function in conjunction with `cyg_modbus_response_nocopy()` to re-use the transaction PDU buffer. For example:

```
{
  cyg_uint8 len;
  cyg_uint8 *pdata;
  cyg_modbus_raw_pdu(ctx,&len,&pdata);
  … code to fill pdata buffer …;
  len = response_length;
  cyg_modbus_response_nocopy(ctx,len);
}
```

Name

cyg_modbus_response_nocopy — Provide MODBUS response (no copy)

Synopsis

#include <cyg/modbus.h>

```
cyg_bool cyg_modbus_response_nocopy(ctx, len);
```

Description

Like the `cyg_modbus_response()` call this function is used to provide the MODBUS server with a valid response to a request handed to the backend for processing. This call assumes that the original PDU buffer (as referenced through the opaque *`ctx`* handle) contains the response data.

This “PDU buffer” re-use can simplify the backend processing, since it no longer needs to dynamically manage buffers that are handed-off to the main MODBUS server control.

Return value

Currently only `ENOERR` is returned.

Name

cyg_modbus_response_exception — Provide MODBUS exception response

Synopsis

#include <cyg/modbus.h>

```
cyg_bool cyg_modbus_response_exception(ctx, ecode);
```

Description

If the backend handler detects an error, or needs to notify the remote client of an exceptional result, this function is used to provide such a MODBUS exception response.

Return value

Currently only `ENOERR` is returned.

Name

cyg_modbus_get_uid — Read “Unit ID” of request

Synopsis

#include <cyg/modbus.h>

Cyg_ErrNo **cyg_modbus_get_uid**(*ctx*, *uid*);

Description

This function provides access to the underlying “Unit ID” value for the passed *ctx* request handle parameter. The passed *uid* parameter references the location to be written with the relevant identifier value.

Return value

On success `ENOERR` is returned, and if non-`NULL` then the referenced *uid* location is updated with the “Unit ID”. If the underlying transport does not provide access to a suitable identifier value then the `ENOENT` error code is returned.

ModbusTCP specific API

These functions are used by the user application to configure features of the ModbusTCP specific transport layer.

Name

cyg_modbus_acm_add — Add client address to Access Control Mechanism pool

Synopsis

#include <cyg/modbus.h>

```
Cyg_ErrNo cyg_modbus_acm_add(server, ptype, ipaddr, addrlen);
```

Description

Depending on the `CYGPKG_MODBUS` and user application configuration, the ModbusTCP transport layer may provide support for an Access Control Mechanism (ACM) based on client (remote host) network addresses. This function allows addresses to be registered, so that subsequent connections from the remote host will be allowed.

The *`ptype`* defines which of the ModbusTCP ACM pools the supplied client address will be registered with:

- MODBUS_CONNPOOL_PRI

  The priority pool is for addresses for which the server should never close the socket connection.

- MODBUS_CONNPOOL_NONPRI

  The non-priority pool is for addresses where the server may close the socket connection.

Currently the *`ipaddr`* can describe either a `AF_INET` (for IPv4) or `AF_INET6` (for IPv6) family address, with *`addrlen`* being the size of the relevant address family.

Return value

A standard `Cyg_ErrNo` error code is returned, with `ENOERR` indicating success. If the ModbusTCP ACM is not provided by the attached transport layer then `ENOENT` is returned. The error `EIO` indicates a failure to communicate with the server control thread. The transport specific implementation may return further error indications as appropriate.

Name

cyg_modbus_acm_remove — Remove ACM registered client address

Synopsis

#include <cyg/modbus.h>

`Cyg_ErrNo` **`cyg_modbus_acm_remove`**`(`*`server`*`, `*`ipaddr`*`, `*`addrlen`*`);`

Description

This function will remove the supplied remote host address from all ACM pools that have a matching entry. If ACM is being used then this may stop future requests from the remote host being processed.

Currently the *`ipaddr`* can describe either a `AF_INET` (for IPv4) or `AF_INET6` (for IPv6) family address, with *`addrlen`* being the size of the relevant address family.

Return value

A standard `Cyg_ErrNo` error code is returned, with `ENOERR` indicating success. If the ModbusTCP ACM is not provided by the attached transport layer then `ENOENT` is returned. The error `EIO` indicates a failure to communicate with the server control thread. The transport specific implementation may return further error indications as appropriate.

MODBUS Exceptions

The MODBUS standard defines a fixed set of exception codes, with implicit interpretations. These should be used in conjunction with the `cyg_modbus_exception()` function by backend handlers to notify the requesting client of an error.

MODBUS_EXCEPTION_ILLEGAL_FUNCTION

> The function code received in the request is not an allowable action for the server. This may be because the function code is only applicable to newer devices, and was not implemented in the unit selected. It could also indicate that the server is in the wrong state to process a request of this type, for example because it is unconfigured and is being asked to return register values.

MODBUS_EXCEPTION_ILLEGAL_ADDRESS

> The data address supplied in the request is not an allowable starting address for the server when taken into account with the number of items to be accessed. For example, for a device with 100 registers (accessed as addresses 0 thru 99) then the PDU supplied address 96 is valid when a request for 4 or fewer items is made, but invalid if 5 or more items are requested.

MODBUS_EXCEPTION_ILLEGAL_VALUE

> A value suppled in the request is not valid for the server. This normally indicates a fault in the structure of a request PDU (e.g. unexpected length). It should be noted that it does *NOT* indicate that a supplied data item is outside of the set acceptable for a specific register. The MODBUS protocol does not interpret or apply any significance of any particular data values for any particular registers.

MODBUS_EXCEPTION_DEVICE_FAIL

> Indicates an unrecoverable error occurred whilst the server was performing the requestion MODBUS function.

MODBUS_EXCEPTION_ACKNOWLEDGE

> This is a specialized response used with specific programming commands. This is where the server has accepted a request, and is processing it, but where the action will take a long time to complete. This response is returned to the client to prevent a

client timeout error waiting for a standard response. It is expected that the client will periodically issue the specific “Poll Program Complete” request to check if the processing has completed.

Note

The eCosPro-MODBUS server does not explcitly support this functionality, since it would be a feature of the user-application supplied backend driver.

MODBUS_EXCEPTION_DEVICE_BUSY

This is a specialized response used with specific programming commands. This is used to indicate to the client that the server is busy processing a long-duration command. The client should re-attempt the request in the future.

MODBUS_EXCEPTION_MEMORY_PARITY_ERROR

This exception is specific to the `MODBUS_FUNC_READ_FILE_RECORD` (20) and `MODBUS_FUNC_WRITE_FILE_RECORD` (21) functions, when using with reference type 6. It is used to indicate that the extended file area failed to pass a consistency check. i.e. the server attempted to read the record file, but detected a parity error in the memory. The client may retry the request, but maintenance of the server device may be required.

MODBUS_EXCEPTION_GW_PATH_UNAVAILABLE

This is a specialized exception returned in conjunction with gateway support. It indicates that the gateway was unable to allocate an internal communication path to the requested UID. This exception usually indicates the gateway device is misconfigured or overloaded.

MODBUS_EXCEPTION_GW_TARGET_FAILED

This is a specialized exception returned in conjuction with gateway support. It indicates that no response was obtained from the target device. This usually indicates that the target device is not present on the MODBUS network.

Backend Interface

The `cyg_modbus_backend_t` structure defines the set of handler functions used to implement the support for the relevant MODBUS operation. A simple NUL terminated ASCII `name` can be provided as a human-readable description of the backend. The application attaches the backend descriptor structure via the `cyg_modbus_server_start()` function.

Note

To avoid blocking the main MODBUS server processing loop the attached handler functions should *NEVER* block. The handlers should be written to return control as quickly as possible. If a hardware operation required by the specific MODBUS function then the handler should schedule suitable code, which will asynchronously respond to the request when the required data is available or action has been performed. The provided `tests/modbus_ut.c` example application provides one possible implementation of such “deferred” support.

If a particular MODBUS function is not supported (or required) by the backend implementation then a `NULL` pointer can be used, with the server support returning a `CYG_MB_EXCEPTION_ILLEGAL_FUNCTION` equivalent error to the calling client.

The backend structure is defined in the `<cyg/modbus.h>` header, which also provides the function prototypes for the individual handlers as detailed over the following pages. The `cyg_mbop_` prefix is used for the prototypes to indicate a MODBUS Backend OPeration.

Each handler is passed the parameters `ctx` and `private`. The `ctx` value is an opaque context descriptor for the main server loop, and is used to identify individual transactions via the API. The `private` parameter is the `private` context value specified when the backend is registered, and can be the hook for the user application hardware support.

The backend descriptor also provides the BASIC (mandatory) and REGULAR (optional) NUL terminated ASCII identification strings accessed via the MEI Type 14 function code.

The `timeout` descriptor field is used to force a timely “exception” response back to a client when the backend does not provide a response within the configured number of seconds. This MODBUS control generated timeout exception replaces any subsequent response made by the user-application for the relevant `ctx` handle.

Other than the device `idread` function the handlers do not return any error state. The handlers are passed validated requests, and are expected to return a valid response or exception via the provided MODBUS API.

Name

cyg_mbop_read_discrete_inputs — Read discrete inputs

Synopsis

```
#include <cyg/modbus.h>
```

```
void cyg_mbop_read_discrete_inputs(ctx, private, addr, numinputs);
```

Description

Discrete inputs are read-only status bits.

Name

cyg_mbop_read_coils — Read coils

Synopsis

#include <cyg/modbus.h>

```
void cyg_mbop_read_coils(ctx, private, addr, numcoils);
```

Description

Coils are read/write single-bits.

Name

cyg_mbop_write_single_coil — Write single coil

Synopsis

#include <cyg/modbus.h>

void **cyg_mbop_write_single_coil**(*ctx*, *private*, *addr*, *outval*);

Description

Coils are read/write single-bits.

Name

cyg_mbop_write_multiple_coils — Write multiple coils

Synopsis

#include <cyg/modbus.h>

```
void cyg_mbop_write_multiple_coils(ctx, private, addr, numout, outputs);
```

Description

Coils are read/write single-bits.

Name

cyg_mbop_read_input_regs — Read input registers

Synopsis

#include <cyg/modbus.h>

```
void cyg_mbop_read_input_regs(ctx, private, addr, numin);
```

Description

Input registers are 16-bit read-only registers.

Name

cyg_mbop_read_holding_regs — Read holding registers

Synopsis

#include <cyg/modbus.h>

```
void cyg_mbop_read_holding_regs(ctx, private, addr, numregs);
```

Description

Holding registers are 16-bit read/write registers.

Name

cyg_mbop_write_single_reg — Write single holding register

Synopsis

#include <cyg/modbus.h>

```
void cyg_mbop_write_single_reg(ctx, private, addr, value);
```

Description

Holding registers are 16-bit read/write registers.

Name

cyg_mbop_write_multiple_regs — Write multiple holding registers

Synopsis

#include <cyg/modbus.h>

```
void cyg_mbop_write_multiple_regs(ctx, private, addr, numregs, outputs);
```

Description

Holding registers are 16-bit read/write registers.

Name

cyg_mbop_rw_multiple_regs — Read and/or write multiple holding registers

Synopsis

#include <cyg/modbus.h>

```
void cyg_mbop_rw_multiple_regs(ctx, private, raddr, rnum, waddr, wnum, outputs);
```

Description

Holding registers are read/write 16-bit registers.

Note

The MODBUS standard dictates that the write is performed *BEFORE* the read.

Name

cyg_mbop_mask_reg — Mask holding register

Synopsis

#include <cyg/modbus.h>

```
void cyg_mbop_mask_reg(ctx, private, addr, and, or);
```

Description

Holding registers are read/write 16-bit registers.

Name

cyg_mbop_read_fifo_queue — Read FIFO contents

Synopsis

#include <cyg/modbus.h>

```
void cyg_mbop_read_fifo_queue(ctx, private, addr);
```

Description

FIFO queues are sets of 16-bit registers. This operation should not empty the queue.

Name

cyg_mbop_read_file_record — Read file records

Synopsis

#include <cyg/modbus.h>

```
void cyg_mbop_read_file_record(ctx, private, numrec, records);
```

Description

Read one or more file records.

Name

cyg_mbop_write_file_record — Write file records

Synopsis

#include <cyg/modbus.h>

```
void cyg_mbop_write_file_record(ctx, private, numrec, records);
```

Description

Write one or more file records.

Name

cyg_mbop_read_id — Return specific extended device ID

Synopsis

#include <cyg/modbus.h>

Cyg_ErrNo **cyg_mbop_read_id**(*private*, *objectid*, *nextid*, *len*, *buffer*);

Description

This function does not provide a response, but purely supplies “Extended” `ObjectID` data to the common MEI Type 14 request handler.

The *objectid* parameter specifies the object to be returned. The caller supplies an optional data buffer pointer in the *buffer* parameter, with the value referenced by the *len* pointer being the space available in the buffer. If the *objectid* is supported then the buffer is filled, and *len* is updated with the actual length of the object. The use of a `NULL` *buffer* allows a caller to ascertain the length of an object prior to calling again with a supplied buffer of the required size.

If the *nextid* pointer is supplied then the next available object ID after *objectid* is written, or `0` if no more objects are available.

Return value

On success `ENOERR` is returned and the *len* and *nextid* values updated accordingly. If the handler does not recognise the passed *objectid* then `ENOENT` is returned and **len* is *NOT* updated. If the passed **len* is not large enough for the object then `EAGAIN` is returned and **len* is update with the required length.

Name

cyg_mbop_canopen — Perform CANOPEN operation

Synopsis

#include <cyg/modbus.h>

```
void cyg_mbop_canopen(ctx, private, len, data);
```

Description

The CYGPKG_MODBUS does not implement any specific CANOPEN support, but does pass through the raw request to the backend driver which may implement support as required.

Note

Please refer to the MODBUS website or the CiA (CAN in Automation) website for a copy and terms of use that cover Function Code 32 MEI Type 13.

Example backend

The `packages/services/modbus/current/tests/backend_dummy.c` source file provides an example of the backend support required for a MODBUS server implementation. It can form the starting point for a customer hardware solution.

The source as provided implements a very basic system to allow verification of the MODBUS server support. The backend does not provide real hardware I/O interaction, but “simulates” responses.

Chapter 111. Internals

The main MODBUS control thread serialises MODBUS backend processing by accepting requests from a mailbox. It parses the request data, and where valid (and supported by the user-application supplied backend) will pass the data to the relevant handler function for processing.

Note

Since the backend handler functions are called from the main MODBUS control thread context they should *NEVER* block. This limits the operations that can be performed within the specific handler functions. If scheduling is required to complete an operation then control should be passed to a suitable user-application thread.

The MODBUS requests are posted to the main MODBUS control thread from the transport listener code. The transports are responsible for the specific communication layer wrapping of the MODBUS PDU packets.

Multiple transport layers can be configured and the main control loop will handle requests from all, directing responses back to the original requesting transport layer. This allows a single application (for example) to provide ModbusTCP support as well as serial line server support. Normally transport implementations would be limited to ModbusTCP, MODBUS-RTU or MODBUS-ASCII however the user could provide support for arbitrary transport mechanisms, e.g. a MODBUS-LOCALTEST special transport for doing internal memory-based loopback messaging for unit-testing.

Chapter 112. Debug and Test

Debugging

Asserts

If the target platform resources allow then the first step in debugging should be to enable ASSERTs. The inclusion of assert checking will increase the code footprint and lower the performance, but does allow the code to catch internal errors from unexpected data values. e.g. when the application/client is not able to guarantee the validity of data passed into the MODBUS code.

The MODBUS asserts are controlled via the standard eCos Infrastructure CYGPKG_INFRA package CYGDBG_USE_ASSERTS option. If enabled then run-time assertion checks are performed by the MODBUS package.

If assertions are enabled, and a debugger is being used it is normally worthwhile setting a breakpoint at start-up on the cyg_assert_fail symbol, so that the debugger will stop prior to entering the default busy-loop assert processing.

Diagnostic Output

In conjuction with the CYGDBG_MODBUS_DEBUG CDL configuration setting and sub-options, the header-file src/modbus_diag.h implements the MODBUS specific debug control.

When CYGDBG_MODBUS_DEBUG is enabled a set of individually selectable sub-systems are available to control the diagnostic output generated.

However, when developing or debugging the MODBUS implementation it may be simpler (with less build side-effects) to control the debugging output via direct uncommenting of the necessary manifests at the head of the src/modbus_diag.h source file, than re-configuring the complete eCos configuration via the CDL. That way only the MODBUS package will be re-built.

Note

Some diagnostic output if enabled may adversely affect the operation of the MODBUS support as seen by 3rd-party code. For example, “slow” serial diagnostic output of the packet parsing and response generation could mean that a significant amount of time passes, such that the MODBUS support no longer adheres to the timeout limits imposed by external code.

Testing

The configuration option CYGPKG_MODBUS_TESTS defines a set of tests that are built.

By default the package will only build the deterministic, automatic, tests. However, the option CYGPKG_MODBUS_TESTS_MANUAL can be defined to build extra tests that may required manual user-intervention, or are more realistic real-world example applications.

modbus_ut

The modbus_ut test application performs some unit-testing of the ModbusTCP server implementation. The test assumes specific features of the tests/backend_dummy.c backend implementation to perform a variety of fixed tests.

Since the backend handler funtions can *NEVER* block the test code will explicitly test pending responses to simulate application situations where hardware responses may take arbitrary periods of time, and hence are passed to other user-application threads for processing.

Note

When configured against the lwIP network stack the unit-testing relies on lwIP being configured with CYGFUN_LWIP_NETIF_LOOPBACK. This is required to allow the test code to connect to the ModbusTCP server running on the same network interface.

modbus_server

The modbus_server is only built when CYGPKG_MODBUS_TESTS_MANUAL is configured. It uses the tests/backend_dummy.c as the backend for a simple example server.

The application just runs for a fixed period of time before terminating, and can be used with external ModbusTCP clients.

Part XXXI. Direct Memory Access Controller (DMAC) Device Drivers

Documentation for drivers of this type may also be integrated into the eCos board support documentation. You should review the documentation for your target board for details. Standalone and more generic drivers are documented in the following sections.

Table of Contents

Chapter 113. Atmel DMA Controller (DMAC)

Name

DMAC — eCos Support for the Atmel DMA Controller

Synopsis

#include CYGBLD_DEV_DMA_H

ch = **atmel_dmac_chan_alloc**(*cyg_uint32 descriptor*, *atmel_dmac_callback *cb*, *CYG_ADDRWORD priv*);

void **atmel_dmac_chan_config**(*atmel_dmac_channel *ch*, *cyg_uint32 extended*);

void **atmel_dmac_chan_free**(*atmel_dmac_channel *ch*);

void **atmel_dmac_start**(*atmel_dmac_channel *ch*, *CYG_ADDRWORD src*, *CYG_ADDRWORD dst*, *cyg_uint32 size*);

void **atmel_dmac_stop**(*atmel_dmac_channel *ch*);

void **atmel_dmac_dma_poll**(*atmel_dmac_channel *ch*);

Description

This package provides access to the Atmel DMAC (DMA Controller) devices. This support is not intended to expose the full functionality of these devices. It is mainly limited to supporting peripheral DMA (e.g. USART, SPI, etc.). It is currently limited to single DMA transfers.

There is currently no cross-platform/variant standardised eCos DMA I/O interface package, since DMA features and functionality vary greatly between architectures, and event within variants of an architecture. This stand-alone device package allows common DMA support to be shared between devices that implement Atmel DMA Controllers.

The user is directed towards the relevant Atmel documentation for a full description of the DMAC devices, and to the variant device drivers for examples of the use of this API. This documentation only gives a brief description of the functions available.

A DMAC instance is defined by a controller number (0 or 1), and each controller has support for a number of (variant defined) channels. The API uses a simple 32-bit encoding to describe how a specific DMA channel should be used, with this package providing helper macros to combine the necessary information into a unique descriptor. These descriptors may be stored with a device driver as required.

The following are examples of how definitions can be made:

```
// USART0 TX on controller 0, 8-bit mem-to-peripheral transfers using AHB_IF2
#define AT91_SERIAL0_DMA_TX      CYGHWR_ATMEL_DMA_M2P(0,USART0_TX,8,IF2)

// USART0 RX on controller 0, 8-bit peripheral-to-mem transfers using AHB_IF2
#define AT91_SERIAL0_DMA_RX      CYGHWR_ATMEL_DMA_P2M(0,USART0_RX,8,IF2)
```

Before DMA transfers can be performed, a DMA channel must be claimed. This is done by calling `atmel_dmac_chan_alloc()`. The *`descriptor`* argument describes the majority of the DMA transfer configuration that will be used. As shown in the examples above the passed descriptor not only encodes the source and destination interfaces, but also the transfer sizes. Also, depending on the descriptor construction macros used, it is possible to control the direction and modification of addresses during transfers. The *`cb`* argument is used to register a client function that will be called when a requested transfer completes. The *`priv`* argument is a client specified value that will be passed to the callback function, and can be used to reference client driver specific data.

If DMA chunk transfers of more than one item per transaction are required then an “extension” 32-bit configuration descriptor can be specifed using the `atmel_dmac_chan_config()` function. The extended descriptor allows for non-default FIFO configurations and transfer chunk sizes to be specified. The `<cyg/hal/sama5d3.h>` header file contains examples of defining extended descriptors. For example, see the `ATMEL_AES_DMA_TX_EXT` manifest.

Most drivers will allocate a DMA channel object and keep it active throughout the system lifetime. However, if it is necessary to share a channel, or otherwise disable the use of a stream, the driver may call `atmel_dmac_chan_free()` to return a channel to an unused state. It will be necessary to call `atmel_dmac_chan_alloc()` before specific DMA descriptor operations can be performed again.

The register callback function has the following prototype:

```
typedef void atmel_dmac_callback( atmel_dmac_channel  *ch,
                                  cyg_uint32          cbid,
                                  cyg_uint32          count,
                                  CYG_ADDRWORD        data );
```

The *`ch`* is the channel structure describing the transfer. The *`cbid`* argument is a completion identifier:

Table 113.1. Completion Codes

`CYGHWR_ATMEL_DMA_COMPLETE`	A valid transfer completion. The *`count`* argument should match the *`size`* passed to the `atmel_dmac_start()` call.
`CYGHWR_ATMEL_DMA_AHBERR`	This code indicates that the DMA Controller has detected an AHB read or write access error. This may indicate invalid memory addresses have been passed, or invalid AHB_IF mappings have been used.
`CYGHWR_ATMEL_DMA_DICERR`	For configurations where Descriptor Integrity Check support is available, and enabled, then if an error is detected in a referenced memory-based transfer structure this result will be raised.

The *`count`* argument is the number of data items successfully transferred. The *`data`* argument is the client private data registered for the callback.

A transfer is configured and started by calling `atmel_dmac_start()`. The *`ch`* argument describes the DMA channel, with the descriptor used when allocating the channel defining how the other arguments are used. The *`src`* argument defines the peripheral or memory address from which the transfer will be made. The *`dst`* argument supplies the peripheral or memory address to which the transfer will write. The *`size`* argument defines the number of data items to be transferred. Once this function call completes the channel is operational and will transfer data once the relevant peripheral starts triggering transfers.

When the transfer completes the registered callback is called from DSR mode.

Notes:

1. Since the callback function is executed as a DSR, only a subset of eCos operations are valid.
2. It is expected that the client driver will perform any necessary CACHE operations within either its supplied callback handler functions, or *before* calling `atmel_dmac_start()` as required.

Chapter 114. Atmel DMA Controller (XDMAC)

Name

XDMAC — eCos Support for the Atmel XDMAC Controller

Synopsis

#include CYGBLD_DEV_DMA_H

`ch = `**`atmel_dmac_chan_alloc`**`(`*`cyg_uint32 descriptor`*`, `*`atmel_dmac_callback *cb`*`, `*`CYG_ADDRWORD priv`*`);`

`void `**`atmel_dmac_chan_config`**`(`*`atmel_dmac_channel *ch`*`, `*`cyg_uint32 extended`*`);`

`void `**`atmel_dmac_chan_free`**`(`*`atmel_dmac_channel *ch`*`);`

`void `**`atmel_dmac_start`**`(`*`atmel_dmac_channel *ch`*`, `*`CYG_ADDRWORD src`*`, `*`CYG_ADDRWORD dst`*`, `*`cyg_uint32 size`*`);`

`void `**`atmel_dmac_stop`**`(`*`atmel_dmac_channel *ch`*`);`

`void `**`atmel_dmac_dma_poll`**`(`*`atmel_dmac_channel *ch`*`);`

Description

This package provides access to the Atmel XDMAC (Extended DMA Controller) devices. This support is not intended to expose the full functionality of these devices. It is mainly limited to supporting peripheral DMA (e.g. USART, SPI, etc.). It is currently limited to single DMA transfers.

There is currently no cross-platform/variant standardised eCos DMA I/O interface package, since DMA features and functionality vary greatly between architectures, and event within variants of an architecture. This stand-alone device package allows common DMA support to be shared between devices that implement Atmel DMA Controllers.

The user is directed towards the relevant Atmel documentation for a full description of the XDMAC devices, and to the variant device drivers for examples of the use of this API. This documentation only gives a brief description of the functions available.

The API of this controller is designed to be compatible with that for the Atmel DMAC controller and is a drop-in replacement for it. Thus the API refers to the DMAC in its naming, not the XDMAC.

A XDMAC instance is defined by a controller number (0 or 1), and each controller has support for a number of (variant defined) channels. The API uses a simple 32-bit encoding to describe how a specific DMA channel should be used, with this package providing helper macros to combine the necessary information into a unique descriptor. These descriptors may be stored with a device driver as required.

The following are examples of how definitions can be made:

```
// USART0 TX on controller 0, 8-bit mem-to-peripheral transfers using default memory interfaces
#define AT91_SERIAL0_DMA_TX      CYGHWR_ATMEL_DMA_M2P(0,USART0_TX,8)

// USART0 RX on controller 0, 8-bit peripheral-to-mem transfers using default memory interfaces
#define AT91_SERIAL0_DMA_RX      CYGHWR_ATMEL_DMA_P2M(0,USART0_RX,8)
```

Before DMA transfers can be performed, a DMA channel must be claimed. This is done by calling `atmel_dmac_chan_alloc()`. The *`descriptor`* argument describes the majority of the DMA transfer configuration that will be used. As shown in the examples above the passed descriptor not only encodes the source and destination interfaces, but also the transfer sizes. Also, depending on the descriptor construction macros used, it is possible to control the direction and modification of addresses during transfers. The *`cb`* argument is used to register a client function that will be called when a requested transfer completes. The *`priv`* argument is a client specified value that will be passed to the callback function, and can be used to reference client driver specific data.

The `atmel_dmac_chan_config()` function is present for compatibility with the DMAC driver. It is not currently needed, but device drivers that may use both drivers may call this with not effect.

Most drivers will allocate a DMA channel object and keep it active throughout the system lifetime. However, if it is necessary to share a channel, or otherwise disable the use of a stream, the driver may call `atmel_dmac_chan_free()` to return a channel to an unused state. It will be necessary to call `atmel_dmac_chan_alloc()` before specific DMA descriptor operations can be performed again.

The register callback function has the following prototype:

```
typedef void atmel_dmac_callback( atmel_dmac_channel  *ch,
                                  cyg_uint32          cbid,
                                  cyg_uint32          count,
                                  CYG_ADDRWORD        data );
```

The *`ch`* is the channel structure describing the transfer. The *`cbid`* argument is a completion identifier:

Table 114.1. Completion Codes

`CYGHWR_ATMEL_DMA_COMPLETE`	A valid transfer completion. The *`count`* argument should match the *`size`* passed to the `atmel_dmac_start()` call.
`CYGHWR_ATMEL_DMA_AHBERR`	This code indicates that the DMA Controller has detected an AHB read or write access error. This may indicate invalid memory addresses have been passed, or invalid AHB_IF mappings have been used.
`CYGHWR_ATMEL_DMA_DICERR`	For configurations where Descriptor Integrity Check support is available, and enabled, then if an error is detected in a referenced memory-based transfer structure this result will be raised.

The *`count`* argument is the number of data items successfully transferred. The *`data`* argument is the client private data registered for the callback.

A transfer is configured and started by calling `atmel_dmac_start()`. The *`ch`* argument describes the DMA channel, with the descriptor used when allocating the channel defining how the other arguments are used. The *`src`* argument defines the peripheral or memory address from which the transfer will be made. The *`dst`* argument supplies the peripheral or memory address to which the transfer will write. The *`size`* argument defines the number of data items to be transferred. Once this function call completes the channel is operational and will transfer data once the relevant peripheral starts triggering transfers.

When the transfer completes the registered callback is called from DSR mode.

Notes:

1. Since the callback function is executed as a DSR, only a subset of eCos operations are valid.
2. It is expected that the client driver will perform any necessary CACHE operations within either its supplied callback handler functions, or *before* calling `atmel_dmac_start()` as required.

Part XXXII. RPMSG Support

Name

Overview — eCosPro Support for RPMSG

Description

This package provides support for RPMSG communications between separate CPU, microcontrollers or virtual machines. It provides an implementation of the OpenAMP RPMSG API for application use as well as an interface for hardware drivers of communication devices.

At present this package is mainly oriented to using RPMSG under the Xvisor hypervisor. As such it takes certain short cuts and does not support all RPMSG features.

Name

API — Functions

Synopsis

#include <cyg/io/rpmsg.h>

```
struct rpmsg_device *cyg_rpmsg_open(char *devname);

int cyg_rpmsg_close(struct rpmsg_device *dev);

void cyg_rpmsg_poll(void);

typedef int (*rpmsg_ept_cb)(struct rpmsg_endpoint *ept, const void *data, int len,
uint32_t src, void *priv);

typedef int (*rpmsg_ns_unbind_cb)(struct rpmsg_endpoint *ept);

int rpmsg_create_ept(struct rpmsg_endpoint *ept, struct rpmsg_device *dev, const char
*name, uint32_t src, uint32_t dst, rpmsg_ept_cb cb, rpmsg_ns_unbind_cb ns_unbind_cb);

void rpmsg_destroy_ept(struct rpmsg_endpoint *ept);

unsigned int is_rpmsg_ept_ready(struct rpmsg_endpoint *ept);

int rpmsg_send(struct rpmsg_endpoint *ept, const void *data, int len);

int rpmsg_sendto(struct rpmsg_endpoint *ept, const void *data, int len, uint32_t dst);

int rpmsg_send_offchannel(struct rpmsg_endpoint *ept, uint32_t src, uint32_t dst, const
void *data, int len);

int rpmsg_trysend(struct rpmsg_endpoint *ept, const void *data, int len);

int rpmsg_trysendto(struct rpmsg_endpoint *ept, const void *data, int len, uint32_t dst);

int rpmsg_trysend_offchannel(struct rpmsg_endpoint *ept, uint32_t src, uint32_t dst,
const void *data, int len);

int rpmsg_send_offchannel_raw(struct rpmsg_endpoint *ept, uint32_t src, uint32_t dst,
const void *data, int len, int wait);
```

Description

The RPMSG API follows the standard API in most details. The main difference is that currently it is oriented towards supporting the Xvisor VMSG infrastructure, which differs from the standard in some ways. As a result the name service functionality is not supported, and there are some eCos specific extensions.

The eCos extensions provide a mechanism for managing RPMSG devices. The function `cyg_rpmsg_open()` looks for an RPMSG device with the given name and returns a pointer to it. The function `cyg_rpmsg_close()` closes a device down when it is no longer needed. The function `cyg_rpmsg_poll()` is used in systems where interrupts are disabled to process RPMSG transfers; where interrupts are enabled the RPMSG package calls this function internally and the application does not need to do it itself.

The remaining API functions follow the OpenAMP API with the exception of the name server functionality.

The function `rpmsg_create_ept()` initializes an application-supplied endpoint object with a name, addresses and callbacks; any messages directed to the given destination address will cause the callback on this endpoint to be called. An RPMSG client should create an endpoint to communicate with a remote node. The client should provide at least a channel name and a callback for message notification. By default the endpoint source address should be set to RPMSG_ADDR_ANY.

The function `rpmsg_destroy_ept()` unregisters the endpoint from the device.

The function `is_rpmsg_ept_ready()` returns 1 if the endpoint has both local and destination addresses set, 0 otherwise.

The function `rpmsg_send()` sends *`data`* of length *`len`* based on the endpoint *`ept`*. The message will be sent to the remote node which the channel belongs to, using the endpoint's source and destination addresses. If there are no TX buffers available, the function will block until one becomes available, or a timeout of 15 seconds elapses. When the latter happens, `RPMSG_ERR_NO_BUFF` is returned. The function returns the number of bytes sent if successful, or a negative error code on failure.

The function `rpmsg_sendto()` functions in the same way as `rpmsg_send()` except that the destination address is taken from the *`dst`* argument and not the endpoint.

The function `rpmsg_send_offchannel()` functions in the same way as `rpmsg_send()` except that both the source and destination addresses are taken from the *`src`* and *`dst`* argument and not the endpoint.

The functions `rpmsg_trysend()`, `rpmsg_trysendto()` and `rpmsg_trysend_offchannel()` function in the same way as their non-try equivalents except that when no buffers are available they return immediately with `RPMSG_ERR_NO_BUFF`.

The function `rpmsg_send_offchannel_raw()` allows all options supported by the previous functions to be specified as arguments.

When a message is received for an endpoint it is passed to the application by calling the callback function registered when the endpoint was created. The endpoint, data pointer and length and the source endpoint address are passed in. The callback is made from a thread that is part of the RPMSG subsystem. The callback is permitted to call any functions that are allowed from a thread but should avoid doing anything that might take a long time since it would block processing of new messages for other endpoints.

The following code shows an example of the use of this API. For clarity, error checking code is omitted.

```
void rpmsg_test( void )
{
    struct rpmsg_device *rdev = NULL;
    struct rpmsg_endpoint ept;
    int err;
    int i;
    uint32_t src = 0x501;
    uint32_t dst = 0x502;

    // Locate device "rpmsg0"
    rdev = cyg_rpmsg_open( "rpmsg0" );

    // Create an endpoint for the given src and dst pair
    err = rpmsg_create_ept( &ept, rdev,
                            "ecos-ept0", src, dst,
                            rpmsg_ept_cb, NULL );

    // Send some messages, we assume the peer will
    // reply in some way.
    for( i = 0; i < 100; i++ )
    {
        char msg[40];

        int len = diag_sprintf( msg, "%3d 1234567", i );

        err = rpmsg_send( &ept, msg, len );

        diag_printf("SENT: %4d %04lx %s\n", len, dst, (char *)msg);
    }

    // Close the device when we are finished
    cyg_rpmsg_close(rdev);
}

// Receive callback
int rpmsg_ept_cb(struct rpmsg_endpoint *ept, void *data,
                 size_t len, uint32_t src, void *priv)
{
    // Handle received message

    diag_printf("GOT : %4d %04lx %s\n", len, src, data);

    return RPMSG_SUCCESS;
}
```

Part XXXIII. Serial Device Drivers

Documentation for drivers of this type is often integrated into the eCos board support documentation. You should review the documentation for your target board for details. Standalone and more generic drivers are documented in the following sections.

Table of Contents

Chapter 115. Freescale MCFxxxx Serial Driver

Name

CYGPKG_DEVS_SERIAL_MCFxxxx — eCos Support for the MCFxxxx On-chip Serial Devices

Description

All members of the Freescale MCFxxxx ColdFire family of processors contain a number of on-chip UARTs for serial communication. They all use very similar hardware. There are some variations such as different fifo sizes, and some processors contain extra functionality such as autobaud detection, but a single eCos device driver can cope with most of these differences. The CYGPKG_DEVS_SERIAL_MCFxxxx package provides this driver. It will use definitions provided by the variant HAL CYGPKG_HAL_M68K_MCFxxxx, the processor HAL and the platform HAL.

The driver provides partial support for hardware flow control and for serial line status. Only CTS/RTS hardware flow control is supported since the UART does not provide DTR/DSR lines. Similarly only line breaks, and certain communication errors are supported for line status since the UART does not provide other lines such as DCD or RI. On some platforms it should be possible to emulate these lines using GPIO pins, but currently there is no support for this.

Once application code accesses a UART through the serial driver, for example by opening a device /dev/ser0, the driver assumes that it has sole access to the hardware. This means that the UART should not be used for any other purpose, for example HAL diagnostics or gdb debug traffic. Instead such traffic has to go via another communication channel such as ethernet.

Configuration Options

The MCFxxxx serial driver should be loaded automatically when selecting a platform containing a suitable processor, and it should never be necessary to load it explicitly. The driver as a whole is inactive unless the generic serial support, CYGPKG_IO_SERIAL_DEVICES, is enabled. Exactly which UART or UARTs are accessible on a given platform is determined by the platform because even if the processor contains a UART the platform may not provide a connector. Support for a given UART, say uart0, is controlled by a configuration option CYGPKG_DEVS_SERIAL_MCFxxxx_SERIAL0. The device driver configuration option in turn depends on a HAL configuration option CYGHWR_HAL_M68K_MCFxxxx_UART0 to indicate that the UART is actually present and connected on the target hardware. If a given UART is of no interest to an application developer then it is possible to save some memory by disabling this option.

For every enabled UART there are a set of configuration options. The following use SERIAL0 as an example, though each UART device available will have its own unique SERIAL*n* naming:

CYGDAT_DEVS_SERIAL_MCFxxxx_SERIAL0_NAME

> Each serial device should have a unique name so that application code can open it. The default device names are /dev/ser0, /dev/ser1, and so on. It is only necessary to change these if the platform contains additional off-chip UARTs with clashing names.

CYGNUM_DEVS_SERIAL_MCFxxxx_SERIAL0_ISR_PRIORITY

> By default the driver arranges for the UARTs to interrupt at a low interrupt priority. Usually there will be no need to change this because the driver does not actually do very much processing at ISR level, and anyway UARTs are not especially fast devices so do not require immediate attention. On some Coldfires with MCF5282-compatible interrupt controllers care has to be taken that all interrupt priorities are unique.

CYGNUM_DEVS_SERIAL_MCFxxxx_SERIAL0_BAUD

> Each UART will be initialized to a given baud rate. The default baud rate is 38400 because in most scenarios this is fast enough yet does not suffer from excess data corruption. Lower baud rates can be used if the application will operate in an electrically noisy environment, or higher baud rates up to 230400 can be used if 38400 does not provide sufficient throughput.

CYGNUM_DEVS_SERIAL_MCFxxxx_SERIAL0_BUFSIZE

The serial driver will maintain software buffers for incoming and outgoing data. The former allows data to continue to arrive even if the application is still busy processing the previous transfer, and thus potentially improves throughput. The latter allows the application to transmit data without immediately blocking until the transfer is complete, often eliminating the need for a separate thread. The size of these buffers can be controlled via this configuration option, or alternatively these buffers can be disabled completely to save memory.

CYGNUM_DEVS_SERIAL_MCFxxxx_SERIAL0_ISR_BUFSIZE

If the serial driver has been configured with buffering (non-zero CYGNUM_DEVS_SERIAL_MCFxxxx_SERIAL*n*_BUFSIZE option) then this RX ISR specific option allows extra ISR buffering to be used to minimise the chances of dropped characters at high baud rates. The use of this extra ISR->DSR buffer reduces the number of DSR calls needed to pass data to the higher I/O layer whilst also minimising the latency in processing individual received characters.

Note

If debugging using RedBoot “GDB over Ethernet”, due to the increased ISR and DSR latency from the debugging support, the I/O and ISR buffers may need to be very large to minimise the chance of dropped characters at high baud rates (e.g. 230400). For example the CYGPKG_IO_SERIAL test serial3 (against the hosted ser_filter) can, when debugging via BDM (i.e. no RedBoot) and configured with diagnostics at 115200 using UART0, run the test at 230400 using UART1 successfully with CYGNUM_DEVS_SERIAL_MCFxxxx_SERIAL1_BUFSIZE of 128-bytes and a CYGNUM_DEVS_SERIAL_MCFxxxx_SERIAL1_ISR_BUFSIZE of 32-bytes. However, if RedBoot “GDB over Ethernet” debugging is used, with diagnostics over the Ethernet/RedBoot channel, then the test running at 230400 over UART1 will complete without dropped characters with CYGNUM_DEVS_SERIAL_MCFxxxx_SERIAL1_BUFSIZE of 8192-bytes and a CYGNUM_DEVS_SERIAL_MCFxxxx_SERIAL1_ISR_BUFSIZE of 4096-bytes. If even larger (e.g. 64K) continuous 230400 transfers are required then those buffers need to be further increased. This is an unfortunate side-effect of the ISR/DSR latencies introduced by RedBoot Ethernet based debugging.

Careful application specific tuning of the buffer sizes, and potentially the CYGNUM_DEVS_SERIAL_MCFxxxx_SERIAL*n*_ISR_PRIORITY, may be needed to ensure the desired baud rate and bandwidth requirements are met. If the ISR latency is too high, or the application thread reading data from the serial I/O layer cannot empty the CYGNUM_DEVS_SERIAL_MCFxxxx_SERIAL*n*_BUFSIZE quickly enough, then characters can still be dropped even with large buffers.

It is important that any thread priorities and/or scheduling requirements match the desired UART device use, and that the higher-level buffering and DMA receive buffers are tuned to match the worse-case load.

If the processor HAL is capable of supporting eDMA and the specific UART device is configured to use buffering, by setting a suitable CYGNUM_DEVS_SERIAL_MCFxxxx_SERIAL*n*_BUFSIZE value, then DMA specific options are made available:

CYGPKG_DEVS_SERIAL_MCFxxxx_SERIAL0_EDMA

This option, when enabled, allows eDMA support for the UART device to be configured via the following configuration options:

CYGNUM_DEVS_SERIAL_MCFxxxx_SERIAL0_EDMA_TX

If I/O buffering is configured then this option allows eDMA to be used for transmissions. This can greatly reduce the number of ISR and DSR operations needed to support transmission, reducing the CPU load needed.

CYGNUM_DEVS_SERIAL_MCFxxxx_SERIAL0_EDMA_PRIORITY

By default the driver arranges for the UART DMA to interrupt at the highest priority. For lower baud rates this is less critical, but when configured for high baud rates (e.g. 115200 or higher, and especially for ColdFire processors with UARTs with a shallow hardware FIFO) the aim is to minimise the interrupt latency for DMA buffer switch events.

Note

eDMA use for RX is not yet implemented.

There are additional options in the generic serial I/O package `CYGPKG_IO_SERIAL` which will affect this driver. For example `CYGPKG_IO_SERIAL_FLOW_CONTROL` and its sub-options determine what flow control mechanism (if any) should be used.

This package also defines some configuration options related to testing. Usually these options are of no interest to application developers and can be ignored.

Porting

The generic driver needs some information from other packages about the exact hardware, for example how many UARTs are available and where in memory they can be accessed.

1. Another package, usually the processor HAL, should provide one or more options `CYGHWR_HAL_M68K_MCFxxxx_UART0`, `CYGHWR_HAL_M68K_MCFxxxx_UART1` or `CYGHWR_HAL_M68K_MCFxxxx_UART2`. These may be calculated or user-configurable depending on the processor.
2. The device driver will also look for symbol definitions `CYGHWR_HAL_M68K_MCFxxxx_UART0_RTS` and `CYGHWR_HAL_M68K_MCFxxxx_UART0_CTS`, and the equivalents for the other UARTs, to determine whether or not these handshake lines are connected. These may be configuration options or they may be statically defined in a HAL I/O header file. The platform HAL should also implement the generic serial package's interface `CYGINT_IO_SERIAL_FLOW_CONTROL_HW` if appropriate.
3. If RTS is connected then the driver will also look for a symbol `CYGHWR_HAL_M68K_MCFxxxx_UART0_RS485_RTS`. This enables partial support for RS485 communication in that the device driver will arrange for the RTS line to be asserted during a transmit. The driver has no support for more advanced RS485 functionality such as multidrop.

In addition the driver assumes the standard MCFxxxx HAL macros are defined for the UART base addresses and the registers. The driver primarily targets MCF5282-compatible UARTs but there is also some support for functionality available on other members of the Coldfire range, for example the MCF5272's fractional baud rate support.

The HAL can optionally define the `HAL_PLF_DEVS_SERIAL_MCFxxx_XMIT_HOOK`. macro to provide code called to indicate when the driver transmissions are started and stopped. The macro is passed the base H/W address for the device in use (to identify the port when multiple serial interfaces are active) as well as a boolean indicating the start (`true`) and stop (`false`) status for TX.

Chapter 116. NXP PNX8310 Serial Driver

Name

`CYGPKG_DEVS_SERIAL_MIPS_PNX8310` — eCos Support for the PNX8310 On-chip UARTs

Description

The PNX8310 comes with two on-chip UARTs for serial communication. Other PNX83xx processors come with compatible devices. The `CYGPKG_DEVS_SERIAL_MIPS_PNX8310` package provides an eCos serial device driver. It can support up to three UARTs, depending on the processor and on which of the UARTs are connected on any given platform. If the CTS and RTS lines are connected then hardware flow control is supported. Line status is supported for line breaks and for certain communication errors. The UARTs do not have any support for DTR, DSR, DCD or RI lines. On some platforms these lines may be emulated using GPIO pins, but the driver does not currently have any support for this.

Once application code accesses a UART through the serial driver, for example by opening a device `/dev/ser0`, the driver assumes that it has sole access to the hardware. This means that the UART should not be used for any other purpose, for example HAL diagnostics or gdb debug traffic. Instead such traffic has to go via another communication channel such as ethernet.

Configuration Options

The PNX8310 serial driver should be loaded automatically when selecting a platform containing a suitable processor, and it should never be necessary to load it explicitly. The driver as a whole is inactive unless the generic serial support, `CYGPKG_IO_SERIAL_DEVICES`, is enabled. Exactly which UART or UARTs are accessible on a given platform is determined by the platform because even if the processor contains a UART the platform may not provide a connector. Support for a given UART, say uart0, is controlled by a configuration option `CYGPKG_DEVS_SERIAL_PNX8310_SERIAL0`, which will be active only if the platform enables the device. If a given UART is of no interest to an application developer then it is possible to save some memory by disabling this option.

For every enabled UART there are a further four configuration options:

`CYGDAT_DEVS_SERIAL_PNX8310_SERIAL0_NAME`

> Each serial device should have a unique name so that application code can open it. The default device names are `/dev/ser0`, `/dev/ser1`, and so on. It is only necessary to change these if the platform contains additional off-chip UARTs with clashing names.

`CYGNUM_DEVS_SERIAL_PNX8310_SERIAL0_ISR_PRIORITY`

> By default the driver arranges for the UARTs to interrupt at a low interrupt priority. Usually there will be no need to change this because the driver does not actually do very much processing at ISR level, and anyway UARTs are not especially fast devices so do not require immediate attention.

`CYGNUM_DEVS_SERIAL_PNX8310_SERIAL0_BAUD`

> Each UART will be initialized to a given baud rate. The default baud rate is 38400 because in most scenarios this is fast enough yet does not suffer from excess data corruption. Lower baud rates can be used if the application will operate in an electrically noisy environment, or higher baud rates up to 230400 can be used if 38400 does not provide sufficient throughput.

`CYGNUM_DEVS_SERIAL_PNX8310_SERIAL0_BUFSIZE`

> The serial driver will maintain software buffers for incoming and outgoing data. The former allows data to continue to arrive even if the application is still busy processing the previous transfer, and thus potentially improves throughput. The latter allows the application to transmit data without immediately blocking until the transfer is complete, often eliminating the need for a separate thread. The size of these buffers can be controlled via this configuration option, or alternatively these buffers can be disabled completely to save memory.

There are additional options in the generic serial I/O package CYGPKG_IO_SERIAL which will affect this driver. For example CYGPKG_IO_SERIAL_FLOW_CONTROL and its sub-options determine what flow control mechanism (if any) should be used.

This package also defines some configuration options related to testing. Usually these options are of no interest to application developers and can be ignored.

Porting

The generic driver needs some information from other packages about the exact hardware, for example how many UARTs are available and whether or not they are connected.

1. Another package, usually the platform HAL, should implement one or more of the interfaces CYGINT_DEVS_SERIAL_PNX8310_UART0, CYGINT_DEVS_SERIAL_PNX8310_UART1, or CYGINT_DEVS_SERIAL_PNX8310_UART2. Typically this is left to the platform HAL because even if the processor contains the UART device it may not be accessible on a given platform because there is no suitable connector.

2. If the RTS and CTS are connected for a given UART then the platform HAL should also implement the appropriate interface, for example CYGINT_DEVS_SERIAL_PNX8310_UART0_RS232_RTSCTS. This will enable driver support for hardware handshaking.

3. If a given UART is supported then the generic driver will need to know where it is mapped in the address space. Typically this is handled by the processor or variant HAL package via a definition HAL_PNX8310_UART0_BASE in cyg/hal/proc_io.h or cyg/hal/var_io.h.

4. On some platforms or processors additional initialization may be needed, for example to connect certain pins to the internal UART rather than to other on-chip devices. The processor or platform HAL can define a macro HAL_PNX8310_UART0_PROC_INIT for this purpose.

Chapter 117. Nios II Avalon UART Serial Driver

Name

CYGPKG_DEVS_SERIAL_NIOS2_AVALON_UART — eCos Serial Driver for Nios II Avalon UARTs

Description

A Nios II hardware design can include one or more Avalon uart devices for serial communication. On typical hardware each uart requires an external transceiver chip on the board to convert between FGPA and RS232 voltage levels, so the actual number of uarts is more a property of the board than of the FPGA hardware design. CYGPKG_DEVS_SERIAL_NIOS2_AVALON_UART provides an RS232 serial device driver for up to eight Avalon uarts in the design. It should be noted that in typical eCos configurations the first uart will be used for the HAL diagnostics and debug channel, either directly or via virtual vector calls to the RedBoot ROM monitor, so that uart should not be accessed via the serial driver.

Configuration Options

The Nios II Avalon uart serial driver should be loaded automatically when creating an eCos configuration for a hardware design which includes a suitable device, and it should never be necessary to load the package explicitly. The driver as a whole is inactive unless the generic serial support CYGPKG_IO_SERIAL_DEVICES is enabled.

For each uart in the h/w design the driver package provides a component, for example, CYGPKG_DEVS_SERIAL_NIOS2_AVALON_UART0, allowing driver support for that uart to be enabled or disabled. When enabled this component contains a number of options related to that uart:

CYGDAT_DEVS_SERIAL_NIOS2_AVALON_UART0_NAME

> Each serial device should have a unique name so that application code can open it. The default names may be provided by the h/w design HAL, otherwise they are strings such as /dev/ser0 and /dev/ser1. Usually it is only necessary to change these names if the h/w design involves different types of uarts and hence multiple serial device drivers.

CYGDAT_DEVS_SERIAL_NIOS2_AVALON_UART0_BAUD

> If the h/w design supports variable baud rates then this option can be used to set the default baud rate. This may be different from the default set in the h/w design. It will be installed when the device is opened by application code. If the h/w design does not support variable baud rates then this option will be inactive.

CYGDAT_DEVS_SERIAL_NIOS2_AVALON_UART0_BUFSIZE

> The serial driver will maintain software buffers for incoming and outgoing data. The former allows data to continue to arrive even if the application is still busy processing the previous transfer, and thus potentially improves throughput. Also since Avalon uarts do not have fifos a software buffer reduces the risk of lost data. The latter allows the application to transmit data without immediately blocking until the transfer is complete, often eliminating the need for a separate thread. The size of these buffers can be controlled via this configuration option, or alternatively these buffers can be disabled completely to save memory.

In addition the package provides control over the compiler flags used to build the driver code and some support for serial testing. There are further options in the generic serial I/O package CYGPKG_IO_SERIAL which will affect this driver. For example CYGPKG_IO_SERIAL_FLOW_CONTROL and its sub-options determine what flow control mechanism (if any) should be used. Hardware flow control will only be available if the h/w design includes support for the RTS/CTS lines.

Porting

This serial driver package needs information from the h/w design HAL about the number of Avalon uarts and how they have been configured. Note that this information can also affect the implementation of the HAL diagnostics and debug channel in the Nios II architectural HAL package so the relevant CDL interfaces are provided there. Also note that in accordance with eCos conventions uart numbering starts with 0.

1. The CDL interface `CYGHWR_HAL_NIOS2_AVALON_UARTS` should be implemented once for every Avalon uart in the h/w design.

2. Avalon uarts can be designed with RTS/CTS support if desired. If a given uart has this support enabled then the h/w design HAL should implement the appropriate CDL interface, for example `CYGHWR_HAL_NIOS2_AVALON_UART0_RTSCTS`. In the absence of RTS/CTS support the serial driver will not attempt hardware flow control on that uart.

3. Avalon uarts can be designed with either a fixed or a variable baud rate. If the latter then the h/w design HAL should implement the appropriate CDL interface, for example `CYGHWR_HAL_NIOS2_AVALON_UART0_VARIABLE_BAUD`.

4. For each uart the h/w design HAL should define the base address, the interrupt vector, the uart's input clock, the default baud rate, the number of stop bits, the parity, and the word length. This information should come from the `cyg/hal/nios2_hwconfig.h` header.

5. Default device names and baud rates can be provided using configuration options such as `CYGDAT_DEVS_SERIAL_NIOS2_AVALON_UART0_DEFAULT_NAME` and `CYGNUM_DEVS_SERIAL_NIOS2_AVALON_UART0_DEFAULT_BAUD`.

Unlike more conventional uarts, with an Avalon uart the number of bits per word, the number of stop bits, and the parity support are all fixed when the hardware is designed. Any attempt to change these settings at run-time will fail. The baud rate may also be fixed in the h/w design, or run-time changes may be supported. If the latter then the baud rate will be constrained by the input clock and a 16-bit divisor register, so for example with a 100MHz input clock the lowest standard baud rate that can be supported is 1800.

The serial driver does not provide any support for DMA transfers so it ignores any end-of-packet register that may have been included in the h/w design. Only RS232 is supported, not RS485 networking. There is no support for the DTR, DSR, DCD or RI lines. If these are needed then the lines would have to be wired up to an Avalon GPIO unit and handled by application code.

Part XXXIV. USB Support

Name

Overview — eCosPro Support for USB

Description

Important

This eCosPro-USB Middleware package is STRICTLY LICENSED FOR NON-COMMERCIAL PURPOSES ONLY. It may not be used for Commercial purposes in full or in part in any format, including source code, binary code and object code format.

A Commercial eCosPro License version 3 (or above) which explicity includes this Middleware Package is required for Commercial use.

The eCosPro-USB package only provides USB functionality for embedded systems. Where USB connectivity between eCos-based USB devices and host systems is required, it is your responsibility to ensure that a suitable host operating system class driver is available. Common host operating systems such as Windows, Linux and OS X include support for several standard USB classes. Unfortunately the specific classes supported varies considerably between them. If a host operating system you wish to support does not provide the necessary class support, then a suitable driver will need to be sourced from a third party USB class provider.

All USB Vendor and Product IDs used within this stack are for testing purposes only. It is your responsibility to obtain and deploy valid identifiers for your product; information about this can be found at the following link: www.usb.org/developers/vendor . It is also your responsibility to undertake any compliance testing needed to use the USB logo. Information regarding the USB-IF compliance program can be found here: http://www.usb.org/developers/compliance/.

Class Support

The individual USB classes are documented in the sections of the eCosPro documentation most relevant to that functionality. For example, information regarding the configuration and use of the Host Mass Storage Class is organized together with other file system related documentation.

Table 5. USB class support

USB Class	eCos Package Name and Documentation Link	Description
Host Mass Storage Class (MSC)	Host mass storage driver package `(CYGPKG_DEVS_DISK_USBMS)`	Supports use of USB memory sticks, hard disk drives and similar storage devices. USB mass storage devices essentially provide a block level interface, which by convention, a FAT format file system is layered over.
Host Communication Device Class (CDC), Abstract Control Model (ACM) subclass	Host CDC ACM protocol driver package `(CYGPKG_IO_USB_CDC_ACM_HOST)`	This class enables USB to serial converters conforming to the USB-IF defined CDC ACM class to be connected to embedded hardware. This can be used to expand the number of serial port connections available and can also be used for straightforward interconnection of embedded hardware; where one side implements the USB host class, and the other the device (target) side support. Read the USB Serial Support section for an overview of general USB serial functionality.
Host FTDI serial adapter class	Host FTDI protocol driver package `(CYGPKG_IO_USB_FTDI)`	This class enables FTDI USB to serial adapters to be connected to the embedded hardware, expanding the number of serial port connections available. This is a proprietary standard defined and supported by FTDI.

USB Class	eCos Package Name and Documentation Link	Description
		Read the USB Serial Support section for an overview of general USB serial functionality.
Target Communication Device Class (CDC), Abstract Control Model (ACM) subclass	Target CDC ACM protocol driver package (`CYGPKG_IO_USB_CDC_ACM`)	This class enables simple, straightforward connectivity and communication between embedded hardware and host PC's, or other embedded hardware. It is a USB-IF defined class that is supported by all major host operating systems. Read the USB Serial Support section for an overview of general USB serial functionality.
Target Remote Network Driver Interface Specification (RNDIS) Class	Target RNDIS driver package (`CYGPKG_DEVS_ETH_USB_RNDIS`)	The RNDIS support provides network level connectivity to host PCs. Note that this does not provide physical network access, but rather enables generic networking level communications between the USB attached Host and Target devices. RNDIS is a proprietary Microsoft defined class and is particularly suited for use with Windows-based systems. Compatible drivers are also available for other major operating systems.
Target Communication Device Class (CDC), Ethernet Emulation Model (EEM) subclass	Target CDC EEM driver package (`CYGPKG_DEVS_ETH_USB_CDCEEM`)	CDC-EEM provides direct network level connectivity between embedded targets and host PCs. Note that this does not provide physical network access, but rather enables generic networking level communications between the USB attached Host and Target devices. It is a USB-IF defined USB standard and is supported natively by both Linux and Mac OS.

Stack Limitations

The eCosPro USB stack stack currently implements a subset of the complete USB functionality. It is intended that it will eventually fully support not just host and target sides of the USB protocol, but also On-The-Go functionality. The current state of the stack is as follows:

- OTG support is not implemented.
- The host stack implements only control and bulk transfers. Interrupt and Isochronous transfers are not currently implemented. Additionally, only fixed size transfers are supported, short transfers are not, and scatter/gather support is not implemented.
- Hub and root hub support is limited to those control requests directly needed by the stack. Hub connect state is polled, hub interrupt status transfers are not supported.
- There is a limited set of host controllers supported. Currently only OHCI and the Synopsys DWC_usb controller are supported.
- The target stack implements control, bulk and interrupt transfers. Isochronous transfers are not implemented. Limited support for short OUT transfers is implemented.
- A relatively limited set of peripheral controllers is currently supported. For example, the Synopsys DWC_usb controller, as used on the STM32 Cortex-M family and the Atmel UDPHS controller driver.
- There is no intrinsic support for composite devices. In target mode applications can define and service composite devices directly. In host mode class drivers should be able to extract the specific interfaces they need from a composite device, although this has never been tested.
- The stack can currently only function within eCos with the kernel present. Support for unthreaded, non-interrupting environments such as RedBoot has not been implemented.

Terminology

USB has its own terminology, some of which conflicts with similar terms used in eCos. Some terms are used several times for slightly different but related concepts. The following lists some of the terms used and how they relate to the eCosPro USB stack.

Device

A device is something that is plugged into a USB bus, for example a flash drive, a serial adaptor or an ethernet adaptor. When its attachment is detected it is given an address and descriptors are fetched. These will generally be referred to by the term *physical device*.

In the eCosPro USB host stack the term *device* is also used to refer to the data structure allocated in the host stack to represent and manage the physical device while it is attached.

Target

A target is the name used in the eCosPro USB stack for a peripheral device implemented by eCosPro.

Host Controller (HC)

A host controller is the hardware device that provides access to a USB bus. It typically controls the transfer of packets between the USB stack and devices. It usually also provides support for one or more external ports into which devices and hubs are plugged.

Host Controller Driver (HCD)

A host controller driver is the software component that controls the host controller hardware and provides a standard interface by which the rest of the USB stack accesses the bus.

Peripheral Controller (PC)

A peripheral controller is the hardware device that provides access to a USB bus in peripheral mode. It typically controls the transfer of packets between the USB stack and the USB host.

Peripheral Controller Driver (PCD)

A peripheral controller driver is the software component that controls the peripheral controller hardware and provides a standard interface by which the rest of the USB stack accesses the bus.

Hub

A hub is a USB bus component that splits the USB bus into a number of additional ports. A typical external hub has four downstream ports for connecting new devices and a single upstream port that connects to a downstream port of another hub or host controller.

A host controller also acts as a hub, known as a *root hub*. It typically contains one, two or more ports which form the external interface to the USB bus. These are usually controlled by a handful of registers in the controller and accessed by the HCD. The USB host stack usually virtualizes the root hub so that it can be controlled in exactly the same way as an external hub through the exchange of control messages.

Endpoint

In the USB specification an endpoint is the terminus of communication between the host and a device. All USB transfers are directed to a combination of device address and endpoint number. Endpoints are uni-directional, and each device can have up to 16 endpoints in each direction, except endpoint 0 which is bidirectional.

In the eCosPro USB stack, the term *endpoint* is often used to refer to data structures that represent and manage transfers between the host and the target. Endpoint data structures can exist in both the USB stack itself and in the HCD/PCD.

Transfer

All communication is handled through a `usb_tfr` object. Internally to an HCD/PCD, the term transfer may also refer to an internal data structure. There may be a many-to-one relationship between HCD/PCD transfers and `usb_tfr` objects.

Port

A port is the connection point for attached devices. Both external hubs and host controllers have downstream ports. A hub's port is the target of various control operations to enable/disable and reset the connected device.

Class Driver	A class driver is a software component that converts a USB device into a particular eCos device type. Typically it operates between two interfaces, acting as a client to the USB stack and providing an eCos device interface to higher level components.

To configure and use the USB stack support, you should mainly refer to the class specific documentation linked to above. The following sections of this document are intended for developers who wish to build a deeper understanding of how the USB stack functions, for example to add their own class support.

Host Support Overview

Host support comprises the USB stack itself, one or more Host Controller Drivers and one or more Class Drivers.

During initialization the USB stack calls the init routine of each HCD. Platform-specific code called by the HCD locates all the instances of that host controller and configures it for host usage. This usually involves supplying power and clocks to the HC, configuring external pin multiplexing and enabling any external PHY, charge pump or power switches. Each HC is then registered with the USB stack as a separate USB bus. The first thing that happens for each bus is that a new device object is created to represent the root hub for that bus. The root hub is a virtual USB hub implemented by the USB stack to control the downstream ports of the HC. The USB stack controls the downstream ports by calling functions in the HCD.

All hubs, including the root hubs, are polled on a regular basis by the USB stack to detect any new device connections. If a new connection is detected then a device object is allocated and initialized. This object is then driven through a state machine that resets the device, assigns an address to it and fetches its device and configuration descriptors. Once this is done the device type is inspected. If it is a hub then the device object is passed over to the hub state machine which fetches further descriptors and starts polling the downstream ports for new connections. If it is not a hub, then the list of registered class drivers is searched for a class driver that recognizes the device type.

Class drivers interact with the device by allocating and initializing USB transfer objects and passing them to the USB stack for execution. Each transfer object identifies the device it applies to together with the endpoint and transfer type. It also points to a data buffer to/from which the data will be transferred and for control messages also contains the SETUP packet. Transfers are passed into the USB stack and are processed asynchronously to the client; completion of a transfer is signalled by calling a callback function in the transfer. The transfer's status indicates whether this was a successful exchange of data, some error has occurred, or the device has been disconnected. At this point the class driver is free to release the transfer, or resubmit it.

Device disconnection usually results in any active transfers receiving an error. It is also noticed by the hub polling machine and results in the device being detached from its parent hub. A call is also made to the class driver, which should then cancel any active transfers, dismantle any device specific data structures and optionally signal the event to its clients in turn.

The existence of transfers and device objects is controlled by reference counts. For each device the presence of the physical device counts as a reference, as should its use by a class driver; each active transfer also counts as a reference. So, when a device is detached, the physical device reference will be removed, but it will only be returned to the free pool once all transfers have completed and the class driver has finished with it. Similarly, transfers are created with a single reference, but the HCD will take one or more references while the transfer is active in the controller. To release a transfer, the class driver simply needs to decrement the reference counter in the completion callback, which will return it to the pool immediately, or maybe a little later when the HCD has finished with it.

Target Support Overview

Target, or peripheral, support comprises the USB stack itself plus one or more Peripheral Controller Drivers. There is currently no equivalent of the host class driver mechanism for targets and instead peripheral functionality is provided by application code. See the `acm_example.c` source for an example; this can be found in the `packages/io/usb/<version>/tests` directory.

Initialization of the target USB stack is similar to that for the host stack as far as Peripheral Controller Drivers are concerned. The USB stack calls the init routine of each PCD. Platform-specific code called by the PCD locates all the instances of that peripheral controller and configures them for peripheral usage. This usually involves supplying power and clocks to the Peripheral Controller, configuring external pin multiplexing and enabling any external PHY, charge pump or power switches. For dual mode

OTG controllers, much of this can be shared with the host controller mode. Finally, the peripheral controller is registered with the USB stack under a unique name.

Application code instantiates a target by creating a target object and populating it with pointers to device, string and configuration descriptors. Most of this can be done statically, and apart from the target object, can be constant data stored in ROM or flash memory. The target object also contains pointers to a number of callback routines that the USB stack uses to signal events to the application. The target is activated by attaching it to the specific peripheral controller that controls the external USB socket on which the peripheral is to be presented.

The USB stack remains quiescent until the PCD detects that the peripheral port has been attached to a host controller. From this point the USB stack handles most of the initial interaction with the host as it resets the peripheral port, assigns an address and fetches descriptors. Most descriptors will be fetched from those statically defined in the target object, but if any are not found then a callback to the application is made which allows it to manufacture any special descriptors dynamically. Also during this process, callbacks will be made to indicate the progress of the peripheral through the USB peripheral state machine.

Once the target reaches `CONFIGURED` state it is ready for operation and application code can start exchanging messages with the host. This is done using exactly the same transfer objects as are used in host mode. Each transfer object identifies the endpoint, transfer type and data to be exchanged, and its completion is signalled by calling a callback function. Transfers are controlled by reference counts in the same way as for host support; since the target is statically allocated by the application it is not subject to reference count control in the same way as host device objects.

If the peripheral is detached then any active transfers will be cancelled and the target is returned to an earlier state. Reconnection will restart the configuration process and eventually return the target to `CONFIGURED` state. Application code should avoid submitting transfers unless the target is in `CONFIGURED` state.

Structure of this Document

This document is divided into a number of sections. Some of these are of interest to all users, others will only be useful to writers of class or host controller drivers. Other parts are only useful to those who are working on the USB stack itself:

- The Configuration section describes the CDL configuration options that can be applied to the USB stack.
- The Transfer Objects section describes the message-like transfer objects that are used for all data transfers in the USB stack, together with various API functions that apply to them.
- The Host Device Object section describes the data structure used to represent a device connected to the stack in host mode. This section also describes the lifecycle of a device and a hub.
- The Class Drivers section describes the interface to host device class drivers.
- The Host Controller Drivers section describes the interface to Host Controller Drivers.
- The Target Objects section describes the data structure used to represent a target device connected to the stack in peripheral mode. This section also describes the support for string descriptor encoding and the target object API together with a description of what an application should do to create a target peripheral device.
- The Peripheral Controller Drivers section describes the interface to Peripheral Controller Drivers.

Name

Configuration — USB System Configuration

Description

USB support consists of several packages: the main USB subsystem `CYGPKG_IO_USB`; at least one HCD or PCD package, for example `CYGPKG_DEVS_USB_OHCI` or `CYGPKG_DEVS_USB_PCD_DWC`; and a target specific configuration package, for example `CYGPKG_DEVS_USB_STM32`. The driver and configuration packages are usually part of the target specification in `ecos.db`. Once these packages are installed, there are a number of configuration options that may be set. Those for the controllers and configuration packages are described in their own documentation, those for the USB subsystem are described here.

To enable USB support in any configuration it should simply be necessary to add the `CYGPKG_IO_USB` package and then enable `CYGPKG_IO_USB_HOST` and/or `CYGPKG_IO_USB_TARGET` to provide the appropriate stack modes.

Options

The following configuration options control the behaviour of the USB subsystem.

cdl_component CYGPKG_IO_USB_HOST

> This component enables USB host support in the USB stack.

cdl_option CYGNUM_IO_USB_HOST_BUFFER_SIZE

> The USB subsystem uses a shared buffer for some purposes. This option defines the size of that buffer. Class drivers may require that this buffer be increased in size. The default size is 64 bytes, which is sufficient for the configurations of most devices.

cdl_component CYGSEM_IO_USB_HUB

> This option controls the level of HUB support. When enabled, full HUB support is provided, for both root and external HUBs. When disabled, only rudimentary root hub support is provided, external hubs are not supported, and all devices must be plugged in to the USB host device. This option is enabled by default.

cdl_option CYGNUM_IO_USB_HUB_POLL_INTERVAL

> This option defines the interval in milliseconds between polls of a hub's ports for device attach and detach events. This value effectively defines how responsive the system is to device attach/detach. The default value is 1000, causing the hubs to be polled once a second.

cdl_option CYGNUM_IO_USB_HUB_PORT_MAX

> This option defines the maximum number of ports allowed per hub. The default value is four since most external hubs have just four downstream ports; virtual root hubs seldom have more and usually have fewer.

cdl_component CYGPKG_IO_USB_TARGET

> This component enables USB target support in the USB stack.

cdl_option CYGNUM_IO_USB_MEMORY_TABLE_TARGET_ENDPOINT_SIZE

> Number of endpoints allocated for target use.

cdl_option CYGNUM_IO_USB_OS_THREAD_STACK_SIZE

> The USB subsystem uses an internal thread to handle asynchronous actions. This option defines the size of this stack in addition to CYGNUM_HAL_STACK_SIZE_TYPICAL. The default value is 4KiB.

cdl_option CYGNUM_IO_USB_OS_THREAD_PRIORITY

The USB subsystem uses an internal thread to handle asynchronous actions. This option defines the priority at which this thread is scheduled. The default value is set to 10, which makes the USB thread run at a medium high priority.

cdl_component CYGSEM_IO_USB_MEMORY_SYSTEM_HEAP

This option enables use of the system heap for all USB memory allocations. It is orthogonal to the `PRIVATE_HEAP` and `TABLES` options. This is the default memory allocation option.

cdl_option CYGNUM_IO_USB_MEMORY_SYSTEM_HEAP_LIMIT

This option sets a limit on the amount of memory allocated from the system heap. Once the USB stack has allocated this much, it will refuse further allocations until memory is freed. If this value is set to 0 (the default) the limit is not enforced.

cdl_component CYGSEM_IO_USB_MEMORY_PRIVATE_HEAP

This option enables use of a fixed size private heap for all USB memory allocations. It is orthogonal to the `SYSTEM_HEAP` and `TABLES` options.

cdl_option CYGNUM_IO_USB_MEMORY_PRIVATE_HEAP_SIZE

This option defines the size of the private heap. Once this heap is exhausted, no more memory will be allocated.

cdl_component CYGSEM_IO_USB_MEMORY_TABLES

This option enables use of fixed sized static tables for all USB memory allocations. It is orthogonal to the `SYSTEM_HEAP` and `PRIVATE_HEAP` options.

cdl_option CYGNUM_IO_USB_MEMORY_TABLE_TFR_SIZE

Number of transfer objects in table. This option defaults to 4.

cdl_option CYGNUM_IO_USB_MEMORY_TABLE_DEVICE_SIZE

Number of device objects in table. This option defaults to 4.

cdl_option CYGNUM_IO_USB_MEMORY_TABLE_ENDPOINT_SIZE

Number of endpoints allocated. The endpoint object table will be the device table size multiplied by this value plus the value of `CYGNUM_IO_USB_MEMORY_TABLE_TARGET_ENDPOINT_SIZE`. The default value is 3. Typically devices need an entry for endpoint zero plus input and output endpoints. If any class driver needs more it should require a larger value for this option.

cdl_option CYGNUM_IO_USB_MEMORY_TABLE_DESC_SIZE

Number of descriptor objects in table. This option defaults to five times the number of entries in the device table. Typically each device needs an entry each for one configuration, one interface and three endpoints. Class drivers can increase this value if necessary.

cdl_option CYGNUM_IO_USB_TRANSFER_POOL_SIZE

This option defines the number of transfer objects maintained in the free pool for fast allocation. When the memory allocation strategy is to use tables, this option is ignored and the pool is set to the same size as the transfer table, and all transfers are kept in the pool. The default value is 4.

cdl_option CYGDBG_IO_USB_DIAG

This option controls the inclusion of diagnostics in the USB stack. The exact set of diagnostic messages can be further controlled at runtime on a per-subsystem basis.

cdl_option CYGDBG_IO_USB_LOG

This option controls the inclusion of logging in the USB stack. This differs from diagnostics in that only major events like device attach/detach are logged.

cdl_option CYGDBG_IO_USB_STATISTICS

This option controls the inclusion of statistics gathering in the USB stack.

cdl_option CYGNUM_IO_USB_STATISTICS_INTERVAL

This option defines the interval between reporting USB stack statistics. The time is given in seconds. A value of 0 disables the regular reports and statistics will only be reported if the application calls `usb_statistics_log()`.

Name

Transfer Object — Structure and Interface

Synopsis

#include <cyg/io/usb.h>

`usb_tfr *`**`usb_device_tfr_alloc`**`(`*`dev`*`);`

`usb_tfr *`**`usb_target_tfr_alloc`**`(`*`tgt`*`);`

`void `**`usb_tfr_ref`**`(`*`tfr`*`);`

`void `**`usb_tfr_unref`**`(`*`tfr`*`);`

`void `**`usb_tfr_init`**`(`*`tfr`*`, `*`endpoint`*`, `*`attr`*`, `*`buffer`*`, `*`size`*`, `*`callback`*`, `*`callback_data`*`);`

`#define `**`usb_request_init`**`(`*`req`*`, `*`request_type`*`, `*`request`*`, `*`value`*`, `*`index`*`, `*`length`*`);`

`void `**`usb_tfr_control`**`(`*`tfr`*`, `*`endpoint`*`, `*`req`*`, `*`buffer`*`, `*`size`*`, `*`callback`*`, `*`callback_data`*`);`

`usb_tfr *`**`usb_control_tfr`**`(`*`dev`*`, `*`request_type`*`, `*`request`*`, `*`value`*`, `*`index`*`, `*`callback`*`, `*`callback_data`*`);`

`void `**`usb_tfr_bulk`**`(`*`tfr`*`, `*`endpoint`*`, `*`req`*`, `*`buffer`*`, `*`size`*`, `*`callback`*`, `*`callback_data`*`);`

`void `**`usb_tfr_submit`**`(`*`tfr`*`);`

`void `**`usb_tfr_cancel`**`(`*`tfr`*`);`

Description

Most interaction between clients of the USB stack and the stack itself happens through the use of transfer objects (or just transfers). These data structures are allocated and initialized by the client and passed to the USB stack. When the transfer completes, the client is notified via a callback, following which it may release or reuse the object for another transfer.

Transfer Class Object

A transfer has the following structure:

```
struct usb_tfr
{
    usb_node                    node;           // List/queue node

    union
    {
        usb_device              *dev;           // Host device
        usb_target              *tgt;           // Target device
    };

    usb_uint16                  refcount;       // Reference count
    usb_uint16                  endpoint;       // Endpoint address
    usb_uint8                   attr;           // Transfer type
    usb_uint8                   flags;          // Additional flags
    int                         status;         // Current/returned status

    void                        *tfr_buffer;    // Data buffer
    usb_uint16                  tfr_size;       // Size of transfer/buffer
    usb_uint16                  tfr_actual;     // Actual size transferred

    usb_tfr_callback            *callback[2];   // Completion callback stack
    void                        *callback_data; // Callback data

    // The following fields are used by the HCD or PCD
    void                        *hcd_endpoint;  // HCD endpoint private data
    usb_list                    hcd_list;       // HCD transfer list

    // Per-transfer-type fields.
    union
    {
        struct
        {
            usb_uint8           setup[8];       // Setup packet
        } control;
        struct
        {
        } bulk;
#ifdef USB_CONFIG_INTERRUPT
        struct
        {
        } interrupt;
#endif
#ifdef USB_CONFIG_ISOCHRONOUS
        struct
        {
            int                 start_frame;    // Start frame
            int                 interval;       // transfer interval
            int                 pkt_count;      // Number of packets
            usb_iso_packet_desc *iso_packet;    // packet descriptors
        } isochronous;
#endif
    };
};
```

The fields of the transfer are as follows:

node
: A list node, which is used to link this transfer into internal lists in the USB stack. It may also be used by the client when the object is in its possession, but should be unlinked whenever passed to `usb_tfr_submit()`.

dev
: A pointer to the host device object on which this transfer will operate. This is filled in when the transfer is allocated by `usb_device_tfr_alloc()`. This value should not be changed directly by the client since other fields in this object may depend on this field. The transfer also holds a reference to the device which may not be decremented properly if this field is changed. If the client needs to communicate with a different device, it should allocate a new transfer object.

dev
: A pointer to the target object on which this transfer will operate. This is filled in when the transfer is allocated by `usb_target_tfr_alloc()`. This value should not be changed directly by the client since other fields in this object may depend on this field. The transfer also holds a reference to the target which may not be decremented properly if this field is changed.

refcount
: Transfer reference count. This controls the existence of this transfer. It can be incremented with a call to `usb_tfr_ref()` and decremented with a call `usb_tfr_unref()`. This field is set to 1 when the transfer is allocated, and the reference count on the associated device or target is also incremented. If the refcount is decremented to zero then the device or target reference is decremented, and the transfer returned to the free pool.

endpoint
: This is the endpoint number and direction. This field has the same format as the *`bEndpointAddress`* field of an endpoint descriptor and is initialized from the descriptor.

attr

This is the endpoint attributes; it mainly defines the type of transfer: control, bulk, interrupt or isochronous. It may contain other information for some transfer types. This field is initialized from the *`bmAttributes`* field of an endpoint descriptor and has the same format.

flags

This field contains a number of flag bits that control the nature of the transfer. If `USB_TFR_FLAGS_TARGET` is set then this is a target mode transfer and the *`tgt`* field is is valid, otherwise it is a host mode transfer and the *`dev`* field is valid. The flags `USB_TFR_FLAGS_START` and `USB_TFR_FLAGS_END` allow transfers to be chained together to provide a scatter/gather facility; this is currently not implemented. The `USB_TFR_FLAGS_CALLBACK` flag indicates that this transfer's callback should be called when it is complete. The `USB_TFR_FLAGS_SHORT_OK` flag indicates that a short transfer should be treated as a success and not a failure; this is currently not implemented for host transfers, but is for target OUT transfers.

By default, a transfer is initialized with the `START`, `END` and `CALLBACK` flags set. Additionally a target mode transfer is initialized with the `TARGET` flag set.

status

This field contains the transfer status. While it is in the possession of the USB stack, this field may be used to record internal state transitions. When it is returned to the client via a callback, it will contain either `USB_OK` to indicate the transfer was successful, an error code, or a status code (e.g. `USB_TFR_CANCELLED`).

tfr_buffer

A pointer to a buffer containing the data to be transmitted, or where the received data should be places. There are no explicit alignment requirements on this buffer, but on some platforms this buffer may need to be synchronized to external memory or flushed from the data cache, so if it is not cache line aligned these operations may have a side-effect on other data.

tfr_size

The size of the data buffer, in bytes, and hence the size of the transfer. The transfer size is not limited by the maximum packet size of the addressed endpoint, the driver may split the transfer into a sequence of packets if necessary.

tfr_actual

When the transfer is complete, this will contain the number of bytes actually transferred. This should only differ from *`tfr_size`* if the `SHORT_OK` flag is set.

callback

When a transfer has completed, this is signalled to the client by calling a callback routine. Callbacks are managed as a stack, with the client callback as the lowest, last, callback. This mechanism allow the USB stack to interpose its own finalization processing for a transfer if required. This callback stacking is opaque to the client.

callback_data

This is a client-supplied data value that the client can supply to ensure continuity between the submitter and the callback; it is usually a pointer to some controlling data structure. The client's callback can retrieve this value from the transfer by accessing this field. While callbacks are stacked, the callback_data is not, internal callbacks only make use of the standard transfer fields.

hcd_endpoint

This field is for use by the HCD or PCD, and typically contains a pointer to the data structure in the driver that controls the endpoint to which this transfer is directed.

hcd_list

This field is for use by the HCD or PCD, and typically is the root of a chain of internal data structures that define the data transfer.

control

This sub-structure is part of an anonymous union that defines per-transfer-type fields consisting of this field and the following *`bulk`*, *`interrupt`* and *`isochronous`* fields. This structure contains the contents of the setup packet. In host mode normally the setup packet is not assigned directly, but is copied here by `usb_tfr_control()` after being initialized elsewhere with `usb_request_init()`.

bulk

This field contains bulk transfer control fields. It is currently empty.

interrupt

This field contains interrupt transfer control fields. It is currently empty and is only defined if interrupt transfer support is configured. It's contents may change significantly when interrupt transfer support is implemented.

isochronous This field contains isochronous transfer control fields. It is currently empty and is only defined if isochronous transfer support is configured. At present isochronous transfers are not supported. It's contents may change significantly when isochronous transfer support is implemented.

API

There are a number of API functions associated with the management of transfer objects.

Host mode clients should allocate a transfer by calling `usb_device_tfr_alloc()`. The result will be a pointer to a transfer which has been zeroed except that the *`node`* field will be initialized, the *`dev`* field will be set to the supplied device pointer and the *`refcount`* will be set to 1. Additionally, `usb_device_ref()` will have been called on the device. This function will return a NULL pointer if there are no transfers available for allocation.

Target mode clients should allocate a transfer by calling `usb_target_tfr_alloc()`. The result will be a pointer to a transfer which has been zeroed except that the *`node`* field will be initialized, the *`tgt`* field will be set to the supplied target pointer, the *`refcount`* will be set to 1 and `usb_target_ref()` will have been called on the target. The *`flags`* field will be initialized with the `USB_TFR_FLAGS_TARGET` flag. This function will return a NULL pointer if there are no transfers available for allocation.

If a client needs to take further reference to the transfer it can call `usb_tfr_ref()`. References are released by calling `usb_tfr_unref()`. When a client is finished with a transfer it should call `usb_tfr_unref()` a last time to release the initial reference. This may have the side-effect of calling `usb_device_unref()` or `usb_target_unref()` which may result in the device object being freed.

In general, a transfer is initialized by one of several routines to create a specific type of transfer. The most general initialization routine is `usb_tfr_init()` which sets up the transfer with the endpoint address and attributes, a data buffer, and a callback. This is the only initialization function that should be applied to target mode transfers, the remaining initialization functions only apply to host mode transfers.

Control transfer initialization is supported by a number of functions. The function `usb_tfr_control()` initializes a general control transfer. The *`endpoint`* argument is used to look up the endpoint in the device and initialize the *`endpoint`* and *`attr`* fields in the transfer. The *`req`* argument points to a completed USB request that will be copied into the *`setup`* buffer. The data buffer and callback are initialized too. The USB request can be initialized using the `usb_request_init()` macro. The first argument to this is the name of the request to initialize, not a pointer. The remaining arguments give values for the various fields; the 16 bit fields will potentially be byte swapped into little endian order.

The function `usb_control_tfr()` provided a higher level interface to create control transfers that do not transfer additional data. This is given the destination device plus values for the request type, request code, value and index field (but not the length), and the callback. Using these parameters, a transfer is allocated, a request buffer created and the transfer initialized with an endpoint address of zero. If a transfer cannot be allocated, a NULL pointer is returned.

A bulk transfer can be initialized using `usb_tfr_bulk()`. The *`endpoint`* argument is used to look up the endpoint in the device and initialize the *`endpoint`* and *`attr`* fields in the transfer. The buffer and callback are also initialized.

Interrupt and Isochronous transfers are not currently supported, but when they are similar functions to initialize those will be available.

A transfer is submitted to the USB stack by calling `usb_submit()`. This function performs some simple checks before passing the transfer on to the appropriate driver. If this function is passed a NULL transfer, it will return `USB_ERR_TFR_ALLOC`. The transfer initialization routines also return if a NULL transfer is passed to them. This allows detection and handling of transfer allocation failures in most cases to be deferred until this call, keeping error handling simpler.

A transfer can be cancelled by calling `usb_cancel()`. This may simply mark the transfer for cancellation, the transfer may not actually be cancelled until some time after this function returns. When the transfer is actually cancelled, its callback will be called with a status of `USB_TFR_CANCELLED`. However, if the transfer was already finished, or caused an error, the callback status may be `USB_OK` or an error code. Thus client code cannot rely on the transfer completing with a CANCELLED status; this call just ensures that the transfer will be returned to the client in some way. Note that it is not very useful to cancel host mode control or bulk transfers since they will usually be processed as soon as submitted and will be returned quickly; the client is unlikely to catch them in time.

Name

Host Device Object — Structure and Interface

Synopsis

#include <cyg/io/usb.h>

void **usb_device_ref**(*dev*);

void **usb_device_unref**(*dev*);

Description

Whenever a new device or hub is attached to a USB, a device object is created to represent it in the USB stack. Most users of the USB stack do not need to concern themselves with the contents of a device object, so most of the information here is for the use of USB stack developers.

Host Device Object

This structure does double duty for both standard devices and for hubs. The internal union separates out the role specific fields. This structure also has two typedef names, usb_device and usb_hub, which can be used interchangeably, but help keep track of which role the structure is currently being used in. A device object has the following structure:

```
struct usb_device
{
    usb_node                    node;           // Node in per-hub list
    usb_bus                     *bus;           // Controlling bus
    usb_hub                     *parent;        // Parent hub

    usb_uint32                  flags;          // Flags
    usb_uint8                   refcount;       // Reference counter
    usb_uint8                   id;             // Device ID
    usb_uint8                   port;           // Port on parent hub
    usb_uint8                   state;          // Current device state
    usb_uint8                   state_data;     // Associated data

    usb_device_descriptor       desc;           // Device descriptor
    usb_descriptor              *config;        // Current configuration
    usb_descriptor              *interface;     // Current interface
    usb_descriptor              *desc_chain;    // Chain of all descriptors

    usb_resource_client         res_client;     // Resource client

    void                        *hcd_priv;      // HCD private data

    usb_device_endpoint         *endpoints;     // List of active endpoints

    union
    {
        struct
        {
            union
            {
                // Used only during initialization
                usb_uint8               *buf;          // Descriptor read buffer
                usb_config_descriptor   *config;       // Config descriptor view of buf

                // Used only when state >= RUNNING
                usb_class_driver        *class_driver; // Attached class driver

            };
        } device;
        struct
        {
            int                 port_count;     // Number of downstream ports
            usb_hub_port_status port_status[USB_HUB_PORT_MAX+1];

            usb_uint8           *buf;           // Descriptor read buffer

            usb_list            devices;        // List of attached devices

            // Status tfr state
            usb_tfr             *status_tfr;    // Current status change tfr
            usb_uint8           status_buf[4];  // Status change buffer
            usb_descriptor      *intr_desc;     // Interrupt endpoint descriptor

            // TODO: Power allocation stuff

            // TODO: Transaction translator stuff
        } hub;
    };
};
```

The fields of the device are as follows:

node
A list node that is used to link this device into a list in the hub object to which this device is attached.

bus
A pointer to the object representing the bus to which this device is attached. This pointer provides access to the HCD that communicates with this device. The bus object also controls the allocation of device IDs.

parent
A pointer to the parent hub, the same hub in whose device list this device must be linked via the *node* field.

flags
Flag bits controlling aspects of this device. The `USB_DEVICE_FLAGS_HUB` indicates that this device is a hub, and `USB_DEVICE_FLAGS_HUB_ROOT` indicates that this is a root hub.

refcount
Device reference count. This is initialized to 1 when the device is first attached, representing a reference held by the physical device, and incremented for each active transfer for this device. Class drivers may also take their own references. When the physical device is detached the refcount is decremented, which should result in the device object being freed.

id
Device ID. Each device starts with an ID of zero while it is being configured. This field will be set to the allocated ID once the physical device has had its address set successfully.

port
This is the port number within the parent hub to which this device is attached.

state
Device state. During their lifetime devices pass through a set of different states. This field described what state the device is currently in.

state_data
Some device states need some additional data in addition to the state. That data is stored here.

desc
During initialization the USB stack will read the device device descriptor and store it here.

config
When the device has been configured, this will point to the descriptor for the configuration that has been set in the physical device.

interface
When the device has been configured, this will point to the descriptor for the interface that has been set in the physical device.

desc_chain
This points to a chain of usb_descriptor objects which contain all the configuration, interface and endpoint descriptors that have been read from the physical device. They are stored in a single linear chain with the descriptors for each configuration chained on to the end of the previous descriptor chain.

res_client
: This structure is used internally by the USB stack to enable devices to wait for resources such as memory, or to implement delays.

hcd_priv
: This is a pointer to private data defined by the HCD that controls the physical device. It is copied from the *hcd_priv* field of the usb_bus object from which the device attachment was detected.

endpoints
: This points to a chain of usb_device_endpoint objects which associate an endpoint descriptor with an HCD supplied pointer that implements that endpoint. Only the endpoints for the interface currently selected appear in this list, together with endpoint 0 for control packets.

device
: This sub-structure is part of an anonymous union that provides fields for either devices or hubs, depending on the *flags* field. At present this contains an anonymous union that contains either a pointer to a buffer used to read configurations, or during normal running a pointer to the class driver that is using this device.

hub
: This is the second element of the anonymous union and contains field used if this device is a hub. This contains a number of fields that are mainly used internally by the USB stack. Included are a count of the number of downstream ports the hub contains together with the most recent reported state of each and a list of the devices attached to this hub. As hub support evolves, this sub-structure will acquire further fields.

Device Lifecycle

From initial attachment through configuration, data transfer and final detachment, a device goes through a lifecycle in the USB stack. This section looks at this lifecycle.

When a physical device is attached to a port on a hub the state change is detected by the hub state machine (see Hub Lifecycle). This results in a device object being created and initialized. The *refcount* is set to 1, representing a reference held by the physical device.

A device runs through a state machine that is generally run in the callbacks of transfers, delays and resource allocations. Each state assesses the result of the previous operation, issues new transfers/delays/allocations, sets the next state and waits for completion. States are generally named for the operation for which they waiting to complete. The device moves through the following states:

NEW

> This is the initial state, the device is further initialized to have an endpoint for device 0 endpoint 0. A request is set up to wait for allocation of the shared configuration buffer.

BUFFER

> This state is entered when the shared configuration buffer has been allocated. This buffer allocation has two purposes. First, it provides us with a buffer large enough to read entire configurations into. Second, and more importantly, it serializes all device initializations, which is necessary before setting the device address. The device sends a control packet to the hub to clear the connect change bit. The *state_data* field is initialized to contain a pair of 4 bit counters which count the number of reset and port status retries that have been tried.

CLEAR_CONNECT

> Once the connect change bit has been cleared, the device sends a control command to the hub to reset the physical device. This puts the device into a state where it responds to commands sent to device ID 0.

RESETTING

> After the reset command has been sent, the device waits 200ms for the reset to complete.

RESET_DELAY

> After the delay, a control request is sent to the hub to get the status of the port.

PORT_STATUS

The result of the port status request is analyzed. If the device appears to have disconnected, then the state machine is terminated, and the detach event will be detected by the hub state machine. If the port status indicates that the device has not been reset, then the port status retry counter is decremented, and after a delay the state machine goes back to the RESETTING state, to re-submit the port status request. If the port status counter is zero, then a clear port enable command is sent, the reset retry counter is decremented, the port status counter reset to its original value and the next state set to CLEAR_CONNECT. This will cause the port to be reset again. If both retry counters are zero, then the device is considered unusable and the device state set to UNDEFINED. If the device has been rest and enabled then the reset is successful. A control command is sent to the hub to clear the reset change bit in the port and the next state set to CLEAR_RESET.

CLEAR_RESET

In this state we are reasonably sure that the device has been reset correctly, it should respond to control commands sent to device ID 0. A new device ID is allocated from the usb_bus object and stored in the state_data field. A control command is now sent to device ID 0 to set the address of the device to the allocated value. The next state is set to ADDRESS.

ADDRESS

If the attempt to set the address failed then the device is disabled, the ID freed and the state set to CLEAR_CONNECT to go through the rest and port status cycle again. If it was successful then the device ID is set to the allocated value and a new control endpoint is attached in the HCD. A control request is sent to the device to read the device descriptor into the buffer and the next state set to DEV_DESC.

DEV_DESC

Once the device descriptor has been successfully read, it is copied into the *desc* field of the device object. A request is now sent to read the first 9 bytes of the first configuration descriptor into the buffer. The next state is set to CFG_DESC and the state_data set to zero.

CFG_DESC

From the first 9 bytes of the configuration descriptor it is possible to get the whole size of the configuration. This is used to send a request to read the entire configuration into the buffer. The next state is set to CFG_ALL.

CFG_ALL

The read configuration is parsed and converted into a chain of usb_descriptor objects, which are then appended to the *desc_chain* in the device. If there are more configurations to read, then a new request to read the first 9 bytes of the next descriptor is sent and the state set to CFG_DESC; the state_data field is used to keep track of which descriptor is currently being read.

If all the descriptors have been read then the device configuration is inspected. If the device is a hub, then the hub state machine is started. Otherwise, the shared buffer is released and a class driver is sought to support this device. If no class driver is found, the device state is set to UNSUPPORTED, otherwise it is set to RUNNING.

RUNNING

This is the eventual state for a device supported by a class driver. The device will stay in this state until the physical device detaches.

A device can end up in two other states instead of this one. UNSUPPORTED state is similar to RUNNING except that there is no class driver. UNDEFINED state is reached if the device appears to be attached to the hub port, but does not communicate with the USB stack.

When a device detach is detected by the hub state machine, `usb_device_detach()` is called. This function puts the device into DETACH state, deallocates the device ID and unlinks the device from the parent hub. If the device is a hub, then it also recursively detaches any devices attached to the ports of this hub. If the device has a class driver attached to it, then the driver's detach routine is called. Finally, `usb_device_unref()` is called to remove the physical device's reference. This should result in the device being deallocated once any pending transfers have terminated.

Hub Lifecycle

Initially a hub passes through the same state machine as any other device to reset it, allocate an ID and read the descriptors. Once this is done and the device is identified as a hub, control moves to the hub state machine, which is an additional set of states to the device state machine.

RUNNING

> The hub starts out in device RUNNING state. The shared buffer is still allocated and a control command is sent to the hub to read its hub descriptor. The next state is set to DESC.

DESC

> When the hub descriptor has been read, the number of downstream ports is extracted and saved. A control command is sent to power up port 1 and the state_data is set to 1. The next state is set to PORT_POWER.

PORT_POWER

> The state machine loops in this state sending a command to power up each hub port in turn, using state_data to keep track of the current port. Once all ports have been powered up, a command to fetch the port status of port 1 is sent and status_data set to 1. The next state is set to PORT_STATUS.

PORT_STATUS

> The port status result is analyzed and if it shows a connection status change then `usb_device_attach()` or `usb_device_detach()` are called as appropriate. If there are more ports to poll, then a port status command is sent for the next port, and state_data incremented to track which port is being polled. If all the ports have been polled, then the state is set to READY and a delay set up for some number of milliseconds in the future.

READY

> This is the default state of a hub when it is not polling the ports. When the delay set up in PORT_STATUS expires, this state is processed. A new port status request is sent for port 1, state_data set to 1, and the next state set to PORT_STATUS. This re-executes the loop in PORT_STATUS state to poll all the ports and act on any attach/detach events.

Name

Class Drivers — Structure and Interface

Synopsis

#include <cyg/io/usb.h>

int **usb_class_driver_register**(*class_driver*);

int **usb_class_driver_deregister**(*class_driver*);

usb_descriptor ***usb_descriptor_find**(*desc*, *type*);

usb_descriptor ***usb_device_class_find**(*dev*, *class*, *subclass*, *protocol*, *configuration*, *interface*);

int **usb_device_configure**(*dev*, *config*, *callback*, *callback_data*);

Description

A class driver translates between operations on a standard eCos device interface and operations on a USB device. For example the USB mass storage class driver translates between disk driver operations and USB mass storage operations.

USB Class Object

A class driver interfaces initially to the USB stack through a usb_class_driver object:

```
struct usb_class_driver
{
    usb_node            node;           // Link in class driver list
    int                 priority;       // Priority in list

    int                 (*attach)( usb_class_driver *class_driver,
                                   usb_device *device );

    int                 (*detach)( usb_class_driver *class_driver,
                                   usb_device *device );

    void                (*poll)( usb_class_driver *class_driver,
                                 usb_uint32 interval );
};
```

The fields in this structure are as follows:

node

A list node, which is used to link this object into a prioritized list of class drivers. This is initialized by `usb_class_driver_register()` so does not need to be initialized by the class driver.

priority

The priority of this class driver. This should be a positive integer. When looking for a class driver to handle a newly attached device, the list is scanned in increasing priority order. So lower values are handled first.

attach

Whenever a new device is attached to a hub, the USB stack assigns it an address, fetches any descriptors, and then tried to find a class driver for it. It does this by calling the `attach()` functions of all registered class drivers until one indicates that it is willing to handle this device. The `attach()` function indicates acceptance by returning `USB_OK`, it indicates non-acceptance by returning an error code, preferably `USB_ERR_NO_SUPPORT`.

If the `attach()` function returns `USB_OK` then it should also set the device's *`device.class_driver`* field to point to the usb_class_driver object. It should call `usb_device_ref()` on the USB device to ensure that it remains valid. It must also call `usb_device_select_interface`.

detach — This function is called when the USB device detaches. In addition to cleaning up its own data structures, the main thing this function should do it arrange for the reference to the USB device taken in the attach function to be released by calling `usb_device_unref()`, directly or otherwise.

poll — This function is called periodically from the USB subsystem. The *`interval`* parameter indicates the number of milliseconds since the last call to this function. It can be used by the class driver to operate timeouts and retries. The exact interval between calls will depend on the level of activity of the USB stack and the resolution of the main system timer. Class drivers should therefore not depend on this for accurate timing operation and may need to make their own arrangements for such things.

USB Class API

The USB stack exports a number of functions that are intended for use by class drivers.

The function `usb_class_driver_register()` is used by a class driver to register itself with the USB stack. Typically a class driver is initialized as an instance of the driver type that it is intending to serve (e.g. disk, network, serial etc.) and during the initialization function for that driver will call `usb_class_driver_register()`. If the driver ever needs to detach itself from the USB stack then it can call `usb_class_driver_deregister()`.

Within the `attach()` function, the class driver can call some USB stack functions to help it decide whether a device is one that it can support. The most important of these is `usb_device_class_find()` which scans the descriptors attached to a a device for an interface that implements the given *`class`*, *`subclass`* and *`protocol`* types. A value of -1 for *`subclass`* and *`protocol`* acts as a wildcard. If successful it returns pointers to the configuration and interface found. The function `usb_descriptor_find()` can be used on *`dev->desc_chain`*, or any other descriptor pointer, to find the next descriptor of a given type. The class driver can also just inspect the device and parse the descriptor chain itself if necessary.

Before returning, the `attach()` function should call `usb_device_configure()`. Which will configure the device to use the configuration descriptor is supplied. The class driver must also supply a callback which will be called when the configuration has been done, or has failed. Further device setup can then be done in the callback.

Putting all that together, the functions of a class driver should have the following approximate form:

```
//--------------------------------------------------------------------------
// Attach device call

static int mydev_attach( usb_class_driver *class_driver, usb_device *dev )
{
    int result = USB_OK;
    mydev_data *mydev;

    // Look for configuration and interface. Here we assume that the
    // first configuration is the one we want to use and that we are
    // not worried about the function type, hence the wildcard.
    usb_descriptor *cdesc, *idesc;

    usb_device_class_find( dev, USB_CLASS_MYCLASS, USB_SUBCLASS_MYSUBCLASS, -1, &cdesc, &idesc );

    if( cdesc == NULL || idesc == NULL )
        return USB_ERR_NO_SUPPORT;

    // Set up device data structures here...
    mydev = mydev_alloc();
    mydev->dev = dev;

    // Extract any useful information from the interface descriptor
    // chain, such as endpoint addresses...

    // Reference the device
    usb_device_ref( dev );
    // Send off a command to select the configuration
    // we have found.
    result = usb_device_configure( dev, cdesc, mydev_attach_tfr_done, mydev );

    // If it all worked, set the device's class driver to point to us.
    if( result == USB_OK )
        dev->device.class_driver = class_driver;

    return result;
}

//-----------------------------------------------------------------------------
// Configuration callback

static int mydev_attach_tfr_done( usb_tfr *tfr )
{
    int result = tfr->status;
    mydev_data *mydev = tfr->callback_data;
    usb_device *dev = tfr->dev;

    // Release tfr object
    usb_tfr_unref( tfr );

    if( result != USB_OK )
    {
        // Handle configuration error by detaching from USB device and
        // freeing local resources.
        mydev_free( mydev );
        usb_device_unref( dev );
        return result;
    }

    // Continue device initialization...

    return result;
}

//-----------------------------------------------------------------------------
// Device detach call

static int mydev_detach( usb_class_driver *class_driver, usb_device *dev )
{
    int result = USB_OK;

    // Find my device data from device pointer.
    mydev_data *mydev = mydev_find( dev );

    // Shut down device...

    // Free device data structure
    mydev_free( mydev );

    // Release reference to device
    usb_device_unref( dev );

    return result;
}

//-----------------------------------------------------------------------------
// Device poll call

static void mydev_poll( usb_class_driver *class_driver, usb_uint32 interval )
{
    // Handle timeouts and delays in active devices...
}
```

Name

Host Controller Drivers — Structure and Interface

Description

This section is mainly of interest to developers who want to write a new host controller driver. It describes the interface used by the USB stack to initiate HCD operations and the API that an HCD can use to interact with the USB stack.

HCD Object

The main interface between the USB stack and each type of HCD is the usb_hcd object:

```
struct usb_hcd
{
    const char                  *name;          // Driver name

    // Initialization etc.
    void (*init)( void );                       // Initialize controller(s)
    int  (*attach)( usb_bus *bus );             // Attach to hardware
    int  (*detach)( usb_bus *bus );             // Detach from hardware

    // Endpoint handling
    int (*endpoint_attach)( usb_device *dev, usb_device_endpoint *dep );
    int (*endpoint_detach)( usb_device *dev, void *hcd_endpoint );

    // Transfer handling
    int  (*submit)( usb_device *dev, usb_tfr *tfr ); // Submit transfer (chain)
    int  (*cancel)( usb_device *dev, usb_tfr *tfr ); // Cancel transfer

    // Controller operation
    void (*poll)( usb_bus *bus );               // Poll controller for events

    int  (*frame_number)( usb_bus *bus );       // Get current frame number


    // Root hub support

    int (*port_status)( usb_bus *bus, int port, usb_hub_port_status *status);
    int (*set_port_feature)(usb_bus *bus, int port, usb_uint16 feature );
    int (*clear_port_feature)(usb_bus *bus, int port, usb_uint16 feature );

    // TODO: Bandwidth support

} CYG_HAL_TABLE_TYPE;
```

The fields are as follows:

name

This is a pointer to a string that names this device. It is mainly used for debugging.

init

This is called once by the USB stack to initialize all HCDs of this type. In combination with platform code this function should enumerate all the HCDs of the supported type and eventually call `usb_hcd_register()` to make the controller available to the USB stack.

The call to `usb_hcd_register()` is passed a hcd_bus object that the HCD should allocate in its private data structures. Within this object the *`hcd`* field should be set to this HCDs usb_hcd object. The *`hcd_priv`* field should be set to point to the HCDs per-controller private data structure; this value will be copied to the *`hcd_priv`* field of any device attached to this bus. The *`hcd_ep0`* field should be set to point to an HCD control endpoint for device 0; this will be used to communicate with a newly attached device before its ID has been set. The second argument to `usb_hcd_register()` is a count of the number of downstream, ports the root hub contains.

While this function should locate the devices and initialize the HCD data structures it should not access the Host Controller hardware at this point.

attach
: This is called to attach the HCD to the hardware. This is when the hardware should be initialized, interrupt handlers registered and everything made ready for transfers to occur.

detach
: This is called to detach the HCD from the hardware. It should undo the initialization done by the attach function, leaving the device free for other software to take control.

The main reason for this attach/detach mechanism is to allow OTG devices to be shared between host and peripheral drivers.

endpoint_attach
: This is called to create an endpoint in the host controller. The HCD should use the id of the device plus the endpoint descriptor in the usb_device_endpoint object to create an endpoint of the correct type and direction for the device.

The HCD will typically allocate controller and driver data structures to represent this endpoint. If the underlying controller only supports a limited number of endpoints, then the driver should either fail excess endpoint attachments, or arrange to share the physical endpoints between a larger number of virtual endpoints. If the HCD endpoint is created successfully the it should assign a pointer to it to the hcd field in the usb_device_endpoint object.

endpoint_detach
: This is called when the device is detached, or changes its active interface. It undoes the resource allocation made in `endpoint_attach`. Additionally, this function must cancel any transfers that are pending on the endpoint. Depending on the nature of the controller, these transfer cancellations and the eventual deallocation of the endpoint may happen after this function returns.

submit
: This is called to submit a transfer to a device. Internally, this function should extract the HCD private data from the device *`hcd_priv`* and the endpoint from the device's usb_device_endpoint object for the transfer's endpoint address. The HCD is free to use the *`hcd_endpoint`* and *`hcd_list`* fields in the usb_tfr object; the latter should be initialized before use.

cancel
: This is called to cancel a pending transfer. In general this is only necessary for interrupt or isochronous transfers, control and bulk transfers will always terminate within a finite time. The transfer will not necessarily be available for reuse once this function returns. This is only guaranteed once the transfer's callback is invoked, either with a `USB_TFR_CANCELLED` status, some other error, or even `USB_OK`.

poll
: This is called from the main USB handling loop to give the HCD the chance to service the hardware. In general all controller operations should be handled in this function rather than the ISR or DSR. The HCD should test the hardware for transfer completion, device attach/detach and errors and handle them here.

If a transfer completes in this polling routine its callback may either be invoked directly by calling `usb_tfr_callback_pop()` or may be deferred for later processing by calling `usb_tfr_complete_async()`. The latter is preferable since it avoids any problems of recursion if the callback submits another transfer.

The simplest way to write an HCD is to do all device event handling in the `poll()` routine. If the controller supports interrupts then the HCD can call `usb_signal_poll()` to cause the poll routine to be called. If it makes sense to handle device events in the ISR or DSR, callbacks, such as returning transfers, should still happen in the poll routine.

frame_number
: This simply returns the current USB frame number.

port_status

This call fills in the *status* buffer with information on the state of the given port. This routine should query the port in the host controller's root hub registers and translate the results into the standard format expected in the status result, which should be returned in little endian order.

set_port_feature

This is called to set a port feature. The *feature* argument is a standard hub port feature code as defined in the USB standard. Only the subset of features relevant to a root hub are supported.

clear_port_feature

This is called to clear a port feature. The *feature* argument is a standard hub port feature code as defined in the USB standard. Only the subset of features relevant to a root hub are supported.

Name

Target Object — Structure and Interface

Synopsis

#include <cyg/io/usb.h>

void **usb_target_ref**(*tgt*);

void **usb_target_unref**(*tgt*);

int **usb_target_attach**(*tgt*, *pcdi*);

int **usb_target_detach**(*tgt*);

int **usb_target_stall**(*tgt*, *ep*, *stall*);

usb_pcdi ***usb_pcdi_find_by_name**(*name*);

int **usb_string_descriptor_utf8**(*buf*, *len*, *u8*);

int **usb_string_descriptor_create**(*buf*, *len*, *index*, *strings[]*, *strings_num*);

Description

In order to support a target device, application code must create and initialize a usb_target object, which is then passed to the USB stack.

Throughout this section the code that instantiates and uses the target object is referred to an an "application". Normally this will be a device driver or other middleware that translates USB operations into some other interface that eCosPro understands. Examples would be the CDC ACM driver that translates USB traffic into a serial device, or the RNDIS driver that translates into a Ethernet driver interface.

The target mode stack retains the USB terminology for data transfer direction, which can be a little confusing. So a transfer which involves data being passed from the host to be received by the target, is referred to as an OUT transfer. Similarly, transmission from the target to the host is referred to an an IN transfer.

Target Object

A target object has the following structure:

```
struct usb_target
{
    usb_pcdi                    *pcdi;          // PCD device instance

    usb_uint32                  flags;          // Flags
    usb_uint8                   refcount;       // Reference counter
    usb_uint8                   id;             // Device ID
    usb_uint8                   state;          // Current target state

    const usb_device_descriptor *desc;          // Device descriptor
    const usb_device_qual_descriptor *qdesc;    // Device qualifier descriptor

    const usb_config_descriptor **configs;      // Array of configurations
    int                         config_count;   // Size of array
    const usb_config_descriptor *config_current;// Currently selected config
    const usb_string_descriptor **strings;      // Array of string descriptors
    int                         string_count;   // Size of array

    usb_target_endpoint         ep0;            // Control endpoint

    usb_target_interface        *interfaces;    // List of active interfaces

    void                        *data;          // Client private data pointer


    // Optional callbacks to user code

    // Control message escape
    int (*control)(usb_target *tgt, usb_request *req, void **buf, usb_uint16 *len );

    int (*new_state)(usb_target *tgt);          // Signals state change

    // Dynamic descriptor callback, called if the addressed static
    // descriptor pointer is NULL.
    int (*get_descriptor)(usb_target *tgt, usb_uint8 type, usb_uint8 index,
                          usb_uint8 **buf, usb_uint16 *len );

#if CYGINT_IO_USB_TARGET_INTERFACE_CALLBACK>0
    // Interface change callback
    int (*set_interface)(usb_target *tgt, usb_uint8 intf, usb_uint8 alt);
#endif
};
```

The fields of the target object are as follows:

pcdi

This is a pointer to the PCD instance to which this target object is attached. This is initialized during the call to `usb_target_attach()` and cleared by `usb_target_detach()`. This field should not be initialized or changed by the application.

flags

Various flag bits. At present only two flags are defined, `USB_TGT_FLAG_CALLBACK_CTRL` and `USB_TGT_FLAG_CALLBACK_DESC` which control the invocation of the `control()` and `get_descriptor()` callbacks.

refcount

Target reference count. Since target objects are allocated by the application, zeroing this count does not cause the target to be zeroed. This count is used to keep track of the number of transfers associated with the target and for consistency checking.

id

The device ID. This is the ID that the host has set via a SET ADDRESS command. Before that happens, and when the target is reset, this will be set to zero.

state

Target state. This moves through states as defined by the USB specification and controls how the target reacts to bus events such as suspend, resume and reset.

desc

A pointer to the device descriptor for this target. If this pointer is NULL, then the `get_descriptor()` callback is called to supply the descriptor.

qdesc

A pointer to the device qualifier descriptor for this target. If this pointer is NULL, then the `get_descriptor()` callback is called to supply the descriptor.

configs

A pointer to an array of pointers to configuration descriptors. If this pointer is NULL, or a pointer in the array is NULL, then the `get_descriptor()` callback is called to supply the descriptor.

config_count

The size of the *`configs`* array. If a descriptor index greater than this value is requested, then the `get_descriptor()` callback is called to supply the descriptor.

config_current

When a target passes into the CONFIGURED state, this field will be set to point to the configuration selected. When the target is reset, this will be set back to NULL.

strings

A pointer to an array of pointers to string descriptors. If this pointer is NULL, or a pointer in the array is NULL, then the `get_descriptor()` callback is called to supply the descriptor.

There are a number of issues with string descriptors and their encoding which are covered in the section titled String Descriptor Encoding.

string_count

The size of the *`strings`* array. If a descriptor index greater than this value is requested, then the `get_descriptor()` callback is called to supply the descriptor.

ep0

A target endpoint structure. This is initialized and attached to the PCDI when the target is initially connected to the bus. It is used for all control endpoint transfers.

The *`desc`* sub-field of this object may be initialized to point to a descriptor for endpoint 0. If this is left NULL, it will be initialized to point to a default descriptor that allows a maximum packet size of 64 bytes. If the client needs to use a different maximum packet size on endpoint zero, it should set this sub-field.

interfaces

A chain of dynamically allocated target interface objects. When the target is configured by the host the selected configuration is scanned and for each endpoint in each active interface an endpoint is created and attached to the PCDI. A usb_target_endpoint object is allocated and attached to this list for each interface, and itself contains a list of usb_target_endpoint objects for each attached endpoint. When the target is reset, this list is scanned, the endpoints detached from the PCDI, and all the objects freed. A switch to a different alternate setting for an interface will result in the interface object in this list being detached, its endpoints detached from the PCDI, and a new interface and endpoints for the selected alternate created.

data

A data pointer that the application may use for its own purposes. Normally this will point to some data structure associated with the application.

`control()`

Normally, the USB target mode stack will handle all control SETUP messages to read descriptors, set the address, set the configuration and other commands. If a SETUP packet arrives that has a request type or code that is not recognized, then this function will be called. The return code defines what will happen next:

`USB_OK`

> This indicates that the command was recognized and processed and there is no further action required. The USB stack will return a status packet to the host and then return to looking for the next SETUP packet.

`USB_TARGET_CONTROL_DATAIN`

> This indicates that the USB stack should return data to the host. The data to be returned should be described by setting `*buf` and `*len` to the address and size of a buffer.
>
> When the data has been successfully sent, this function will be called again, with the same request structure, the buffer pointer as passed, and `*len` set to the actual quantity of data sent. This is done so that the application can release or reuse the buffer; it can distinguish this call from the first by looking at `*buf` which will be NULL in the first call and non-NULL in the second. On return from this second call a status packet will be received from the host and the USB stack will return to looking for the next SETUP packet.

`USB_TARGET_CONTROL_DATAOUT`

This indicates that the USB stack should receive data from the host. A buffer into which the data should be received is described by setting `*buf` and `*len` to the address and size of the buffer.

When the data has been successfully received, this function will be called again, with the same request structure, the buffer pointer as passed, and `*len` set to the actual quantity of data received. The application can distinguish this call from the first by looking at `*buf` which will be NULL in the first call and non-NULL in the second. On return from this second call a status packet will be return to the host and the USB stack will return to looking for the next SETUP packet.

`USB_ERR_COMMAND_INVALID`

If the callback returns this error code, then the USB stack will generate a STALL condition on the bus, which will act to abort the control transfer. The stack will then return to looking for the next SETUP packet.

If this callback is NULL, then any unrecognized SETUP packets will cause a STALL. So, unless the target device protocol contains extra control operations, it is not necessary for the application to supply this callback.

`new_state()`

This callback is called each time the target moves into a new state. The application can perform any processing of its own in response to this call. If the application does no need to process these events, it can set this pointer to NULL.

`get_descriptor()`

If any of the descriptor pointers in the target object is NULL, or a descriptor outside the supplied set is fetched, this callback will be called. The *`type`* and *`index`* values identify the descriptor being read. If the application can generate the descriptor itself, it should set `*buf` and `*len` to point to the descriptor and return.

When the descriptor has been returned to the host, this callback will be called again with the same type and index values. This is done so that the application can release or reuse the buffer; it can distinguish this call from the first by looking at `*buf` which will be NULL in the first call and non-NULL in the second.

`get_interface()`

If a target implements interfaces with alternate settings, the CDL interface CYGINT_IO_USB_TARGET_INTERFACE_CALLBACK should be implemented to cause this callback to be present. Subsequently, whenever the host sends a `SET_INTERFACE` operation to select an alternate setting, this function will be called. This allows the client code to adapt to the potential change in endpoint configuration.

Since relatively few targets implement alternate interface settings, this callback is only present if the CDL interface is implemented.

String Descriptor Encoding

USB string descriptors are in Unicode, encoded in UTF-16LE. Unfortunately, this is not a character encoding that is directly supported by the GCC toolchain. There are a number of ways to work around this. The first, and simplest is to use the `-fshort-char` compiler option to force `wchar_t` to be 16 bits rather than the default 32 bits. Strings can then be prefixed by `L` to ensure 16 bit Unicode encoding. A typical static descriptor can then be defined as follows:

```
static const usb_string_descriptor mytgt_string_manufacturer =
{
    .bLength                            = 2+2*11,
    .bDescriptorType                    = USB_DESC_STRING,
    .bString                            = L"eCosCentric"
};
```

However, there are a number of problems with this. The encoding is strictly 16 bits, and any code points that require a surrogate pair cannot be defined. Compiling files with the `-fshort-char` option will throw up compiler warnings since it differs from the defaults with which the libraries will have been built. But, most importantly, it only works for little endian targets; big endian targets will generate the 16 bit values in big endian byte order.

A more portable approach would be to encode the UTF-16LE directly using optional byte swaps where necessary, as in the following example:

```
static const usb_string_descriptor mytgt_string_manufacturer =
{
    .bLength                            = 2+2*11,
    .bDescriptorType                    = USB_DESC_STRING,
    .bString                            = { USB_CPU_TO_LE16('e'),
                                            USB_CPU_TO_LE16('C'),
                                            USB_CPU_TO_LE16('o'),
                                            USB_CPU_TO_LE16('s'),
                                            …
                                          }
};
```

However, this approach is clumsy and does not allow the size or contents of the string to be made a configuration option, or even easy to change in the code.

The preferred approach in the USB stack is to generate and store string descriptors in UTF-8 and to convert them to UTF-16LE at run time, when the descriptor is requested. A UTF-8 string is just a sequence of bytes and can be defined and manipulated like any other byte array. Most text editors will allow a UTF-8 string to be created or pasted in from some other source without any problems. Most UTF-8 strings occupy less space than their UTF-16LE equivalents. A standard ASCII string is just a UTF-8 string that contains no code points beyond the basic ASCII set.

To simplify use of UTF-8 strings, the USB stack exports a couple of helper functions. The function `usb_string_descriptor_utf8()` takes a pointer and length of a buffer in which a string descriptor is created, and a pointer to a UTF-8 string. It recodes the UTF-8 string into UTF-16LE in the buffer together with setting the descriptor size and type. If successful, the buffer will contain a string descriptor ready to be transmitted. If the buffer is not large enough for the descriptor, an error code will be returned.

The function `usb_string_descriptor_create()` is passed a buffer pointer and length, the index of the descriptor to be returned and a pointer to an array of UTF-8 strings. It checks that the index is correct and then creates a new string descriptor in the buffer using the indexed string from the array; it calls `usb_string_descriptor_utf8()` to do this.

To put all this together, the strings for a device can be defined statically as follows:

```
static const usb_string_descriptor mytgt_string_langid =
{
    .bLength                            = 2+2*1,
    .bDescriptorType                    = USB_DESC_STRING,
    .bString                            = { USB_CPU_TO_LE16(0x0809) },
};

static const usb_string_descriptor *mytgt_string_descriptors[1] =
{
    [0]                 = &mytgt_string_langid,
};

static const char *mytgt_descriptor_strings[] =
{
    [1]                 = "eCosCentric",
    [2]                 = "My Device",
    [3]                 = "01234567890",
};
```

In the target object, the *strings* field is set to point to `mytgt_string_descriptors` and string_count set to 1. This will cause the USB target stack to call the target's get_descriptor function for the remaining string descriptors. This function should look like the following example:

```
static usb_uint8 mytgt_dynamic_desc[64];       // Dynamic descriptor buffer

static int mytgt_get_descriptor(usb_target *tgt, usb_uint8 type, usb_uint8 index,
                                usb_uint8 **buf, usb_uint16 *len )
{
    int result = USB_OK;

    // A non-NULL buffer pointer is the USB stack returning the buffer
    // to us for reuse.
    if( *buf != NULL )
        return USB_OK;

    if( type == USB_DESC_STRING )
    {
        result = usb_string_descriptor_create( mytgt_dynamic_desc, sizeof(mytgt_dynamic_desc),
                                               index,
                                               mytgt_descriptor_strings,
                                               sizeof(mytgt_descriptor_strings)/sizeof(char *) );
        if( result == USB_OK )
        {
            *buf = mytgt_dynamic_desc;
            *len = mytgt_dynamic_desc[0];
        }
    }
    else
    {
        // Handle any other descriptor types
    }

    return result;
}
```

Note that in this example the serial number string is a constant. For devices where a unique serial number is required for each unit, a different approach may be needed. First the unit must have a unique identifier that can be used for this purpose. Depending on the platform this could be fetched from EPROM, FLASH, a serial number chip or a built-in chip ID. The simplest approach is to convert this value into an ASCII string. Then in the `get_descriptor()` function this string can be converted to a string descriptor when the serial number is requested. For an example take a look at the `hid_test.c` test program where a serial number is manufactured from a checksum of the executable.

Target Object API

The USB stack exports a number of functions that are intended for use by applications using targets.

The function `usb_target_attach()` must be called to attach a target to a specific peripheral interface. Peripheral interfaces are named and a pointer to a particular interface can be obtained by calling `usb_pcdi_find_by_name()`. Following this, most target events will be handled by the USB stack with calls to the callbacks as necessary. If the application wants to stop the target, it should call `usb_target_detach()`.

Some USB protocols require an endpoint stall to signal various conditions. The function `usb_target_endpoint_stall()` allows this to be done. The *ep* argument contains the endpoint address, and should have `USB_ENDPOINT_ADDR_IN` set for IN endpoints. The *stall* argument is 1 to stall the endpoints and zero to clear the stall condition.

An application must instantiate a usb_target object and initialize it before calling `usb_target_attach()`. Typically this object can be defined statically as in the following example:

```
static usb_target mytgt_target =
{
    .desc               = &mytgt_device_descriptor,
    .qdesc              = &mytgt_device_qual_descriptor,

    .configs            = mytgt_config_descriptors,
    .config_count       = 1,

    .strings            = mytgt_string_descriptors,
    .string_count       = 1,

    .control            = mytgt_control,
    .new_state          = mytgt_new_state,
    .get_descriptor     = mytgt_get_descriptor,

    .data               = &mytgt_data,
};
```

This defines the target object, initializes the static descriptors and callbacks. No fields beyond those shown above need to be initialized. While none of the fields is mandatory, if a static descriptor is not present, the `get_descriptor()` callback will be called, so it is not sensible to have both NULL descriptors and no `get_descriptor()`.

Once the application has started, it should locate the PCDI it wants to attach the target to and call `usb_target_attach()`, as in the following example:

```
void mytgt_init( void )
{
    usb_pcdi *pcdi;

    // Find PCDI by name
    pcdi = usb_pcdi_find_by_name( "usb_fs" );

    if( pcdi == NULL )
    {
        // Handle error
    }

    // Attach our target to the PCDI
    result = usb_target_attach( &mytgt_target, pcdi );

    if( result != USB_OK )
    {
        // Handle error
    }
}
```

Once the target has been attached, all further interaction with the application will be via the callbacks. Most `new_state()` callbacks can be ignored while the target is going through the initial connect/reset/address sequence. The transition to CONFIGURED state is the most important since this is when the target should become ready to interact with the host. Normally this will be the point at which it submits transfers to the OUT endpoints to receive packets from the host and maybe starts sending data via the IN endpoints.

For a complete example take a look at the `acm_example.c` test program. This source is annotated with extra comments. The `usbms_tgt.c` and `hid_test.c` test programs in `packages/io/usb/<version>/tests` directory. The CDC/ACM protocol driver can also examined for example code.

Name

Peripheral Controller Drivers — Structure and Interface

Description

This section is mainly of interest to developers who want to write a new peripheral controller driver. It describes the interface used by the USB stack to initiate PCD operations and the API that a PCD can use to interact with the USB stack.

PCD Objects

The main interface between the USB stack and each type of PCD is the usb_pcd object:

```
struct usb_pcd
{
    const char                  *name;                          // Driver name

    int pad;

    // Initialization etc.
    void (*init)( void );                                       // Initialize controller(s)
    int  (*attach)( usb_pcdi *pcdi, usb_target *tgt );          // Attach to hardware
    int  (*detach)( usb_pcdi *pcdi, usb_target *tgt );          // Detach from hardware

    // Endpoint attach/detach
    int (*endpoint_attach)( usb_pcdi *pcdi, usb_target_endpoint *tep );
    int (*endpoint_detach)( usb_pcdi *pcdi, usb_target_endpoint *tep );

    // Set/clear endpoint stall
    int (*endpoint_stall)( usb_pcdi *pcdi, usb_target_endpoint *tep, int stall );

    // Transfer handling
    int  (*submit)( usb_target *tgt, usb_tfr *tfr );            // Submit transfer (chain)
    int  (*cancel)( usb_target *tgt, usb_tfr *tfr );            // Cancel transfer

    // Controller operation
    void (*poll)( usb_pcdi *pcdi );                             // Poll controller for events

    int (*set_address)( usb_pcdi *pcdi, usb_uint8 addr );       // Set new target address

};
```

The fields are as follows:

name

This is a pointer to a string that names this device. It is mainly used for debugging.

init

This is called once by the USB stack to initialize all PCDs of this type. In combination with platform code this function should enumerate all the PCDs of the supported type and eventually call `usb_pcdi_register()` to make the controller available to the USB stack.

The call to `usb_pcdi_register()` is passed a usb_pcdi object that the PCD should allocate in its private data structures.

While this function should locate the devices and initialize the PCD data structures it should not access the Peripheral Controller hardware at this point.

attach

This is called to attach the PCD to the hardware. This is when the hardware should be initialized, interrupt handlers registered and everything made ready for transfers to occur.

detach

This is called to detach the PCD from the hardware. It should undo the initialization done by the attach function, leaving the device free for other software to take control.

The main reason for this attach/detach mechanism is to allow OTG devices to be shared between host and peripheral drivers.

endpoint_attach

This is called to create an endpoint in the peripheral controller. The PCD should use the endpoint descriptor in the usb_target_endpoint object to create an endpoint of the correct type and direction for the device.

The PCD will typically allocate controller and driver data structures to represent this endpoint. If the underlying controller only supports a limited number of endpoints, then the driver should either fail excess endpoint attachments, or arrange to share the physical endpoints between a larger number of virtual endpoints. If the PCD endpoint is created successfully the it should assign a pointer to it to the *`pcd`* field in the usb_target_endpoint object.

endpoint_detach

This is called when the target is detached, or changes its active interface. It undoes the resource allocation made in `endpoint_attach`. Additionally, this function must cancel any transfers that are pending on the endpoint. Depending on the nature of the controller, these transfer cancellations and the eventual deallocation of the endpoint may happen after this function returns.

submit

This is called to submit a transfer to the controller. Internally, this function should extract the PCD private data from the target *`pcdi`* field and the endpoint from the target's usb_target_endpoint object for the transfer's endpoint address. The PCD is free to use the *`hcd_endpoint`* and *`hcd_list`* fields in the usb_tfr object; the latter should be initialized before use.

cancel

This is called to cancel a pending transfer. The transfer will not necessarily be available for reuse once this function returns. This is only guaranteed once the transfer's callback is invoked, either with a `USB_TFR_CANCELLED` status, some other error, or even `USB_OK`.

poll

This is called from the main USB handling loop to give the PCD the chance to service the hardware. Controller operations may be handled either in this function or in the ISR or DSR; however, callbacks must be made from this function. The PCD should test the hardware for transfer completion, device attach/detach and errors and handle them here.

If a transfer completes in this polling routine its callback may either be invoked directly by calling `usb_tfr_callback_pop()` or may be deferred for later processing by calling `usb_tfr_complete_async()`. The latter is preferable since it avoids any problems of recursion if the callback submits another transfer.

The simplest way to write an PCD is to do all device event handling in the `poll()` routine. If the controller supports interrupts then the PCD can call `usb_signal_poll()` to cause the poll routine to be called. If it makes sense to handle device events in the ISR or DSR, callbacks, such as returning transfers, should still happen in the poll routine.

set_address

This is called to set the address of the target in the peripheral controller. This is called after the host has sent a SET_ADDRESS command.

There is just one instance of the usb_pcd object for each type of peripheral controller. However, there may be more than one physical device of each type on the board. Each of these is represented by a usb_pcdi object:

```
struct usb_pcdi
{
    usb_node                    node;                   // Link in PDCI list

    char                        *name;                  // Instance name

    usb_uint8                   state;                  // Controller state

    usb_pcd                     *pcd;                   // Pointer to PCD
    usb_target                  *tgt;                   // Current attached target
    void                        *pcdi;                  // Driver private data
};
```

node Node in list of active PCD instances. This need not be initialized by the PCD, it is initialize by `usb_pcdi_register()`.

name Name of this interface. This should distinguish it from all other PCDIs, and is used by `usb_pcdi_find_by_name()` to locate this PCDI. This field must be initialized by the PCD before calling `usb_pcdi_register()`.

pcd A pointer to the usb_pcd object for the controlling driver. This field must be initialized by the PCD before calling `usb_pcdi_register()`.

tgt When a target is attached to a PCDI by calling `usb_target_attach()`, a pointer to the target is placed here. This field should be initialized to NULL by the PCD before calling `usb_pcdi_register()`.

pcdi A pointer to a per-instance data structure in the PCD. This field must be initialized by the PCD before calling `usb_pcdi_register()`.

Part XXXV. USB Serial Support

Documentation for drivers of this type is often integrated into the eCos board support documentation. You should review the documentation for your target board for details. Standalone and more generic drivers are documented in the following sections.

Table of Contents

Chapter 118. USB Serial Support

Name

Overview — eCosPro Support for USB Serial devices

Description

eCosPro USB serial support is divided into a number of packages. The USB serial driver package (`CYGPKG_IO_SERIAL_USB`) provides the common part of the USB serial device support. It communicates with one or more packages that implement a specific USB serial protocol. Serial support is implemented in both peripheral and host modes. The currently supported USB serial protocol packages include the Target CDC ACM protocol driver, Host CDC ACM protocol driver and Host FTDI protocol driver.

Applications access USB serial devices just as they would a physical UART, using the standard eCos I/O package API to send and receive data, and to set and get configuration and status information.

Example USB serial test applications can be found in the `packages/devs/serial/usb/<VER>/tests` directory. In particular the `usb_echo.c` test demonstrates access to all potentially attached USB serial protocol device types. It also includes code that shows the use of callbacks to determine when a host adapter has been attached or detached.

The following sections describe the configuration, internal data structures and workings of the USB serial package. They will generally only be of interest to someone with specific configuration requirements, or to gain a deeper understanding of the interface between the USB serial and protocol packages.

Data Structures

For various practical reasons, the interface between this package and the protocol drivers is defined in the `usb_serial.h` file in the main USB package.

The interface between the eCos serial driver and the USB protocol driver is defined by a data structure, usb_serial_if. This contains a number of fields that control the transfer of data between the drivers. Many of these are for internal use of the drivers, however, a number must be initialized by the user:

`dev`

In host mode this should be set to point to the usb_device object for the device being used when it has attached to the bus. The protocol driver must take a reference to this device in order to prevent it from being deallocated.

`tgt`

In target mode this points to the usb_target object that represents this peripheral. This target should be populated with suitable descriptors for the protocol being implemented.

`call`

This points to a table of functions that are used to communicate between the two drivers. This is described later.

`chan`

A pointer to the eCos serial device channel.

`rx_buf`

A pointer to a buffer used to receive data from the protocol package.

`rx_maxpkt`

The size of the *`rx_buf`* buffer.

`target.pcdi_name`

In target mode, this points to a string that names the peripheral controller to which the target will be attached.

`host.id`

In host mode, this should be the channel index number. This effectively defines the order in which host channels are searched for a VID/PID match. See the CDC/ACM host driver for details.

`host.vid`

In host mode, this controls whether this serial channel matches a particular USB Vendor ID. If zero it will match any ID, otherwise this channel will only match a device with the given Vendor ID. See the CDC/ACM host driver for details.

`host.pid`

In host mode, this controls whether this serial channel matches a particular USB Product ID. If zero it will match any ID, otherwise this channel will only match a device with the given Product ID. See the CDC/ACM host driver for details.

The *`call`* field points to a table of function calls. The functions in this table provide communication between the serial driver and the protocol driver.

`int (*init)(usb_serial_if *usb_if);`

This is the initialization routine for the protocol driver, it is called from the serial driver during it's initialization. This function should perform any USB stack initialization, such as attaching the target object to the PCDI in target mode, or register the class driver in host mode.

`int (*attach)(usb_serial_if *usb_if);`

This is called from the protocol driver whenever an attach event is detected in both host and target modes. The main side effect of this will be to invoke any registered serial callback.

`int (*detach)(usb_serial_if *usb_if);`

This is called from the protocol driver whenever an attach event is detected in both host and target modes. As with the attach call, this will cause the serial callback to be invoked.

`int (*send_data_start)(usb_serial_if *usb_if);`

This is called from the protocol driver to kick the serial driver transmitter into activity. It should be called when the USB device, host or target, moves into a state where data transfers can be started.

`int (*send_data)(usb_serial_if *usb_if, usb_uint8 *buf, usb_int16 len);`

This is called by the serial driver to transmit data on the USB device. The *`buf`* and *`len`* arguments describe the raw data to be sent. The protocol driver may need to wrap this data in any protocol headers or trailers and send it via the USB stack.

`int (*send_data_done)(usb_serial_if *usb_if, usb_uint8 *buf, usb_int16 len);`

This is called by the protocol driver when the transfer requested by `send_data` has completed. The *`buf`* will match the buffer pointer in the `send_data` call, and *`len`* will give the amount of data transmitted. It is likely that this function will call `send_data` to start another transmission; so the protocol driver must be ready for this.

`int (*recv_data)(usb_serial_if *usb_if, usb_uint8 *buf, usb_int32 len);`

This is called by the serial driver to supply a buffer for asynchronous data reception. It will use the *`rx_buf`* and *`rx_maxpkt`* fields from the common data structure. The protocol driver should use this to submit the necessary USB reception transfers to the USB device.

int (*recv_data_done)(usb_serial_if *usb_if, usb_uint8 *buf, usb_int16 len);

The protocol driver calls this when some data has been received. The *buf* and *len* describe the data received; while these will describe a portion of the *rx_buf* buffer, they may not describe all of it since protocol headers and trailers may be skipped. As with `send_data_done`, the serial driver may call back into the protocol driver during this call.

int (*dev_line_coding)(usb_serial_if *usb_if, usb_line_coding *line_coding);

This is called from the protocol driver when it receives a command from the USB peer to set the line parameters. The parameters passed are encoded in the usb_line_coding as described in the `usb_serial.h` header.

int (*dev_control_line_state)(usb_serial_if *usb_if, usb_uint16 line_state);

This is called from the protocol driver when it receives a command from the USB peer to set the RTS and DTR control line states. The state is encoded in the *line_state* argument as described in the `usb_serial.h` header.

int (*usb_line_coding)(usb_serial_if *usb_if, usb_line_coding *line_coding);

This is called from the serial driver when the eCos client sets any of the serial line parameters. The parameters passed are encoded in the usb_line_coding as described in the `usb_serial.h` header.

int (*usb_control_line_state)(usb_serial_if *usb_if, usb_uint16 line_state);

This is called from the serial driver when the eCos client sets the state of either the RTS or DTR lines. The state is encoded in the *line_state* argument as described in the `usb_serial.h` header.

int (*usb_set_config)(usb_serial_if *usb_if, cyg_uint32 key, const void *xbuf, cyg_uint32 *len);

This may be called from the serial driver if it is passed any `set_config()` keys that it does not recognize. It allows the protocol driver to handle any config options itself. This entry may be set to NULL, in which case no call will be made.

int (*usb_get_descriptor)(usb_serial_if *usb_if, usb_uint8 type, usb_uint8 index, usb_uint8 **buf, usb_uint16 *len);

This is used only for target mode drivers. The target `get_descriptor()` callback will be called if a given descriptor is not statically defined in the target object. If that routine cannot supply the descriptor then this callback should be invoked. The arguments follow the pattern of the target `get_descriptor()` function, except for the first argument, which is a pointer to the serial interface object and not the target object. This entry may be set to NULL if there are no descriptors to be fetched.

The line coding and control line entries in this list provide functionality that in the context of a pseudo-USB-serial connection between two machines have no real purpose. They only make sense if there is a genuine UART being controlled at one end.

Example Target Setup

A USB serial device needs some data structures to be defined and initialized. For a target mode device the following example for a notional target shows what needs to be done. This is usually done in a platform specific USB configuration package to associate a hardware peripheral with the serial protocol driver. This example shows a CDC ACM device, although the same approach should serve for any target protocol.

```
//=============================================================================
// USB serial device ACM0

#define USB_SUBSYSTEM USB_SUBSYSTEM_PCD

#include <cyg/usb/usb.h>
#include <pkgconf/io_usb_cdc_acm.h>
#include <cyg/usb/usb_serial.h>
#include <cyg/usb/cdc_acm.h>

#include CYGDAT_IO_USB_SERIAL_DEVICE_HEADER
```

```
//-----------------------------------------------------------------------------
// Connection calls
//
// Functions that start with usb_serial are supplied by the serial driver.
// Functions that start with cdc_acm are supplied by the CDC ACM protocol
// driver and would be substituted with functions for another protocol
// driver.

static const usb_serial_calls example_serial_calls =
{
    .init                        = cdc_acm_init,
    .attach                      = usb_serial_attach,
    .detach                      = usb_serial_detach,
    .send_data_start             = usb_serial_send_data_start,
    .send_data                   = cdc_acm_send_data,
    .send_data_done              = usb_serial_send_data_done,
    .recv_data                   = cdc_acm_recv_data,
    .recv_data_done              = usb_serial_recv_data_done,
    .dev_line_coding             = usb_serial_line_coding,
    .dev_control_line_state      = usb_serial_control_line_state,
    .usb_line_coding             = cdc_acm_line_coding,
    .usb_control_line_state      = cdc_acm_control_line_state,
    .usb_get_descriptor          = cdc_acm_get_descriptor,
};

//-----------------------------------------------------------------------------
// Interface object
//
// Preceeded by some forward definitions and the declaration of the receive
// buffer.

static usb_target example_acm0_target;
static serial_channel example_acm0;

static usb_uint8 example_acm0_rx_buf[CYGNUM_IO_USB_CDC_ACM_MAXPKT];

static usb_serial_if example_acm0_serial_if =
{
    .tgt                         = &example_acm0_target,
    .call                        = &example_serial_calls,
    .chan                        = &example_acm0,
    .target.pcdi_name            = "usb_fs",

    .rx_buf                      = example_acm0_rx_buf,
    .rx_maxpkt                   = CYGNUM_IO_USB_CDC_ACM_MAXPKT,
};

//-----------------------------------------------------------------------------
// USB target
//
// The CDC ACM protocol driver supplies the device, configuration and string
// descriptors, However, string descriptor 3, the serial number, is not provided
// and must be generated by a call to the target get_descriptor callback. The
// control and new_state callbacks are also supplied by the protocol driver.

static int example_acm0_get_descriptor(usb_target *tgt, usb_uint8 type, usb_uint8 index,
                                       usb_uint8 **buf, usb_uint16 *len );

static usb_target example_acm0_target =
{
    .desc               = &cdc_acm_device_descriptor,

    .configs            = cdc_acm_config_descriptors,
    .config_count       = 1,

    .strings            = cdc_acm_string_descriptors,
    .string_count       = 4,
    .get_descriptor     = example_acm0_get_descriptor,

    .control            = cdc_acm_control,
    .new_state          = cdc_acm_new_state,

    .data               = &example_acm0_serial_if,
};

//-----------------------------------------------------------------------------
// Serial channel
//
// This is the standard serial device channel structure, and needs to be
// initialized in the standard way with default settings and transmit and
// receive buffers.

// The baud rate is irrelevant, but we must choose a default value
#define CYGNUM_DEVS_USB_EXAMPLE_ACM0_BAUD         9600

static unsigned char example_acm_out_buf0[CYGNUM_DEVS_USB_EXAMPLE_ACM0_BUFSIZE];
static unsigned char example_acm_in_buf0[CYGNUM_DEVS_USB_EXAMPLE_ACM0_BUFSIZE];

static SERIAL_CHANNEL_USING_INTERRUPTS(example_acm0,
                                       usb_serial_funs,
                                       example_acm0_serial_if,
                                       CYG_SERIAL_BAUD_RATE(CYGNUM_DEVS_USB_EXAMPLE_ACM0_BAUD),
                                       CYG_SERIAL_STOP_DEFAULT,
                                       CYG_SERIAL_PARITY_DEFAULT,
                                       CYG_SERIAL_WORD_LENGTH_DEFAULT,
                                       CYG_SERIAL_FLAGS_DEFAULT,
                                       &example_acm_out_buf0[0], sizeof(example_acm_out_buf0),
                                       &example_acm_in_buf0[0], sizeof(example_acm_in_buf0)
    );

//-----------------------------------------------------------------------------
// Device table entry
//
// This generates an entry in the device table for the ACM0 device.

DEVTAB_ENTRY(example_serial_io0,
             "/dev/acm0",
             0,                     // Does not depend on a lower level interface
             &cyg_io_serial_devio,
             usb_serial_init,
             usb_serial_lookup,     // Serial driver may need initializing
             &example_acm0
    );

//-----------------------------------------------------------------------------
// Descriptor callback
//
// String descriptor 3 is not defined by the CDC ACM driver. Instead it must
// be supplied by a platform-specific callback. The following example simply
// returns a constant descriptor; in real systems, a descriptor may need to be
// synthesized from a board-specific serial number (fetched from flash or EEPROM).
// Any other descriptors are generated by the CDC ACM driver via the
// usb_get_descriptor() callback.

static const usb_string_serial cdc_acm_string_product =
{
    .bLength                            = 2+2*10,
    .bDescriptorType                    = USB_DESC_STRING,
    .bString                            = L"0123456789",
};

static int example_acm0_get_descriptor(usb_target *tgt, usb_uint8 type, usb_uint8 index,
                                       usb_uint8 **buf, usb_uint16 *len )
```

```
{
    int result = USB_OK;

    if( (type == USB_DESC_STRING) && (index == tgt->desc->iSerialNumber) )
    {
        *buf = (usb_uint8 *)&usb_string_serial;
        *len = usb_string_serial->bLength;
    }
    else
    {
        usb_serial_if *usb_if = tgt->data;
        if( usb_if->call->usb_get_descriptor )
            result = usb_if->call->usb_get_descriptor( usb_if, type, index, buf, len );
    }

    return result;
}
```

Example Host Setup

Host mode devices need largely the same set of data structures as for target mode, but initialized in a slightly different way. This is usually done in the USB serial protocol driver where a number of channels will be instantiated. The following shows the data structures for CDC ACM channel 0.

```
//------------------------------------------------------------------------------
// Connection calls
//
// Functions that start with usb_serial are supplied by the serial driver.
// Functions that start with cdc_acm_host are supplied by the CDC ACM host protocol
// driver and would be substituted with functions for another protocol
// driver.

static const usb_serial_calls cdc_acm_host_serial_calls =
{
    .init                        = cdc_acm_host_init,
    .attach                      = usb_serial_attach,
    .detach                      = usb_serial_detach,
    .send_data_start             = usb_serial_send_data_start,
    .send_data                   = cdc_acm_host_send_data,
    .send_data_done              = usb_serial_send_data_done,
    .recv_data                   = cdc_acm_host_recv_data,
    .recv_data_done              = usb_serial_recv_data_done,
    .dev_line_coding             = usb_serial_line_coding,
    .dev_control_line_state      = usb_serial_control_line_state,
    .usb_line_coding             = cdc_acm_host_line_coding,
    .usb_control_line_state      = cdc_acm_host_control_line_state,
};

//------------------------------------------------------------------------------
// Interface object
//
// Preceeded by some forward definitions and the declaration of the receive
// buffer.

static serial_channel cdc_acm0_host;

static usb_uint8 cdc_acm0_host_rx_buf[CDC_ACM_HOST_MAXPKT];

static usb_serial_if cdc_acm0_host_serial_if =
{
    .call                        = &cdc_acm0_host_serial_calls,
    .chan                        = &cdc_acm0_host,

    .rx_buf                      = cdc_acm0_host_rx_buf,
    .rx_maxpkt                   = CDC_ACM_HOST_MAXPKT,
    .host.id                    = __n,
    .host.vid                   = CYGNUM_IO_USB_CDC_ACM_HOST_SERIAL0_VID,
    .host.pid                   = CYGNUM_IO_USB_CDC_ACM_HOST_SERIAL0_PID,
};

//-----------------------------------------------------------------------------
// Serial channel
//
// This is the standard serial device channel structure, and needs to be
// initialized in the standard way with default settings and transmit and
// receive buffers.

static unsigned char cdc_acm_out_buf0_host[CYGNUM_IO_USB_CDC_ACM_HOST_SERIAL0_BUFSIZE];
static unsigned char cdc_acm_in_buf0_host[CYGNUM_IO_USB_CDC_ACM_HOST_SERIAL0_BUFSIZE];

static SERIAL_CHANNEL_USING_INTERRUPTS(cdc_acm0_host,
                                       usb_serial_funs,
                                       cdc_acm0_host_serial_if,
                                       CYG_SERIAL_BAUD_RATE(CYGNUM_IO_USB_CDC_ACM_HOST_SERIAL0_BAUD),
                                       CYG_SERIAL_STOP_DEFAULT,
                                       CYG_SERIAL_PARITY_DEFAULT,
                                       CYG_SERIAL_WORD_LENGTH_DEFAULT,
                                       CYG_SERIAL_FLAGS_DEFAULT,
                                       &cdc_acm_out_buf0_host[0], sizeof(cdc_acm_out_buf0_host),
                                       &cdc_acm_in_buf0_host[0], sizeof(cdc_acm_in_buf0_host)
    );

//-----------------------------------------------------------------------------
// Device table entry
//
// This generates an entry in the device table for the ACM0 device.

DEVTAB_ENTRY(cdc_acm_host_serial_io0,
             CYGPKG_IO_USB_CDC_ACM_HOST_SERIAL0_NAME,
             0,
             &cyg_io_serial_devio,
             usb_serial_init,
             usb_serial_lookup,
             &cdc_acm0_host
    );
```

Chapter 119. USB Target CDC ACM Protocol Driver

Name

Overview — eCosPro Support for CDC ACM Protocol in Peripheral Mode

Description

This package provides support for a peripheral mode serial connection using the CDC ACM protocol. It needs to be used in conjunction with the USB serial driver package and the reader is referred to that document for additional details.

Configuration

To use this package, various other packages need to be included and configuration options need to be set. The packages that need including are CYGPKG_IO_USB, CYGPKG_IO_SERIAL_USB, CYGPKG_IO_SERIAL and this package CYGPKG_IO_USB_CDC_ACM. To get USB target and serial support running, the options CYGPKG_IO_USB_TARGET and CYGPKG_IO_SERIAL_DEVICES need to be enabled. So long as the hardware USB configuration package defines the target data structures described in the USB serial driver package, a CDC/ACM peripheral will be presented on the selected device port.

The configuration options CYGNUM_IO_USB_CDC_ACM_VID and CYGNUM_IO_USB_CDC_ACM_PID *must* be configured, respectively, with suitable VendorID and ProductID values. These values are used by hosts to identify a specific USB device. The www.usb.org site provides more information, specifically the Getting a Vendor ID page.

The configuration options CYGPKG_IO_USB_CDC_ACM_MANUFACTURER and CYGPKG_IO_USB_CDC_ACM_PRODUCT allow for human-readable identification strings to be supplied by the device. These strings are returned as part of the device USB description, and may be used by the host O/S in its description of the product as presented to end-users.

The configuration option CYGNUM_IO_USB_CDC_ACM_MAXPKT defines the maximum packet size used for USB transfers. It is used to define fields in the CDC ACM descriptors and must be used to define the size of the the *rx_buf* in the usb_serial_if structure.

Configuration Packages

The USB driver configuration packages for some target families (e.g. AT91 and STM32) contain configuration for a CDC ACM device instance. These have some common configuration options. In the following descriptions, XXXX should be replaced by the target family name.

The option CYGNUM_DEVS_USB_XXXX_ACM0_BUFSIZE defines the size of the circular buffer for ACM0. This must be sized such that there is always enough space in the buffer for a maximum sized USB packet after the high water mark is reached.

The option CYGPKG_DEVS_USB_XXXX_ACM0_SERIALNO defines the source of the serial number reported in USB string descriptor 3. The CSUM option generates an executable-unique checksum, which is necessary for the eCosCentric test system. The HAL option calls a function supplied by the platform HAL to supply the serial number. The CONST option uses the value of CYGINT_DEVS_USB_XXXX_ACM0_SERIALNO, which is useful for testing. In all cases the serial number is a 32 bit unsigned value which is encoded into the string descriptor in hexadecimal combined with the test crash ID.

The option CYGINT_DEVS_USB_XXXX_ACM0_SERIALNO supplies the constant value for the serial number if the CONST option is selected. This is useful for testing, but is not a viable solution for production devices.

Protocol Support

This driver only supports a minimal subset of the CDC ACM protocol, sufficient to provide a serial-like interface between a host and the target board. Any unrecognized messages will generate an error response; normally a STALL on the control endpoint.

CDC ACM uses two bulk endpoints to transfer raw data bytes between the host and the target board. Additionally, control messages are sent to the control endpoint to adjust the configuration of the channel. The following commands are supported:

`SET LINE CODING`
: This sets the baud rate, stop bits, parity and character size for the channel. These parameters are simply passed on to the USB serial driver by calling the `dev_line_coding` callback.

`GET LINE CODING`
: This returns the baud rate, stop bits, parity and character size for the channel. Usually this just returns the value set by the last `SET LINE CODING` command. The data structure is initialized to a default set of values in case it is queried before being set.

`SET CONTROL LINE STATE`
: This sets the state of the RTS and DTR lines. These are passed on to the USB serial driver by calling the `dev_control_line_state` callback.

`SEND BREAK`
: This sends a BREAK condition on the line. At present this command is accepted and acknowledged but nothing is done to act on it.

Since the protocol driver is not controlling a real UART, these commands have no real effect and are supported mainly to keep host OS drivers happy.

Host Driver Advice

The Linux operating system includes generic support for CDC ACM class devices and automatically supports standards conforming CDC ACM based devices.

Whilst Windows also includes built-in generic CDC ACM support, it has to be enabled with a specially crafted .inf file. You may alternatively elect to use a third party CDC ACM class driver as this can provide better functionality, reliability and install support compared to the Windows built-in driver.

In-depth information concerning Windows USB class support along with the use, configuration and installation of associated .inf files, are beyond the scope of this documentation. The Microsoft Developer Network documentation should be consulted for specific details. Relevant sections include Windows USB support and USB class drivers included in Windows.

As an aid in the use of Windows built-in CDC ACM class support an example .inf file has been provided here: `packages/devs/usb/pcd/class/cdc_acm/<version>/host/cdcacm-generic.inf`

The .inf file will need to be tailored to match your specific manufacturer VID and PID numbers, as well as manufacturer and device description strings. You will need to modify all the VID and PID references in the [DeviceList] sections, as well as the MFGNAME and DESCRIPTION strings at the end of the file. To enable CDC ACM on a specific Windows system the user should right-click on the .inf file and select the "Install" option from the pop-up menu. A Windows driver validation dialog box will appear, from which the "Install this driver software anyway" option should be selected.

Depending on the Windows version you are using, the CDC ACM driver installation may not be automatic on the first connection of your CDC ACM class device. An error may be reported in finding a suitable device driver during device insertion, or the Device Manager may show your device, but with an associated warning triangle. In these cases you will need to manually select the correct class driver. To accomplish this, right-click on the new entry in the "Ports" section of the Device Manager and select "Update Driver Software" from the pop-up menu. In the following dialog select "Browse my computer for driver software". You should then select the "Let me pick from a list of device drivers on my computer" from the available options that are then displayed. At this point you will then be presented with a list of device types - select "Ports (COM & LPT)" and then the "Next" button. You should then be able to find your company name in the Manufacturer list, and the Model associated with your driver entry - select this and hit the "Next" button. A Warning dialog concerning device verification will appear, select "Yes" to continue and install the driver. A confirmation message indicating the driver has installed should then appear.

Each time an eCos CDC ACM device is plugged in an entry should become visible within the "Ports" section of the Device Manager. The entry's device name should include an associated automatically assigned COM port number in brackets. You should be able to connect to this assigned COM port to communicate with your device.

Chapter 120. USB Host CDC ACM Protocol Driver

Name

Overview — eCosPro Support for CDC ACM Protocol in Host Mode

Description

This package provides protocol driver support for host mode serial adapter connections that use the CDC ACM protocol. It needs to be used in conjunction with the USB serial driver package and the reader is referred to that document for additional details.

Usage Model

The eCos serial driver subsystem does not support the dynamic creation and deletion of serial devices. Instead the CDC/ACM driver allocates a number of permanently available serial channels which are allocated to USB devices as they are attached. In order to retain some continuity, it is possible to assign specific Vendor and Product ID values to individual channels so that the USB device will be allocated the same channel each time it is attached.

Writing data to a detached channel will result in that data being lost, just as if it were being written to a disconnected serial line. A detached channel will not produce any data. Changes to the line configuration (baud rate, parity, stop bits, data size) will be stored and applied to the device once it is attached.

Applications may install a serial line callback function which will be called with the *which* field set to CYGNUM_SERIAL_STATUS_ATTACH and the *value* field set to zero for a detach and one for an attach.

Configuration

To use this package, various other packages need to be included and configuration options need to be set. The packages that need to be included are: USB Support (CYGPKG_IO_USB), USB serial driver (CYGPKG_IO_SERIAL_USB), Serial device drivers (CYGPKG_IO_SERIAL) and this package USB host cdc acm protocol driver (CYGPKG_IO_USB_CDC_ACM_HOST). Packages can be added directly using **ecosconfig add** on the command line, or the **Build->Packages…** menu item within the eCos Configuration Tool. Depending on your platform some of these may already be present in the default eCos configuration. To activate the requisite USB host and serial support, the options CYGPKG_IO_USB_HOST and CYGPKG_IO_SERIAL_DEVICES need to be enabled. Following this at least one CDC/ACM channel needs to be configured using the options below.

The configuration option CYGNUM_IO_USB_CDC_ACM_MAXPKT defines the maximum packet size used for USB transfers. It is used define the size of the the *rx_buf* in the usb_serial_if structure. The actual packet size used by any device is specified in its descriptors. Most CDC/ACM devices will use a maximum packet size of 64 bytes, and many will be smaller. So this value should not be reduced unless it is known that only devices with smaller packet sizes will be used.

Each serial channel has a number of configuration options associated with it. The following descriptions show the options for serial channel 0, for other channels the zero should be replaced with the number of the channel, currently up to 4.

CYGINT_IO_USB_CDC_ACM_HOST_SERIAL0

> This interface may be implemented by the platform HAL to instantiate this serial channel.

CYGPKG_IO_USB_CDC_ACM_HOST_SERIAL0

> This is the main component that defines this serial channel, unless this component is enabled, the remaining options will remain undefined. Its default value is derived from CYGINT_IO_USB_CDC_ACM_HOST_SERIAL0, but it may also be enabled with a requires statement or from the configtool.

CYGNUM_IO_USB_CDC_ACM_HOST_SERIAL0_NAME

> This option controls the name that an eCos application should use to access this device via cyg_io_lookup(), open(), or similar calls. The default is "/dev/acm0" and so on.

`CYGNUM_IO_USB_CDC_ACM_HOST_SERIAL0_BAUD`

This option specifies the default baud rate for this channel. Its default value is set to `CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL_BAUD`.

`CYGNUM_IO_USB_CDC_ACM_HOST_SERIAL0_BUFSIZE`

This option specifies the size of both the input and output buffer for the common serial I/O driver layer. This should be at least equal to `CYGNUM_IO_USB_CDC_ACM_HOST_MAXPKT` and preferably somewhat larger. The default value makes it twice the maximum packet size.

`CYGNUM_IO_USB_CDC_ACM_HOST_SERIAL0_VID`

This option controls whether this serial channel matches a particular USB Vendor ID. If zero it will match any ID, otherwise this channel will only match a device with the given vendor ID.

`CYGNUM_IO_USB_CDC_ACM_HOST_SERIAL0_PID`

This option controls whether this serial channel matches a particular USB Product ID. If zero it will match any ID, otherwise this channel will only match a device with the given product ID.

Channels are searched for a VID and PID match in numerical order, skipping any that are already in use. Therefore lower numbered channels should have more specific VID and PID values and generic channels should be at the end of the list.

Protocol Support

This driver only supports a subset of the CDC ACM protocol, sufficient to provide a serial-like interface between the host and the device.

CDC ACM uses two bulk endpoints to transfer raw data bytes between the host and the target board. Additionally, control messages are sent to the control endpoint to adjust the configuration of the channel. Only the following commands are currently sent:

`SET LINE CODING` This sets the baud rate, stop bits, parity and character size for the channel. This is generated in response to a `CYG_IO_SET_CONFIG_SERIAL_INFO` set_config key from the application.

Supported Devices

Any USB serial adapter that implements the standard USB-IF defined CDC ACM protocol should be compatible with the eCosPro host CDC ACM protocol driver. The driver has been tested successfully with the Microchip MCP2200 chip based Microchip MCP2200EV-VCP evaluation board.

Chapter 121. USB Host FTDI Protocol Driver

Name

Overview — eCosPro Support for FTDI Protocol in Host Mode

Description

This package provides protocol support for a host mode connection to FTDI USB to serial adaptors. It needs to be used in conjunction with the USB serial driver package and the reader is referred to that document for additional details.

Usage Model

The eCos serial driver subsystem does not support the dynamic creation and deletion of serial devices. Instead the FTDI driver allocates a number of permanently available serial channels which are allocated to USB devices as they are attached. In order to retain some continuity, it is possible to assign specific Vendor and Product ID values to individual channels so that the USB device will be allocated the same channel each time it is attached.

Writing data to a detached channel will result in that data being lost, just as if it were being written to a disconnected serial line. A detached channel will not produce any data. Changes to the line configuration (baud rate, parity, stop bits, data size, flow control) will be stored and applied to the device once it is attached.

Applications may install a serial line callback function which will be called with the *which* field set to `CYGNUM_SERIAL_STATUS_ATTACH` and the *value* field set to zero for a detach and one for an attach.

Configuration

To use this package, various other packages need to be included and configuration options need to be set. The packages that need to be included are: USB Support (`CYGPKG_IO_USB`), USB serial driver (`CYGPKG_IO_SERIAL_USB`), Serial device drivers (`CYGPKG_IO_SERIAL`) and this package USB host ftdi protocol driver (`CYGPKG_IO_USB_FTDI`). Packages can be added directly using **ecosconfig add** on the command line, or the **Build->Packages…** menu item within the eCos Configuration Tool. Depending on your platform some of these may already be present in the default eCos configuration. To activate the requisite USB host and serial support, the options `CYGPKG_IO_USB_HOST` and `CYGPKG_IO_SERIAL_DEVICES` need to be enabled. Following this at least one FTDI channel needs to be configured using the options below.

The configuration option `CYGNUM_IO_USB_FTDI_MAXPKT` defines the maximum packet size used for USB transfers. It is used to define the size of the the *rx_buf* in the usb_serial_if structure. The actual packet size used by any device is specified in its descriptors. Most FTDI devices will use a maximum packet size of 64 bytes, and many will be smaller. So this value should not be reduced unless it is known that only devices with smaller packet sizes will be used.

The option `CYGPKG_IO_USB_FTDI_SUPPORTED_EXTRA` defines an additional set of VID/PID pairs that the driver can support. A user or HAL package can define this to add entries to the table of VID/PID pairs that the driver recognizes as valid devices. This option consists of a comma separated list of initializers for for a VID/PID structure. For example "{ 0x0403, 0x6001 }, …".

Each serial channel has a number of configuration options associated with it. The following descriptions show the options for serial channel 0, for other channels the zero should be replaced with the number of the channel, currently up to 4.

`CYGINT_IO_USB_FTDI_SERIAL0`

This interface may be implemented by the platform HAL to instantiate this serial channel.

`CYGPKG_IO_USB_FTDI_SERIAL0`

This is the main component that defines this serial channel, unless this component is enabled, the remaining options will remain undefined. Its default value is derived from `CYGINT_IO_USB_FTDI_SERIAL0`, but it may also be enabled with a requires statement or from the configtool.

`CYGNUM_IO_USB_FTDI_SERIAL0_NAME`

This option defines the name of this channel, which eCos applications should use to access the device via cyg_io_lookup(), open() or similar calls. The default is to set it to "/dev/ftdi0" and so on.

`CYGNUM_IO_USB_FTDI_SERIAL0_BAUD`

This option specifies the default baud rate for this channel. Its default value is set to `CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL_BAUD`.

`CYGNUM_IO_USB_FTDI_SERIAL0_BUFSIZE`

This option specifies the size of both the input and output buffer for the common serial I/O driver layer. This should be at least equal to `CYGNUM_IO_USB_FTDI_MAXPKT` and preferably somewhat larger. The default value makes it twice the maximum packet size.

`CYGNUM_IO_USB_FTDI_SERIAL0_VID`

This option controls whether this serial channel matches a particular USB Vendor ID. If zero it will match any ID, otherwise this channel will only match a device with the given vendor ID.

`CYGNUM_IO_USB_FTDI_SERIAL0_PID`

This option controls whether this serial channel matches a particular USB Product ID. If zero it will match any ID, otherwise this channel will only match a device with the given product ID.

Channels are searched for a VID and PID match in numerical order, skipping any that are already in use. Therefore lower numbered channels should have more specific VID and PID values and generic channels should be at the end of the list.

`CYGNUM_IO_USB_FTDI_SERIAL0_LATENCY`

This option defines the latency in milliseconds between packets containing data from the FTDI device. If the device has 62 or more bytes available, it will send a packet immediately. Otherwise it will wait this number of milliseconds after the last packet before sending what it has. Decreasing this value will improve responsiveness for low data rate applications at the cost of more activity from the host driver as it processes more packets.

Supported Devices

There is a large family of FTDI devices, some of which support slightly different variants of the communication protocol. It is therefore important to detect what kind of device has been attached. FTDI devices do not have standard class. subclass or protocol codes defined by the USB-IF. Device descriptors have these fields set to zero, to redirect them to the interfaces. The interface fields are set to `0xFF`, which is a reserved vendor specific code. Additionally, FTDI devices can have internal or external PROMs from which `idVendor`, `idProduct` and string descriptors may be fetched. The result of all this is that the exact identity of an FTDI device can be difficult to determine.

The eCosPro FTDI driver employs a number of strategies to determine whether a device is one that can be supported. The first part of this is a table of VID/PID pairs. Only if a device can be found in this table will it be accepted. By default the table contains the standard FTDI VID/PID values and will be extended to contain the values for any other devices that have been tested. There is also a mechanism that allows users or other packages to add entries to this table using the `CYGPKG_IO_USB_FTDI_SUPPORTED_EXTRA` option.

If a device passes the VID/PID test, then other descriptor values such as `bcdDevice` `bNumInterface` are used to determine which FTDI device model this is likely to be and adjust the protocol used to talk to it accordingly.

The eCos FTDI driver has been successfully tested against the following standard FTDI USB serial adapters:

- FTDI CHIPI-X10 (FT231X based)

- FTDI US232R (FT232RL based)
- FTDI US232B (FT232BM based)

In addition an older FT232R and a FT2232C dual channel based devices from manufacturers other than FTDI have also been tested.

Part XXXVI. VirtIO Support

Name

Overview — eCosPro Support for VirtIO

Description

The eCosPro VirtIO package supports access to VirtIO devices. It provides general management of the device and the buffer queues associated with it. Normally a next-level driver will use the facilities provided by this driver to then implement an eCos compliant driver for a particular class of device.

A VirtIO device is described by a cyg_vio_driver structure. Data transfers are described by a cyg_vio_tfr structure. Operations are handled through an API that is used by the class drivers.

VirtIO Device Structure

A VirtIO device is instantiated by creating one of these structures and populating its fields with suitable values. The following fields need to be set; other fields in the structure will be initialized by the VirtIO driver.

base
Base address of the device.

vector
Device interrupt vector. Interrupts are not handled by the VirtIO driver, instead they need to be handled by the parent class driver. See the API description for details.

vector_pri
Interrupt vector priority.

priv
A private pointer for the parent driver. This will usually be a pointer to a data structure that the driver uses to store state for this device.

legacy
If set to 1, this indicates that the device is a legacy device. At present only legacy devices are supported.

pci
If set to 1 this indicates that this is a PCI device. At present PCI devices are not supported.

drv_features
A bit mask corresponding to the driver feature bits defined for this class of driver. This will be set in the DRV_FEATURES field of the device during feature negotiation.

queue_count
The number of queues in the queue array.

queue
A pointer to an array of pointers to cyg_vio_queue structures. Queues are defined using the `VIO_QUEUE(__name, __size)` macro and then collected together into an array which is pointed to by this field.

The following example shows how a VirtIO console driver would be instantiated:

```

// Define driver feature set
#define VIO_CONSOLE_FEATURES            VIRTIO_F_RING_INDIRECT_DESC     | \
                                        VIRTIO_F_NOTIFY_ON_EMPTY        | \
                                        VIRTIO_F_ANY_LAYOUT             | \
                                        VIRTIO_CONSOLE_F_MULTIPORT      | \
                                        VIRTIO_CONSOLE_F_EMERG_WRITE

// Set queue size
#define CONSOLE_QUEUE_SIZE            128

// Define queues. Here we have two console channels and a control channel.
VIO_QUEUE( virtual_console_rxq0, CONSOLE_QUEUE_SIZE );
VIO_QUEUE( virtual_console_txq0, CONSOLE_QUEUE_SIZE );
VIO_QUEUE( virtual_control_rxq, CONSOLE_QUEUE_SIZE );
VIO_QUEUE( virtual_control_txq, CONSOLE_QUEUE_SIZE );
VIO_QUEUE( virtual_console_rxq1, CONSOLE_QUEUE_SIZE );
VIO_QUEUE( virtual_console_txq1, CONSOLE_QUEUE_SIZE );

// Collect queues together into an array
cyg_vio_queue *virtual_console_queues[] = { &virtual_console_rxq0, &virtual_console_txq0,
                                         &virtual_control_rxq, &virtual_control_txq,
                                         &virtual_console_rxq1, &virtual_console_txq1 };

// Declare the driver
cyg_vio_driver console_vio_driver =
{
     // Hardware parameters
    .base               = CYGHWR_HAL_CONSOLE0_BASE,
    .vector             = CYGNUM_HAL_INTERRUPT_CONSOLE0,
    .vector_pri         = 0xa0,

    // Console driver private data
    .priv               = &virtual_console,

    // This is a legacy, non PCI device
    .legacy             = 1,
    .pci                = 0,

    // Set driver features for use in negotiation
    .drv_features       = VIO_CONSOLE_FEATURES,

    // Attach queues.
    .queue_count        = 6,
    .queue              = virtual_console_queues,

};
```

VirtIO Transfer Structure

Data transfers are described using a cyg_vio_tfr structure. The parent driver prepares a transfer object, passes it to the VirtIO driver and receives it back via a callback when the transfer is complete.

A transfer object has the following structure:

```
#define VIO_IOV_MAX             8

struct cyg_vio_tfr
{
    cyg_uint16          queue;          // Queue number
    cyg_uint16          head;           // head descriptor index
    cyg_uint16          iov_len;        // IOV entry count
    cyg_uint32          actual;         // Total actual bytes transferred

    void                (*callback)( cyg_vio_tfr *tfr );
    void                *priv;          // Private data pointer

    struct
    {
        void            *buffer;
        cyg_uint32      size;
        cyg_uint32      flags;
    } iov[VIO_IOV_MAX];
};

#define VIO_IOV_FLAGS_WRITE             CYGHWR_VIRTIO_DESC_FLAGS_WRITE
```

The fields are as follows:

queue
: The index in the queue array of the queue to apply the transfer to.

head
: This is used to store the index of the head buffer descriptor associated with this transfer. It does not need to be set by the client, but may be monitored to detect transfer completion, it will be set to 0xFFFF when the transfer is done.

iov_len
: The number of entries in the *`iov`* field that are in use.

actual
: The actual number of bytes transferred. This is the value of the used queue element length field and therefore depends on the hypervisors device emulation to set it correctly.

callback
: A function that is called when the transfer is completed. The single argument is a pointer to the completed transfer. This will only be called from within the `cyg_vio_poll()`.

priv
: A private pointer that can be used by the parent driver to supply context to the callback.

iov
: An array of buffer pointers with their sizes. The *`flags`* field for each will either be zero, or contain the `VIO_IOV_FLAGS_WRITE` flag. If the flag is zero, then this buffer is for transfer from the VM to the hypervisor, and if set, then the buffer is for transfer from the hypervisor to the VM.

: While the default size of this array is 8, this is not a fixed limit. Larger arrays could be passed by treating a transfer object as an initial substructure of a larger object that contains space for a longer IOV array.

Name

API — Functions

Synopsis

#include <cyg/io/virtio.h>

```
cyg_uint32 cyg_vio_avail(cyg_vio_driver *driver, int queue);

cyg_bool cyg_vio_queue_ready(cyg_vio_driver *driver, int queue);

cyg_bool cyg_vio_submit(cyg_vio_driver *driver, cyg_vio_tfr *tfr);

void cyg_vio_poll(cyg_vio_driver *driver);

void cyg_vio_driver_init(cyg_vio_driver *driver);

void hal_vio_init(void);
```

Description

This API is intended to be used by client drivers to access the VirtIO device and provide the functionality expected of a driver of the given class.

Functions `cyg_vio_avail()` and `cyg_vio_queue_ready()` test the state of the queue. The first returns the number of buffer descriptors available for transfer; it can be used to check that there is enough resource to start a transfer before submitting it. The second is used to check that a queue has completed initialization.

The function `cyg_vio_submit()` submits a transfer to the VirtIO device. All fields in the transfer should be initialized before submission. If the transfer is successfully queued, this function returns `CYG_VIO_DONE`. If the submission fails, a non-zero error code is returned.

The function `cyg_vio_poll()` polls a given driver for completed transfers. If a transfer is complete, then its callback function is called. Calling this poll routine is the only way in which transfer completions are recognized. It is the responsibility of the client driver to arrange to call it. This may be done from a thread context if that exists, or may be done from a DSR if the device interrupt has been enabled. When a callback is called, the only fields in the transfer that will have been updated are the *head* and *actual* fields; so the transfer may be immediately resubmitted to the driver from the callback with no changes if the same transfer is to be repeated.

The function `cyg_vio_driver_init()` is called to initialize the common parts of a VirtIO driver. This function will perform startup negotiation with the hypervisor device and initialize all valid queues. On return, the driver will be ready for submission of transfers. If this function is called for a driver that has already been initialized, it will return immediately, so it may be called from multiple locations safely.

The function `hal_vio_init()` is not supplied by the VirtIO package but is expected to be defined by the variant or platform HAL. The VirtIO package calls this function from a constructor during initialization. This function is responsible for detecting any VirtIO devices, installing base address and interrupt vector values and calling `cyg_vio_driver_init()` for each. Detection may involve searching a memory area for valid VirtIO devices or scanning a PCI bus. If the VirtIO devices are at known fixed addresses then this function should just call `cyg_vio_driver_init()` for each device to be initialized.

Part XXXVII. Wallclock Device Drivers

Documentation for drivers of this type is often integrated into the eCos board support documentation. You should review the documentation for your target board for details. Standalone and more generic drivers are documented in the following sections.

Table of Contents

Chapter 122. Wallclock Support

Name

CYGPKG_IO_WALLCLOCK — eCos Support Wallclock devices

Overview

The wallclock device provides real time stamps, as opposed to the eCos kernel timers which typically just count the number of clock ticks since the hardware was powered up. Depending on the target platform this device may involve interacting with a suitable on-chip device or an external clock chip, or it may be emulated by using the kernel timers.

The wallclock package operates using the standard UNIX epoch of midnight 1st January 1970. When times and dates are expressed in seconds, this is zero point for that count. However, many wallclock devices only have a two digit year field. In order to get a further 30 years duration from this field, and to avoid Y2K issues, many drivers actually store dates starting from the year 2000. It is therefore inavisable to try setting the wallclock to any date before 2000.

Configuration

The Wallclock package contains a number of configuration options, most of which are set by either the wallclock device driver, or the platform HAL.

CYGINT_WALLCLOCK_HW_IMPLEMENTATIONS

> This interface is implemented by the wallclock device driver to signal to this package that a hardware wallclock device is present. If this interface is zero, then this package will implement the wallclock using the kernel timer.

CYGINT_WALLCLOCK_SET_GET_MODE_SUPPORTED

> This interface is implement by the wallclock device driver to signal to this package that the device can set the current value as well as get it and that the set value is preserved when the power is off.

CYGSEM_WALLCLOCK_MODE

> The wallclock driver can be used in one of two modes. Set/get mode allows time to be kept during power off (assuming there's a battery backed clock). Init/get mode is slightly smaller and can be used when there is no battery backed clock - in this mode time 0 is the time of the board power up. The default value of this option depends on whether CYGINT_WALLCLOCK_SET_GET_MODE_SUPPORTED but may be changed by the user.

CYGPKG_WALLCLOCK_EMULATE

> When this option is enabled, a wallclock device will be emulated using the kernel real-time clock. The default value depends on the value of CYGINT_WALLCLOCK_HW_IMPLEMENTATIONS but may be changed by the user.

CYGIMP_WALLCLOCK_NONE

> This option disables the wallclock even if a hardware driver is present. The default value is depends on the value of CYGINT_WALLCLOCK_HW_IMPLEMENTATIONS.

CYGINT_IO_WALLCLOCK_HAS_SCRATCHSPACE

> If the underlying wallclock driver implements this interface, that means that it supports scratch space. This feature exists in some hardware to allow users to store some information in battery backed RAM, alongside the wallclock RTC's data. An API is provided by this package to access it.

CYGINT_IO_WALLCLOCK_HAS_ALARM

> Some wallclock hardware provides alarms which will generate an interrupt when a certain date/time is reached. This interface can be implemented by the underlying wallclock driver to indicate that such support can be used. An API is provided by this package to set and disable these alarms. Note these alarms are wholly separate from eCos kernel alarms.

Wallclock Tests

This package contains a number of test programs. The **wallclock** and **wallclock2** programs test basic functionality of the wallclock device. The **alarm** program tests the functionality of any alarms supported by the device. The **subsec** program tests the functionality and accuracy of device subsecond support if it is present.

Name

C API — Details

Synopsis

#include <cyg/io/wallclock.h>

`cyg_uint32` **`cyg_wallclock_get_current_time`**`(`*`void`*`);`

`void` **`cyg_wallclock_set_current_time`**`(`*`cyg_uint32 time_stamp`*`);`

`Cyg_ErrNo` **`cyg_wallclock_get_time_timespec`**`(`*`struct timespec *tp`*`);`

`Cyg_ErrNo` **`cyg_wallclock_set_time_timespec`**`(`*`struct timespec *tp`*`);`

`Cyg_ErrNo` **`cyg_wallclock_get_time_date`**`(`*`cyg_uint16 *year`*`,` *`cyg_uint8 *month`*`,` *`cyg_uint8 *day`*`,` *`cyg_uint8 *hour`*`,` *`cyg_uint8 *min`*`,` *`cyg_uint8 *sec`*`,` *`cyg_uint32 *nsec`*`);`

`Cyg_ErrNo` **`cyg_wallclock_set_time_date`**`(`*`cyg_uint16 year`*`,` *`cyg_uint8 month`*`,` *`cyg_uint8 day`*`,` *`cyg_uint8 hour`*`,` *`cyg_uint8 min`*`,` *`cyg_uint8 sec`*`,` *`cyg_uint32 nsec`*`);`

`Cyg_ErrNo` **`cyg_wallclock_get_info`**`(` *`wallclock_info_key key`*`,` *`wallclock_info *info`*`);`

`Cyg_ErrNo` **`cyg_wallclock_set_alarm_timespec`**`(`*`cyg_uint8 alarm_index`*`,` *`struct timespec *alarm_tp`*`);`

`Cyg_ErrNo` **`cyg_wallclock_set_alarm_date`**`(`*`cyg_uint8 alarm_index`*`,` *`cyg_uint16 year`*`,` *`cyg_uint8 month`*`,` *`cyg_uint8 day`*`,` *`cyg_uint8 hour`*`,` *`cyg_uint8 min`*`,` *`cyg_uint8 sec`*`,` *`cyg_uint32 nsec`*`);`

`Cyg_ErrNo` **`cyg_wallclock_disable_alarm`**`(`*`cyg_uint8 alarm_index`*`);`

`Cyg_ErrNo` **`cyg_wallclock_read_scratch`**`(`*`cyg_uint32 offset`*`,` *`cyg_uint8 *buf`*`,` *`cyg_uint32 len`*`);`

`Cyg_ErrNo` **`cyg_wallclock_write_scratch`**`(`*`cyg_uint32 offset`*`,` *`cyg_uint8 *buf`*`,` *`cyg_uint32 len`*`);`

Description

The wallclock package exports a C API for interacting directly with the wallclock. This API is a veneer over a C++ API which is used internally within eCos. Applications should generally use the C API. Wallclock support is also integrated into the POSIX and C++ library packages and if these are part of the configuration the APIs provided by these libraries should be used in preference to the API described here. This API will sidestep any mechanisms present in these other packages for maintenance of the current time and may give rise to inconsistencies between the times that different parts of the system percieve.

The main part of the API consists of functions to set and get the current wallclock time. There are three versions of each set and get function which take the time to be set or got in different formats. The functions `cyg_wallclock_get_current_time` and `cyg_wallclock_set_current_time` use a timestamp consisting of seconds since the epoch; no fractions of a second are supported. The functions `cyg_wallclock_get_time_timespec` and `cyg_wallclock_set_time_timespec` use a struct timespec consisting of seconds and nanoseconds values; the nanoseconds value is only used or returned non-zero if the underlying wallclock device supports sub-second resolution. Finally, the `cyg_wallclock_get_time_date` and `cyg_wallclock_set_time_date` use a date and time broken down into its component parts from the year down to nanoseconds; again the nanoseconds argument is only used or returned non-zero if sub-second support is present.

The function `cyg_wallclock_get_info` returns information about the wallclock device. The key may be one of the following values:

CYG_WALLCLOCK_INFO_RES

Resolution in microseconds/tick (NB different from kernel). Returns in "resolution" member of wallclock_info.

CYG_WALLCLOCK_INFO_MAXYEAR

The maximum year supported. This is a rough hint of the exact date. Returned in uint32val.

CYG_WALLCLOCK_INFO_GET_SCRATCH_SIZE

Many modern RTCs have a battery backed scratch space accompanying the RTC. If there is one, this returns its size in bytes in the "uint32val" member of the wallclock_info. If unsupported, either ENOSUPP is returned, or size may be set to 0.

CYG_WALLCLOCK_INFO_GET_NUM_ALARMS

Number of alarms available. Number returned in the "uint32val" member of wallclock_info. If unsupported, either ENOSUPP returned, or uint32val set to 0.

CYG_WALLCLOCK_INFO_GET_ALARM_INTVEC

Must be called with an alarm index number in the "uint32val" member of supplied wallclock_info argument. Will return the interrupt vector number for that alarm in the same "uint32val" member. If there is no alarm or no interrupt for it, ENOSUPP is returned.

CYG_WALLCLOCK_INFO_GET_SUBSECOND_FRACTION

Sub-second fractions supported. If the wallclock driver supports sub-second resolution, this returns in "uint32val" the number of fractions each second is divided into. If the driver does not support sub-seconds, then this will either return ENOSUPP, or "uint32val" will be zero.

CYG_WALLCLOCK_INFO_GET_ALARM_PERIOD_MIN

Since not all RTCs will have support for second granularity alarms this call is used to ascertain the minimum alarm period. Must be called with an alarm index number in the "uint32val" member of supplied wallclock_info argument. Will return in "uint32val" the smallest alarm delta as a microsecond value. If the driver returns `ENOSUPP`, or the value 0, then the default of 1-second granularity can be assumed.

If the wallclock device supports alarms then the functions `cyg_wallclock_set_alarm_timespec`, `cyg_wallclock_set_alarm_date` and `cyg_wallclock_disable_alarm` will be defined and provide support for setting and disabling individual alarms. Expiry of an alarm will cause a given interrupt vector to be raised (as defined bt the `CYG_WALLCLOCK_INFO_GET_ALARM_INTVEC` key). It is the responsibility of the application to attach an ISR and DSR to this vector and handle any subsequent processing. See the **alarm** test program for an example.

If the wallclock device supports scratch space then the functions `cyg_wallclock_read_scratch` and `cyg_wallclock_write_scratch` will be defined to read and write *`len`* bytes at the given `offset` in the scratch space.

Chapter 123. Dallas DS1302 Wallclock Device Driver

Name

CYGPKG_DEVICES_WALLCLOCK_DALLAS_DS1302 — eCos Support for the Dallas DS1302 Real-Time Clock

Description

This package CYGPKG_DEVICES_WALLCLOCK_DALLAS_DS1302 provides a device driver for the wallclock device in the Dallas DS1302 Real-Time Clock chips. This combines a real-time clock and 31 bytes of battery-backed RAM in a single package. The driver can also be used with any other chips that provide the same interface to the clock hardware.

The package will usually be loaded into the configuration automatically whenever selecting a target which contains a compatible chip. By default it will provide the standard eCos wallclock device, although another implementation such as software emulation may be selected if desired. The only other configuration options related to this package allow users to change the compiler flags. If the application does not actually use the wallclock device, directly or indirectly, then the code should get removed automatically at link-time to ensure that the application does not suffer any unnecessary overheads.

Functionality

This wallclock device driver package implements the standard functionality required by the generic wallclock support CYGPKG_IO_WALLCLOCK. The functionality is not normally accessed directly. Instead it is used by the C library time package to implement standard calls such as time and gmtime. The eCos C library also provides a non-standard function cyg_libc_time_settime for changing the current wallclock setting. In addition RedBoot provides a **date** command which interacts with the wallclock device.

The package provides a number of additional functions that are specific to a DS1302:

```
#include <cyg/io/wallclock/ds1302.h>

externC unsigned char cyg_wallclock_ds1302_read_tcs_ds_rs(void);
externC void cyg_wallclock_ds1302_write_tcs_ds_rs(unsigned char val);
externC void cyg_wallclock_ds1302_read_ram(int offset,
                        unsigned char* buf, int len);
externC void cyg_wallclock_ds1302_write_ram(int offset,
                        unsigned char* buf, int len);
```

The _tcs_ds_rs functions allow applications to read and update the trickle charge register in the DS1302. The manufacturer's data sheet should be consulted for further information on this register.

The _ram functions allow applications to read and modify the contents of the 31 bytes of battery-backed RAM. The offset specifies the starting address within the RAM and should be between 0 and 31. The buffer provides the destination or source of the data, and the length gives the number of bytes transferred. Wrap-around is not supported so the sum of the offset and length should also be less than 31. The package's ds1302.c testcase provides example code.

The wallclock package is initialized by a static constructor with a priority immediately after CYG_INIT_DEV_WALLCLOCK. Applications should not call any wallclock-related functions nor any of the DS1302-specific functions before that constructor has run.

Porting

The DS1302 is accessed via a 3-wire bus. At the time of writing there is no generic 3-wire support package within eCos, so instead the wallclock driver expects to bit-bang some GPIO lines. Typically the platform HAL provides appropriate hardware-specific macros for this, via the header file cyg/hal/plf_io.h. The required macros are:

```
HAL_DS1302_CE(_setting_);
HAL_DS1302_SCLK(_setting_);
HAL_DS1302_OUT(_setting_);
HAL_DS1302_IN(_setting_);
```

```
HAL_DS1302_SELECT_OUT(_setting_);
```

The argument to the first three macros will always be 0 or 1 and corresponds to the desired state of the chip-enable, clock, or I/O line. For example, at the start of a transfer the wallclock driver will invoke:

```
    …
    HAL_DS1302_CE(1);
    …
```

Asserting the CE line should activate the DS1302 chip. The `HAL_DS1302_IN` macro is used to sample the state of the I/O line and should set its argument to 0 or 1. The `HAL_DS1302_SELECT_OUT` macro is used to switch the I/O line between output (1) or input (0).

Platform HALs may provide two additional macros:

```
HAL_DS1302_DATA
HAL_DS1302_INIT();
```

`HAL_DS1302_DATA` can be used to define one or more static variables needed by the other macros, for example to hold a shadow copy of the GPIO output register. If defined, `HAL_DS1302_INIT` will be invoked during driver initialization and typically sets up the GPIO lines such that the CE and SCLK lines are outputs.

In addition the DS1302 device driver package `CYGPKG_DEVICES_WALLCLOCK_DALLAS_DS1302` should be included in the CDL target entry so that it gets loaded automatically whenever eCos is configured for that target.

Chapter 124. Dallas DS1306 Wallclock Device Driver

Name

CYGPKG_DEVICES_WALLCLOCK_DALLAS_DS1306 — eCos Support for the Dallas DS1306 Real-Time Clock

Description

This package CYGPKG_DEVICES_WALLCLOCK_DALLAS_DS1306 provides a device driver for the wallclock device in the Dallas DS1306 Real-Time Clock chips. This combines a real-time clock and 96 bytes of battery-backed RAM in a single package. The driver can also be used with any other chips that provide the same interface to the clock hardware.

The package will usually be loaded into the configuration automatically whenever selecting a target which contains a compatible chip. By default it will provide the standard eCos wallclock device, although another implementation such as software emulation may be selected if desired. The only other configuration options related to this package allow users to change the compiler flags. If the application does not actually use the wallclock device, directly or indirectly, then the code should get removed automatically at link-time to ensure that the application does not suffer any unnecessary overheads.

Functionality

This wallclock device driver package implements the standard functionality required by the generic wallclock support CYGPKG_IO_WALLCLOCK. The functionality is not normally accessed directly. Instead it is used by the C library time package to implement standard calls such as time and gmtime. The eCos C library also provides a non-standard function cyg_libc_time_settime for changing the current wallclock setting. In addition RedBoot provides a **date** command which interacts with the wallclock device.

The package provides a number of additional functions that are specific to a DS1306:

```
#include <cyg/io/wallclock/ds1306.h>

externC void cyg_wallclock_ds1306_read_regs(int offset,
                                  unsigned char* buf, int len);
externC void cyg_wallclock_ds1306_write_regs(int offset,
                                  const unsigned char* buf, int len);
externC void cyg_wallclock_ds1306_read_ram(int offset,
                                  unsigned char* buf, int len);
externC void cyg_wallclock_ds1306_write_ram(int offset,
                                  const unsigned char* buf, int len);
```

The read_regs and write_regs functions allow direct access to all of the wallclock-related registers including the alarms, the control register 0x0F, the status register 0x10, and the trickle charger register 0x11. The offset should be between 0x00 and 0x1F, specifying the first register that should be read or written. For full details of the DS1306 registers see the manufacturer's data sheet.

The _ram functions allow applications to read and modify the contents of the 96 bytes of battery-backed RAM. The offset specifies the starting address within the RAM and should be between 0x00 and 0x5F. The buffer provides the destination or source of the data, and the length gives the number of bytes transferred. The package's ds1306.c testcase provides example code.

The wallclock package is initialized by a static constructor with a priority immediately after CYG_INIT_DEV_WALLCLOCK. Applications should not call any wallclock-related functions nor any of the DS1306-specific functions before that constructor has run.

Porting

The DS1306 can be either attached to an SPI bus or it can be accessed via a 3-wire interface. The driver supports both modes of operation, with a bit of support from the platform HAL. For SPI, the platform HAL should implement the CDL interface CYGHWR_WALLCLOCK_DALLAS_DS1306_SPI and provided an SPI device instance cyg_spi_wallclock_ds1306. The exact details of this device instantiation will depend on the SPI bus driver. For 3-wire the platform HAL should implement the CDL interface CYGHWR_WALLCLOCK_DALLAS_DS1306_3WIRE and provide a bit-bang function:

```
#include <cyg/io/wallclock/ds1306.h>

cyg_bool
hal_ds1306_bitbang(cyg_ds1306_bitbang_op op)
{
    cyg_bool result = 0;

    switch(op) {
      case CYG_DS1306_BITBANG_INIT: …
      case CYG_DS1306_BITBANG_CE_HIGH: …
      case CYG_DS1306_BITBANG_CE_LOW: …
      case CYG_DS1306_BITBANG_SCLK_HIGH: …
      case CYG_DS1306_BITBANG_SCLK_LOW: …
      case CYG_DS1306_BITBANG_DATA_HIGH: …
      case CYG_DS1306_BITBANG_DATA_LOW: …
      case CYG_DS1306_BITBANG_DATA_READ: …
      case CYG_DS1306_BITBANG_INPUT: …
      case CYG_DS1306_BITBANG_OUTPUT: …
    }
    return result;
}
```

The INIT operation should set the 3-wire bus to its default settings: all lines should be output low. The HIGH and LOW operations should set the specified line to the appropriate level. INPUT switches the data line from an output to an input, and OUTPUT switches it back to an output. READ should return the current state of the data line, and is the only operation for which the return value matters.

In addition the DS1306 device driver package `CYGPKG_DEVICES_WALLCLOCK_DALLAS_DS1306` should be included in the CDL target entry so that it gets loaded automatically whenever eCos is configured for that target.

Chapter 125. Dallas DS1307 Wallclock Device Driver

Name

CYGPKG_DEVICES_WALLCLOCK_DALLAS_DS1307 — eCos Support for the Dallas DS1307 Serial Real-Time Clock

Description

This package CYGPKG_DEVICES_WALLCLOCK_DALLAS_DS1307 provides a device driver for the wallclock device in the Dallas DS1307 Serial Real-Time Clock chips. This combines a real-time clock and 56 bytes of battery-backed RAM in a single package. The driver can also be used with any other chips that provide the same interface to the clock hardware.

The package will usually be loaded into the configuration automatically whenever selecting a target which contains a compatible chip. By default it will provide the standard eCos wallclock device, although another implementation such as software emulation may be selected if desired. The only other configuration options related to this package allow users to change the compiler flags. If the application does not actually use the wallclock device, directly or indirectly, then the code should get removed automatically at link-time to ensure that the application does not suffer any unnecessary overheads.

Functionality

This wallclock device driver package implements the standard functionality required by the generic wallclock support CYGPKG_IO_WALLCLOCK. The functionality is not normally accessed directly. Instead it is used by the C library time package to implement standard calls such as time and gmtime. The eCos C library also provides a non-standard function cyg_libc_time_settime for changing the current wallclock setting. In addition RedBoot provides a **date** command which interacts with the wallclock device.

Porting

DS1307 platform support can be implemented in one of two ways. The preferred approach involves the generic I²C API, as defined by the package CYGPKG_IO_I2C. The platform HAL can just provide a cyg_i2c_device structure cyg_i2c_wallclock_ds1307 and implement the CDL interface CYGINT_DEVICES_WALLCLOCK_DALLAS_DS1307_I2C. The DS1307 driver will now use I²C rx and tx operations to interact with the chip.

Alternatively the DS1307 driver can use macros or functions provided by another package to access the chip. This is intended primarily for older platforms that predate the CYGPKG_IO_I2C package. The other package should export a header file containing macros DS_GET and DS_PUT that transfer the eight bytes corresponding to the chip's clock registers. It should also export the name of this header via a #define CYGDAT_DEVS_WALLCLOCK_DS1307_INL in the global configuration header pkgconf/system.h. For full details see the source code.

In addition the DS1307 device driver package CYGPKG_DEVICES_WALLCLOCK_DALLAS_DS1307 should be included in the CDL target entry so that it gets loaded automatically whenever eCos is configured for that target.

Chapter 126. Dallas DS1390 Wallclock Device Driver

Name

CYGPKG_DEVICES_WALLCLOCK_DALLAS_DS1390 — eCos Support for the Dallas DS1390 Serial Real-Time Clock

Description

This package CYGPKG_DEVICES_WALLCLOCK_DALLAS_DS1390 provides a device driver for the wallclock device in the Dallas DS1390 Serial Real-Time Clock chips. The driver can also be used with any other chips that provide the same interface to the clock hardware.

The package will usually be loaded into the configuration automatically whenever selecting a target which contains a compatible chip. By default it will provide the standard eCos wallclock device, although another implementation such as software emulation may be selected if desired. The only other configuration options related to this package allow users to change the compiler flags. If the application does not actually use the wallclock device, directly or indirectly, then the code should get removed automatically at link-time to ensure that the application does not suffer any unnecessary overheads.

Functionality

This wallclock device driver package implements the standard functionality required by the generic wallclock support CYGPKG_IO_WALLCLOCK. The functionality is not normally accessed directly. Instead it is used by the C library time package to implement standard calls such as time and gmtime. The eCos C library also provides a non-standard function cyg_libc_time_settime for changing the current wallclock setting. In addition RedBoot provides a **date** command which interacts with the wallclock device.

Porting

The DS1390 driver uses the SPI driver API defined by the package CYGPKG_IO_SPI. A suitable SPI device driver must be available for the target. The platform HAL must provide a cyg_spi_device structure cyg_spi_wallclock_ds1390. The platform HAL should initialize this structure and any associated SPI driver specific struture with the correct phase, polarity and chip select parameters for this device.

In addition the DS1390 device driver package CYGPKG_DEVICES_WALLCLOCK_DALLAS_DS1390 should be included in the CDL target entry so that it gets loaded automatically whenever eCos is configured for that target.

Extra API Calls

In addition to the standard wallclock API calls, this driver exports a number of additional functions to permit direct access to additional features of the device. A header, cyg/io/wallclock/ds1390.h is available to define this API.

```
cyg_uint8 cyg_ds1390_read_reg( int addr )
```

Read and return a single 8-bit register from the DS1390, *addr* should be in the range 0x00 to 0x0F.

```
void cyg_ds1390_write_reg( int addr, int val )
```

Write a single 8-bit register to the DS1390, *addr* should be in the range 0x00 to 0x0F and *val* in the range 0x00 to 0xFF.

```
void cyg_ds1390_set_control( cyg_uint8 val )
```

Write the DS1390 control register with the content of *val*.

```
cyg_uint8 cyg_ds1390_get_control( void )
```

Read and return the value of the DS1390 control register.

`void cyg_ds1390_set_status( cyg_uint8 val )`

Write the DS1390 status register with the content of *val*.

`cyg_uint8 cyg_ds1390_get_status( void )`

Read and return the value of the DS1390 control register.

`void cyg_ds1390_set_charger( cyg_uint8 val )`

Write the DS1390 trickle-charge register with the content of *val*.

`cyg_uint8 cyg_ds1390_get_charger( void )`

Read and return the value of the DS1390 trickle-charge register.

`int cyg_wallclock_set_alarm( cyg_uint32 secs )`

Set the DS1390 alarm to trigger when the wallclock time matches the value of *secs*. The DS1390 alarm will match only up to days of the month, so the alarm cannot be set more than one month in the future. This function only initializes the DS1390 to generate the alarm interrupt; it is the responsibility of the caller to attach an ISR to the appropriate vector and unmask it in the interrupt controller.

Chapter 127. Freescale MCFxxxx On-Chip Wallclock Device Driver

Name

CYGPKG_DEVS_WALLCLOCK_MCFxxxx — eCos Support for the Freescale MCFxxxx On-Chip Real-Time Clock

Description

Some members of the Freescale ColdFire range of processors come with an on-chip Real-Time Clock device which can act as an eCos wallclock. The device will not always be appropriate for an application's requirement. Typically it does not have its own low-current battery input so it will only operate when the whole processor is powered up. Hence either the entire system needs to be powered by a battery or have a battery backup. Otherwise the device will lose its settings when the power fails, requiring an application-level recovery mechanism, which means that there is no real advantage to using the RTC rather than a software emulation.

For those scenarios where the on-chip RTC does meet the application's requirements, this package CYGPKG_DEVS_WALLCLOCK_MCFxxxx provides an eCos device driver. The package will usually be loaded into the configuration automatically whenever selecting a target which contains a compatible device. By default it will provide the standard eCos wallclock device, although another implementation such as software emulation may be selected if desired. The only other configuration options related to this package allow users to change the compiler flags. If the application does not actually use the wallclock device, directly or indirectly, then the code should get removed automatically at link-time to ensure that the application does not suffer any unnecessary overheads.

Functionality

This wallclock device driver package implements the standard functionality required by the generic wallclock support CYGPKG_IO_WALLCLOCK. The functionality is not normally accessed directly. Instead it is used by the C library time package to implement standard calls such as time and gmtime. The eCos C library also provides a non-standard function cyg_libc_time_settime for changing the current wallclock setting. In addition RedBoot provides a **date** command which interacts with the wallclock device.

The wallclock driver does not provide any support for other functionality provided by the RTC device such as the alarm or stopwatch. It only manipulates the HOURMIN, SECONDS, DAYS, and CR registers. Applications can access the remaining registers as required without affecting the driver.

The wallclock package is initialized by a static constructor with a priority immediately after CYG_INIT_CLOCK. Applications should not call any wallclock-related functions before that constructor has run.

Porting

The driver requires only minimal porting. The HAL packages, typically the processor HAL, should supply the register definitions and the device base address HAL_MCFxxxx_RTC_BASE. In addition the platform HAL should define the crystal frequency using a #define of HAL_MCFxxxx_RTC_XTAL: legal values are 32768, 32000, and 38400. Finally the driver package CYGPKG_DEVS_WALLCLOCK_MCFxxxx should be included in the CDL target entry so that it gets loaded automatically whenever eCos is configured for that target.

Chapter 128. Intersil ISL1208 Wallclock Device Driver

Name

CYGPKG_DEVICES_WALLCLOCK_INTERSIL_ISL1208 — eCos Support for the Intersil ISL1208 Real-Time Clock

Description

This package CYGPKG_DEVICES_WALLCLOCK_INTERSIL_ISL1208 provides a device driver for the wallclock device in the Intersil ISL1208 Real-Time Clock chips. These combine a real-time clock, alarm functionality, a selectable frequency output, and two bytes of non-volatile memory in a single package. The driver can also be used with any other chips that provide the same interface to the clock hardware.

The package will usually be loaded into the configuration automatically whenever selecting a target which contains a compatible chip. By default it will provide the standard eCos wallclock device, although another implementation such as software emulation may be selected if desired. The only other configuration options related to this package allow users to change the compiler flags. If the application does not actually use the wallclock device, directly or indirectly, then the code should get removed automatically at link-time to ensure that the application does not suffer any unnecessary overheads.

Functionality

This wallclock device driver package implements the standard functionality required by the generic wallclock support CYGPKG_IO_WALLCLOCK. The functionality is not normally accessed directly. Instead it is used by the C library time package to implement standard calls such as time and gmtime. The eCos C library also provides a non-standard function cyg_libc_time_settime for changing the current wallclock setting. In addition RedBoot provides a **date** command which interacts with the wallclock device.

The driver does not provide direct access to any of the other functionality provided by the chip. Instead if an application wishes to access the alarms or the non-volatile bytes then it can do so itself, via the generic I²C API. However any such application code does need to synchronize with the wallclock driver to prevent concurrent accesses to the device. The driver exports a mutex lock to allow for this:

```
#include <cyg/io/wallclock/isl1208.h>

extern cyg_drv_mutex_t cyg_isl1208_lock;
```

The mutex should be locked via cyg_drv_mutex_lock to prevent the wallclock driver from accessing the chip, and then unlocked via cyg_drv_mutex_unlock when the driver can safely access the chip again.

The wallclock package is initialized by a static constructor with a priority immediately after CYG_INIT_DEV_WALLCLOCK. Applications should not call any wallclock-related functions before that constructor has run.

Porting

The ISL1208 is accessed via an I²C serial bus, and the driver assumes the presence of the generic I²C support package CYGPKG_IO_I2C and a suitable hardware driver. In addition it requires that some other package, typically the platform HAL, exports a cyg_i2c_device structure cyg_i2c_wallclock_isl1208. The ISL1208 device driver package CYGPKG_DEVICES_WALLCLOCK_INTERSIL_ISL1208 can then be included in the CDL target entry so that it gets loaded automatically whenever eCos is configured for that target.

Chapter 129. Intersil ISL12028 Wallclock Device Driver

Name

CYGPKG_DEVICES_WALLCLOCK_INTERSIL_ISL12028 — eCos Support for the Intersil ISL12028 Real-Time Clock

Description

This package CYGPKG_DEVICES_WALLCLOCK_INTERSIL_ISL12028 provides a device driver for the wallclock device in the Intersil ISL12028 Real-Time Clock chips. These combine a real-time clock, alarm functionality, and a bank of EEPROM in a single package. The driver can also be used with any other chips that provide the same interface to the clock hardware.

The package will usually be loaded into the configuration automatically whenever selecting a target which contains a compatible chip. By default it will provide the standard eCos wallclock device, although another implementation such as software emulation may be selected if desired. The only other configuration options related to this package allow users to change the compiler flags. If the application does not actually use the wallclock device, directly or indirectly, then the code should get removed automatically at link-time to ensure that the application does not suffer any unnecessary overheads.

Functionality

This wallclock device driver package implements the standard functionality required by the generic wallclock support CYGPKG_IO_WALLCLOCK. The functionality is not normally accessed directly. Instead it is used by the C library time package to implement standard calls such as time and gmtime. The eCos C library also provides a non-standard function cyg_libc_time_settime for changing the current wallclock setting. In addition RedBoot provides a **date** command which interacts with the wallclock device.

The driver does not provide direct access to any of the other functionality provided by the chip. Instead if an application wishes to access the alarms or the EEPROM memory then it can do so itself, via the generic I²C API. However any such application code does need to synchronize with the wallclock driver to prevent concurrent accesses to the device. The driver exports a mutex lock to allow for this:

```
#include <cyg/io/wallclock/isl12028.h>

extern cyg_drv_mutex_t cyg_isl12028_lock;
```

The mutex should be locked via cyg_drv_mutex_lock to prevent the wallclock driver from accessing the chip, and then unlocked via cyg_drv_mutex_unlock when the driver can safely access the chip again.

The wallclock package is initialized by a static constructor with a priority immediately after CYG_INIT_DEV_WALLCLOCK. Applications should not call any wallclock-related functions before that constructor has run.

Porting

The ISL12028 is accessed via an I²C serial bus, and the driver assumes the presence of the generic I²C support package CYGPKG_IO_I2C and a suitable hardware driver. In addition it requires that some other package, typically the platform HAL, exports a cyg_i2c_device structure cyg_i2c_wallclock_isl12028. The ISL12028 device driver package CYGPKG_DEVICES_WALLCLOCK_INTERSIL_ISL12028 can then be included in the CDL target entry so that it gets loaded automatically whenever eCos is configured for that target.

Chapter 130. ST M41TXX Wallclock Device Driver

Name

CYGPKG_DEVICES_WALLCLOCK_ST_M41TXX — eCos Support for the ST M41TXX Serial Real-Time Clock

Description

This package CYGPKG_DEVICES_WALLCLOCK_ST_M41TXX provides a device driver for the wallclock device in the ST M41TXX Serial Real-Time Clock chips. This is a real-time clock, alarm and watchdog in a single package. eCos only currently supports the real-time clock function.

The package will usually be loaded into the configuration automatically whenever selecting a target which contains a compatible chip. By default it will provide the standard eCos wallclock device, although another implementation such as software emulation may be selected if desired. The only other configuration options related to this package allow users to change the compiler flags. If the application does not actually use the wallclock device, directly or indirectly, then the code should get removed automatically at link-time to ensure that the application does not suffer any unnecessary overheads.

Functionality

This wallclock device driver package implements the standard functionality required by the generic wallclock support CYGPKG_IO_WALLCLOCK. The functionality is not normally accessed directly. Instead it is used by the C library time package to implement standard calls such as time and gmtime. The eCos C library also provides a non-standard function cyg_libc_time_settime for changing the current wallclock setting. In addition RedBoot provides a **date** command which interacts with the wallclock device.

Porting

M41TXX platform support is implemented via the generic I²C API, as defined by the package CYGPKG_IO_I2C. The platform HAL can just provide a cyg_i2c_device structure cyg_i2c_wallclock_m41txx. The M41TXX driver will now use I²C rx and tx operations to interact with the chip.

In addition the M41TXX device driver package CYGPKG_DEVICES_WALLCLOCK_ST_M41TXX should be included in the CDL target entry so that it gets loaded automatically whenever eCos is configured for that target.

Chapter 131. ST M48T Wallclock Device Driver

Name

CYGPKG_DEVS_WALLCLOCK_ST_M48Txxx — eCos Support for the ST M48T TimeKeeper SRAM chips and compatibles

Description

This package CYGPKG_DEVS_WALLCLOCK_ST_M48Txxx provides a device driver for the wallclock device in the ST M48T family of TimeKeeper SRAM chips (e.g. the M48T35AV part). These combine an amount of battery-backed SRAM and a real-time clock in a single package. The driver can also be used with any other chips that provide the same interface to the clock hardware.

The package will usually be loaded into the configuration automatically whenever selecting a target which contains a compatible chip. By default it will provide the standard eCos wallclock device, although another implementation such as software emulation may be selected if desired. The only other configuration options related to this package allow users to change the compiler flags. If the application does not actually use the wallclock device, directly or indirectly, then the code should get removed automatically at link-time to ensure that the application does not suffer any unnecessary overheads.

Functionality

This wallclock device driver package implements the standard functionality required by the generic wallclock support CYGPKG_IO_WALLCLOCK. The functionality is not normally accessed directly. Instead it is used by the C library time package to implement standard calls such as time and gmtime. The eCos C library also provides a non-standard function cyg_libc_time_settime for changing the current wallclock setting. In addition RedBoot provides a **date** command which interacts with the wallclock device.

M48T devices provide some support for a calibration value. If the application has some alternative way of getting a reliable time value, for example NTP over a TCP/IP network, then the wallclock can be made to tick slightly faster or slower. The calibration value is a small integer between -31 and +31. A positive value x adds 512x extra cycles every 125829120 actual cycles, speeding up the clock by approximately 10.7x seconds per month. Alternatively a negative value x subtracts 256x cycles, slowing down the clock by 5.35x seconds per month. The package provides two functions for examining and changing the current calibration value:

```
#include <cyg/io/wallclock_m48txxx.h>

externC cyg_int32   cyg_wallclock_m48t_get_calibration(void);
externC void        cyg_wallclock_m48t_set_calibration(cyg_int32);
```

Porting

For most platforms adding support for the M48T wallclock device requires just two steps. The package must be added to the appropriate CDL target entry so that it gets loaded automatically, and selects the relevant ST M48T family device, whenever configuring eCos for that target. Also the platform HAL should specify the location of the clock hardware in the address space, by defining the symbol HAL_WALLCLOCK_M48Txxx_BASE. The definition should go into cyg/hal/hal_io.h or more commonly into a platform-specific header cyg/hal/plf_io.h which gets included automatically by the former. The value should be the address of the control register of the clock device. The driver provides the CYGNUM_DEVS_WALLCLOCK_ST_M48Txxx_OFFSET_CLOCK value which is set to the appropriate offset value for the CYGHWR_DEVS_WALLCLOCK_ST_M48Txxx selected device. For example, given a battery-backed 32K TimeKeeper chip at 0x30000000, the clock hardware will occupy the last eight bytes at 0x30007ff8 and that is the value which should be used, and CYGNUM_DEVS_WALLCLOCK_ST_M48Txxx_OFFSET_CLOCK will have the value 0x7ff8.

The package provides some support for hardware where the clock is mapped into memory in strange ways. The platform HAL can define an additional symbol HAL_WALLCLOCK_M48Txxx_STRIDE and macros HAL_WALLCLOCK_M48Txxx_READ_UINT8 and HAL_WALLCLOCK_M48Txxx_WRITE_UINT8 to change the way in which the driver accesses the hardware. The source code should be consulted for further details of how these work.

If the selected ST M48T device implements the Century bit then the configurarion will define CYGINT_DEVS_WALLCLOCK_ST_M48Txxx_CENTURY_BIT as a non-zero value. The platform can override the CDL device based configuration by defining `HAL_WALLCLOCK_M48Txxx_NO_CENTURY_BIT` which can be used to notify the driver that a compatible device does not support the Century bit, or that the feature should be explicitly disabled.

If the Century bit is not supported the driver will instead use a heuristic for determining the century: if the year register is < 70 then this is treated as relative to 2000; otherwise it is treated as relative to 1900; this gives an effective range of Jan 1st 1970 to Dec 31st 2069.

Part XXXVIII. Watchdog Drivers

Documentation for drivers of this type is often integrated into the eCos board support documentation. You should review the documentation for your target board for details. Standalone and more generic drivers are documented in the following sections.

Table of Contents

Chapter 132. Freescale Kinetis Watchdog Driver

Name

CYGPKG_DEVICES_WATCHDOG_CORTEXM_KINETIS — eCos Support for the Kinetis on-chip Watchdog timer device (WDOG)

Description

The Freescale Kinetis processor provides a Watchdog Timer (WDOG) module. Once the watchdog timer is enabled it will automatically reset the processor unless software resets the timer within the configured period.

The package CYGPKG_DEVICES_WATCHDOG_CORTEXM_KINETIS provides an eCos driver for this device, complementing the generic package CYGPKG_IO_WATCHDOG. The functionality should be accessed via the standard eCos watchdog functions watchdog_start, watchdog_reset and watchdog_get_resolution.

The driver only supports “reset” mode.

Configuration Options

The Kinetis watchdog driver package should be loaded automatically when selecting a platform containing a Kinetis processor, and it should never be necessary to load it explicitly into the configuration. The package is inactive unless the generic watchdog support CYGPKG_IO_WATCHDOG is loaded and the watchdog explicitly started. Depending on the choice of eCos template it may be necessary to load the CYGPKG_IO_WATCHDOG package. By default the HAL startup for the Kinetis platform disables the watchdog shortly after reset, so even when the I/O watchdog package is loaded the watchdog will not trigger unless the watchdog support is explicitly started by calling watchdog_start.

The package provides four main configuration options.

CYGIMP_WATCHDOG_HARDWARE can be used to disable the use of the hardware watchdog and switch to a software emulation provided by the generic watchdog package instead. This may prove useful during debugging.

CYGNUM_DEVS_WATCHDOG_CORTEXM_KINETIS_CLK is used to select the clock source used to drive the watchdog timer module. The setting of this variable affects the minimum and maximum values that can be configured for the watchdog timeout.

CYGNUM_DEVS_WATCHDOG_CORTEXM_KINETIS_CLK_PRESCALER can further extend the maximum possible value for the watchdog timeout.

CYGNUM_DEVS_WATCHDOG_CORTEXM_KINETIS_DESIRED_TIMEOUT_US determines the timeout before the hardware watchdog resets the system. The default setting gives a 1-second timeout. The minimum and maximum timeout values are determined by the selected source clock and prescaler values. Depending on the configuration the range can extend from 9 microseconds to 4000 seconds.

Porting

The watchdog device driver does not require any platform-specific support. The only porting effort required is to list CYGPKG_DEVICES_WATCHDOG_CORTEXM_KINETIS as one of the hardware packages in the ecos.db target entry.

Chapter 133. Freescale MCFxxxx SCM Watchdog Driver

Name

CYGPKG_DEVS_WATCHDOG_MCFxxxx_SCM — eCos Support for the MCFxxxx SCM On-chip Watchdog Device

Description

ColdFire MCFxxxx processors typically come with two on-chip watchdog devices. The main watchdog is not readily usable by eCos: it comes up enabled and, once disabled, it can never be reenabled. Hence in a typical development environment that watchdog device needs to be disabled early on or it will interfere with debugging, and cannot be used again. There is a second watchdog device embedded in the System Control Module which is usable. This package CYGPKG_DEVS_WATCHDOG_MCFxxxx_SCM provides an eCos driver for that device, complementing the generic package CYGPKG_IO_WATCHDOG. The driver functionality should be accessed via the standard eCos watchdog functions watchdog_start, watchdog_reset and watchdog_get_resolution.

Configuration Options

The MCFxxxx SCM watchdog driver package should be loaded automatically when selecting a platform containing a suitable MCFxxxx ColdFire processor. It should never be necessary to load it explicitly into the configuration. The package is inactive unless the generic watchdog support CYGPKG_IO_WATCHDOG is loaded. Depending on the choice of eCos template it may be necessary to load the latter.

There are a number of configuration options. The first is CYGIMP_WATCHDOG_HARDWARE, which can be used to disable the use of the hardware watchdog and switch to a software emulation provided by the generic watchdog package instead. This may prove useful during debugging.

By default the watchdog device is set to reset the system when the timeout expires. It can be configured to raise an interrupt instead by disabling CYGIMP_DEVS_WATCHDOG_MCFxxxx_SCM_RESET. The interrupt ISR will invoke any installed application action handlers.

The watchdog timeout is controlled by CYGNUM_DEVS_WATCHDOG_MCFxxxx_SCM_TICKS. This corresponds to the CWT field in the SCM's CWCR register. It can take a value between 8 and 31, with a default of 28. That means 2^28 peripheral bus clock ticks have to elapse before the watchdog triggers. Typically that means a timeout of a small number of seconds. There is a calculated CDL option CYGNUM_DEVS_WATCHDOG_MCFxxxx_SCM_DELAY which gives the current delay in nanoseconds.

The watchdog device has a bit which turns it read-only, preventing any errant code from accidentally disabling it. By default the driver will set this bit after starting the watchdog. If for some reason the application needs to access the device directly then the option CYGIMP_DEVS_WATCHDOG_MCFxxxx_SCM_WRITE_ONCE should be disabled.

By default the watchdog is set to continue ticking even if the core is halted by an idle thread action or by power management code. This can cause problems if the application code halts the core for an extended period of time, so the behaviour can be changed by disabling CYGIMP_DEVS_WATCHDOG_MCFxxxx_SCM_RUN_WHILE_HALTED.

If the watchdog device is configured to raise interrupts rather than generate a reset then CYGNUM_DEVS_WATCHDOG_MCFxxxx_SCM_ISR_PRIORITY controls the interrupt priority. There are also configuration options allowing developers to tweak the compiler flags used for building this package.

Porting

The watchdog device driver usually does not require any platform-specific support. The processor HAL should provide the device definitions needed by the code. The only porting effort required is to list CYGPKG_DEVS_WATCHDOG_MCFxxxx_SCM as one of the hardware packages in the ecos.db target entry.

Chapter 134. Freescale MCFxxxx Watchdog Driver

Name

CYGPKG_DEVS_WATCHDOG_MCFxxxx — eCos Support for the MCFxxxx On-chip Watchdog Device

Description

Several members of the MCFxxxx ColdFire family have a simple watchdog device embedded in the System Control Module or SCM. This package CYGPKG_DEVS_WATCHDOG_MCFxxxx provides an eCos device driver for the watchdog device, complementing the generic package CYGPKG_IO_WATCHDOG. The driver functionality should be accessed via the standard eCos watchdog functions watchdog_start, watchdog_reset and watchdog_get_resolution.

The hardware has limited functionality: instead of automatically causing a reset when the watchdog triggers it can only raise an interrupt. By default the watchdog driver installs a non-maskable interrupt with the highest possible priority, and a custom interrupt VSR will immediately perform a reset using the chip's Reset Controller Module. This is not quite as good as a watchdog device which performs the reset automatically: corruption of the interrupt vector table, the interrupt controller, the SCM module, or the VSR code may prevent a reset from occurring. However in most circumstances a watchdog timeout will still result in a full reset. Alternatively the driver can be configured to generate an ordinary interrupt, leaving it up to application code to perform recovery from the timeout.

Configuration Options

The MCFxxxx watchdog driver package should be loaded automatically when selecting a platform containing a suitable ColdFire processor, and it should never be necessary to load it explicitly into the configuration. The package is inactive unless the generic watchdog support CYGPKG_IO_WATCHDOG is loaded. Depending on the choice of eCos template it may be necessary to load the latter.

The first configuration option is CYGIMP_WATCHDOG_HARDWARE, which can be used to disable the use of the hardware watchdog and switch to a software emulation provided by the generic watchdog package instead. This may prove useful during debugging.

If the hardware watchdog is enabled then CYGIMP_DEVS_WATCHDOG_MCFxxxx_ATTEMPT_RESET controls whether the driver will install a non-maskable interrupt VSR which performs a reset, or an ordinary interrupt handler which calls into application code. The default is to attempt the reset.

The watchdog timeout is controlled by CYGNUM_DEVS_WATCHDOG_MCFxxxx_TICKS. It is measured in system clock ticks and only a limited number of values are available: 2^9, 2^{11}, 2^{13}, 2^{15}, 2^{19}, 2^{23}, 2^{27} and 2^{31}. The default is 2^{27} clock ticks. For a processor running at 64MHz that corresponds to just over two seconds. With the same clock 2^{23} ticks would give 0.13 seconds and 2^{31} would give 33 seconds. For convenience there is a calculated configuration option CYGNUM_DEVS_WATCHDOG_MCFxxxx_DELAY which gives the actual delay in nanoseconds.

If the watchdog is configured to generate an ordinary interrupt rather than attempt a reset then CYGNUM_DEVS_WATCHDOG_MCFxxxx_ISR_PRIORITY determines the interrupt priority. The default will be provided by the processor HAL.

Porting

The watchdog device driver usually does not require any platform-specific support. The only porting effort required is to list CYGPKG_DEVS_WATCHDOG_MCFxxxx as one of the hardware packages in the ecos.db target entry. However if the driver has been configured to generate a reset then it will use HAL_VSR_SET to install a custom VSR cyg_mcfxxxx_watchdog_vsr. On platforms where the exception vectors are in flash and hence read-only this will be a problem and the function will have to be placed in the appropriate slot instead of hal_m68k_interrupt_vsr.

Chapter 135. Freescale MCF5272 Watchdog Driver

Name

CYGPKG_DEVS_WATCHDOG_MCF5272 — eCos Support for the MCF5272 On-chip Watchdog Device

Description

The Freescale MCF5272 ColdFire processor has a built-in watchdog device. Once started it will automatically reset the processor unless software updates the device at regular intervals. The package CYGPKG_DEVS_WATCHDOG_MCF5272provides an eCos driver for this device, complementing the generic package CYGPKG_IO_WATCHDOG. The functionality should be accessed via the standard eCos watchdog functions watchdog_start, watchdog_reset and watchdog_get_resolution.

The watchdog driver only supports reset mode. The hardware can also be configured to raise an interrupt when the watchdog times out, but this mode of operation is not supported.

Configuration Options

The MCF5272 watchdog driver package should be loaded automatically when selecting a platform containing an MCF5272 processor, and it should never be necessary to load it explicitly into the configuration. The package is inactive unless the generic watchdog support CYGPKG_IO_WATCHDOG is loaded. Depending on the choice of eCos template it may be necessary to load the latter.

The package provides two main configuration options. CYGIMP_WATCHDOG_HARDWARE can be used to disable the use of the hardware watchdog and switch to a software emulation provided by the generic watchdog package instead. This may prove useful during debugging. CYGNUM_DEVS_WATCHDOG_MCF5272_TICKS determines the timeout before the hardware watchdog resets the processor. It should be a 15-bit number, with each tick corresponding to 32768 system clock cycles. For a processor operating at 66MHz each tick is approximately 0.5 milliseconds so the default value of 2000 corresponds to a one-second timeout. The maximum value of 32767 gives a timeout of approximately 16 seconds.

Porting

The watchdog device driver does not require any platform-specific support. The only porting effort required is to list CYGPKG_DEVS_WATCHDOG_MCF5272 as one of the hardware packages in the ecos.db target entry.

Chapter 136. Freescale MCF5282 Watchdog Driver

Name

CYGPKG_DEVS_WATCHDOG_MCF5282 — eCos Support for the MCF5282 On-chip Watchdog Device

Description

The Freescale MCF5282 Coldfire processor has two built-in watchdog devices. The System Control Module or SCM has a simple watchdog device which can only generate an interrupt when the watchdog triggers. The Watchdog Timer Module has a more advanced watchdog device, but unfortunately it has a write-once register which makes it difficult to use in a typical development environment. The package CYGPKG_DEVS_WATCHDOG_MCF5282 provides an eCos device driver for the SCM device, complementing the generic package CYGPKG_IO_WATCHDOG. The functionality should be accessed via the standard eCos watchdog functions watchdog_start, watchdog_reset and watchdog_get_resolution.

By default the watchdog driver installs a non-maskable interrupt with the highest possible priority, and a custom interrupt VSR will immediately perform a reset using the chip's Reset Controller Module. This is not quite as good as a watchdog device which performs the reset automatically: corruption of the interrupt vector table, the interrupt controller, the SCM module, or the VSR code may prevent a reset from occurring. However in most circumstances a watchdog timeout will still result in a full reset. Alternatively the driver can be configured to generate an ordinary interrupt, leaving it up to application code to perform recovery from the timeout.

Configuration Options

The MCF5282 watchdog driver package should be loaded automatically when selecting a platform containing an MCF5282 processor, and it should never be necessary to load it explicitly into the configuration. The package is inactive unless the generic watchdog support CYGPKG_IO_WATCHDOG is loaded. Depending on the choice of eCos template it may be necessary to load the latter.

The first configuration option is CYGIMP_WATCHDOG_HARDWARE, which can be used to disable the use of the hardware watchdog and switch to a software emulation provided by the generic watchdog package instead. This may prove useful during debugging.

If the hardware watchdog is enabled then CYGIMP_DEVS_WATCHDOG_MCF5282_ATTEMPT_RESET controls whether the driver will install a non-maskable interrupt VSR which performs a reset, or an ordinary interrupt handler which calls into application code. The default is to attempt the reset.

The watchdog timeout is controlled by CYGNUM_DEVS_WATCHDOG_MCF5282_TICKS. It is measured in system clock ticks and only a limited number of values are available: 2^9, 2^{11}, 2^{13}, 2^{15}, 2^{19}, 2^{23}, 2^{27} and 2^{31}. The default is 2^{27} clock ticks. For a processor running at 64MHz that corresponds to just over two seconds. With the same clock 2^{23} ticks would give 0.13 seconds and 2^{31} would give 33 seconds.

If the watchdog is configured to generate an ordinary interrupt rather than attempt a reset then CYGNUM_DEVS_WATCHDOG_MCF5282_ISR_PRIORITY determines the interrupt priority. The default will be provided by the processor HAL. On an MCF5282 all interrupt priorities must be unique.

Porting

The watchdog device driver does not require any platform-specific support. The only porting effort required is to list CYGPKG_DEVS_WATCHDOG_MCF5282 as one of the hardware packages in the ecos.db target entry.

Chapter 137. Freescale MCF532x Watchdog Driver

Name

CYGPKG_DEVS_WATCHDOG_MCF532x — eCos Support for the MCF532x On-chip Watchdog Device

Description

The ColdFire MCF532x family of processors come with two on-chip watchdog devices. The main watchdog is not readily usable by eCos: it comes up enabled and, once disabled, it can never be reenabled. Hence in a typical development environment that watchdog device needs to be disabled early on or it will interfere with debugging, and cannot be used again. There is a second watchdog device embedded in the System Control Module which is usable. This package CYGPKG_DEVS_WATCHDOG_MCF532x provides an eCos driver for that device, complementing the generic package CYGPKG_IO_WATCHDOG. The driver functionality should be accessed via the standard eCos watchdog functions watchdog_start, watchdog_reset and watchdog_get_resolution.

Configuration Options

The MCF532x watchdog driver package should be loaded automatically when selecting a platform containing an MCF532x ColdFire processor, or any other ColdFire with a compatible device. It should never be necessary to load it explicitly into the configuration. The package is inactive unless the generic watchdog support CYGPKG_IO_WATCHDOG is loaded. Depending on the choice of eCos template it may be necessary to load the latter.

There are a number of configuration options. The first is CYGIMP_WATCHDOG_HARDWARE, which can be used to disable the use of the hardware watchdog and switch to a software emulation provided by the generic watchdog package instead. This may prove useful during debugging.

By default the watchdog device is set to reset the system when the timeout expires. It can be configured to raise an interrupt instead by disabling CYGIMP_DEVS_WATCHDOG_MCF532x_RESET. The interrupt ISR will invoke any installed application action handlers.

The watchdog timeout is controlled by CYGNUM_DEVS_WATCHDOG_MCF532x_TICKS. This corresponds to the CWT field in the SCM's CWCR register. It can take a value between 8 and 31, with a default of 27. That means 2^27 clock ticks have to elapse before the watchdog triggers. For a processor operating at 240/80MHz that corresponds to approximately 1.67 seconds. There is also a calculated CDL option CYGNUM_DEVS_WATCHDOG_MCF532x_DELAY which gives the current delay in nanoseconds.

The watchdog device has a bit which turns it read-only, preventing any errant code from accidentally disabling it. By default the driver will set this bit after starting the watchdog. If for some reason the application needs to access the device directly then the option CYGIMP_DEVS_WATCHDOG_MCF532x_WRITE_ONCE should be disabled.

By default the watchdog is set to continue ticking even if the core is halted by an idle thread action or by power management code. This can cause problems if the application code halts the core for an extended period of time, so the behaviour can be changed by disabling CYGIMP_DEVS_WATCHDOG_MCF532x_RUN_WHILE_HALTED.

If the watchdog device is configured to raise interrupts rather than generate a reset then CYGNUM_DEVS_WATCHDOG_MCF532x_ISR_PRIORITY controls the interrupt priority. There are also configuration options allowing developers to tweak the compiler flags used for building this package.

Porting

The watchdog device driver usually does not require any platform-specific support. The only porting effort required is to list CYGPKG_DEVS_WATCHDOG_MCF532x as one of the hardware packages in the ecos.db target entry.

Chapter 138. Nios II Avalon Timer Watchdog Driver

Name

CYGPKG_DEVS_WATCHDOG_NIOS2_AVALON_TIMER — eCos Support for a Nios II Avalon Timer-based Watchdog Device

Description

A Nios II hardware design can include an Avalon timer which acts as a watchdog device. Once started it will automatically reset the processor unless software updates the device at regular intervals. The package CYGPKG_DEVS_WATCHDOG_NIOS2_AVALON_TIMER provides an eCos driver for this device, complementing the generic package CYGPKG_IO_WATCHDOG. The functionality should be accesssed via the standard eCos watchdog functions.

Configuration Options

The Avalon watchdog driver package should be loaded automatically when creating an eCos configuration for a hardware design which includes a suitable watchdog device, and it should never be necessary to load the package explicitly. The package is inactive unless the generic watchdog support CYGPKG_IO_WATCHDOG is loaded. Depending on the choice of eCos template it may be necessary to load the latter.

The package provides a configuration option CYGIMP_WATCHDOG_HARDWARE. This can be used to disable the use of the hardware watchdog and switch to a software emulation provided by the generic watchdog package instead. This may prove useful during debugging. The package also provides two configuration options for manipulating the compiler flags used to build the driver.

Porting

A hardware design requiring a watchdog should include an Avalon timer labelled “watchdog”. This timer should use the presets for a watchdog device: no writeable period, no readable snapshot, no start/stop control bits, no timeout pulse, system reset on timeout. The watchdog period can be set to any desired value, subject to the constraints of a 32-bit counter and the hardware's input clock. For example if the hardware runs at 100MHz then the watchdog period is limited to at most 42.9 seconds.

The hardware design HAL package should include definitions for HAL_NIOS2_AVALON_TIMER_WATCHDOG_BASE and HAL_NIOS2_AVALON_TIMER_WATCHDOG_PERIOD. In addition the ecos.db target entry should list CYGPKG_DEVS_WATCHDOG_NIOS2_AVALON_TIMER as one of the hardware packages.

Chapter 139. NXP PNX8310 Watchdog Driver

Name

CYGPKG_DEVS_WATCHDOG_MIPS_PNX8310 — eCos Support for the PNX8310 On-chip Watchdog Device

Description

The NXP PNX8310 processor is based around a PR1910 core. This core supports three timers, one of which can act as a watchdog device. Once the timer is started it will automatically reset the processor unless software resets the timer at regular intervals. The package CYGPKG_DEVS_WATCHDOG_MIPS_PNX8310 provides an eCos driver for this device, complementing the generic package CYGPKG_IO_WATCHDOG. The functionality should be accessed via the standard eCos watchdog functions watchdog_start, watchdog_reset and watchdog_get_resolution.

The watchdog driver only supports reset mode. The hardware can also be configured to raise an interrupt half way through the timeout period, but the driver does not use this functionality. An application can install an interrupt handler on CYGNUM_HAL_ISR_TMR3 if desired.

Configuration Options

The PNX8310 watchdog driver package should be loaded automatically when selecting a platform containing a PNX8310 processor, and it should never be necessary to load it explicitly into the configuration. The package is inactive unless the generic watchdog support CYGPKG_IO_WATCHDOG is loaded. Depending on the choice of eCos template it may be necessary to load the latter.

The package provides two main configuration options. CYGIMP_WATCHDOG_HARDWARE can be used to disable the use of the hardware watchdog and switch to a software emulation provided by the generic watchdog package instead. This may prove useful during debugging. CYGNUM_DEVS_WATCHDOG_MIPS_PNX8310_MILLISECONDS determines the timeout before the hardware watchdog resets the system. The default setting gives a 10-second timeout. The maximum timeout is determined by the CPU clock frequency, approximately 70 seconds for a 120MHz processor.

Porting

The watchdog device driver does not require any platform-specific support. The only porting effort required is to list CYGPKG_DEVS_WATCHDOG_MIPS_PNX8310 as one of the hardware packages in the ecos.db target entry.

Chapter 140. NXP PNX8330 Watchdog Driver

Name

CYGPKG_DEVS_WATCHDOG_MIPS_PNX8330 — eCos Support for the PNX8330 On-chip Watchdog Device

Description

The NXP PNX8330 processor Configuration unit supports three timers, one of which will act as a watchdog device. Once the timer is started it will automatically reset the processor unless software resets the timer at regular intervals. The package CYGPKG_DEVS_WATCHDOG_MIPS_PNX8330 provides an eCos driver for this device, complementing the generic package CYGPKG_IO_WATCHDOG. The functionality should be accessed via the standard eCos watchdog functions watchdog_start, watchdog_reset and watchdog_get_resolution.

The hardware, and thus the watchdog driver, only supports reset mode.

Configuration Options

The PNX8330 watchdog driver package should be loaded automatically when selecting a platform containing a PNX8330 processor, and it should never be necessary to load it explicitly into the configuration. The package is inactive unless the generic watchdog support CYGPKG_IO_WATCHDOG is loaded. Depending on the choice of eCos template it may be necessary to load the latter.

The package provides two main configuration options. CYGIMP_WATCHDOG_HARDWARE can be used to disable the use of the hardware watchdog and switch to a software emulation provided by the generic watchdog package instead. This may prove useful during debugging. CYGNUM_DEVS_WATCHDOG_MIPS_PNX8330_MILLISECONDS determines the timeout before the hardware watchdog resets the system. The default setting gives a 10-second timeout. The maximum timeout is determined by the CPU clock frequency, approximately 70 seconds for a 120MHz processor.

Porting

The watchdog device driver does not require any platform-specific support. The only porting effort required is to list CYGPKG_DEVS_WATCHDOG_MIPS_PNX8330 as one of the hardware packages in the ecos.db target entry.

Chapter 141. Synthetic Target Watchdog Device

Name

Synthetic Target Watchdog Device — Emulate watchdog hardware in the synthetic target

Overview

Some target hardware comes equipped with a watchdog timer. Application code can start this timer and after a certain period of time, typically a second, the watchdog will trigger. Usually this causes the hardware to reboot. The application can prevent this by regularly resetting the watchdog. An automatic reboot can be very useful when deploying hardware in the field: a hardware glitch could cause the unit to hang; or the software could receive an unexpected sequence of inputs, never seen in the laboratory, causing the system to lock up. Often the hardware is still functional, and a reboot sorts out the problem with only a brief interruption in service.

The synthetic target watchdog package emulates watchdog hardware. During system initialization watchdog device will be instantiated, and the `watchdog.tcl` script will be loaded by the I/O auxiliary. When the eCos application starts the watchdog device, the `watchdog.tcl` script will start checking the state of the eCos application at one second intervals. A watchdog reset call simply involves a message to the I/O auxiliary. If the `watchdog.tcl` script detects that a second has elapsed without a reset then it will send a `SIGPWR` signal to the eCos application, causing the latter to terminate. If gdb is being used to run the application, the user will get a chance to investigate what is happening. This behaviour is different from real hardware in that there is no automatic reboot, but the synthetic target is used only for development purposes, not deployment in the field: if a reboot is desired then this can be achieved very easily by using gdb commands to run another instance of the application.

Installation

Before a synthetic target eCos application can use a watchdog device it is necessary to build and install host-side support. The relevant code resides in the `host` subdirectory of the synthetic target watchdog package, and building it involves the standard **configure**, **make** and **make install** steps. The implementation of the watchdog support does not require any executables, just a Tcl script `watchdog.tcl` and some support files, so the **make** step is a no-op.

There are two main ways of building the host-side software. It is possible to build both the generic host-side software and all package-specific host-side software, including the watchdog support, in a single build tree. This involves using the **configure** script at the toplevel of the eCos repository. For more information on this, see the `README.host` file at the top of the repository. Note that if you have an existing build tree which does not include the synthetic target watchdog support then it will be necessary to rerun the toplevel configure script: the search for appropriate packages happens at configure time.

The alternative is to build just the host-side for this package. This requires a separate build directory, building directly in the source tree is disallowed. The **configure** options are much the same as for a build from the toplevel, and the `README.host` file can be consulted for more details. It is essential that the watchdog support be configured with the same `--prefix` option as other eCos host-side software, especially the I/O auxiliary provided by the architectural synthetic target HAL package, otherwise the I/O auxiliary will be unable to locate the watchdog support.

Target-side Configuration

The watchdog device depends on the generic watchdog support, `CYGPKG_IO_WATCHDOG`: if the generic support is absent then the watchdog device will be inactive. Some templates include this generic package by default, but not all. If the configuration does not include the generic package then it can be added using the eCos configuration tools, for example:

```
$ ecosconfig add CYGPKG_IO_WATCHDOG
```

By default the configuration will use the hardware-specific support, i.e. this package. However the generic watchdog package contains an alternative implementation using the kernel alarm facility, and that implementation can be selected if desired. However usually it will be better to rely on an external watchdog facility as provided by the I/O auxiliary and the `watchdog.tcl` script: if there are serious problems within the application, for example memory corruption, then an internal software-only implementation will not be reliable.

The watchdog resolution is currently fixed to one second: if the device does not receive a reset signal at least once a second then the watchdog will trigger and the eCos application will be terminated with a SIGPWR signal. The current implementation does not allow this resolution to be changed.

On some targets the watchdog device does not perform a hard reset. Instead the device works more or less via the interrupt subsystem, allowing application code to install action routines that will be called when the watchdog triggers. The synthetic target watchdog support effectively does perform a hard reset, by sending a SIGPWR signal to the eCos application, and there is no support for action routines.

The synthetic target watchdog package provides some configuration options for manipulating the compiler flags used for building the target-side code. That code is fairly simple, so for nearly all applications the default flags will suffice.

It should be noted that the watchdog device is subject to selective linking. Unless some code explicitly references the device, for example by calling the start and reset functions, the watchdog support will not appear in the final executable. This is desirable because a watchdog device has no effect until started.

Wallclock versus Elapsed Time

On real hardware the watchdog device uses wallclock time: if the device does not receive a reset signal within a set period of time then the watchdog will trigger. When developing for the synthetic target this is not always appropriate. There may be other processes running, using up some or most of the cpu time. For example, the application may be written such that it will issue a reset after some calculations which are known to complete within half a second, well within the one-second resolution of the watchdog device. However if other Linux processes are running then the synthetic target application may get timesliced, and half a second of computation may take several seconds of wallclock time.

Another problem with using wallclock time is that it interferes with debugging: if the application hits a breakpoint then it is unlikely that the user will manage to restart it in less than a second, and the watchdog will not get reset in time.

To avoid these problems the synthetic target watchdog normally uses consumed cpu time rather than wallclock time. If the application is timesliced or if it is halted inside gdb then it does not consume any cpu time. The application actually has to spend a whole second's worth of cpu cycles without issuing a reset before the watchdog triggers.

However using consumed cpu time is not a perfect solution either. If the application makes blocking system calls then it is not using cpu time. Interaction with the I/O auxiliary involves system calls, but these should take only a short amount of time so their effects can be ignored. If the application makes direct system calls such as `cyg_hal_sys_read` then the system behaviour becomes undefined. In addition by default the idle thread will make blocking `select` system calls, effectively waiting until an interrupt occurs. If an application spends much of its time idle then the watchdog device may take much longer to trigger than expected. It may be desirable to enable the synthetic target HAL configuration option CYGIMP_HAL_IDLE_THREAD_SPIN, causing the idle thread to spin rather than block, at the cost of wasted cpu cycles.

The default is to use consumed cpu time, but this can be changed in the target definition file:

```
synth_device watchdog {
    use wallclock_time
    …
}
```

User Interface

When the synthetic target is run in graphical mode the watchdog device extends the user interface in two ways. The Help menu is extended with an entry for the watchdog-specific documentation. There is also a graphical display of the current state of the watchdog. Initially the watchdog is asleep:

When application code starts the device the watchdog will begin to keep an eye on things (or occasionally both eyes).

If the watchdog triggers the display will change again, and optionally the user can receive an audible alert. The location of the watchdog display within the I/O auxiliary's window can be controlled via a **watchdog_pack** entry in the target definition file. For example the following can be used to put the watchdog display to the right of the central text window:

```
synth_device watchdog {
    watchdog_pack -in .main.e -side top
    …
}
```

The user interface section of the generic synthetic target HAL documentation can be consulted for more information on window packing.

By default the watchdog support will not generate an audible alert when the watchdog triggers, to avoid annoying colleagues. Sound can be enabled in the target definition file, and two suitable files `sound1.au` and `sound2.au` are supplied as standard:

```
synth_device watchdog {
    sound sound1.au
    …
}
```

An absolute path can be specified if desired:

```
synth_device watchdog {
    sound /usr/share/emacs/site-lisp/emacspeak/sounds/default-8k/alarm.au
    …
}
```

Sound facilities are not built into the I/O auxiliary itself, instead an external program is used. The default player is **play**, a front-end to the sox application shipped with some Linux distributions. If another player should be used then this can be specified in the target definition file:

```
synth_device watchdog {
    …
    sound_player my_sound_player
```

The specified program will be run in the background with a single argument, the sound file.

Command Line Arguments

The watchdog support does not use any command line arguments. All configuration is handled through the target definition file.

Hooks

The watchdog support does not provide any hooks for use by other scripts. There is rarely any need for customizing the system's behaviour when a watchdog triggers because those should be rare events, even during application development.

Additional Tcl Procedures

The watchdog support does not provide any additional Tcl procedures or variables for use by other scripts.

Part XXXIX. eCos POSIX compatibility layer

Table of Contents

Chapter 142. POSIX Standard Support

eCos contains support for the POSIX Specification (ISO/IEC 9945-1)[POSIX].

POSIX support is divided between the POSIX and the FILEIO packages. The POSIX package provides support for threads, signals, synchronization, timers and message queues. The FILEIO package provides support for file and device I/O. The two packages may be used together or separately, depending on configuration.

This document takes a functional approach to the POSIX library. Support for a function implies that the data types and definitions necessary to support that function, and the objects it manipulates, are also defined. Any exceptions to this are noted, and unless otherwise noted, implemented functions behave as specified in the POSIX standard.

This document only covers the differences between the eCos implementation and the standard; it does not provide complete documentation. For full information, see the POSIX standard [POSIX]. Online, the Open Group Single Unix Specification [SUS2] provides complete documentation of a superset of POSIX. If you have access to a Unix system with POSIX compatibility, then the manual pages for this will be of use. There are also a number of books available. [Lewine] covers the process, signal, file and I/O functions, while [Lewis1], [Lewis2], [Nichols] and [Norton] cover Pthreads and related topics (see Bibliography, xref). However, many of these books are oriented toward using POSIX in non-embedded systems, so care should be taken in applying them to programming under eCos.

The remainder of this chapter broadly follows the structure of the POSIX Specification. References to the appropriate section of the Standard are included.

Omitted functions marked with “// TBA” are potential candidates for later implementation.

Process Primitives [POSIX Section 3]

Functions Implemented

```
int kill(pid_t pid, int sig);
int pthread_kill(pthread_t thread, int sig);
int sigaction(int sig, const struct sigaction *act, struct sigaction *oact);
int sigqueue(pid_t pid, int sig, const union sigval value);
int sigprocmask(int how, const sigset_t *set, sigset_t *oset);
int pthread_sigmask(int how, const sigset_t *set, sigset_t *oset);
int sigpending(sigset_t *set);
int sigsuspend(const sigset_t *set);
int sigwait(const sigset_t *set, int *sig);
int sigwaitinfo(const sigset_t *set, siginfo_t *info);
int sigtimedwait(const sigset_t *set, siginfo_t *info, const struct timespec *timeout);
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signo);
int sigdelset(sigset_t *set, int signo);
int sigismember(const sigset_t *set, int signo);
unsigned int alarm( unsigned int seconds );
int pause( void );
unsigned int sleep( unsigned int seconds );
```

Functions Omitted

```
pid_t fork(void);
int execl( const char *path, const char *arg, … );
int execv( const char *path, char *const argv[] );
int execle( const char *path, const char *arg, … );
int execve( const char *path, char *const argv[], char *const envp[] );
int execlp( const char *path, const char *arg, … );
int execvp( const char *path, char *const argv[] );
int pthread_atfork( void(*prepare)(void), void (*parent)(void),  void (*child)() );
pid_t wait( int *stat_loc );
pid_t waitpid( pid_t pid, int *stat_loc, int options );
void _exit( int status );
```

Notes

- Signal handling may be enabled or disabled with the CYGPKG_POSIX_SIGNALS option. Since signals are used by other POSIX components, such as timers, disabling signals will disable those components too.
- *kill()* and *sigqueue()* may only take a **pid** argument of zero, which maps to the current process.
- The *SIGEV_THREAD* notification type is not currently implemented.
- Job Control and Memory Protection signals are not supported.
- An extra implementation defined *si_code* value, *SI_EXCEPT*, is defined to distinguish hardware generated exceptions from others.
- Extra signals are defined: _SIGTRAP_, _SIGIOT_, _SIGEMT_, _SIGSYS_ These are largely to maintain compatibility with the signal numbers used by GDB.
- Signal delivery may currently occur at unexpected places in some API functions. Using *longjmp()* to transfer control out of a signal handler may result in the interrupted function not being able to complete properly. This may result in later function calls failing or deadlocking.

Process Environment [POSIX Section 4]

Functions Implemented

```
int uname( struct utsname *name );
time_t time( time_t *tloc );
char *getenv( const char *name );
int isatty( int fd );
long sysconf( int name );
```

Functions Omitted

```
pid_t getpid( void );
pid_t getppid( void );
uid_t getuid( void );
uid_t geteuid( void );
gid_t getgid( void );
gid_t getegid( void );
int setuid( uid_t uid );
int setgid( gid_t gid );
int getgroups( int gidsetsize, gid_t grouplist[] );
char *getlogin( void );
int getlogin_r( char *name, size_t namesize );
pid_t getpgrp( void );
pid_t setsid( void );
int setpgid( pid_t pid, pid_t pgid );
char *ctermid( char *s);
char *ttyname( int fd );                             // TBA
int ttyname_r( int fd, char *name, size_t namesize); // TBA
clock_t times( struct tms *buffer );                 // TBA
```

Notes

- The fields of the *utsname* structure are initialized as follows:

  ```
  sysname “eCos”
  nodename  “”  (gethostname() is currently not available)

  release   Major version number of the kernel
  version       Minor version number of the kernel
  machine “”  (Requires some config tool changes)
  ```

  The sizes of these strings are defined by CYG_POSIX_UTSNAME_LENGTH and CYG_POSIX_UTSNAME_NODENAME_LENGTH. The latter defaults to the value of the former, but may also be set independently to accommodate a longer node name.

- The *time()* function is currently implemented in the C library.

- A set of environment strings may be defined at configuration time with the CYGDAT_LIBC_DEFAULT_ENVIRONMENT option. The application may also define an environment by direct assignment to the **environ** variable.

- At present *isatty()* assumes that any character device is a tty and that all other devices are not ttys. Since the only kind of device that eCos currently supports is serial character devices, this is an adequate distinction.

- All system variables supported by sysconf will yield a value. However, those that are irrelevant to eCos will either return the default minimum defined in `<limits.h>`, or zero.

Files and Directories [POSIX Section 5]

Functions Implemented

```
DIR *opendir( const char *dirname );
struct dirent *readdir( DIR *dirp );
int readdir_r( DIR *dirp, struct dirent *entry, struct dirent **result );
void rewinddir( DIR *dirp );
int closedir( DIR *dirp );
int chdir( const char *path );
char *getcwd( char *buf, size_t size );
int open( const char * path , int oflag , … );
int creat( const char * path, mode_t mode );
int link( const char *existing, const char *new );
int mkdir( const char *path, mode_t mode );
int unlink( const char *path );
int rmdir( const char *path );
int rename( const char *old, const char *new );
int stat( const char *path, struct stat *buf );
int fstat( int fd, struct stat *buf );
int access( const char *path, int amode );
long pathconf(const char *path, int name);
long fpathconf(int fd, int name);
```

Functions Omitted

```
mode_t umask( mode_t cmask );
int mkfifo( const char *path, mode_t mode );
int chmod( const char *path, mode_t mode );                 // TBA
int fchmod( int fd, mode_t mode );                          // TBA
int chown( const char *path, uid_t owner, gid_t group );
int utime( const char *path, const struct utimbuf *times ); // TBA
int ftruncate( int fd, off_t length );                      // TBA
```

Notes

- If a call to `open()` or `creat()` supplies the third _mode_ parameter, it will currently be ignored.
- Most of the functionality of these functions depends on the underlying filesystem.
- Currently *access()* only checks the *F_OK* mode explicitly, the others are all assumed to be true by default.
- The maximum number of open files allowed is supplied by the CYGNUM_FILEIO_NFILE option. The maximum number of file descriptors is supplied by the CYGNUM_FILEIO_NFD option.

Input and Output [POSIX Section 6]

Functions Implemented

```
int dup( int fd );
int dup2( int fd, int fd2 );
int close(int fd);
ssize_t read(int fd, void *buf, size_t nbyte);
ssize_t write(int fd, const void *buf, size_t nbyte);
int fcntl( int fd, int cmd, … );
off_t lseek(int fd, off_t offset, int whence);
int fsync( int fd );
int fdatasync( int fd );
```

Functions Omitted

```
int pipe( int fildes[2] );
int aio_read( struct aiocb *aiocbp );                        // TBA
int aio_write( struct aiocb *aiocbp );                       // TBA
int lio_listio( int mode, struct aiocb *const list[],
                int nent, struct sigevent *sig);             // TBA
int aio_error( struct aiocb *aiocbp );                       // TBA
int aio_return( struct aiocb *aiocbp );                      // TBA
int aio_cancel( int fd, struct aiocb *aiocbp );              // TBA
int aio_suspend( const struct aiocb *const list[],
                 int nent, const struct timespec *timeout );  // TBA
int aio_fsync( int op, struct aiocb *aiocbp );               // TBA
```

Notes

- Only the *F_DUPFD* command of *fcntl()* is currently implemented.
- Most of the functionality of these functions depends on the underlying filesystem.

Device and Class Specific Functions [POSIX Section 7]

Functions Implemented

```
speed_t cfgetospeed( const struct termios *termios_p );
int cfsetospeed( struct termios *termios_p, speed_t speed );
speed_t cfgetispeed( const struct termios *termios_p );
int cfsetispeed( struct termios *termios_p, speed_t speed );
int tcgetattr( int fd, struct termios *termios_p );
int tcsetattr( int fd, int optional_actions,  const struct termios *termios_p );
int tcsendbreak( int fd, int duration );
int tcdrain( int fd );
int tcflush( int fd, int queue_selector );
int tcsendbreak( int fd, int action );
```

Functions Omitted

```
pid_t tcgetpgrp( int fd );
int tcsetpgrp( int fd, pid_t pgrp );
```

Notes

- Only the functionality relevant to basic serial device control is implemented. Only very limited support for canonical input is provided, and then only via the “tty” devices, not the “serial” devices. None of the functionality relevant to job control, controlling terminals and sessions is implemented.
- Only *MIN* = 0 and *TIME* = 0 functionality is provided.
- Hardware flow control is supported if the underlying device driver and serial port support it.
- Support for break, framing and parity errors depends on the functionality of the hardware and device driver.

C Language Services [POSIX Section 8]

Functions Implemented

```
char *setlocale( int category, const char *locale );
int fileno( FILE *stream );
FILE *fdopen( int fd, const char *type );
int getc_unlocked( FILE *stream);
int getchar_unlocked( void );
int putc_unlocked( FILE *stream );
int putchar_unlocked( void );
char *strtok_r( char *s, const char *sep, char **lasts );
char *asctime_r( const struct tm *tm, char *buf );
char *ctime_r( const time_t *clock, char *buf );
struct tm *gmtime_r( const time_t *clock, struct tm *result );
struct tm *localtime_r( const time_t *clock, struct tm *result );
int rand_r( unsigned int *seed );
```

Functions Omitted

```
void flockfile( FILE *file );
int ftrylockfile( FILE *file );
void funlockfile( FILE *file );
int sigsetjmp( sigjmp_buf env, int savemask );      // TBA
void siglongjmp( sigjmp_buf env, int val );         // TBA
void tzset(void);                                   // TBA
```

Notes

- *setlocale()* is implemented in the C library Internationalization package.
- Functions *fileno()* and *fdopen()* are implemented in the C library STDIO package.
- Functions *getc_unlocked()*, *getchar_unlocked()*, *putc_unlocked()* and *putchar_unlocked()* are defined but are currently identical to their non-unlocked equivalents.

- *strtok_r()*, *asctime_r()*, *ctime_r()*, *gmtime_r()*, *localtime_r()* and *rand_r()* are all currently in the C library, alongside their non-reentrant versions.

System Databases [POSIX Section 9]

Functions Implemented

<none>

Functions Omitted

```
struct group *getgrgid( gid_t gid );
int getgrgid( gid_t gid, struct group *grp, char *buffer, size_t bufsize, struct group **result );
struct group *getgrname( const char *name );
int getgrname_r( const char *name,    struct group *grp,      char *buffer,
                 size_t     bufsize,  struct group **result );
struct passwd *getpwuid( uid_t uid );
int getpwuid_r( uid_t   uid,      struct passwd *pwd,         char *buffer,
                size_t  bufsize,  struct passwd **result );
struct passwd *getpwnam( const char *name );
int getpwnam_r( const char *name,    struct passwd *pwd,      char *buffer,
                size_t      bufsize, struct passwd **result );
```

Notes

- None of the functions in this section are implemented.

Data Interchange Format [POSIX Section 10]

This section details *tar* and *cpio* formats. Neither of these is supported by eCos.

Synchronization [POSIX Section 11]

Functions Implemented

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_destroy(sem_t *sem);
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_post(sem_t *sem);
int sem_getvalue(sem_t *sem, int *sval);
int pthread_mutexattr_init( pthread_mutexattr_t *attr);
int pthread_mutexattr_destroy( pthread_mutexattr_t *attr);
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *mutex_attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_timedlock(pthread_mutex_t *mutex, const struct timespec *abstime);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_condattr_init(pthread_condattr_t *attr);
int pthread_condattr_destroy(pthread_condattr_t *attr);
int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr);
int pthread_cond_destroy(pthread_cond_t *cond);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex,
                           const struct timespec *abstime);
```

Functions Omitted

```
sem_t *sem_open(const char *name, int oflag, …);   // TBA
int sem_close(sem_t *sem);          // TBA
int sem_unlink(const char *name);       // TBA
int pthread_mutexattr_getpshared( const pthread_mutexattr_t *attr,
                                  int                       *pshared );
int pthread_mutexattr_setpshared( const pthread_mutexattr_t *attr,
                                  int                       pshared );
int  pthread_condattr_getpshared( const pthread_condattr_t  *attr,
                                  int                       *pshared);
int  pthread_condattr_setpshared( const pthread_condattr_t  *attr,
                                  int                       pshared);
```

Notes

- The presence of semaphores is controlled by the CYGPKG_POSIX_SEMAPHORES option. This in turn causes the _POSIX_SEMAPHORES feature test macro to be defined and the semaphore API to be made available.

- The **pshared** argument to *sem_init()* is not implemented, its value is ignored.

- Functions *sem_open()*, *sem_close()* and *sem_unlink()* are present but always return an error (ENOSYS).

- The exact priority inversion protocols supported may be controlled with the _POSIX_THREAD_PRIO_INHERIT and _POSIX_THREAD_PRIO_PROTECT configuration options.

- {_POSIX_THREAD_PROCESS_SHARED} is not defined, so the **process-shared** mutex and condition variable attributes are not supported, and neither are the functions *pthread_mutexattr_getpshared()*, *pthread_mutexattr_setpshared()*, *pthread_condattr_getpshared()* and *pthread_condattr_setpshared()*.

- Condition variables do not become bound to a particular mutex when *pthread_cond_wait()* is called. Hence different threads may wait on a condition variable with different mutexes. This is at variance with the standard, which requires a condition variable to become (dynamically) bound by the first waiter, and unbound when the last finishes. However, this difference is largely benign, and the cost of policing this feature is non-trivial.

Memory Management [POSIX Section 12]

Functions Implemented

<none>

Functions Omitted

```
int mlockall( int flags );
int munlockall( void );
int mlock( const void *addr, size_t len );
int munlock( const void *addr, size_t len );
void mmap( void *addr, size_t len, int prot, int flags, int fd, off_t off );
int munmap( void *addr, size_t len );
int mprotect( const void *addr, size_t len, int prot );
int msync( void *addr, size_t len, int flags );
int shm_open( const char *name, int oflag, mode_t mode );
int shm_unlink( const char *name );
```

Notes

None of these functions are currently provided. Some may be implemented in a restricted form in the future.

Execution Scheduling [POSIX Section 13]

Functions Implemented

```
int sched_yield(void);
int sched_get_priority_max(int policy);
int sched_get_priority_min(int policy);
int sched_rr_get_interval(pid_t pid, struct timespec *t);
int pthread_attr_setscope(pthread_attr_t *attr, int scope);
int pthread_attr_getscope(const pthread_attr_t *attr, int *scope);
int pthread_attr_setinheritsched(pthread_attr_t *attr, int inherit);
int pthread_attr_getinheritsched(const pthread_attr_t *attr, int *inherit);
int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
int pthread_attr_getschedpolicy(const pthread_attr_t *attr, int *policy);
int pthread_attr_setschedparam(pthread_attr_t *attr, const struct sched_param *param);
int pthread_attr_getschedparam(const pthread_attr_t *attr, struct sched_param *param);
int pthread_setschedparam(pthread_t thread, int policy, const struct sched_param *param);
int pthread_getschedparam(pthread_t thread, int *policy, struct sched_param *param);
int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr, int protocol);
int pthread_mutexattr_getprotocol(pthread_mutexattr_t *attr, int *protocol);
int pthread_mutexattr_setprioceiling(pthread_mutexattr_t *attr, int prioceiling);
int pthread_mutexattr_getprioceiling(pthread_mutexattr_t *attr, int *prioceiling);
int pthread_mutex_setprioceiling(pthread_mutex_t *mutex, int prioceiling, int *old_ceiling);
int pthread_mutex_getprioceiling(pthread_mutex_t *mutex, int *prioceiling);
```

Functions Omitted

```
int sched_setparam(pid_t pid, const struct sched_param *param);
int sched_getparam(pid_t pid, struct sched_param *param);
int sched_setscheduler(pid_t pid, int policy, const struct sched_param *param);
int sched_getscheduler(pid_t pid);
```

Notes

- The functions *sched_setparam()*, *sched_getparam()*, *sched_setscheduler()* and *sched_getscheduler()* are present but always return an error.
- The scheduler policy *SCHED_OTHER* is equivalent to *SCHED_RR*.
- Only *PTHREAD_SCOPE_SYSTEM* is supported as a **contentionscope** attribute.
- The default thread scheduling attributes are:

  ```
  contentionscope         PTHREAD_SCOPE_SYSTEM
  inheritsched            PTHREAD_INHERIT_SCHED
  schedpolicy             SCHED_OTHER
  chedparam.sched        0
  ```

- Mutex priority inversion protection is controlled by a number of kernel configuration options. If CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_INHERIT is defined then {_POSIX_THREAD_PRIO_INHERIT} will be defined and PTHREAD_PRIO_INHERIT may be set as the protocol in a *pthread_mutexattr_t* object. If CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_CEILING is defined then {_POSIX_THREAD_PRIO_PROTECT} will be defined and PTHREAD_PRIO_PROTECT may be set as the protocol in a *pthread_mutexattr_t* object.

- The default attribute values set by *pthread_mutexattr_init()* is to set the protocol attribute to PTHREAD_PRIO_NONE and the prioceiling attribute to zero.

Clocks and Timers [POSIX Section 14]

Functions Implemented

```
int clock_settime( clockid_t clock_id,
const struct timespec *tp);
int clock_gettime( clockid_t clock_id, struct timespec *tp);
int clock_getres( clockid_t clock_id, struct timespec *tp);
int timer_create( clockid_t clock_id, struct sigevent *evp, timer_t *timer_id);
int timer_delete( timer_t timer_id );
int timer_settime( timer_t                  timerid,
                   int                      flags,
                   const struct itimerspec  *value,
                   struct itimerspec        *ovalue );
int timer_gettime( timer_t timerid, struct itimerspec *value );
int timer_getoverrun( timer_t timerid );
int nanosleep( const struct timespec *rqtp, struct timespec *rmtp);
int gettimeofday(struct timeval *tv, struct timezone* tz);
```

Functions Omitted

<none>

Notes

- Currently *timer_getoverrun()* only reports timer notifications that are delayed in the timer subsystem. If they are delayed in the signal subsystem, due to signal masks for example, this is not counted as an overrun.

- The option CYGPKG_POSIX_TIMERS allows the timer support to be enabled or disabled, and causes _POSIX_TIMERS to be defined appropriately. This will cause other parts of the POSIX system to have limited functionality.

Message Passing [POSIX Section 15]

Functions Implemented

```
mqd_t mq_open( const char *name, int  oflag, … );
int mq_close( mqd_t  mqdes );
int mq_unlink( const char *name );
int mq_send( mqd_t mqdes, const char *msg_ptr,  size_t msg_len, unsigned int msg_prio );
ssize_t mq_receive( mqd_t mqdes, char *msg_ptr, size_t msg_len, unsigned int *msg_prio );
int mq_setattr( mqd_t mqdes, const struct mq_attr *mqstat, struct mq_attr *omqstat );
int mq_getattr( mqd_t mqdes, struct mq_attr *mqstat );
int mq_notify( mqd_t mqdes, const struct sigevent *notification );
```

From POSIX 1003.1d draft:

```
int mq_send( mqd_t        mqdes,
             const char   *msg_ptr,
             size_t       msg_len,
             unsigned int msg_prio,
             const struct timespec *abs_timeout );

ssize_t mq_receive( mqd_t         mqdes,
                    char          *msg_ptr,
                    size_t        msg_len,
                    unsigned int  *msg_prio,
                    const struct timespec *abs_timeout );
```

Functions Omitted

<none>

Notes

- The presence of message queues is controlled by the CYGPKG_POSIX_MQUEUES option. Setting this will cause [_POSIX_MESSAGE_PASSING] to be defined and the message queue API to be made available.
- Message queues are not currently filesystem objects. They live in their own name and descriptor spaces.

Thread Management [POSIX Section 16]

Functions Implemented

```
int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
int pthread_attr_getdetachstate(const pthread_attr_t *attr, int *detachstate);
int pthread_attr_setstackaddr(pthread_attr_t *attr,  void *stackaddr);
int pthread_attr_getstackaddr(const pthread_attr_t *attr, void **stackaddr);
int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);
int pthread_attr_getstacksize(const pthread_attr_t *attr, size_t *stacksize);
int pthread_create( pthread_t             *thread,
                    const pthread_attr_t  *attr,
                    void                  *(*start_routine)(void *),
                    void                  *arg);
pthread_t pthread_self( void );
int pthread_equal(pthread_t thread1, pthread_t thread2);
void pthread_exit(void *retval);
int pthread_join(pthread_t thread, void **thread_return);
int pthread_detach(pthread_t thread);
int pthread_once(pthread_once_t *once_control, void (*init_routine)(void));
```

Functions Omitted

<none>

Notes

- The presence of thread support as a whole is controlled by the the CYGPKG_POSIX_PTHREAD configuration option. Note that disabling this will also disable many other features of the POSIX package, since these are intimately bound up with the thread mechanism.
- The default (non-scheduling) thread attributes are:

  ```
  detachstate             PTHREAD_CREATE_JOINABLE
  stackaddr               unset
  stacksize               unset
  ```

- Dynamic thread stack allocation is only provided if there is an implementation of *malloc()* configured (i.e. a package implements the CYGINT_MEMALLOC_MALLOC_ALLOCATORS interface). If there is no malloc() available, then the thread creator must supply a stack. If only a stack address is supplied then the stack is assumed to be PTHREAD_STACK_MIN bytes long.

This size is seldom useful for any but the most trivial of threads. If a different sized stack is used, both the stack address and stack size must be supplied.

- The value of PTHREAD_THREADS_MAX is supplied by the CYGNUM_POSIX_PTHREAD_THREADS_MAX option. This defines the maximum number of threads allowed. The POSIX standard requires this value to be at least 64, and this is the default value set.
- When the POSIX package is installed, the thread that calls *main()* is initialized as a POSIX thread. The priority of that thread is controlled by the CYGNUM_POSIX_MAIN_DEFAULT_PRIORITY option.

Thread-Specific Data [POSIX Section 17]

Functions Implemented

```
int pthread_key_create(pthread_key_t *key, void (*destructor)(void *));
int pthread_setspecific(pthread_key_t key, const void *pointer);
void *pthread_getspecific(pthread_key_t key);
int pthread_key_delete(pthread_key_t key);
```

Functions Omitted

<none>

Notes

- The value of PTHREAD_DESTRUCTOR_ITERATIONS is provided by the CYGNUM_POSIX_PTHREAD_DESTRUCTOR_ITERATIONS option. This controls the number of times that a key destructor will be called while the data item remains non-NULL.
- The value of PTHREAD_KEYS_MAX is provided by the CYGNUM_POSIX_PTHREAD_KEYS_MAX option. This defines the maximum number of per-thread data items supported. The POSIX standard calls for this to be a minimum of 128, which is rather large for an embedded system. The default value for this option is set to 128 for compatibility but it should be reduced to a more usable value.

Thread Cancellation [POSIX Section 18]

Functions Implemented

```
int pthread_cancel(pthread_t thread);
int pthread_setcancelstate(int state, int *oldstate);
int pthread_setcanceltype(int type, int *oldtype);
void pthread_testcancel(void);
void pthread_cleanup_push( void (*routine)(void *), void *arg);
void pthread_cleanup_pop( int execute);
```

Functions Omitted

<none>

Notes

Asynchronous cancellation is only partially implemented. In particular, cancellation may occur in unexpected places in some functions. It is strongly recommended that only synchronous cancellation be used.

Non-POSIX Functions

In addition to the standard POSIX functions defined above, the following non-POSIX functions are defined in the FILEIO package.

General I/O Functions

```
int ioctl(int fd, CYG_ADDRWORD com, CYG_ADDRWORD data);
int select(int nfd, fd_set *in, fd_set *out, fd_set *ex, struct timeval *tv);
```

Socket Functions

```
int socket( int domain, int type, int protocol);
int bind(int s, const struct sockaddr *sa, unsigned int len);
int listen(int s, int len);
int accept(int s, struct sockaddr *sa, socklen_t *addrlen);
int connect(int s, const struct sockaddr *sa, socklen_t len);
int getpeername(int s, struct sockaddr *sa, socklen_t *len);
int getsockname(int s, struct sockaddr *sa, socklen_t *len);
int setsockopt(int s, int level, int optname, const void *optval, socklen_t optlen);
int getsockopt(int s, int level, int optname, void *optval, socklen_t *optlen);
ssize_t recvmsg(int s, struct msghdr *msg, int flags);
ssize_t recvfrom(int s, void *buf, size_t len, int flags, struct sockaddr *from, socklen_t *fromlen);
ssize_t recv(int s, void *buf, size_t len, int flags);
ssize_t sendmsg(int s, const struct msghdr *msg, int flags);
ssize_t sendto(int                    s,
               const void             *buf,
               size_t                 len,
               int                    flags,
               const struct sockaddr  *to,
               socklen_t              tolen);
ssize_t send(int s, const void *buf, size_t len, int flags);
int shutdown(int s, int how);
```

Notes

- The precise behaviour of these functions depends mainly on the functionality of the underlying filesystem or network stack to which they are applied.

References and Bibliography

[Lewine] Donald A. Lweine Posix Programmer,s Guide: Writing Portable Unix Programs With the POSIX.1 Standard O,Reilly & Associates; ISBN: 0937175730.

[Lewis1] Bil Lewis Daniel J. Berg Threads Primer: A Guide to Multithreaded Programming Prentice Hall ISBN: 013443698

[Lewis2] Bil Lewis Daniel J. Berg Multithreaded Programming With Pthreads Prentice Hall Computer Books ISBN: 0136807291

[Nichols] Bradford Nichols Dick Buttlar Jacqueline Proulx Farrell Pthreads Programming: A POSIX Standard for Better Multiprocessing (O,Reilly Nutshell) O,Reilly & Associates ISBN: 1565921151

[Norton] Scott J. Norton Mark D. Depasquale Thread Time: The MultiThreaded Programming Guide Prentice Hall ISBN: 0131900676

[POSIX] Portable Operating System Interface(POSIX) - Part 1: System Application Programming Interface (API)[C Language] ISO/IEC 9945-1:1996, IEEE

[SUS2] Open Group; Single Unix Specification, Version 2 http://www.opengroup.org/public/pubs/online/7908799/index.html

Part XL. µITRON

Table of Contents

Chapter 143. µITRON API

Introduction to µITRON

The µITRON specification defines a highly flexible operating system architecture designed specifically for application in embedded systems. The specification addresses features which are common to the majority of processor architectures and deliberately avoids virtualization which would adversely impact real-time performance. The µITRON specification may be implemented on many hardware platforms and provides significant advantages by reducing the effort involved in understanding and porting application software to new processor architectures.

Several revisions of the µITRON specification exist. In this release, *eCos* supports the µITRON version 3.02 specification, with complete “Standard functionality” (level S), plus many “Extended” (level E) functions. An exception is get_tid() which has µITRON 4 semantics. The definitive reference on µITRON is Dr. Sakamura,s book: *µITRON 3.0, An Open and Portable Real-Time Operating System for Embedded Systems*. The book can be purchased from the IEEE Press, and an ASCII version of the standard can be found online at http://www.t-engine.org/specifications#d.

µITRON and *eCos*

The *eCos* kernel implements the functionality used by the µITRON compatibility subsystem. The configuration of the kernel influences the behavior of µITRON programs.

In particular, the default configuration has time slicing (also known as round-robin scheduling) switched on; this means that a task can be moved from RUN state to READY state at any time, in order that one of its peers may run. This is not strictly conformant to the µITRON specification, which states that timeslicing may be implemented by periodically issuing a rot_rdq(0) call from within a periodic task or cyclic handler; otherwise it is expected that a task runs until it is pre-empted in consequence of synchronization or communication calls it makes, or the effects of an interrupt or other external event on a higher priority task cause that task to become READY. To disable timeslicing functionality in the kernel and µITRON compatibility environment, please disable the CYGSEM_KERNEL_SCHED_TIMESLICE configuration option in the kernel package. A description of kernel scheduling is in Kernel Overview.

For another example, the semantics of task queueing when waiting on a synchronization object depend solely on the way the underlying kernel is configured. As discussed above, the multi-level queue scheduler is the only one which is µITRON compliant, and it queues waiting tasks in FIFO order. Future releases of that scheduler might be configurable to support priority ordering of task queues. Other schedulers might be different again: for example the bitmap scheduler can be used with the µITRON compatibility layer, even though it only allows one task at each priority and as such is not µITRON compliant, but it supports only priority ordering of task queues. So which queueing scheme is supported is not really a property of the µITRON compatibility layer; it depends on the kernel.

In this version of the µITRON compatibility layer, the calls to disable and enable scheduling and interrupts (dis_dsp(), ena_dsp(), loc_cpu() and unl_cpu()) call underlying kernel functions; in particular, the xxx_dsp() functions lock the scheduler entirely, which prevents dispatching of DSRs; functions implemented by DSRs include clock counters and alarm timers. Thus time “stops” while dispatching is disabled with dis_dsp().

Like all parts of the *eCos* system, the detailed semantics of the µITRON layer are dependent on its configuration and the configuration of other components that it uses. The µITRON configuration options are all defined in the file pkgconf/uitron.h, and can be set using the configuration tool or editing the .ecc file in your build directory by hand.

An important configuration option for the µITRON compatibility layer is “Option: Return Error Codes for Bad Params” (CYGSEM_UITRON_BAD_PARAMS_RETURN_ERRORS), which allows a lot of the error checking code in the µITRON compatibility layer to be removed. Of course this leaves a program open to undetected errors, so it should only be used once an application is fully debugged and tested. Its benefits include reduced code size and faster execution. However, it affects the API significantly, in that with this option enabled, bad calls do not return errors, but cause an assert failure (if that is itself enabled) or malfunction internally. There is discussion in more detail about this in each section below.

We now give a brief description of the µITRON functions which are implemented in this release. Note that all C and C++ source files should have the following `#include` statement:

```
#include <cyg/compat/uitron/uit_func.h>
```

Task Management Functions

The following functions are fully supported in this release:

```
ER sta_tsk(
  ID tskid,
  INT stacd )
```

```
void ext_tsk( void )
```

```
void exd_tsk( void )
```

```
ER dis_dsp( void )
```

```
ER ena_dsp( void )
```

```
ER chg_pri(
  ID tskid,
  PRI tskpri )
```

```
ER rot_rdq(
  PRI tskpri )
```

```
ER get_tid(
  ID *p_tskid )
```

```
ER ref_tsk(
  T_RTSK *pk_rtsk,
  ID tskid )
```

```
ER ter_tsk(
  ID tskid )
```

```
ER rel_wai(
  ID tskid )
```

The following two functions are supported in this release, when enabled with the configuration option `CYGPKG_UITRON_TASKS_CREATE_DELETE` with some restrictions:

```
ER cre_tsk(
  ID tskid,
  T_CTSK *pk_ctsk )
```

```
ER del_tsk(
  ID tskid )
```

These functions are restricted as follows:

Because of the static initialization facilities provided for system objects, a task is allocated stack space statically in the configuration. So while tasks can be created and deleted, the same stack space is used for that task (task ID number) each time. Thus the stack size (pk_ctsk->stksz) requested in `cre_tsk()` is checked for being less than that which was statically allocated, and otherwise ignored. This ensures that the new task will have enough stack to run. For this reason `del_tsk()` does not in any sense free the memory that was in use for the task's stack.

The task attributes (pk_ctsk->tskatr) are ignored; current versions of *eCos* do not need to know whether a task is written in assembler or C/C++ so long as the procedure call standard appropriate to the CPU is followed.

Extended information (pk_ctsk->exinf) is ignored.

Error checking

For all these calls, an invalid task id (tskid) (less than 1 or greater than the number of configured tasks) only returns E_ID when bad params return errors (`CYGSEM_UITRON_BAD_PARAMS_RETURN_ERRORS` is enabled, see above).

Similarly, the following conditions are only checked for, and only return errors if `CYGSEM_UITRON_BAD_PARAMS_RETURN_ERRORS` is enabled:

- pk_crtk in `cre_tsk()` is a valid pointer, otherwise return E_PAR
- `ter_tsk()` or `rel_wai()` on the calling task returns E_OBJ
- the CPU is not locked already in `dis_dsp()` and `ena_dsp()` ; returns E_CTX
- priority level in `chg_pri()` and `rot_rdq()` is checked for validity, E_PAR
- return value pointer in `get_tid()` and `ref_tsk()` is a valid pointer, or E_PAR

The following conditions are checked for, and return error codes if appropriate, regardless of the setting of `CYGSEM_UITRON_BAD_PARAMS_RETURN_ERRORS` :

- When create and delete functions `cre_tsk()` and `del_tsk()` are supported, all calls which use a valid task ID number check that the task exists; if not, E_NOEXS is returned
- When supported, `cre_tsk()` : the task must not already exist; otherwise E_OBJ
- When supported, `cre_tsk()` : the requested stack size must not be larger than that statically configured for the task; see the configuration options “Static initializers”, and “Default stack size”. Else E_NOMEM
- When supported, `del_tsk()` : the underlying *eCos* thread must not be running - this would imply either a bug or some program bypassing the µITRON compatibility layer and manipulating the thread directly. E_OBJ
- `sta_tsk()` : the task must be dormant, else E_OBJ
- `ter_tsk()` : the task must not be dormant, else E_OBJ
- `chg_pri()` : the task must not be dormant, else E_OBJ
- `rel_wai()` : the task must be in `WAIT` or `WAIT-SUSPEND` state, else E_OBJ

Task-Dependent Synchronization Functions

These functions are fully supported in this release:

```
ER sus_tsk(
    ID tskid )
```

```
ER rsm_tsk(
    ID tskid )
```

```
ER frsm_tsk(
    ID tskid )
```

```
ER slp_tsk( void )
```

```
ER tslp_tsk(
    TMO tmout )
```

```
ER wup_tsk(
    ID tskid )
```

```
ER can_wup(
    INT *p_wupcnt,    ID tskid )
```

Error checking

The following conditions are only checked for, and only return errors if CYGSEM_UITRON_BAD_PARAMS_RETURN_ERRORS is enabled (see the configuration option “Return Error Codes for Bad Params”):

- invalid tskid; less than 1 or greater than CYGNUM_UITRON_TASKS returns E_ID
- wup_tsk() , sus_tsk() , rsm_tsk() , frsm_tsk() on the calling task returns E_OBJ
- dispatching is enabled in tslp_tsk() and slp_tsk() , or E_CTX
- tmout must be positive, otherwise E_PAR
- return value pointer in can_wup() is a valid pointer, or E_PAR

The following conditions are checked for, and can return error codes, regardless of the setting of CYGSEM_UITRON_BAD_PARAMS_RETURN_ERRORS:

- When create and delete functions cre_tsk() and del_tsk() are supported, all calls which use a valid task ID number check that the task exists; if not, E_NOEXS is returned
- sus_tsk() : the task must not be dormant, else E_OBJ
- frsm/rsm_tsk() : the task must be suspended, else E_OBJ
- tslp/slp_tsk() : return codes E_TMOUT, E_RLWAI and E_DLT are returned depending on the reason for terminating the sleep
- wup_tsk() and can_wup() : the task must not be dormant, or E_OBJ is returned

Synchronization and Communication Functions

These functions are fully supported in this release:

```
ER sig_sem(
    ID semid )
```

```
ER wai_sem(
    ID semid )
```

```
ER preq_sem(
    ID semid )
```

```
ER twai_sem(
    ID semid,    TMO tmout )
```

```
ER ref_sem(
    T_RSEM *pk_rsem ,    ID semid )
```

```
ER set_flg(
    ID flgid,    UINT setptn )
```

```
ER clr_flg(
    ID flgid,    UINT clrptn )
```

```
ER wai_flg(
    UINT *p_flgptn,    ID flgid ,
    UINT waiptn ,    UINT wfmode )
```

```
ER pol_flg(
    UINT *p_flgptn,    ID flgid ,
    UINT waiptn ,    UINT wfmode )
```

```
ER twai_flg(
    UINT *p_flgptn    ID flgid ,
    UINT waiptn ,    UINT wfmode,    TMO tmout )
```

```
ER ref_flg(
    T_RFLG *pk_rflg,    ID flgid )
```

```
ER snd_msg(
    ID mbxid,    T_MSG *pk_msg )
```

```
ER rcv_msg(
    T_MSG **ppk_msg,    ID mbxid )
```

```
ER prcv_msg(
    T_MSG **ppk_msg,    ID mbxid )
```

```
ER trcv_msg(
    T_MSG **ppk_msg,    ID mbxid ,    TMO tmout )
```

```
ER ref_mbx(
    T_RMBX *pk_rmbx,    ID mbxid )
```

The following functions are supported in this release (with some restrictions) if enabled with the appropriate configuration option for the object type (for example `CYGPKG_UITRON_SEMAS_CREATE_DELETE`):

```
ER cre_sem(
    ID semid,    T_CSEM *pk_csem )
```

```
ER del_sem(
    ID semid )
```

```
ER cre_flg(
    ID flgid,    T_CFLG *pk_cflg )
```

```
ER del_flg(
    ID flgid )
```

```
ER cre_mbx(
    ID mbxid,    T_CMBX *pk_cmbx )
```

```
ER del_mbx(
    ID mbxid )
```

In general the queueing order when waiting on a synchronization object depends on the underlying kernel configuration. The multi-level queue scheduler is required for strict µITRON conformance and it queues tasks in FIFO order, so requests to create an object with priority queueing of tasks (`pk_cxxx->xxxatr = TA_TPRI`) are rejected with E_RSATR. Additional undefined bits in the attributes fields must be zero.

In general, extended information (pk_cxxx->exinf) is ignored.

For semaphores, the initial semaphore count (pk_csem->isemcnt) is supported; the new semaphore's count is set. The maximum count is not supported, and is not in fact defined in type pk_csem.

For flags, multiple tasks are allowed to wait. Because single task waiting is a subset of this, the W bit (TA_WMUL) of the flag attributes is ignored; all other bits must be zero. The initial flag value is supported.

For mailboxes, the buffer count is defined statically by kernel configuration option `CYGNUM_KERNEL_SYNCH_MBOX_QUEUE_SIZE`; therefore the buffer count field is not supported and is not in fact defined in type pk_cmbx. Queueing of messages is FIFO ordered only, so TA_MPRI (in pk_cmbx->mbxatr) is not supported.

Error checking

The following conditions are only checked for, and only return errors if `CYGSEM_UITRON_BAD_PARAMS_RETURN_ERRORS` is enabled:

- invalid object id; less than 1 or greater than `CYGNUM_UITRON_TASKS/SEMAS/MBOXES` as appropriate returns E_ID
- dispatching is enabled in any call which can sleep, or E_CTX
- tmout must be positive, otherwise E_PAR
- pk_cxxx pointers in `cre_xxx()` must be valid pointers, or E_PAR
- return value pointer in `ref_xxx()` is valid pointer, or E_PAR
- flag wait pattern must be non-zero, and mode must be valid, or E_PAR
- return value pointer in flag wait calls is a valid pointer, or E_PAR

The following conditions are checked for, and can return error codes, regardless of the setting of `CYGSEM_UITRON_BAD_PARAMS_RETURN_ERRORS` :

- When create and delete functions `cre_xxx()` and `del_xxx()` are supported, all calls which use a valid object ID number check that the object exists. If not, E_NOEXS is returned.
- In create functions `cre_xxx()` , when supported, if the object already exists, then E_OBJ
- In any call which can sleep, such as `twai_sem()` : return codes E_TMOUT, E_RLWAI, E_DLT or of course E_OK are returned depending on the reason for terminating the sleep
- In polling functions such as `preq_sem()` return codes E_TMOUT or E_OK are returned depending on the state of the synchronization object
- In creation functions, the attributes must be compatible with the selected underlying kernel configuration: in `cre_sem()` `pk_csem->sematr` must be equal to `TA_TFIFO` else E_RSATR.
- In `cre_flg()` `pk_cflg->flgatr` must be either `TA_WMUL` or `TA_WSGL` else `E_RSATR`.
- In `cre_mbx()` `pk_cmbx->mbxatr` must be `TA_TFIFO + TA_MFIFO` else E_RSATR.

Extended Synchronization and Communication Functions

None of these functions are supported in this release.

Interrupt management functions

These functions are fully supported in this release:

```
void   ret_int( void )
```

```
ER loc_cpu( void )
```

```
ER unl_cpu( void )
```

```
ER dis_int(
    UINT eintno )
```

```
ER ena_int(
    UINT eintno )
```

```
void  ret_wup(
    ID tskid )
```

```
ER iwup_tsk(
    ID tskid )
```

```
ER isig_sem(
    ID semid )
```

```
ER iset_flg(
    ID flgid ,
    UID setptn )
```

```
ER isend_msg(
    ID mbxid ,
    T_MSG *pk_msg )
```

Note that `ret_int()` and the `ret_wup()` are implemented as macros, containing a “return” statement.

Also note that `ret_wup()` and the `ixxx_yyy()` style functions will only work when called from an ISR whose associated DSR is `cyg_uitron_dsr()`, as specified in include file `<cyg/compat/uitron/uit_ifnc.h>`, which defines the `ixxx_yyy()` style functions also.

If you are writing interrupt handlers more in the *eCos* style, with separate ISR and DSR routines both of your own devising, do not use these special functions from a DSR: use plain `xxx_yyy()` style functions (with no ’i, prefix) instead, and do not call any µITRON functions from the ISR at all.

The following functions are not supported in this release:

```
ER def_int(
    UINT dintno,
  T_DINT *pk_dint )
```

```
ER chg_iXX(
    UINT iXXXX )
```

```
ER ref_iXX(
    UINT * p_iXXXX )
```

These unsupported functions are all Level C (CPU dependent). Equivalent functionality is available via other *eCos*-specific APIs.

Error checking

The following conditions are only checked for, and only return errors if `CYGSEM_UITRON_BAD_PARAMS_RETURN_ERRORS` is enabled:

- `loc/unl_cpu()` : these must only be called in a µITRON task context, else E_CTX.
- `dis/ena_int()` : the interrupt number must be in range as specified by the platform HAL in qustion, else E_PAR.

Memory pool Management Functions

These functions are fully supported in this release:

```
ER get_blf(
    VP *p_blf,    ID mpfid )
```

```
ER pget_blf(
    VP *p_blf,    ID mpfid )
```

```
ER tget_blf(
    VP *p_blf,    ID mpfid,    TMO tmout )
```

```
ER rel_blf(
    ID mpfid,    VP blf )
```

```
ER ref_mpf(
    T_RMPF *pk_rmpf,    ID mpfid )
```

```
ER get_blk(
    VP *p_blk,    ID mplid,    INT blksz )
```

```
ER pget_blk(
    VP *p_blk,    ID mplid,    INT blksz )
```

```
ER tget_blk(
    VP *p_blk,    ID mplid,    INT blksz,    TMO tmout )
```

```
ER rel_blk(
    ID mplid,    VP blk )
```

```
ER ref_mpl(
    T_RMPL *pk_rmpl,    ID mplid )
```

Note that of the memory provided for a particular pool to manage in the static initialization of the memory pool objects, some memory will be used to manage the pool itself. Therefore the number of blocks * the blocksize will be less than the total memory size.

The following functions are supported in this release, when enabled with `CYGPKG_UITRON_MEMPOOLVAR_CREATE_DELETE` or `CYGPKG_UITRON_MEMPOOLFIXED_CREATE_DELETE` as appropriate, with some restrictions:

```
ER cre_mpl(
    ID mplid,    T_CMPL *pk_cmpl )
```

```
ER del_mpl(
    ID mplid )
```

```
ER cre_mpf(
    ID mpfid,    T_CMPF *pk_cmpf )
```

```
ER del_mpf(
    ID mpfid )
```

Because of the static initialization facilities provided for system objects, a memory pool is allocated a region of memory to manage statically in the configuration. So while memory pools can be created and deleted, the same area of memory is used for that memory pool (memory pool ID number) each time. The requested variable pool size (pk_cmpl->mplsz) or the number of fixed-size blocks (pk_cmpf->mpfcnt) times the block size (pk_cmpf->blfsz) are checked for fitting within the statically allocated memory area, so if a create call succeeds, the resulting pool will be at least as large as that requested. For this reason `del_mpl()` and `del_mpf()` do not in any sense free the memory that was managed by the deleted pool for use by other pools; it may only be managed by a pool of the same object id.

For both fixed and variable memory pools, the queueing order when waiting on a synchronization object depends on the underlying kernel configuration. The multi-level queue scheduler is required for strict µITRON conformance and it queues tasks in FIFO order, so requests to create an object with priority queueing of tasks (pk_cxxx->xxxatr = TA_TPRI) are rejected with E_RSATR. Additional undefined bits in the attributes fields must be zero.

In general, extended information (pk_cxxx->exinf) is ignored.

Error checking

The following conditions are only checked for, and only return errors if `CYGSEM_UITRON_BAD_PARAMS_RETURN_ERRORS` is enabled:

- invalid object id; less than 1 or greater than `CYGNUM_UITRON_MEMPOOLVAR/MEMPOOLFIXED` as appropriate returns E_ID
- dispatching is enabled in any call which can sleep, or E_CTX
- tmout must be positive, otherwise E_PAR
- pk_cxxx pointers in `cre_xxx()` must be valid pointers, or E_PAR
- return value pointer in `ref_xxx()` is a valid pointer, or E_PAR
- return value pointers in get block routines is a valid pointer, or E_PAR
- blocksize request in get variable block routines is greater than zero, or E_PAR

The following conditions are checked for, and can return error codes, regardless of the setting of `CYGSEM_UITRON_BAD_PARAMS_RETURN_ERRORS`:

- When create and delete functions `cre_xxx()` and `del_xxx()` are supported, all calls which use a valid object ID number check that the object exists. If not, E_NOEXS is returned.
- When create functions `cre_xxx()` are supported, if the object already exists, then E_OBJ
- In any call which can sleep, such as `get_blk()` : return codes E_TMOUT, E_RLWAI, E_DLT or of course E_OK are returned depending on the reason for terminating the sleep
- In polling functions such as `pget_blk()` return codes E_TMOUT or E_OK are returned depending on the state of the synchronization object
- In creation functions, the attributes must be compatible with the selected underlying kernel configuration: in `cre_mpl()` `pk_cmpl->mplatr` must be equal to `TA_TFIFO` else E_RSATR.
- In `cre_mpf()` `pk_cmpf->mpfatr` must be equal to `TA_TFIFO` else E_RSATR.
- In creation functions, the requested size of the memory pool must not be larger than that statically configured for the pool else E_RSATR; see the configuration option “Option: Static initializers”. In `cre_mpl()` `pk_cmpl->mplsz` is the field of interest
- In `cre_mpf()` the product of `pk_cmpf->blfsz` and `pk_cmpf->mpfcnt` must be smaller than the memory statically configured for the pool else E_RSATR
- In functions which return memory to the pool `rel_blk()` and `rel_blf()` , if the free fails, for example because the memory did not come from that pool originally, then E_PAR is returned

Time Management Functions

These functions are fully supported in this release:

```
ER set_tim(
    SYSTIME *pk_tim )
```

Caution

Setting the time may cause erroneous operation of the kernel, for example a task performing a wait with a time-out may never awaken.

```
ER get_tim(
    SYSTIME *pk_tim )
```

```
ER dly_tsk(
    DLYTIME dlytim )
```

```
ER def_cyc(
    HNO cycno,    T_DCYC *pk_dcyc )
```

```
ER act_cyc(
    HNO cycno,    UINT cycact )
```

```
ER ref_cyc(
    T_RCYC *pk_rcyc,    HNO cycno )
```

```
ER def_alm(
    HNO almno,    T_DALM *pk_dalm )
```

```
ER ref_alm(
    T_RALM *pk_ralm,    HNO almno )
```

```
void ret_tmr( void )
```

Error checking

The following conditions are only checked for, and only return errors if `CYGSEM_UITRON_BAD_PARAMS_RETURN_ERRORS` is enabled:

- invalid handler number; less than 1 or greater than `CYGNUM_UITRON_CYCLICS/ALARMS` as appropriate, or E_PAR
- dispatching is enabled in `dly_tsk()` , or E_CTX
- dlytim must be positive or zero, otherwise E_PAR
- return value pointer in `ref_xxx()` is a valid pointer, or E_PAR
- params within pk_dalm and pk_dcyc must be valid, or E_PAR
- cycact in `act_cyc()` must be valid, or E_PAR
- handler must be defined in `ref_xxx()` and `act_cyc()` , or E_NOEXS
- parameter pointer must be a good pointer in `get_tim()` and `set_tim()` , otherwise E_PAR is returned

The following conditions are checked for, and can return error codes, regardless of the setting of `CYGSEM_UITRON_BAD_PARAMS_RETURN_ERRORS` :

- `dly_tsk()` : return code E_RLWAI is returned depending on the reason for terminating the sleep

System Management Functions

These functions are fully supported in this release:

```
ER get_ver(
    T_VER *pk_ver )
```

```
ER ref_sys(
    T_RSYS *pk_rsys )
```

```
ER ref_cfg(
    T_RCFG *pk_rcfg )
```

Note that the information returned by each of these calls may be configured to match the user's own versioning system, and the values supplied by the default configuration may be inappropriate.

These functions are not supported in this release:

```
ER def_svc(
    FN s_fncd,
    T_DSVC *pk_dsvc )
```

```
ER def_exc(
    UINT exckind,
    T_DEXC *pk_dexc )
```

Error checking

The following conditions are only checked for, and only return errors if `CYGSEM_UITRON_BAD_PARAMS_RETURN_ERRORS` is enabled:

- return value pointer in all calls is a valid pointer, or E_PAR

Network Support Functions

None of these functions are supported in this release.

µITRON Configuration FAQ

Q: How are µITRON objects created?

For each type of uITRON object (tasks, semaphores, flags, mboxes, mpf, mpl) these two quantities are controlled by configuration:

- The *maximum* number of this type of object.
- The number of these objects which exist *initially*.

This is assuming that for the relevant object type, *create* and *delete* operations are enabled; enabled is the default. For example, the option `CYGPKG_UITRON_MBOXES_CREATE_DELETE` controls whether the functions `cre_mbx()` and `del_mbx()` exist in the API. If not, then the maximum number of mboxes is the same as the initial number of mboxes, and so on for all µITRON object types.

Mboxes have no initialization, so there are only a few, simple configuration options:

- `CYGNUM_UITRON_MBOXES` is the total number of mboxes that you can have in the system. By default this is 4, so you can use mboxes 1,2,3 and 4. You cannot create mboxes outside this range; trying to `cre_mbx(5,...)` will return an error.
- `CYGNUM_UITRON_MBOXES_INITIALLY` is the number of mboxes created automatically for you, during startup. By default this is 4, so all 4 mboxes exist already, and an attempt to create one of these eg. `cre_mbx(3,...)` will return an error because the mbox in quesion already exists. You can delete a pre-existing mbox, and then re-create it.

If you change `CYGNUM_UITRON_MBOXES_INITIALLY`, for example to 0, no mboxes are created automatically for you during startup. Any attempt to use an mbox without creating it will return E_NOEXS because the mbox does not exist. You can create an mbox, say `cre_mbx(3,...)` and then use it, say `snd_msg(3,&foo)`, and all will be well.

Q: How are µITRON objects initialized?

Some object types have optional initialization. Semaphores are an example. You could have `CYGNUM_UITRON_SEMAS`=10 and `CYGNUM_UITRON_SEMAS_INITIALLY`=5 which means you can use semaphores 1-5 straight off, but you must create semaphores 6-10 before you can use them. If you decide not to initialize semaphores, semaphores 1-5 will have an initial count of zero. If you decide to initialize them, you must supply a dummy initializer for semaphores 6-10 also. For example, in terms of the configuration output in `pkgconf/uitron.h`:

```
   #define CYGDAT_UITRON_SEMA_INITIALIZERS \
        CYG_UIT_SEMA(  1 ),     \
        CYG_UIT_SEMA(  0 ),     \
        CYG_UIT_SEMA(  0 ),     \
        CYG_UIT_SEMA( 99 ),     \
        CYG_UIT_SEMA(  1 ),     \
        CYG_UIT_SEMA_NOEXS,     \
        CYG_UIT_SEMA_NOEXS,     \
        CYG_UIT_SEMA_NOEXS,     \
        CYG_UIT_SEMA_NOEXS,     \
        CYG_UIT_SEMA_NOEXS
```

Semaphore 1 will have initial count 1, semaphores 2 and 3 will be zero, number 4 will be 99 initially, 5 will be one and numbers 6 though 10 do not exist initially.

Aside: this is how the definition of the symbol would appear in the configuration header file `pkgconf/uitron.h` — unfortunately editing such a long, multi-line definition is somewhat cumbersome in the GUI config tool in current releases. The macros `CYG_UIT_SEMA()` — to create a semaphore initializer — and `CYG_UIT_SEMA_NOEXS` — to invoke a dummy initializer — are provided in in the environment to help with this. Similar macros are provided for other object types. The resulting #define symbol is used in the context of a C++ array initializer, such as:

```
Cyg_Counting_Semaphore2 cyg_uitron_SEMAS[ CYGNUM_UITRON_SEMAS ] = {
  CYGDAT_UITRON_SEMA_INITIALIZERS
};
```

which is eventually macro-processed to give

```
Cyg_Counting_Semaphore2 cyg_uitron_SEMAS[ 10 ] = {
  Cyg_Counting_Semaphore2( ( 1 ) ),
  Cyg_Counting_Semaphore2( ( 0 ) ),
  Cyg_Counting_Semaphore2( ( 0 ) ),
  Cyg_Counting_Semaphore2( ( 99 ) ),
  Cyg_Counting_Semaphore2( ( 1 ) ),
  Cyg_Counting_Semaphore2(0),
  Cyg_Counting_Semaphore2(0),
  Cyg_Counting_Semaphore2(0),
  Cyg_Counting_Semaphore2(0),
  Cyg_Counting_Semaphore2(0),
};
```

so you can see how it is necessary to include the dummy entries in that definition, otherwise the resulting code will not compile correctly.

If you choose `CYGNUM_UITRON_SEMAS_INITIALLY`=0 it is meaningless to initialize them, for they must be created and so initialized then, before use.

Q: What about µITRON tasks?

Some object types require initialization. Tasks are an example of this. You must provide a task with a priority, a function to enter when the task starts, a name (for debugging purposes), and some memory to use for the stack. For example (again in terms of the resulting definitions in `pkgconf/uitron.h`):

```
#define CYGNUM_UITRON_TASKS 4           // valid task ids are 1,2,3,4
#define CYGNUM_UITRON_TASKS_INITIALLY 4 // they all exist at start

#define CYGDAT_UITRON_TASK_EXTERNS            \
extern "C" void startup( unsigned int );      \
extern "C" void worktask( unsigned int );     \
extern "C" void lowtask( unsigned int );        \
static char stack1[ CYGNUM_UITRON_STACK_SIZE ], \
            stack2[ CYGNUM_UITRON_STACK_SIZE ], \
            stack3[ CYGNUM_UITRON_STACK_SIZE ], \
            stack4[ CYGNUM_UITRON_STACK_SIZE ];

#define CYGDAT_UITRON_TASK_INITIALIZERS \
 CYG_UIT_TASK("main task", 8, startup,  &stack1, sizeof( stack1 )), \
 CYG_UIT_TASK("worker 2" , 9, worktask, &stack2, sizeof( stack2 )), \
 CYG_UIT_TASK("worker 3" , 9, worktask, &stack3, sizeof( stack3 )), \
 CYG_UIT_TASK("low task" ,20, lowtask,  &stack4, sizeof( stack4 )), \
```

So this example has all four tasks statically configured to exist, ready to run, from the start of time. The “main task” runs a routine called `startup()` at priority 8. Two “worker” tasks run both a priority 9, and a “low priority” task runs at priority 20 to do useful non-urgent background work.

```
Task ID | Exists at | Function | Priority | Stack   | Stack
 number |  startup  |  entry   |          | address | size
--------+-----------+----------+----------+---------+----------
   1    |    Yes    |  startup |    8     | &stack1 | CYGNUM…
   2    |    Yes    | worktask |    9     | &stack2 | CYGNUM…
   3    |    Yes    | worktask |    9     | &stack3 | CYGNUM…
   4    |    Yes    |  lowtask |   20     | &stack4 | CYGNUM…
--------+-----------+----------+----------+---------+----------
```

Q: How can I create µITRON tasks in the program?

You must provide free slots in the task table in which to create new tasks, by configuring the number of tasks existing initially to be smaller than the total. For a task ID which does not initially exist, it will be told what routine to call, and what priority it is, when the task is created. But you must still set aside memory for the task to use for its stack, and give it a name during initialization. For example:

```
#define CYGNUM_UITRON_TASKS 4           // valid task ids are 1-4
#define CYGNUM_UITRON_TASKS_INITIALLY 1 // only task #1 exists

#define CYGDAT_UITRON_TASK_EXTERNS \
extern "C" void startup( unsigned int ); \
static char stack1[ CYGNUM_UITRON_STACK_SIZE ], \
            stack2[ CYGNUM_UITRON_STACK_SIZE ], \
            stack3[ CYGNUM_UITRON_STACK_SIZE ], \
            stack4[ CYGNUM_UITRON_STACK_SIZE ];

#define CYGDAT_UITRON_TASK_INITIALIZERS \
   CYG_UIT_TASK( "main", 8, startup, &stack1, sizeof( stack1 ) ), \
   CYG_UIT_TASK_NOEXS( "slave",      &stack2, sizeof( stack2 ) ), \
   CYG_UIT_TASK_NOEXS( "slave2",     &stack3, sizeof( stack3 ) ), \
   CYG_UIT_TASK_NOEXS( "slave3",     &stack4, sizeof( stack4 ) ), \
```

So tasks numbered 2,3 and 4 have been given their stacks during startup, though they do not yet exist in terms of `cre_tsk()` and `del_tsk()` so you can create tasks 2–4 at runtime.

```
Task ID | Exists at | Function | Priority | Stack   | Stack
 number |  startup  |  entry   |          | address | size
--------+-----------+----------+----------+---------+----------
   1    |    Yes    |  startup |    8     | &stack1 | CYGNUM…
   2    |    No     |   N/A    |   N/A    | &stack2 | CYGNUM…
   3    |    No     |   N/A    |   N/A    | &stack3 | CYGNUM…
   4    |    No     |   N/A    |   N/A    | &stack4 | CYGNUM…
--------+-----------+----------+----------+---------+----------
```

(you must have at least one task at startup in order that the system can actually run; this is not so for other uITRON object types)

Q: Can I have different stack sizes for µITRON tasks?

Simply set aside different amounts of memory for each task to use for its stack. Going back to a typical default setting for the µITRON tasks, the definitions in `pkgconf/uitron.h` might look like this:

```
#define CYGDAT_UITRON_TASK_EXTERNS \
extern "C" void task1( unsigned int ); \
extern "C" void task2( unsigned int ); \
extern "C" void task3( unsigned int ); \
extern "C" void task4( unsigned int ); \
static char stack1[ CYGNUM_UITRON_STACK_SIZE ], \
            stack2[ CYGNUM_UITRON_STACK_SIZE ], \
            stack3[ CYGNUM_UITRON_STACK_SIZE ], \
            stack4[ CYGNUM_UITRON_STACK_SIZE ];

#define CYGDAT_UITRON_TASK_INITIALIZERS \
  CYG_UIT_TASK( "t1", 1, task1, &stack1, CYGNUM_UITRON_STACK_SIZE ), \
  CYG_UIT_TASK( "t2", 2, task2, &stack2, CYGNUM_UITRON_STACK_SIZE ), \
  CYG_UIT_TASK( "t3", 3, task3, &stack3, CYGNUM_UITRON_STACK_SIZE ), \
  CYG_UIT_TASK( "t4", 4, task4, &stack4, CYGNUM_UITRON_STACK_SIZE )
```

Note that `CYGNUM_UITRON_STACK_SIZE` is used to control the size of the stack objects themselves, and to tell the system what size stack is being provided.

Suppose instead stack sizes of 2000, 1000, 800 and 800 were required: this could be achieved by using the GUI config tool to edit these options, or editting the `.ecc` file to get these results in `pkgconf/uitron.h`:

```
#define CYGDAT_UITRON_TASK_EXTERNS \
extern "C" void task1( unsigned int ); \
extern "C" void task2( unsigned int ); \
extern "C" void task3( unsigned int ); \
extern "C" void task4( unsigned int ); \
static char stack1[ 2000 ], \
            stack2[ 1000 ], \
            stack3[  800 ], \
            stack4[  800 ];

#define CYGDAT_UITRON_TASK_INITIALIZERS \
      CYG_UIT_TASK( "t1", 1, task1, &stack1, sizeof( stack1 ) ), \
      CYG_UIT_TASK( "t2", 2, task2, &stack2, sizeof( stack2 ) ), \
      CYG_UIT_TASK( "t3", 3, task3, &stack3, sizeof( stack3 ) ), \
      CYG_UIT_TASK( "t4", 4, task4, &stack4, sizeof( stack4 ) )
```

Note that the sizeof() operator has been used to tell the system what size stacks are provided, rather than quoting a number (which is difficult for maintenance) or the symbol `CYGNUM_UITRON_STACK_SIZE` (which is wrong).

We recommend using (if available in your release) the stacksize symbols provided in the architectural HAL for your target, called `CYGNUM_HAL_STACK_SIZE_TYPICAL` and `CYGNUM_HAL_STACK_SIZE_MINIMUM`. So a better (more portable) version of the above might be:

```
#define CYGDAT_UITRON_TASK_EXTERNS \
extern "C" void task1( unsigned int ); \
extern "C" void task2( unsigned int ); \
extern "C" void task3( unsigned int ); \
extern "C" void task4( unsigned int ); \
static char stack1[ CYGNUM_HAL_STACK_SIZE_TYPICAL + 1200 ], \
            stack2[ CYGNUM_HAL_STACK_SIZE_TYPICAL +  200 ], \
            stack3[ CYGNUM_HAL_STACK_SIZE_TYPICAL        ], \
            stack4[ CYGNUM_HAL_STACK_SIZE_TYPICAL        ];

#define CYGDAT_UITRON_TASK_INITIALIZERS \
      CYG_UIT_TASK( "t1", 1, task1, &stack1, sizeof( stack1 ) ), \
      CYG_UIT_TASK( "t2", 2, task2, &stack2, sizeof( stack2 ) ), \
      CYG_UIT_TASK( "t3", 3, task3, &stack3, sizeof( stack3 ) ), \
      CYG_UIT_TASK( "t4", 4, task4, &stack4, sizeof( stack4 ) )
```

Part XLI. TCP/IP Stack Support for eCos

The Common Networking for eCos package provides support for a complete TCP/IP networking stack. The design allows for the actual stack to be modular allowing for different implementations to be provided. Currently only one version based on FreeBSD is available, with the earlier OpenBSD implementation (circa 2000) deprecated and withdrawn from current eCosPro releases.

For resource-constrained systems, the lightweight networking stack lwip (see Part XLIV, “lwIP - the lightweight IP stack for eCosPro”) has been ported to eCos and supports the IP, TCP, UDP, ICMP, IGMP, ARP, DHCP, AutoIP, DNS, SNMP, SLIP and PPP protocols. lwip was designed from the outset to have a low memory footprint and gains many of its lightweight properties from being highly configurable, making it an excellent eCos add-on package.

Table of Contents

Chapter 144. Ethernet Driver Design

Currently, the networking stack only supports ethernet based networking.

The network drivers use a two-layer design. One layer is hardware independent and contains all the stack specific code. The other layer is platform dependent and communicates with the hardware independent layer via a very simple API. In this way, hardware device drivers can actually be used with other stacks, if the same API can be provided by that stack. We designed the drivers this way to encourage the development of other stacks in eCos while allowing re-use of the actual hardware specific code.

More comprehensive documentation of the ethernet device driver and the associated API can be found in the eCos generic ethernet device driver documentation. The driver and API is the same as the minimal debug stack used by the RedBoot application. See the RedBoot documentation for further information.

Chapter 145. Sample Code

Many examples using the networking support are provided. These are arranged as eCos test programs, primarily for use in verifying the package, but they can also serve as useful frameworks for program design. We have taken a KISS approach to building programs which use the network. A single include file <`network.h`> is all that is required to access the stack. A complete, annotated test program can be found at `net/common/`*`VERSION`*`/tests/ftp_test.c`, with its associated files.

Chapter 146. Configuring IP Addresses

Each interface (“eth0” and “eth1”) has independent configuration of its setup. Each can be set up manually (in which case you must write code to do this), or by using BOOTP/DHCP, or explicitly, with configured values. If additional interfaces are added, these must be configured manually.

The configurable values are:

- IP address
- netmask
- broadcast address
- gateway/router
- server address.

Server address is the DHCP server if applicable, but in addition, many test cases use it as “the machine to talk to” in whatever manner the test exercises the protocol stack.

The initialization is invoked by calling the C routine:

```
void init_all_network_interfaces(void);
```

Additionally, if the system is configured to support IPv6 then each interface may have an address assigned which is a composite of a 64 bit prefix and the 32 bit IPv4 address for that interface. The prefix is controlled by the CDL setting CYGHWR_NET_DRIVER_ETH0_IPV6_PREFIX for “eth0”, etc. This is a CDL booldata type, allowing this address to be suppressed if not desired.

Alternatively, the system can configure its IPv6 address using router solicitation. When the CDL option CYGOPT_NET_IPV6_ROUTING_THREAD is enabled, `init_all_network_interface` will start a thread which sends out router solicit messages, process router advertisements and thus configure an IPv6 address to the interface.

Refer to the test cases, …`/packages/net/common/`*`VERSION`*`/tests/ftp_test.c` for example usage, and the source files in …`/packages/net/common/`*`VERSION`*`/src/bootp_support.c` and `network_support.c` to see what that call does.

This assumes that the MAC address (also known as ESA or Ethernet Station Address) is already defined in the serial EEPROM or however the particular target implements this; support for setting the MAC address is hardware dependent.

DHCP support is active by default, and there are configuration options to control it. Firstly, in the top level of the “Networking” configuration tree, “Use full DHCP instead of BOOTP” enables DHCP, and it contains an option to have the system provide a thread to renew DHCP leases and manage lease expiry. Secondly, the individual interfaces “eth0” and “eth1” each have new options within the “Use BOOTP/DHCP to initialize ’*ethX*,” to select whether to use DHCP rather than BOOTP.

You should not configure the network stack to use BOOTP/DHCP if you are using RedBoot, have configured it also to use BOOTP/DHCP, and are connected via GDB to it over the network. Otherwise the TCP/IP stacks in both RedBoot and the eCos application are likely to be given the same IP address, which will cause problems.

Note that you are completely at liberty to ignore this startup code and its configuration in building your application. `init_all_network_interfaces()` is provided for three main purposes:

- For use by eCos's own test programs.
- As an easy “get you going” utility for newcomers to eCos.

- As readable example code from which further development might start.

If your application has different requirements for bringing up available network interfaces, setting up routes, determining IP addresses and the like from the defaults that the example code provides, you can write your own initialization code to use whatever sequence of `ioctl()` function calls carries out the desired setup. Analogously, in larger systems, a sequence of “ifconfig” invocations is used; these mostly map to `ioctl()` calls to manipulate the state of the interface in question.

By default the supplied `init_all_network_interfaces()` code can configure up to two ethernet interfaces. An alternative implementation is available that can configure more interfaces. This code is used by enabling the `CYGPKG_NET_DRIVER_INIT_NEW` option; this is enabled by default if more than two interfaces are configured but may also be enabled by the user. At preseent support is limited to four interfaces, but can be extended with minimal changes to the CDL and code to any number of interfaces. The alternative code is functionally equivalent to the default, but uses more memory and is therefore not used by default.

Chapter 147. Tests and Demonstrations

Loopback tests

By default, only tests which can execute on any target will be built. These therefore do not actually use external network interfaces (though they may configure and initialize them) but are limited to testing via the loopback interface.

```
ping_lo_test     - ping test of the loopback address
tcp_lo_select    - simple test of select with TCP via loopback
tcp_lo_test      - trivial TCP test via loopback
udp_lo_test      - trivial UDP test via loopback
multi_lo_select  - test of multiple select() calls simultaneously
```

Building the Network Tests

To build further network tests, ensure that the configuration option CYGPKG_NET_BUILD_TESTS is set in your build and then make the tests in the usual way. Alternatively (with that option set) use the following command after building the eCos library, if you wish to build *only* the network tests:

```
make -C net/common/VERSION/ tests
```

This should give test executables in `install/tests/net/common/VERSION/tests` including the following:

```
socket_test      - trivial test of socket creation API
mbuf_test        - trivial test of mbuf allocation API
ftp_test         - simple FTP test, connects to “server”
ping_test        - pings “server” and non-existent host to test timeout
dhcp_test        - ping test, but also relinquishes and
                   reacquires DHCP leases periodically
flood            - a flood ping test; use with care
tcp_echo         - data forwarding program for performance test
nc_test_master   - network characterization master
nc_test_slave    - network characterization slave
server_test      - a very simple server example
tftp_client_test - performs a tftp get and put from/to “server”
tftp_server_test - runs a tftp server for a short while
set_mac_address  - set MAC address(es) of interfaces in NVRAM
bridge           - contributed network bridge code
nc6_test_master  - IPv4/IPv6 network characterization master
nc6_test_slave   - IPv4/IPv6 network characterization slave
ga_server_test   - a very simple IPv4/IPv6 server example
```

Standalone Tests

```
socket_test      - trivial test of socket creation API
mbuf_test        - trivial test of mbuf allocation API
```

These two do not communicate over the net; they just perform simple API tests then exit.

```
ftp_test         - simple FTP test, connects to “server”
```

This test initializes the interface(s) then connects to the FTP server on the “server” machine for for each active interface in turn, confirms that the connection was successful, disconnects and exits. This tests interworking with the server.

```
ping_test        - pings “server” and non-existent host to test timeout
```

This test initializes the interface(s) then pings the server machine in the standard way, then pings address “32 up” from the server in the expectation that there is no machine there. This confirms that the successful ping is not a false positive, and tests the receive timeout. If there is such a machine, of course the 2nd set of pings succeeds, confirming that we can talk to a machine not previously mentioned by configuration or by bootp. It then does the same thing on the other interface, eth1.

If IPv6 is enabled, the program will also ping to the address it last received a router advertisement from. Also a ping will be made to that address plus 32, in a similar way the the IPv4 case.

```
dhcp_test        - ping test, but also manipulates DHCP leases
```

This test is very similar to the ping test, but in addition, provided the network package is not configured to do this automatically, it manually relinquishes and reclaims DHCP leases for all available interfaces. This tests the external API to DHCP. See section below describing this.

```
flood            - a flood ping test; use with care
```

This test performs pings on all interfaces as quickly as possible, and only prints status information periodically. Flood pinging is bad for network performance; so do not use this test on general purpose networks unless protected by a switch.

Performance Test

```
tcp_echo         - data forwarding program for performance test
```

tcp_echo is one part of the standard performance test we use. The other parts are host programs `tcp_source` and `tcp_sink`. To make these (under your *HOST* system) cd to the tests source directory in the eCos repository and type “`make -f make.host`” - this should build `tcp_source` and `tcp_sink`.

The host program “`tcp_source`” sends data to the target. On the target, “`tcp_echo`” sends it onwards to “`tcp_sink`” running on your host. So the target must receive and send on all the data that `tcp_source` sends it; the time taken for this is measured and the data rate is calculated.

To invoke the test, first start `tcp_echo` on the target board and wait for it to become quiescent - it will report work to calibrate a CPU load which can be used to simulate real operating conditions for the stack.

Then on your host machine, in one terminal window, invoke `tcp_sink` giving it the IP address (or hostname) of one interface of the target board. For example “`tcp_sink 10.130.39.66`”. `tcp_echo` on the target will print something like “`SINK connection from 10.130.39.13:1143`” when `tcp_sink` is correctly invoked.

Next, in another host terminal window, invoke `tcp_source`, giving it the IP address (or hostname) of an interface of the target board, and optionally a background load to apply to the target while the test runs. For example, “`tcp_source 194.130.39.66`” to run the test with no additional target CPU load, or “`tcp_source 194.130.39.66 85`” to load it up to 85% used. The target load must be a multiple of 5. `tcp_echo` on the target will print something like “`SOURCE connection from 194.130.39.13:1144`” when tcp_source is correctly invoked.

You can connect tcp_sink to one target interface and tcp_source to another, or both to the same interface. Similarly, you can run `tcp_sink` and `tcp_source` on the same host machine or different ones. TCP/IP and ARP look after them finding one another, as intended.

```
nc_test_master   - network characterization master
nc_test_slave    - network characterization slave
```

These tests talk to each other to measure network performance. They can each run on either a test target or a host computer given some customization to your local environment. As provided, `nc_test_slave` must run on the test target, and `nc_test_master` must be run on a host computer, and be given the test target's IP address or hostname.

The tests print network performance for various packet sizes over UDP and TCP, versus various additional CPU loads on the target.

The programs below are additional forms which support both IPv4 and IPv6 addressing:

```
        nc6_test_slave
        nc6_test_master
```

Interactive Tests

```
server_test - a very simple server example
```

This test simply awaits a connection on port 7734 and after accepting a connection, gets a packet (with a timeout of a few seconds) and prints it.

The connection is then closed. We then loop to await the next connection, and so on. To use it, telnet to the target on port 7734 then type something (quickly!)

```
% telnet 172.16.19.171 7734
Hello target board
```

and the test program will print something like:

```
connection from 172.16.19.13:3369
buf = "Hello target board"
```

```
ga_server_test - another very simple server example
```

This is a variation on the *ga_server_test* test with the difference being that it uses the `getaddrinfo` function to set up its addresses. On a system with IPv6 enabled, it will listen on port 7734 for a TCP connection via either IPv4 or IPv6.

```
tftp_client_test - performs a tftp get and put from/to “server”
```

This is only partially interactive. You need to set things up on the “server” in order for this to work, and you will need to look at the server afterwards to confirm that all was well.

For each interface in turn, this test attempts to read by tftp from the server, a file called `tftp_get` and prints the status and contents it read (if any). It then writes the same data to a file called `tftp_put` on the same server.

In order for this to succeed, both files must already exist. The TFTP protocol does not require that a WRQ request _create_ a file, just that it can write it. The TFTP server on Linux certainly will only allow writes to an existing file, given the appropriate permission. Thus, you need to have these files in place, with proper permission, before running the test.

The conventional place for the tftp server to operate in LINUX is /tftpboot/; you will likely need root privileges to create files there. The data contents of `tftp_get` can be anything you like, but anything very large will waste lots of time printing it on the test,s stdout, and anything above 32kB will cause a buffer overflow and unpredictable failure.

Creating an empty tftp_put file (eg. by copying /dev/null to it) is neatest. So before the test you should have something like:

```
-rw-rw-rw- 1 root        1076 May  1 11:39 tftp_get
-rw-rw-rw- 1 root           0 May  1 15:52 tftp_put
```

note that both files have public permissions wide open. After running the test, `tftp_put` should be a copy of `tftp_get`.

```
-rw-rw-rw-  1 root       1076 May  1 11:39 tftp_get
-rw-rw-rw-  1 root       1076 May  1 15:52 tftp_put
```

If the configuration contains IPv6 support, the test program will also use IPv6. It will attempt to put/get the files listed above from the address it last received a routers solicit from.

```
tftp_server_test - runs a tftp server for a short while
```

This test is truly interactive, in that you can use a standard tftp application to get and put files from the server, during the 5 minutes that it runs. The dummy filesystem which underlies the server initially contains one file, called “uu” which contains part of a familiar text and some padding. It also accommodates creation of 3 further files of up to 1Mb in size and names of up to 256 bytes. Exceeding these limits will cause a buffer overflow and unpredictable failure.

The dummy filesystem is an implementation of the generic API which allows a true filesystem to be attached to the tftp server in the network stack.

We have been testing the tftp server by running the test on the target board, then using two different host computers connecting to the different target interfaces, putting a file from each, getting the “uu” file, and getting the file from the other computer. This verifies that data is preserved during the transfer as well as interworking with standard tftp applications.

Maintenance Tools

```
set_mac_address - set MAC address(es) of interfaces in NVRAM
```

This program makes an example `ioctl()` call SIOCSIFHWADDR “Socket IO Set InterFace HardWare ADDRess” to set the MAC address on targets where this is supported and enabled in the configuration. You must edit the source to choose a MAC address and further edit it to allow this very dangerous operation. Not all ethernet drivers support this operation, because most ethernet hardware does not support it — or it comes pre-set from the factory. *Do not use this program.*

Chapter 148. Support Features

TFTP

The TFTP client and server are described in `tftp_support.h`;

The TFTP client has and new and an older, deprecated, API. The new API works for both IPv4 and IPv6 where as the deprecated API is IPv4 only.

The new API is as follows:

```
int tftp_client_get(const char * const  filename,
                    const char * const  server,
                    const int           port,
                    char                *buf,
                    int                 len,
                    const int           mode,
                    int * const         err);

int tftp_client_put(const char * const  filename,
                    const char * const  server,
                    const int           port,
                    const char          *buf,
                    int                 len,
                    const int           mode,
                    int *const          err);
```

Currently `server` can only be a numeric IPv4 or IPv6 address. The resolver is currently not used, but it is planned to add this feature (patches welcome). If `port` is zero the client connects to the default TFTP port on the server. Otherwise the specified port is used.

The deprecated API is:

```
int tftp_get(const char * const               filename,
             const struct sockaddr_in * const server,
             char                             *buf,
             int                              len,
             const int                        mode,
             int * const                      error);

int tftp_put(const char * const               filename,
             const struct sockaddr_in * const server,
             const char                       *buffer,
             int                              len,
             const int                        mode,
             int * const                      err);
```

The `server` should contain the address of the server to contact. If the `sin_port` member of the structure is zero the default TFTP port is used. Otherwise the specified port is used.

Both API's report errors in the same way. The functions return a value of -1 and `*err` will be set to one of the following values:

```
#define TFTP_ENOTFOUND   1   /* file not found                    */
#define TFTP_EACCESS     2   /* access violation                  */
#define TFTP_ENOSPACE    3   /* disk full or allocation exceeded  */
#define TFTP_EBADOP      4   /* illegal TFTP operation            */
#define TFTP_EBADID      5   /* unknown transfer ID               */
#define TFTP_EEXISTS     6   /* file already exists               */
#define TFTP_ENOUSER     7   /* no such user                      */
#define TFTP_TIMEOUT     8   /* operation timed out               */
#define TFTP_NETERR      9   /* some sort of network error        */
#define TFTP_INVALID    10   /* invalid parameter                 */
```

```
#define TFTP_PROTOCOL   11   /* protocol violation                */
#define TFTP_TOOLARGE   12   /* file is larger than buffer        */
```

If there are no errors the return value is the number of bytes transfered.

The server is more complex. It requires a filesystem implementation to be supplied by the user, and attached to the tftp server by means of a vector of function pointers:

```
struct tftpd_fileops {
             int (*open)(const char *, int);
             int (*close)(int);
             int (*write)(int, const void *, int);
             int (*read)(int, void *, int);
};
```

These functions have the obvious semantics. The structure describing the filesystem is an argument to the `tftpd_start`:

```
int tftp_start(int port,
               struct tftpd_fileops *ops);
```

The first argument is the port to use for the server. If this port number is zero, the default TFTP port number will be used. The return value from `tftpd_start` is a handle which can be passed to `tftpd_stop`. This will kill the tftpd thread. Note that this is not a clean shutdown. The thread will simply be killed. `tftpd_stop` will attempt to close the sockets the thread was listening on and free some of its allocated memory. But if the thread was actively transferreing data at the time `tftpd_stop` is called, it is quite likely some memory and a socket will be leaked. Use this function with caution (or implement a clean shutdown and please contribute the code back :-).

There are two CDL configuration options that control how many servers on how many different ports tftp can be started. CYGSEM_NET_TFTPD_MULTITHREADED, when enabled, allows multiple tftpd threads to operate on the same port number. With only one thread, while the thread is active transferring data, new requests for transfers will not be served until the active transfer is complete. When multiple threads are started on the same port, multiple transfers can take place simultaneous, up to the number of threads started. However a semaphore is required to synchronise the threads. This semaphore is required per port. The CDL option CYGNUM_NET_TFTPD_MULTITHREADED_PORTS controls how many different port numbers multithreaded servers can service.

If CYGSEM_NET_TFTPD_MULTITHREADED is not enabled, only one thread may be run per port number. But this removes the need for a semaphore and so CYGNUM_NET_TFTPD_MULTITHREADED_PORTS is not required and unlimited number of ports can be used.

It should be noted that the TFTPD does not perform any form of file locking. When multiple servers are active, it is assumed the underlying filesystem will refuse to open the same file multiple times, operate correctly with simultaneous read/writes to the same file, or if you are unlucky, corrupt itself beyond all repair.

When IPv6 is enabled the tftpd thread will listen for requests from both IPv4 and IPv6 addresses.

As discussed in the description of the tftp_server_test above, an example filesystem is provided in `net/common/`*`VERSION`*`/src/tftp_dummy_file.c` for use by the tftp server test. The dummy filesystem is not a supported part of the network stack, it exists purely for demonstration purposes.

DHCP

This API publishes a routine to maintain DHCP state, and a semaphore that is signalled when a lease requires attention: this is your clue to call the aforementioned routine.

The intent with this API is that a simple DHCP client thread, which maintains the state of the interfaces, can go as follows: (after `init_all_network_interfaces()` is called from elsewhere)

```
while ( 1 ) {
    while ( 1 ) {
        cyg_semaphore_wait( &dhcp_needs_attention );
        if ( ! dhcp_bind() ) // a lease expired
            break; // If we need to re-bind
    }
    dhcp_down(); // tear down unbound interfaces
    init_all_network_interfaces(); // re-initialize
}
```

and if the application does not want to suffer the overhead of a separate thread and its stack for this, this functionality can be placed in the app,s server loop in an obvious fashion. That is the goal of breaking out these internal elements. For example, some server might be arranged to poll DHCP from time to time like this:

```
while ( 1 ) {
    init_all_network_interfaces();
    open-my-listen-sockets();
    while ( 1 ) {
        serve-one-request();
        // sleeps if no connections, but not forever;
        // so this loop is polled a few times a minute...
        if ( cyg_semaphore_trywait( &dhcp_needs_attention )) {
            if ( ! dhcp_bind() ) {
               close-my-listen-sockets();
               dhcp_down();
               break;
            }
        }
    }
}
```

If the configuration option CYGOPT_NET_DHCP_DHCP_THREAD is defined, then eCos provides a thread as described initially. Independent of this option, initialization of the interfaces still occurs in `init_all_network_interfaces()` and your startup code can call that. It will start the DHCP management thread if configured. If a lease fails to be renewed, the management thread will shut down all interfaces and attempt to initialize all the interfaces again from scratch. This may cause chaos in the app, which is why managing the DHCP state in an application aware thread is actually better, just far less convenient for testing.

If the configuration option CYGOPT_NET_DHCP_OPTION_HOST_NAME is defined, then the TAG_HOST_NAME DHCP option will be included in any DHCP lease requests. The text for the hostname is set by calling `dhcp_set_hostname()`. Any DHCP lease requests made prior to calling `dhcp_set_hostname()` will not include the TAG_HOST_NAME DHCP option. The configuration option CYGNUM_NET_DHCP_OPTION_HOST_NAME_LEN controls the maximum length allowed for the hostname. This permits the hostname text to be determined at run-time. Setting the hostname to the empty string will have the effect of disabling the TAG_HOST_NAME DHCP option.

If the configuration option CYGOPT_NET_DHCP_OPTION_DHCP_CLIENTID_MAC is defined, then the TAG_DHCP_CLIENTID DHCP option will be included in any DHCP lease requests. The client ID used will be the current MAC address of the network interface.

The option CYGOPT_NET_DHCP_PARM_REQ_LIST_ADDITIONAL allows additional DHCP options to be added to the request sent to the DHCP server. This option should be set to a comma separated list of options.

The option CYGOPT_NET_DHCP_PARM_REQ_LIST_REPLACE is similar to CYGOPT_NET_DHCP_PARM_REQ_LIST_ADDITIONAL but in this case it completely replaces the default list of options with the configured set of comma separated options.

Chapter 149. TCP/IP Library Reference

getdomainname

```
GETDOMAINNAME(3)         BSD Library Functions Manual         GETDOMAINNAME(3)

NAME
     getdomainname, setdomainname -- get/set YP domain name of current host

SYNOPSIS
     #include <unistd.h>

     int
     getdomainname(char *name, size_t namelen);

     int
     setdomainname(const char *name, size_t namelen);

DESCRIPTION
     The getdomainname() function returns the YP domain name for the current
     processor, as previously set by setdomainname().  The parameter namelen
     specifies the size of the name array.  If insufficient space is provided,
     the returned name is truncated.  The returned name is always null termi-
     nated.

     setdomainname() sets the domain name of the host machine to be name,
     which has length namelen.  This call is restricted to the superuser and
     is normally used only when the system is bootstrapped.

RETURN VALUES
     If the call succeeds a value of 0 is returned.  If the call fails, a
     value of -1 is returned and an error code is placed in the global vari-
     able errno.

ERRORS
     The following errors may be returned by these calls:

     [EFAULT]           The name or namelen parameter gave an invalid address.

     [EPERM]            The caller tried to set the domain name and was not
                        the superuser.

SEE ALSO
     domainname(1), gethostid(3), gethostname(3), sysctl(3), sysctl(8), yp(8)

BUGS
     Domain names are limited to MAXHOSTNAMELEN (from <sys/param.h>) charac-
     ters, currently 256.  This includes the terminating NUL character.

     If the buffer passed to getdomainname() is too small, other operating
     systems may not guarantee termination with NUL.

HISTORY
     The getdomainname function call appeared in SunOS 3.x.

BSD                               May 6, 1994                              BSD
```

gethostname

```
GETHOSTNAME(3)           BSD Library Functions Manual           GETHOSTNAME(3)

NAME
     gethostname, sethostname -- get/set name of current host
```

```
SYNOPSIS
     #include <unistd.h>

     int
     gethostname(char *name, size_t namelen);

     int
     sethostname(const char *name, size_t namelen);

DESCRIPTION
     The gethostname() function returns the standard host name for the current
     processor, as previously set by sethostname().  The parameter namelen
     specifies the size of the name array.  If insufficient space is provided,
     the returned name is truncated.  The returned name is always null termi-
     nated.

     sethostname() sets the name of the host machine to be name, which has
     length namelen.  This call is restricted to the superuser and is normally
     used only when the system is bootstrapped.

RETURN VALUES
     If the call succeeds a value of 0 is returned.  If the call fails, a
     value of -1 is returned and an error code is placed in the global vari-
     able errno.

ERRORS
     The following errors may be returned by these calls:

     [EFAULT]           The name or namelen parameter gave an invalid address.

     [EPERM]            The caller tried to set the hostname and was not the
                        superuser.

SEE ALSO
     hostname(1), getdomainname(3), gethostid(3), sysctl(3), sysctl(8), yp(8)

STANDARDS
     The gethostname() function call conforms to X/Open Portability Guide
     Issue 4, Version 2 (``XPG4.2'').

HISTORY
     The gethostname() function call appeared in 4.2BSD.

BUGS
     Host names are limited to MAXHOSTNAMELEN (from <sys/param.h>) characters,
     currently 256.  This includes the terminating NUL character.

     If the buffer passed to gethostname() is smaller than MAXHOSTNAMELEN,
     other operating systems may not guarantee termination with NUL.

BSD                              June 4, 1993                              BSD
```

byteorder

```
BYTEORDER(3)             BSD Library Functions Manual             BYTEORDER(3)

NAME
     htonl, htons, ntohl, ntohs, htobe32, htobe16, betoh32, betoh16, htole32,
     htole16, letoh32, letoh16, swap32, swap16 -- convert values between dif-
     ferent byte orderings

SYNOPSIS
     #include <sys/types.h>
     #include <machine/endian.h>
     u_int32_t
     htonl(u_int32_t host32);

     u_int16_t
     htons(u_int16_t host16);

     u_int32_t
     ntohl(u_int32_t net32);

     u_int16_t
     ntohs(u_int16_t net16);

     u_int32_t
     htobe32(u_int32_t host32);

     u_int16_t
     htobe16(u_int16_t host16);

     u_int32_t
     betoh32(u_int32_t big32);

     u_int16_t
     betoh16(u_int16_t big16);

     u_int32_t
     htole32(u_int32_t host32);

     u_int16_t
     htole16(u_int16_t host16);

     u_int32_t
     letoh32(u_int32_t little32);

     u_int16_t
     letoh16(u_int16_t little16);

     u_int32_t
     swap32(u_int32_t val32);

     u_int16_t
     swap16(u_int16_t val16);

DESCRIPTION
     These routines convert 16- and 32-bit quantities between different byte
     orderings.  The ``swap'' functions reverse the byte ordering of the given
     quantity, the others converts either from/to the native byte order used
     by the host to/from either little- or big-endian (a.k.a network) order.

     Apart from the swap functions, the names can be described by this form:
     {src-order}to{dst-order}{size}.  Both {src-order} and {dst-order} can
     take the following forms:

           h    Host order.
           n    Network order (big-endian).
           be   Big-endian (most significant byte first).
           le   Little-endian (least significant byte first).

     One of the specified orderings must be `h'.  {size} will take these
     forms:

           l  Long (32-bit, used in conjunction with forms involving `n').
           s  Short (16-bit, used in conjunction with forms involving `n').
           16
              16-bit.
           32
              32-bit.
     The swap functions are of the form: swap{size}.

     Names involving `n' convert quantities between network byte order and
     host byte order.  The last letter (`s' or `l') is a mnemonic for the tra-
     ditional names for such quantities, short and long, respectively.  Today,
     the C concept of short and long integers need not coincide with this tra-
     ditional misunderstanding.  On machines which have a byte order which is
     the same as the network order, routines are defined as null macros.

     The functions involving either ``be'', ``le'', or ``swap'' use the num-
     bers 16 and 32 for specifying the bitwidth of the quantities they operate
     on.  Currently all supported architectures are either big- or little-
     endian so either the ``be'' or ``le'' variants are implemented as null
     macros.

     The routines mentioned above which have either {src-order} or {dst-order}
     set to `n' are most often used in conjunction with Internet addresses and
     ports as returned by gethostbyname(3) and getservent(3).

SEE ALSO
     gethostbyname(3), getservent(3)

HISTORY
     The byteorder functions appeared in 4.2BSD.

BUGS
     On the vax, alpha, i386, and so far mips, bytes are handled backwards
     from most everyone else in the world.  This is not expected to be fixed
     in the near future.

BSD                              June 4, 1993                              BSD
```

ethers

```
ETHERS(3)                BSD Library Functions Manual                ETHERS(3)

NAME
     ether_aton, ether_ntoa, ether_addr, ether_ntohost, ether_hostton,
     ether_line -- get ethers entry

SYNOPSIS
     #include <netinet/if_ether.h>

     char *
     ether_ntoa(struct ether_addr *e);

     struct ether_addr *
     ether_aton(char *s);

     int
     ether_ntohost(char *hostname, struct ether_addr *e);

     int
     ether_hostton(char *hostname, struct ether_addr *e);

     int
     ether_line(char *l, struct ether_addr *e, char *hostname);

DESCRIPTION
     Ethernet addresses are represented by the following structure:

           struct ether_addr {
                   u_int8_t  ether_addr_octet[6];
           };

     The ether_ntoa() function converts this structure into an ASCII string of
     the form ``xx:xx:xx:xx:xx:xx'', consisting of 6 hexadecimal numbers sepa-
     rated by colons.  It returns a pointer to a static buffer that is reused
     for each call.  The ether_aton() converts an ASCII string of the same
     form and to a structure containing the 6 octets of the address.  It
     returns a pointer to a static structure that is reused for each call.

     The ether_ntohost() and ether_hostton() functions interrogate the data-
     base mapping host names to Ethernet addresses, /etc/ethers.  The
     ether_ntohost() function looks up the given Ethernet address and writes
     the associated host name into the character buffer passed.  This buffer
     should be MAXHOSTNAMELEN characters in size.  The ether_hostton() func-
     tion looks up the given host name and writes the associated Ethernet
     address into the structure passed.  Both functions return zero if they
     find the requested host name or address, and -1 if not.

     Each call reads /etc/ethers from the beginning; if a `+' appears alone on
     a line in the file, then ether_hostton() will consult the ethers.byname
     YP map, and ether_ntohost() will consult the ethers.byaddr YP map.

     The ether_line() function parses a line from the /etc/ethers file and
     fills in the passed struct ether_addr and character buffer with the Eth-
     ernet address and host name on the line.  It returns zero if the line was
     successfully parsed and -1 if not.  The character buffer should be
     MAXHOSTNAMELEN characters in size.

FILES
     /etc/ethers

SEE ALSO
     ethers(5)

HISTORY
     The ether_ntoa(), ether_aton(), ether_ntohost(), ether_hostton(), and
     ether_line() functions were adopted from SunOS and appeared in NetBSD 0.9
     b.

BUGS
     The data space used by these functions is static; if future use requires
     the data, it should be copied before any subsequent calls to these func-
     tions overwrite it.

BSD                            December 16, 1993                           BSD
```

getaddrinfo

```
GETADDRINFO(3)           BSD Library Functions Manual           GETADDRINFO(3)

NAME
     getaddrinfo, freeaddrinfo, gai_strerror -- nodename-to-address transla-
     tion in protocol-independent manner

SYNOPSIS
     #include <sys/types.h>
     #include <sys/socket.h>
     #include <netdb.h>

     int
     getaddrinfo(const char *nodename, const char *servname,
         const struct addrinfo *hints, struct addrinfo **res);

     void
     freeaddrinfo(struct addrinfo *ai);

     char *
     gai_strerror(int ecode);
```

```
DESCRIPTION
     The getaddrinfo() function is defined for protocol-independent nodename-
     to-address translation.  It performs the functionality of
     gethostbyname(3) and getservbyname(3), but in a more sophisticated man-
     ner.

     The addrinfo structure is defined as a result of including the <netdb.h>
     header:

     struct addrinfo {                                                  *
          int     ai_flags;     /* AI_PASSIVE, AI_CANONNAME, AI_NUMERICHOST */
          int     ai_family;    /* PF_xxx */
          int     ai_socktype;  /* SOCK_xxx */
          int     ai_protocol;  /* 0 or IPPROTO_xxx for IPv4 and IPv6 */
          size_t  ai_addrlen;   /* length of ai_addr */
          char   *ai_canonname; /* canonical name for nodename */
          struct sockaddr  *ai_addr; /* binary address */
          struct addrinfo  *ai_next; /* next structure in linked list */
     };

     The nodename and servname arguments are pointers to NUL-terminated
     strings or NULL.  One or both of these two arguments must be a non-null
     pointer.  In the normal client scenario, both the nodename and servname
     are specified.  In the normal server scenario, only the servname is spec-
     ified.  A non-null nodename string can be either a node name or a numeric
     host address string (i.e., a dotted-decimal IPv4 address or an IPv6 hex
     address).  A non-null servname string can be either a service name or a
     decimal port number.

     The caller can optionally pass an addrinfo structure, pointed to by the
     third argument, to provide hints concerning the type of socket that the
     caller supports.  In this hints structure all members other than
     ai_flags, ai_family, ai_socktype, and ai_protocol must be zero or a null
     pointer.  A value of PF_UNSPEC for ai_family means the caller will accept
     any protocol family.  A value of 0 for ai_socktype means the caller will
     accept any socket type.  A value of 0 for ai_protocol means the caller
     will accept any protocol.  For example, if the caller handles only TCP
     and not UDP, then the ai_socktype member of the hints structure should be
     set to SOCK_STREAM when getaddrinfo() is called.  If the caller handles
     only IPv4 and not IPv6, then the ai_family member of the hints structure
     should be set to PF_INET when getaddrinfo() is called.  If the third
     argument to getaddrinfo() is a null pointer, this is the same as if the
     caller had filled in an addrinfo structure initialized to zero with
     ai_family set to PF_UNSPEC.

     Upon successful return a pointer to a linked list of one or more addrinfo
     structures is returned through the final argument.  The caller can
     process each addrinfo structure in this list by following the ai_next
     pointer, until a null pointer is encountered.  In each returned addrinfo
     structure the three members ai_family, ai_socktype, and ai_protocol are
     the corresponding arguments for a call to the socket() function.  In each
     addrinfo structure the ai_addr member points to a filled-in socket
     address structure whose length is specified by the ai_addrlen member.

     If the AI_PASSIVE bit is set in the ai_flags member of the hints struc-
     ture, then the caller plans to use the returned socket address structure
     in a call to bind().  In this case, if the nodename argument is a null
     pointer, then the IP address portion of the socket address structure will
     be set to INADDR_ANY for an IPv4 address or IN6ADDR_ANY_INIT for an IPv6
     address.

     If the AI_PASSIVE bit is not set in the ai_flags member of the hints
     structure, then the returned socket address structure will be ready for a
     call to connect() (for a connection-oriented protocol) or either
     connect(), sendto(), or sendmsg() (for a connectionless protocol).  In
     this case, if the nodename argument is a null pointer, then the IP
     address portion of the socket address structure will be set to the loop-
```

```
       back address.

       If the AI_CANONNAME bit is set in the ai_flags member of the hints struc-
       ture, then upon successful return the ai_canonname member of the first
       addrinfo structure in the linked list will point to a NUL-terminated
       string containing the canonical name of the specified nodename.

       If the AI_NUMERICHOST bit is set in the ai_flags member of the hints
       structure, then a non-null nodename string must be a numeric host address
       string.  Otherwise an error of EAI_NONAME is returned.  This flag pre-
       vents any type of name resolution service (e.g., the DNS) from being
       called.

       The arguments to getaddrinfo() must sufficiently be consistent and unam-
       biguous.  Here are pitfall cases you may encounter:

       +   getaddrinfo() will raise an error if members of the hints structure
           are not consistent.  For example, for internet address families,
           getaddrinfo() will raise an error if you specify SOCK_STREAM to
           ai_socktype while you specify IPPROTO_UDP to ai_protocol.

       +   If you specify a servname which is defined only for certain
           ai_socktype, getaddrinfo() will raise an error because the arguments
           are not consistent.  For example, getaddrinfo() will raise an error
           if you ask for ``tftp'' service on SOCK_STREAM.

       +   For internet address families, if you specify servname while you set
           ai_socktype to SOCK_RAW, getaddrinfo() will raise an error, because
           service names are not defined for the internet SOCK_RAW space.

       +   If you specify a numeric servname, while leaving ai_socktype and
           ai_protocol unspecified, getaddrinfo() will raise an error.  This is
           because the numeric servname does not identify any socket type, and
           getaddrinfo() is not allowed to glob the argument in such case.

       All of the information returned by getaddrinfo() is dynamically allo-
       cated: the addrinfo structures, the socket address structures, and canon-
       ical node name strings pointed to by the addrinfo structures.  To return
       this information to the system the function freeaddrinfo() is called.
       The addrinfo structure pointed to by the ai argument is freed, along with
       any dynamic storage pointed to by the structure.  This operation is
       repeated until a NULL ai_next pointer is encountered.

       To aid applications in printing error messages based on the EAI_xxx codes
       returned by getaddrinfo(), gai_strerror() is defined.  The argument is
       one of the EAI_xxx values defined earlier and the return value points to
       a string describing the error.  If the argument is not one of the EAI_xxx
       values, the function still returns a pointer to a string whose contents
       indicate an unknown error.

     Extension for scoped IPv6 address
       The implementation allows experimental numeric IPv6 address notation with
       scope identifier.  By appending the percent character and scope identi-
       fier to addresses, you can fill sin6_scope_id field for addresses.  This
       would make management of scoped address easier, and allows cut-and-paste
       input of scoped address.

       At this moment the code supports only link-local addresses with the for-
       mat.  Scope identifier is hardcoded to name of hardware interface associ-
       ated with the link.  (such as ne0).  Example would be like
       ``fe80::1%ne0'', which means ``fe80::1 on the link associated with ne0
       interface''.

       The implementation is still very experimental and non-standard.  The cur-
       rent implementation assumes one-by-one relationship between interface and
       link, which is not necessarily true from the specification.
```

```
EXAMPLES
     The following code tries to connect to ``www.kame.net'' service ``http''.
     via stream socket.  It loops through all the addresses available, regard-
     less from address family.  If the destination resolves to IPv4 address,
     it will use AF_INET socket.  Similarly, if it resolves to IPv6, AF_INET6
     socket is used.  Observe that there is no hardcoded reference to particu-
     lar address family.  The code works even if getaddrinfo returns addresses
     that are not IPv4/v6.

           struct addrinfo hints, *res, *res0;
           int error;
           int s;
           const char *cause = NULL;

           memset(&hints, 0, sizeof(hints));
           hints.ai_family = PF_UNSPEC;
           hints.ai_socktype = SOCK_STREAM;
           error = getaddrinfo("www.kame.net", "http", &hints, &res0);
           if (error) {
                   errx(1, "%s", gai_strerror(error));
                   /*NOTREACHED*/
           }
           s = -1;
           for (res = res0; res; res = res->ai_next) {
                   s = socket(res->ai_family, res->ai_socktype,
                       res->ai_protocol);
                   if (s < 0) {
                           cause = "socket";
                           continue;
                   }

                   if (connect(s, res->ai_addr, res->ai_addrlen) < 0) {
                           cause = "connect";
                           close(s);
                           s = -1;
                           continue;
                   }

                   break;  /* okay we got one */
           }
           if (s < 0) {
                   err(1, cause);
                   /*NOTREACHED*/
           }
           freeaddrinfo(res0);

     The following example tries to open a wildcard listening socket onto ser-
     vice ``http'', for all the address families available.

           struct addrinfo hints, *res, *res0;
           int error;
           int s[MAXSOCK];
           int nsock;
           const char *cause = NULL;

           memset(&hints, 0, sizeof(hints));
           hints.ai_family = PF_UNSPEC;
           hints.ai_socktype = SOCK_STREAM;
           hints.ai_flags = AI_PASSIVE;
           error = getaddrinfo(NULL, "http", &hints, &res0);
           if (error) {
                   errx(1, "%s", gai_strerror(error));
                   /*NOTREACHED*/
           }
           nsock = 0;
           for (res = res0; res && nsock < MAXSOCK; res = res->ai_next) {
                   s[nsock] = socket(res->ai_family, res->ai_socktype,
                       res->ai_protocol);
                   if (s[nsock] < 0) {
                           cause = "socket";
                           continue;
                   }

                   if (bind(s[nsock], res->ai_addr, res->ai_addrlen) < 0) {
                           cause = "bind";
                           close(s[nsock]);
                           continue;
                   }
                   (void) listen(s[nsock], 5);

                   nsock++;
           }
           if (nsock == 0) {
                   err(1, cause);
                   /*NOTREACHED*/
           }
           freeaddrinfo(res0);

DIAGNOSTICS
     Error return status from getaddrinfo() is zero on success and non-zero on
     errors.  Non-zero error codes are defined in <netdb.h>, and as follows:

     EAI_ADDRFAMILY  Address family for nodename not supported.
     EAI_AGAIN       Temporary failure in name resolution.
     EAI_BADFLAGS    Invalid value for ai_flags.
     EAI_FAIL        Non-recoverable failure in name resolution.
     EAI_FAMILY      ai_family not supported.
     EAI_MEMORY      Memory allocation failure.
     EAI_NODATA      No address associated with nodename.
     EAI_NONAME      nodename nor servname provided, or not known.
     EAI_SERVICE     servname not supported for ai_socktype.
     EAI_SOCKTYPE    ai_socktype not supported.
     EAI_SYSTEM      System error returned in errno.

     If called with proper argument, gai_strerror() returns a pointer to a
     string describing the given error code.  If the argument is not one of
     the EAI_xxx values, the function still returns a pointer to a string
     whose contents indicate an unknown error.

SEE ALSO
     getnameinfo(3), gethostbyname(3), getservbyname(3), hosts(5),
     resolv.conf(5), services(5), hostname(7), named(8)

     R. Gilligan, S. Thomson, J. Bound, and W. Stevens, Basic Socket Interface
     Extensions for IPv6, RFC2553, March 1999.

     Tatsuya Jinmei and Atsushi Onoe, An Extension of Format for IPv6 Scoped
     Addresses, internet draft, draft-ietf-ipngwg-scopedaddr-format-02.txt,
     work in progress material.

     Craig Metz, "Protocol Independence Using the Sockets API", Proceedings of
     the freenix track: 2000 USENIX annual technical conference, June 2000.

HISTORY
     The implementation first appeared in WIDE Hydrangea IPv6 protocol stack
     kit.

STANDARDS
     The getaddrinfo() function is defined in IEEE POSIX 1003.1g draft speci-
     fication, and documented in ``Basic Socket Interface Extensions for
     IPv6'' (RFC2553).

BUGS
     The text was shamelessly copied from RFC2553.
```

```
BSD                                May 25, 1995                            BSD
```

gethostbyname

```
GETHOSTBYNAME(3)         BSD Library Functions Manual         GETHOSTBYNAME(3)

NAME
     gethostbyname, gethostbyname2, gethostbyaddr, gethostent, sethostent,
     endhostent, hstrerror, herror -- get network host entry

SYNOPSIS
     #include <netdb.h>
     extern int h_errno;

     struct hostent *
     gethostbyname(const char *name);

     struct hostent *
     gethostbyname2(const char *name, int af);

     struct hostent *
     gethostbyaddr(const char *addr, int len, int af);

     struct hostent *
     gethostent(void);

     void
     sethostent(int stayopen);

     void
     endhostent(void);

     void
     herror(const char *string);

     const char *
     hstrerror(int err);

DESCRIPTION
     The gethostbyname() and gethostbyaddr() functions each return a pointer
     to an object with the following structure describing an internet host
     referenced by name or by address, respectively.  This structure contains
     either information obtained from the name server (i.e., resolver(3) and
     named(8)), broken-out fields from a line in /etc/hosts, or database
     entries supplied by the yp(8) system.  resolv.conf(5) describes how the
     particular database is chosen.

     struct  hostent {
             char    *h_name;        /* official name of host */
             char    **h_aliases;    /* alias list */
             int     h_addrtype;     /* host address type */
             int     h_length;       /* length of address */
             char    **h_addr_list;  /* list of addresses from name server */
     };
     #define h_addr  h_addr_list[0]  /* address, for backward compatibility */

     The members of this structure are:

     h_name       Official name of the host.

     h_aliases    A zero-terminated array of alternate names for the host.

     h_addrtype   The type of address being returned.

     h_length     The length, in bytes, of the address.
     h_addr_list  A zero-terminated array of network addresses for the host.
                  Host addresses are returned in network byte order.

     h_addr       The first address in h_addr_list; this is for backward com-
                  patibility.

     The function gethostbyname() will search for the named host in the cur-
     rent domain and its parents using the search lookup semantics detailed in
     resolv.conf(5) and hostname(7).

     gethostbyname2() is an advanced form of gethostbyname() which allows
     lookups in address families other than AF_INET, for example AF_INET6.

     The gethostbyaddr() function will search for the specified address of
     length len in the address family af.  The only address family currently
     supported is AF_INET.

     The sethostent() function may be used to request the use of a connected
     TCP socket for queries.  If the stayopen flag is non-zero, this sets the
     option to send all queries to the name server using TCP and to retain the
     connection after each call to gethostbyname() or gethostbyaddr().  Other-
     wise, queries are performed using UDP datagrams.

     The endhostent() function closes the TCP connection.

     The herror() function prints an error message describing the failure.  If
     its argument string is non-null, it is prepended to the message string
     and separated from it by a colon (`:') and a space.  The error message is
     printed with a trailing newline.  The contents of the error message is
     the same as that returned by hstrerror() with argument h_errno.

FILES
     /etc/hosts
     /etc/resolv.conf

DIAGNOSTICS
     Error return status from gethostbyname(), gethostbyname2(), and
     gethostbyaddr() is indicated by return of a null pointer.  The external
     integer h_errno may then be checked to see whether this is a temporary
     failure or an invalid or unknown host.

     The variable h_errno can have the following values:

     HOST_NOT_FOUND  No such host is known.

     TRY_AGAIN       This is usually a temporary error and means that the
                     local server did not receive a response from an authori-
                     tative server.  A retry at some later time may succeed.

     NO_RECOVERY     Some unexpected server failure was encountered.  This is
                     a non-recoverable error.

     NO_DATA         The requested name is valid but does not have an IP
                     address; this is not a temporary error.  This means that
                     the name is known to the name server but there is no
                     address associated with this name.  Another type of
                     request to the name server using this domain name will
                     result in an answer; for example, a mail-forwarder may be
                     registered for this domain.

SEE ALSO
     resolver(3), getaddrinfo(3), getnameinfo(3), hosts(5), resolv.conf(5),
     hostname(7), named(8)

CAVEAT
     If the search routines in resolv.conf(5) decide to read the /etc/hosts
     file, gethostent() and other functions will read the next line of the
     file, re-opening the file if necessary.

     The sethostent() function opens and/or rewinds the file /etc/hosts.  If
     the stayopen argument is non-zero, the file will not be closed after each
     call to gethostbyname(), gethostbyname2(), or gethostbyaddr().

     The endhostent() function closes the file.

HISTORY
     The herror() function appeared in 4.3BSD.  The endhostent(),
     gethostbyaddr(), gethostbyname(), gethostent(), and sethostent() func-
     tions appeared in 4.2BSD.

BUGS
     These functions use static data storage; if the data is needed for future
     use, it should be copied before any subsequent calls overwrite it.  Only
     the Internet address formats are currently understood.

     YP does not support any address families other than AF_INET and uses the
     traditional database format.

BSD                             March 13, 1997                             BSD
```

getifaddrs

```
GETIFADDRS(3)            BSD Library Functions Manual            GETIFADDRS(3)

NAME
     getifaddrs -- get interface addresses

SYNOPSIS
     #include <sys/types.h>
     #include <sys/socket.h>
     #include <ifaddrs.h>

     int
     getifaddrs(struct ifaddrs **ifap);

     void
     freeifaddrs(struct ifaddrs *ifap);

DESCRIPTION
     The getifaddrs() function stores a reference to a linked list of the net-
     work interfaces on the local machine in the memory referenced by ifap.
     The list consists of ifaddrs structures, as defined in the include file
     <ifaddrs.h>.  The ifaddrs structure contains at least the following
     entries:

         struct ifaddrs   *ifa_next;         /* Pointer to next struct */
         char             *ifa_name;         /* Interface name */
         u_int             ifa_flags;        /* Interface flags */
         struct sockaddr  *ifa_addr;         /* Interface address */
         struct sockaddr  *ifa_netmask;      /* Interface netmask */
         struct sockaddr  *ifa_broadaddr;    /* Interface broadcast address */
         struct sockaddr  *ifa_dstaddr;      /* P2P interface destination */
         void             *ifa_data;         /* Address specific data */

     ifa_next
             Contains a pointer to the next structure on the list.  This field
             is set to NULL in last structure on the list.

     ifa_name
             Contains the interface name.

     ifa_flags
```

```
             Contains the interface flags, as set by ifconfig(8).

     ifa_addr
             References either the address of the interface or the link level
             address of the interface, if one exists, otherwise it is NULL.
             (The sa_family field of the ifa_addr field should be consulted to
             determine the format of the ifa_addr address.)

     ifa_netmask
             References the netmask associated with ifa_addr, if one is set,
             otherwise it is NULL.

     ifa_broadaddr
             This field, which should only be referenced for non-P2P inter-
             faces, references the broadcast address associated with ifa_addr,
             if one exists, otherwise it is NULL.

     ifa_dstaddr
             References the destination address on a P2P interface, if one
             exists, otherwise it is NULL.

     ifa_data
             References address family specific data.  For AF_LINK addresses
             it contains a pointer to the struct if_data (as defined in
             include file <net/if.h>) which contains various interface
             attributes and statistics.  For all other address families, it
             contains a pointer to the struct ifa_data (as defined in include
             file <net/if.h>) which contains per-address interface statistics.

     The data returned by getifaddrs() is dynamically allocated and should be
     freed using freeifaddrs() when no longer needed.

RETURN VALUES
     Upon successful completion, a value of 0 is returned.  Otherwise, a value
     of -1 is returned and errno is set to indicate the error.

ERRORS
     The getifaddrs() may fail and set errno for any of the errors specified
     for the library routines ioctl(2), socket(2), malloc(3), or sysctl(3).

BUGS
     If both <net/if.h> and <ifaddrs.h> are being included, <net/if.h> must be
     included before <ifaddrs.h>.

SEE ALSO
     ioctl(2), socket(2), sysctl(3), networking(4), ifconfig(8)

HISTORY
     The getifaddrs() function first appeared in BSDI BSD/OS.  The function is
     supplied on OpenBSD since OpenBSD 2.7.

                               September 3, 2013
```

getnameinfo

```
GETNAMEINFO(3)           BSD Library Functions Manual           GETNAMEINFO(3)

NAME
     getnameinfo -- address-to-nodename translation in protocol-independent
     manner

SYNOPSIS
     #include <sys/types.h>
     #include <sys/socket.h>
     #include <netdb.h>
     int
     getnameinfo(const struct sockaddr *sa, socklen_t salen, char *host,
         size_t hostlen, char *serv, size_t servlen, int flags);

DESCRIPTION
     The getnameinfo() function is defined for protocol-independent address-
     to-nodename translation.  Its functionality is a reverse conversion of
     getaddrinfo(3), and implements similar functionality with
     gethostbyaddr(3) and getservbyport(3) in more sophisticated manner.

     This function looks up an IP address and port number provided by the
     caller in the DNS and system-specific database, and returns text strings
     for both in buffers provided by the caller.  The function indicates suc-
     cessful completion by a zero return value; a non-zero return value indi-
     cates failure.

     The first argument, sa, points to either a sockaddr_in structure (for
     IPv4) or a sockaddr_in6 structure (for IPv6) that holds the IP address
     and port number.  The salen argument gives the length of the sockaddr_in
     or sockaddr_in6 structure.

     The function returns the nodename associated with the IP address in the
     buffer pointed to by the host argument.  The caller provides the size of
     this buffer via the hostlen argument.  The service name associated with
     the port number is returned in the buffer pointed to by serv, and the
     servlen argument gives the length of this buffer.  The caller specifies
     not to return either string by providing a zero value for the hostlen or
     servlen arguments.  Otherwise, the caller must provide buffers large
     enough to hold the nodename and the service name, including the terminat-
     ing null characters.

     Unfortunately most systems do not provide constants that specify the max-
     imum size of either a fully-qualified domain name or a service name.
     Therefore to aid the application in allocating buffers for these two
     returned strings the following constants are defined in <netdb.h>:

     #define NI_MAXHOST    MAXHOSTNAMELEN
     #define NI_MAXSERV    32

     The first value is actually defined as the constant MAXDNAME in recent
     versions of BIND's <arpa/nameser.h> header (older versions of BIND define
     this constant to be 256) and the second is a guess based on the services
     listed in the current Assigned Numbers RFC.

     The final argument is a flag that changes the default actions of this
     function.  By default the fully-qualified domain name (FQDN) for the host
     is looked up in the DNS and returned.  If the flag bit NI_NOFQDN is set,
     only the nodename portion of the FQDN is returned for local hosts.

     If the flag bit NI_NUMERICHOST is set, or if the host's name cannot be
     located in the DNS, the numeric form of the host's address is returned
     instead of its name (e.g., by calling inet_ntop() instead of
     gethostbyaddr()).  If the flag bit NI_NAMEREQD is set, an error is
     returned if the host's name cannot be located in the DNS.

     If the flag bit NI_NUMERICSERV is set, the numeric form of the service
     address is returned (e.g., its port number) instead of its name.  The two
     NI_NUMERICxxx flags are required to support the -n flag that many com-
     mands provide.

     A fifth flag bit, NI_DGRAM, specifies that the service is a datagram ser-
     vice, and causes getservbyport() to be called with a second argument of
     "udp" instead of its default of "tcp".  This is required for the few
     ports (512-514) that have different services for UDP and TCP.

     These NI_xxx flags are defined in <netdb.h>.
```

```
   Extension for scoped IPv6 address
     The implementation allows experimental numeric IPv6 address notation with
     scope identifier.  IPv6 link-local address will appear as string like
     ``fe80::1%ne0'', if NI_WITHSCOPEID bit is enabled in flags argument.
     Refer to getaddrinfo(3) for the notation.

EXAMPLES
     The following code tries to get numeric hostname, and service name, for
     given socket address.  Observe that there is no hardcoded reference to
     particular address family.

           struct sockaddr *sa;    /* input */
           char hbuf[NI_MAXHOST], sbuf[NI_MAXSERV];

           if (getnameinfo(sa, sa->sa_len, hbuf, sizeof(hbuf), sbuf,
               sizeof(sbuf), NI_NUMERICHOST | NI_NUMERICSERV)) {
                   errx(1, "could not get numeric hostname");
                   /*NOTREACHED*/
           }
           printf("host=%s, serv=%s\n", hbuf, sbuf);

     The following version checks if the socket address has reverse address
     mapping.

           struct sockaddr *sa;    /* input */
           char hbuf[NI_MAXHOST];

           if (getnameinfo(sa, sa->sa_len, hbuf, sizeof(hbuf), NULL, 0,
               NI_NAMEREQD)) {
                   errx(1, "could not resolve hostname");
                   /*NOTREACHED*/
           }
           printf("host=%s\n", hbuf);

DIAGNOSTICS
     The function indicates successful completion by a zero return value; a
     non-zero return value indicates failure.  Error codes are as below:

     EAI_AGAIN          The name could not be resolved at this time.  Future
                        attempts may succeed.

     EAI_BADFLAGS       The flags had an invalid value.

     EAI_FAIL           A non-recoverable error occurred.

     EAI_FAMILY         The address family was not recognized or the address
                        length was invalid for the specified family.

     EAI_MEMORY         There was a memory allocation failure.

     EAI_NONAME         The name does not resolve for the supplied parameters.
                        NI_NAMEREQD is set and the host's name cannot be
                        located, or both nodename and servname were null.

     EAI_SYSTEM         A system error occurred.  The error code can be found
                        in errno.

SEE ALSO
     getaddrinfo(3), gethostbyaddr(3), getservbyport(3), hosts(5),
     resolv.conf(5), services(5), hostname(7), named(8)

     R. Gilligan, S. Thomson, J. Bound, and W. Stevens, Basic Socket Interface
     Extensions for IPv6, RFC2553, March 1999.

     Tatsuya Jinmei and Atsushi Onoe, An Extension of Format for IPv6 Scoped
     Addresses, internet draft, draft-ietf-ipngwg-scopedaddr-format-02.txt,
     work in progress material.
     Craig Metz, "Protocol Independence Using the Sockets API", Proceedings of
     the freenix track: 2000 USENIX annual technical conference, June 2000.

HISTORY
     The implementation first appeared in WIDE Hydrangea IPv6 protocol stack
     kit.

STANDARDS
     The getaddrinfo() function is defined IEEE POSIX 1003.1g draft specifica-
     tion, and documented in ``Basic Socket Interface Extensions for IPv6''
     (RFC2553).

BUGS
     The current implementation is not thread-safe.

     The text was shamelessly copied from RFC2553.

     OpenBSD intentionally uses different NI_MAXHOST value from what RFC2553
     suggests, to avoid buffer length handling mistakes.

BSD                             May 25, 1995                             BSD
```

getnetent

```
GETNETENT(3)             BSD Library Functions Manual             GETNETENT(3)

NAME
     getnetent, getnetbyaddr, getnetbyname, setnetent, endnetent -- get net-
     work entry

SYNOPSIS
     #include <netdb.h>

     struct netent *
     getnetent(void);

     struct netent *
     getnetbyname(char *name);

     struct netent *
     getnetbyaddr(in_addr_t net, int type);

     void
     setnetent(int stayopen);

     void
     endnetent(void);

DESCRIPTION
     The getnetent(), getnetbyname(), and getnetbyaddr() functions each return
     a pointer to an object with the following structure containing the bro-
     ken-out fields of a line in the network database, /etc/networks.

           struct  netent {
                   char            *n_name;        /* official name of net */
                   char            **n_aliases;    /* alias list */
                   int             n_addrtype;     /* net number type */
                   in_addr_t       n_net;          /* net number */
           };

     The members of this structure are:

     n_name      The official name of the network.

     n_aliases   A zero-terminated list of alternate names for the network.
     n_addrtype  The type of the network number returned; currently only
                 AF_INET.

     n_net       The network number.  Network numbers are returned in machine
                 byte order.

     The getnetent() function reads the next line of the file, opening the
     file if necessary.

     The setnetent() function opens and rewinds the file.  If the stayopen
     flag is non-zero, the net database will not be closed after each call to
     getnetbyname() or getnetbyaddr().

     The endnetent() function closes the file.

     The getnetbyname() and getnetbyaddr() functions search the domain name
     server if the system is configured to use one.  If the search fails, or
     no name server is configured, they sequentially search from the beginning
     of the file until a matching net name or net address and type is found,
     or until EOF is encountered.  Network numbers are supplied in host order.

FILES
     /etc/networks

DIAGNOSTICS
     Null pointer (0) returned on EOF or error.

SEE ALSO
     resolver(3), networks(5)

HISTORY
     The getnetent(), getnetbyaddr(), getnetbyname(), setnetent(), and
     endnetent() functions appeared in 4.2BSD.

BUGS
     The data space used by these functions is static; if future use requires
     the data, it should be copied before any subsequent calls to these func-
     tions overwrite it.  Only Internet network numbers are currently under-
     stood.  Expecting network numbers to fit in no more than 32 bits is
     naive.

BSD                             March 13, 1997                             BSD
```

getprotoent

```
GETPROTOENT(3)           BSD Library Functions Manual           GETPROTOENT(3)

NAME
     getprotoent, getprotobynumber, getprotobyname, setprotoent, endprotoent
     -- get protocol entry

SYNOPSIS
     #include <netdb.h>

     struct protoent *
     getprotoent(void);

     struct protoent *
     getprotobyname(char *name);

     struct protoent *
     getprotobynumber(int proto);

     void
     setprotoent(int stayopen);
     void
     endprotoent(void);

DESCRIPTION
     The getprotoent(), getprotobyname(), and getprotobynumber() functions
     each return a pointer to an object with the following structure contain-
     ing the broken-out fields of a line in the network protocol database,
     /etc/protocols.


           struct  protoent {
                   char    *p_name;        /* official name of protocol */
                   char    **p_aliases;    /* alias list */
                   int     p_proto;        /* protocol number */
           };

     The members of this structure are:

     p_name     The official name of the protocol.

     p_aliases  A zero-terminated list of alternate names for the protocol.

     p_proto    The protocol number.

     The getprotoent() function reads the next line of the file, opening the
     file if necessary.

     The setprotoent() function opens and rewinds the file.  If the stayopen
     flag is non-zero, the net database will not be closed after each call to
     getprotobyname() or getprotobynumber().

     The endprotoent() function closes the file.

     The getprotobyname() and getprotobynumber() functions sequentially search
     from the beginning of the file until a matching protocol name or protocol
     number is found, or until EOF is encountered.

RETURN VALUES
     Null pointer (0) returned on EOF or error.

FILES
     /etc/protocols

SEE ALSO
     protocols(5)

HISTORY
     The getprotoent(), getprotobynumber(), getprotobyname(), setprotoent(),
     and endprotoent() functions appeared in 4.2BSD.

BUGS
     These functions use a static data space; if the data is needed for future
     use, it should be copied before any subsequent calls overwrite it.  Only
     the Internet protocols are currently understood.

BSD                             June 4, 1993                             BSD
```

getrrsetbyname

```
GETRRSETBYNAME(3)        BSD Library Functions Manual        GETRRSETBYNAME(3)

NAME
     getrrsetbyname -- retrieve DNS records

SYNOPSIS
```

```
     #include <netdb.h>

     int
     getrrsetbyname(const char *hostname, unsigned int rdclass,
         unsigned int rdtype, unsigned int flags, struct rrsetinfo **res);

     int
     freerrset(struct rrsetinfo **rrset);

DESCRIPTION
     getrrsetbyname() gets a set of resource records associated with a
     hostname, class and type.  hostname is a pointer a to null-terminated
     string.  The flags field is currently unused and must be zero.

     After a successful call to getrrsetbyname(), *res is a pointer to an
     rrsetinfo structure, containing a list of one or more rdatainfo struc-
     tures containing resource records and potentially another list of
     rdatainfo structures containing SIG resource records associated with
     those records.  The members rri_rdclass and rri_rdtype are copied from
     the parameters.  rri_ttl and rri_name are properties of the obtained
     rrset.  The resource records contained in rri_rdatas and rri_sigs are in
     uncompressed DNS wire format.  Properties of the rdataset are represented
     in the rri_flags bitfield. If the RRSET_VALIDATED bit is set, the data
     has been DNSSEC validated and the signatures verified.

     The following structures are used:

     struct  rdatainfo {
             unsigned int            rdi_length;     /* length of data */
             unsigned char           *rdi_data;      /* record data */
     };

     struct  rrsetinfo {
             unsigned int            rri_flags;      /* RRSET_VALIDATED … */
             unsigned int            rri_rdclass;    /* class number */
             unsigned int            rri_rdtype;     /* RR type number */
             unsigned int            rri_ttl;        /* time to live */
             unsigned int            rri_nrdatas;    /* size of rdatas array */
             unsigned int            rri_nsigs;      /* size of sigs array */
             char                    *rri_name;      /* canonical name */
             struct rdatainfo        *rri_rdatas;    /* individual records */
             struct rdatainfo        *rri_sigs;      /* individual signatures */
     };

     All of the information returned by getrrsetbyname() is dynamically allo-
     cated: the rrsetinfo and rdatainfo structures, and the canonical host
     name strings pointed to by the rrsetinfostructure. Memory allocated for
     the dynamically allocated structures created by a successful call to
     getrrsetbyname() is released by freerrset().  rrset is a pointer to a
     struct rrset created by a call to getrrsetbyname().

     If the EDNS0 option is activated in resolv.conf(3), getrrsetbyname() will
     request DNSSEC authentication using the EDNS0 DNSSEC OK (DO) bit.

RETURN VALUES
     getrrsetbyname() returns zero on success, and one of the following error
     codes if an error occurred:

     ERRSET_NONAME    the name does not exist
     ERRSET_NODATA    the name exists, but does not have data of the desired
                      type
     ERRSET_NOMEMORY  memory could not be allocated
     ERRSET_INVAL     a parameter is invalid
     ERRSET_FAIL      other failure

SEE ALSO
     resolver(3), resolv.conf(5), named(8)
```

```
AUTHORS
     Jakob Schlyter <jakob@openbsd.org>

HISTORY
     getrrsetbyname() first appeared in OpenBSD 3.0.  The API first appeared
     in ISC BIND version 9.

BUGS
     The data in *rdi_data should be returned in uncompressed wire format.
     Currently, the data is in compressed format and the caller can't uncom-
     press since it doesn't have the full message.

CAVEATS
     The RRSET_VALIDATED flag in rri_flags is set if the AD (autenticated
     data) bit in the DNS answer is set. This flag should not be trusted
     unless the transport between the nameserver and the resolver is secure
     (e.g. IPsec, trusted network, loopback communication).

BSD                              Oct 18, 2000                              BSD
```

getservent

```
GETSERVENT(3)            BSD Library Functions Manual            GETSERVENT(3)

NAME
     getservent, getservbyport, getservbyname, setservent, endservent -- get
     service entry

SYNOPSIS
     #include <netdb.h>

     struct servent *
     getservent(void);

     struct servent *
     getservbyname(char *name, char *proto);

     struct servent *
     getservbyport(int port, char *proto);

     void
     setservent(int stayopen);

     void
     endservent(void);

DESCRIPTION
     The getservent(), getservbyname(), and getservbyport() functions each
     return a pointer to an object with the following structure containing the
     broken-out fields of a line in the network services database,
     /etc/services.

           struct  servent {
                   char    *s_name;        /* official name of service */
                   char    **s_aliases;    /* alias list */
                   int     s_port;         /* port service resides at */
                   char    *s_proto;       /* protocol to use */
           };

     The members of this structure are:

     s_name     The official name of the service.

     s_aliases  A zero-terminated list of alternate names for the service.
     s_port     The port number at which the service resides.  Port numbers
                are returned in network byte order.

     s_proto    The name of the protocol to use when contacting the service.

     The getservent() function reads the next line of the file, opening the
     file if necessary.

     The setservent() function opens and rewinds the file.  If the stayopen
     flag is non-zero, the net database will not be closed after each call to
     getservbyname() or getservbyport().

     The endservent() function closes the file.

     The getservbyname() and getservbyport() functions sequentially search
     from the beginning of the file until a matching protocol name or port
     number (specified in network byte order) is found, or until EOF is
     encountered.  If a protocol name is also supplied (non-null), searches
     must also match the protocol.

FILES
     /etc/services

DIAGNOSTICS
     Null pointer (0) returned on EOF or error.

SEE ALSO
     getprotoent(3), services(5)

HISTORY
     The getservent(), getservbyport(), getservbyname(), setservent(), and
     endservent() functions appeared in 4.2BSD.

BUGS
     These functions use static data storage; if the data is needed for future
     use, it should be copied before any subsequent calls overwrite it.
     Expecting port numbers to fit in a 32-bit quantity is probably naive.

BSD                             January 12, 1994                           BSD
```

if_nametoindex

```
IF_NAMETOINDEX(3)        BSD Library Functions Manual        IF_NAMETOINDEX(3)

NAME
     if_nametoindex, if_indextoname, if_nameindex, if_freenameindex -- convert
     interface index to name, and vice versa

SYNOPSIS
     #include <net/if.h>

     unsigned int
     if_nametoindex(const char *ifname);

     char *
     if_indextoname(unsigned int ifindex, char *ifname);

     struct if_nameindex *
     if_nameindex(void);

     void
     if_freenameindex(struct if_nameindex *ptr);

DESCRIPTION
     These functions map interface indexes to interface names (such as
     ``lo0''), and vice versa.
     The if_nametoindex() function converts an interface name specified by the
     ifname argument to an interface index (positive integer value).  If the
     specified interface does not exist, 0 will be returned.

     if_indextoname() converts an interface index specified by the ifindex
     argument to an interface name.  The ifname argument must point to a buf-
     fer of at least IF_NAMESIZE bytes into which the interface name corre-
     sponding to the specified index is returned.  (IF_NAMESIZE is also
     defined in <net/if.h> and its value includes a terminating null byte at
     the end of the interface name.)  This pointer is also the return value of
     the function.  If there is no interface corresponding to the specified
     index, NULL is returned.

     if_nameindex() returns an array of if_nameindex structures.
     if_nametoindex is also defined in <net/if.h>, and is as follows:

     struct if_nameindex {
         unsigned int   if_index;  /* 1, 2, … */
         char          *if_name;   /* null terminated name: "le0", … */
     };

     The end of the array of structures is indicated by a structure with an
     if_index of 0 and an if_name of NULL.  The function returns a null
     pointer on error.  The memory used for this array of structures along
     with the interface names pointed to by the if_name members is obtained
     dynamically.  This memory is freed by the if_freenameindex() function.

     if_freenameindex() takes a pointer that was returned by if_nameindex() as
     argument (ptr), and it reclaims the region allocated.

DIAGNOSTICS
     if_nametoindex() returns 0 on error, positive integer on success.
     if_indextoname() and if_nameindex() return NULL on errors.

SEE ALSO
     R. Gilligan, S.  Thomson, J. Bound, and W. Stevens, ``Basic Socket Inter-
     face Extensions for IPv6,'' RFC2553, March 1999.

STANDARDS
     These functions are defined in ``Basic Socket Interface Extensions for
     IPv6'' (RFC2533).

BSD                              May 21, 1998                              BSD
```

inet

```
INET(3)                  BSD Library Functions Manual                  INET(3)

NAME
     inet_addr, inet_aton, inet_lnaof, inet_makeaddr, inet_netof,
     inet_network, inet_ntoa, inet_ntop, inet_pton -- Internet address manipu-
     lation routines

SYNOPSIS
     #include <sys/socket.h>
     #include <netinet/in.h>
     #include <arpa/inet.h>

     in_addr_t
     inet_addr(const char *cp);

     int
     inet_aton(const char *cp, struct in_addr *addr);

     in_addr_t
     inet_lnaof(struct in_addr in);

     struct in_addr
     inet_makeaddr(unsigned long net, unsigned long lna);

     in_addr_t
     inet_netof(struct in_addr in);

     in_addr_t
     inet_network(const char *cp);

     char *
     inet_ntoa(struct in_addr in);

     const char *
     inet_ntop(int af, const void *src, char *dst, size_t size);

     int
     inet_pton(int af, const char *src, void *dst);

DESCRIPTION
     The routines inet_aton(), inet_addr() and inet_network() interpret char-
     acter strings representing numbers expressed in the Internet standard `.'
     notation.  The inet_pton() function converts a presentation format
     address (that is, printable form as held in a character string) to net-
     work format (usually a struct in_addr or some other internal binary rep-
     resentation, in network byte order).  It returns 1 if the address was
     valid for the specified address family, or 0 if the address wasn't
     parseable in the specified address family, or -1 if some system error
     occurred (in which case errno will have been set).  This function is
     presently valid for AF_INET and AF_INET6.  The inet_aton() routine inter-
     prets the specified character string as an Internet address, placing the
     address into the structure provided.  It returns 1 if the string was suc-
     cessfully interpreted, or 0 if the string was invalid.  The inet_addr()
     and inet_network() functions return numbers suitable for use as Internet
     addresses and Internet network numbers, respectively.

     The function inet_ntop() converts an address from network format (usually
     a struct in_addr or some other binary form, in network byte order) to
     presentation format (suitable for external display purposes).  It returns
     NULL if a system error occurs (in which case, errno will have been set),
     or it returns a pointer to the destination string.  The routine
     inet_ntoa() takes an Internet address and returns an ASCII string repre-
     senting the address in `.' notation.  The routine inet_makeaddr() takes
     an Internet network number and a local network address and constructs an
     Internet address from it.  The routines inet_netof() and inet_lnaof()
     break apart Internet host addresses, returning the network number and
     local network address part, respectively.

     All Internet addresses are returned in network order (bytes ordered from
     left to right).  All network numbers and local address parts are returned
     as machine format integer values.

INTERNET ADDRESSES (IP VERSION 4)
     Values specified using the `.' notation take one of the following forms:

           a.b.c.d
           a.b.c
           a.b
           a

     When four parts are specified, each is interpreted as a byte of data and
     assigned, from left to right, to the four bytes of an Internet address.
     Note that when an Internet address is viewed as a 32-bit integer quantity
     on a system that uses little-endian byte order (such as the Intel 386,
     486 and Pentium processors) the bytes referred to above appear as
     ``d.c.b.a''.  That is, little-endian bytes are ordered from right to
     left.

     When a three part address is specified, the last part is interpreted as a
     16-bit quantity and placed in the rightmost two bytes of the network
     address.  This makes the three part address format convenient for speci-
     fying Class B network addresses as ``128.net.host''.

     When a two part address is supplied, the last part is interpreted as a
     24-bit quantity and placed in the rightmost three bytes of the network
     address.  This makes the two part address format convenient for specify-
     ing Class A network addresses as ``net.host''.

     When only one part is given, the value is stored directly in the network
     address without any byte rearrangement.

     All numbers supplied as ``parts'' in a `.' notation may be decimal,
     octal, or hexadecimal, as specified in the C language (i.e., a leading 0x
     or 0X implies hexadecimal; otherwise, a leading 0 implies octal; other-
     wise, the number is interpreted as decimal).

INTERNET ADDRESSES (IP VERSION 6)
     In order to support scoped IPv6 addresses, getaddrinfo(3) and
     getnameinfo(3) are recommended rather than the functions presented here.

     The presentation format of an IPv6 address is given in [RFC1884 2.2]:

     There are three conventional forms for representing IPv6 addresses as
     text strings:

     1.   The preferred form is x:x:x:x:x:x:x:x, where the 'x's are the hexa-
          decimal values of the eight 16-bit pieces of the address.  Examples:

                FEDC:BA98:7654:3210:FEDC:BA98:7654:3210
                1080:0:0:0:8:800:200C:417A

          Note that it is not necessary to write the leading zeros in an indi-
          vidual field, but there must be at least one numeral in every field
          (except for the case described in 2.).

     2.   Due to the method of allocating certain styles of IPv6 addresses, it
          will be common for addresses to contain long strings of zero bits.
          In order to make writing addresses

          containing zero bits easier a special syntax is available to com-
          press the zeros.  The use of ``::'' indicates multiple groups of 16
          bits of zeros.  The ``::'' can only appear once in an address.  The
          ``::'' can also be used to compress the leading and/or trailing
          zeros in an address.

          For example the following addresses:

                1080:0:0:0:8:800:200C:417A  a unicast address
                FF01:0:0:0:0:0:0:43         a multicast address
                0:0:0:0:0:0:0:1             the loopback address
                0:0:0:0:0:0:0:0             the unspecified addresses

          may be represented as:

                1080::8:800:200C:417A       a unicast address
                FF01::43                    a multicast address
                ::1                         the loopback address
                ::                          the unspecified addresses

     3.   An alternative form that is sometimes more convenient when dealing
          with a mixed environment of IPv4 and IPv6 nodes is
          x:x:x:x:x:x:d.d.d.d, where the 'x's are the hexadecimal values of
          the six high-order 16-bit pieces of the address, and the 'd's are
          the decimal values of the four low-order 8-bit pieces of the address
          (standard IPv4 representation).  Examples:

                0:0:0:0:0:0:13.1.68.3
                0:0:0:0:0:FFFF:129.144.52.38

          or in compressed form:

                ::13.1.68.3
                ::FFFF:129.144.52.38

DIAGNOSTICS
     The constant INADDR_NONE is returned by inet_addr() and inet_network()
     for malformed requests.

SEE ALSO
     byteorder(3), gethostbyname(3), getnetent(3), inet_net(3), hosts(5),
     networks(5)

STANDARDS
     The inet_ntop and inet_pton functions conforms to the IETF IPv6 BSD API
     and address formatting specifications.  Note that inet_pton does not
     accept 1-, 2-, or 3-part dotted addresses; all four parts must be speci-
     fied.  This is a narrower input set than that accepted by inet_aton.

HISTORY
     The inet_addr, inet_network, inet_makeaddr, inet_lnaof and inet_netof
     functions appeared in 4.2BSD.  The inet_aton and inet_ntoa functions
     appeared in 4.3BSD.  The inet_pton and inet_ntop functions appeared in
     BIND 4.9.4.

BUGS
     The value INADDR_NONE (0xffffffff) is a valid broadcast address, but
     inet_addr() cannot return that value without indicating failure.  Also,
     inet_addr() should have been designed to return a struct in_addr.  The
     newer inet_aton() function does not share these problems, and almost all
     existing code should be modified to use inet_aton() instead.

     The problem of host byte ordering versus network byte ordering is confus-
     ing.

     The string returned by inet_ntoa() resides in a static memory area.

BSD                              June 18, 1997                             BSD
```

inet6_option_space

```
INET6_OPTION_SPACE(3)    BSD Library Functions Manual    INET6_OPTION_SPACE(3)

NAME
     inet6_option_space, inet6_option_init, inet6_option_append,
     inet6_option_alloc, inet6_option_next, inet6_option_find -- IPv6 Hop-by-
     Hop and Destination Options manipulation

SYNOPSIS
     #include <netinet/in.h>

     int
     inet6_option_space(int nbytes);

     int
     inet6_option_init(void *bp, struct cmsghdr **cmsgp, int type);

     int
     inet6_option_append(struct cmsghdr *cmsg, const u_int8_t *typep,
         int multx, int plusy);
     u_int8_t *
     inet6_option_alloc(struct cmsghdr *cmsg, int datalen, int multx,
         int plusy);;

     int
     inet6_option_next(const struct cmsghdr *cmsg, u_int8_t **tptrp);

     int
     inet6_option_find(const struct cmsghdr *cmsg, u_int8_t **tptrp,
         int type);

DESCRIPTION
     Building and parsing the Hop-by-Hop and Destination options is compli-
     cated due to alignment constranints, padding and ancillary data manipula-
     tion.  RFC2292 defines a set of functions to help the application.  The
     function prototypes for these functions are all in the <netinet/in.h>
     header.

   inet6_option_space
     inet6_option_space() returns the number of bytes required to hold an
     option when it is stored as ancillary data, including the cmsghdr struc-
     ture at the beginning, and any padding at the end (to make its size a
     multiple of 8 bytes).  The argument is the size of the structure defining
     the option, which must include any pad bytes at the beginning (the value
     y in the alignment term ``xn + y''), the type byte, the length byte, and
     the option data.

     Note: If multiple options are stored in a single ancillary data object,
     which is the recommended technique, this function overestimates the
     amount of space required by the size of N-1 cmsghdr structures, where N
     is the number of options to be stored in the object.  This is of little
     consequence, since it is assumed that most Hop-by-Hop option headers and
     Destination option headers carry only one option (appendix B of
     [RFC-2460]).

   inet6_option_init
     inet6_option_init() is called once per ancillary data object that will
     contain either Hop-by-Hop or Destination options.  It returns 0 on suc-
     cess or -1 on an error.

     bp is a pointer to previously allocated space that will contain the
     ancillary data object.  It must be large enough to contain all the indi-
     vidual options to be added by later calls to inet6_option_append() and
     inet6_option_alloc().

     cmsgp is a pointer to a pointer to a cmsghdr structure.  *cmsgp is ini-
     tialized by this function to point to the cmsghdr structure constructed
     by this function in the buffer pointed to by bp.

     type is either IPV6_HOPOPTS or IPV6_DSTOPTS.  This type is stored in the
     cmsg_type member of the cmsghdr structure pointed to by *cmsgp.

   inet6_option_append
     This function appends a Hop-by-Hop option or a Destination option into an
     ancillary data object that has been initialized by inet6_option_init().
     This function returns 0 if it succeeds or -1 on an error.

     cmsg is a pointer to the cmsghdr structure that must have been initial-
     ized by inet6_option_init().

     typep is a pointer to the 8-bit option type.  It is assumed that this
     field is immediately followed by the 8-bit option data length field,
     which is then followed immediately by the option data.  The caller ini-
     tializes these three fields (the type-length-value, or TLV) before call-
     ing this function.
     The option type must have a value from 2 to 255, inclusive.  (0 and 1 are
     reserved for the Pad1 and PadN options, respectively.)

     The option data length must have a value between 0 and 255, inclusive,
     and is the length of the option data that follows.

     multx is the value x in the alignment term ``xn + y''.  It must have a
     value of 1, 2, 4, or 8.

     plusy is the value y in the alignment term ``xn + y''.  It must have a
     value between 0 and 7, inclusive.

   inet6_option_alloc
     This function appends a Hop-by-Hop option or a Destination option into an
     ancillary data object that has been initialized by inet6_option_init().
     This function returns a pointer to the 8-bit option type field that
     starts the option on success, or NULL on an error.

     The difference between this function and inet6_option_append() is that
     the latter copies the contents of a previously built option into the
     ancillary data object while the current function returns a pointer to the
     space in the data object where the option's TLV must then be built by the
     caller.

     cmsg is a pointer to the cmsghdr structure that must have been initial-
     ized by inet6_option_init().

     datalen is the value of the option data length byte for this option.
     This value is required as an argument to allow the function to determine
     if padding must be appended at the end of the option.  (The
     inet6_option_append() function does not need a data length argument since
     the option data length must already be stored by the caller.)

     multx is the value x in the alignment term ``xn + y''.  It must have a
     value of 1, 2, 4, or 8.

     plusy is the value y in the alignment term ``xn + y''.  It must have a
     value between 0 and 7, inclusive.

   inet6_option_next
     This function processes the next Hop-by-Hop option or Destination option
     in an ancillary data object.  If another option remains to be processed,
     the return value of the function is 0 and *tptrp points to the 8-bit
     option type field (which is followed by the 8-bit option data length,
     followed by the option data).  If no more options remain to be processed,
     the return value is -1 and *tptrp is NULL.  If an error occurs, the
     return value is -1 and *tptrp is not NULL.

     cmsg is a pointer to cmsghdr structure of which cmsg_level equals
     IPPROTO_IPV6 and cmsg_type equals either IPV6_HOPOPTS or IPV6_DSTOPTS.

     tptrp is a pointer to a pointer to an 8-bit byte and *tptrp is used by
     the function to remember its place in the ancillary data object each time
     the function is called.  The first time this function is called for a
     given ancillary data object, *tptrp must be set to NULL.

     Each time this function returns success, *tptrp points to the 8-bit
     option type field for the next option to be processed.

   inet6_option_find
     This function is similar to the previously described inet6_option_next()
     function, except this function lets the caller specify the option type to
     be searched for, instead of always returning the next option in the
     ancillary data object.  cmsg is a pointer to cmsghdr structure of which
     cmsg_level equals IPPROTO_IPV6 and cmsg_type equals either IPV6_HOPOPTS
     or IPV6_DSTOPTS.
     tptrp is a pointer to a pointer to an 8-bit byte and *tptrp is used by
     the function to remember its place in the ancillary data object each time
     the function is called.  The first time this function is called for a
     given ancillary data object, *tptrp must be set to NULL.  ~ This function
     starts searching for an option of the specified type beginning after the
     value of *tptrp.  If an option of the specified type is located, this
     function returns 0 and *tptrp points to the 8- bit option type field for
     the option of the specified type.  If an option of the specified type is
     not located, the return value is -1 and *tptrp is NULL.  If an error
     occurs, the return value is -1 and *tptrp is not NULL.

DIAGNOSTICS
     inet6_option_init() and inet6_option_append() return 0 on success or -1
     on an error.

     inet6_option_alloc() returns NULL on an error.

     On errors, inet6_option_next() and inet6_option_find() return -1 setting
     *tptrp to non NULL value.

EXAMPLES
     RFC2292 gives comprehensive examples in chapter 6.

SEE ALSO
     W. Stevens and M. Thomas, Advanced Sockets API for IPv6, RFC2292,
     February 1998.

     S. Deering and R. Hinden, Internet Protocol, Version 6 (IPv6)
     Specification, RFC2460, December 1998.

HISTORY
     The implementation first appeared in KAME advanced networking kit.

STANDARDS
     The functions are documented in ``Advanced Sockets API for IPv6''
     (RFC2292).

BUGS
     The text was shamelessly copied from RFC2292.

BSD                             December 10, 1999                          BSD
```

inet6_rthdr_space

```
INET6_RTHDR_SPACE(3)     BSD Library Functions Manual     INET6_RTHDR_SPACE(3)

NAME
     inet6_rthdr_space, inet6_rthdr_init, inet6_rthdr_add,
     inet6_rthdr_lasthop, inet6_rthdr_reverse, inet6_rthdr_segments,
     inet6_rthdr_getaddr, inet6_rthdr_getflags -- IPv6 Routing Header Options
     manipulation

SYNOPSIS
     #include <netinet/in.h>

     size_t
     inet6_rthdr_space(int type, int segments);

     struct cmsghdr *
     inet6_rthdr_init(void *bp, int type);

     int
     inet6_rthdr_add(struct cmsghdr *cmsg, const struct in6_addr *addr,
         unsigned int flags);

     int
     inet6_rthdr_lasthop(struct cmsghdr *cmsg, unsigned int flags);

     int
     inet6_rthdr_reverse(const struct cmsghdr *in, struct cmsghdr *out);

     int
     inet6_rthdr_segments(const struct cmsghdr *cmsg);

     struct in6_addr *
     inet6_rthdr_getaddr(struct cmsghdr *cmsg, int index);

     int
     inet6_rthdr_getflags(const struct cmsghdr *cmsg, int index);

DESCRIPTION
     RFC2292 IPv6 advanced API defines eight functions that the application
     calls to build and examine a Routing header.  Four functions build a
     Routing header:

     inet6_rthdr_space() return #bytes required for ancillary data

     inet6_rthdr_init() initialize ancillary data for Routing header

     inet6_rthdr_add() add IPv6 address & flags to Routing header

     inet6_rthdr_lasthop() specify the flags for the final hop

     Four functions deal with a returned Routing header:

     inet6_rthdr_reverse() reverse a Routing header

     inet6_rthdr_segments() return #segments in a Routing header

     inet6_rthdr_getaddr() fetch one address from a Routing header

     inet6_rthdr_getflags() fetch one flag from a Routing header

     The function prototypes for these functions are all in the <netinet/in.h>
     header.

   inet6_rthdr_space
     This function returns the number of bytes required to hold a Routing
     header of the specified type containing the specified number of segments
     (addresses).  For an IPv6 Type 0 Routing header, the number of segments
     must be between 1 and 23, inclusive.  The return value includes the size
     of the cmsghdr structure that precedes the Routing header, and any
     required padding.

     If the return value is 0, then either the type of the Routing header is
     not supported by this implementation or the number of segments is invalid
     for this type of Routing header.

     Note: This function returns the size but does not allocate the space
     required for the ancillary data.  This allows an application to allocate
     a larger buffer, if other ancillary data objects are desired, since all
     the ancillary data objects must be specified to sendmsg(2) as a single
     msg_control buffer.

   inet6_rthdr_init
     This function initializes the buffer pointed to by bp to contain a
     cmsghdr structure followed by a Routing header of the specified type.
     The cmsg_len member of the cmsghdr structure is initialized to the size
     of the structure plus the amount of space required by the Routing header.
     The cmsg_level and cmsg_type members are also initialized as required.

     The caller must allocate the buffer and its size can be determined by
     calling inet6_rthdr_space().
     Upon success the return value is the pointer to the cmsghdr structure,
     and this is then used as the first argument to the next two functions.
     Upon an error the return value is NULL.

   inet6_rthdr_add
     This function adds the address pointed to by addr to the end of the Rout-
     ing header being constructed and sets the type of this hop to the value
     of flags.  For an IPv6 Type 0 Routing header, flags must be either
     IPV6_RTHDR_LOOSE or IPV6_RTHDR_STRICT.

     If successful, the cmsg_len member of the cmsghdr structure is updated to
     account for the new address in the Routing header and the return value of
     the function is 0.  Upon an error the return value of the function is -1.

   inet6_rthdr_lasthop
     This function specifies the Strict/Loose flag for the final hop of a
     Routing header.  For an IPv6 Type 0 Routing header, flags must be either
     IPV6_RTHDR_LOOSE or IPV6_RTHDR_STRICT.

     The return value of the function is 0 upon success, or -1 upon an error.

     Notice that a Routing header specifying N intermediate nodes requires N+1
     Strict/Loose flags.  This requires N calls to inet6_rthdr_add() followed
     by one call to inet6_rthdr_lasthop().

   inet6_rthdr_reverse
     This function takes a Routing header that was received as ancillary data
     (pointed to by the first argument, in) and writes a new Routing header
     that sends datagrams along the reverse of that route.  Both arguments are
     allowed to point to the same buffer (that is, the reversal can occur in
     place).

     The return value of the function is 0 on success, or -1 upon an error.

   inet6_rthdr_segments
     This function returns the number of segments (addresses) contained in the
     Routing header described by cmsg.  On success the return value is between
     1 and 23, inclusive.  The return value of the function is -1 upon an
     error.

   inet6_rthdr_getaddr
     This function returns a pointer to the IPv6 address specified by index
     (which must have a value between 1 and the value returned by
     inet6_rthdr_segments()) in the Routing header described by cmsg.  An
     application should first call inet6_rthdr_segments() to obtain the number
     of segments in the Routing header.

     Upon an error the return value of the function is NULL.

   inet6_rthdr_getflags
     This function returns the flags value specified by index (which must have
     a value between 0 and the value returned by inet6_rthdr_segments()) in
     the Routing header described by cmsg.  For an IPv6 Type 0 Routing header
     the return value will be either IPV6_RTHDR_LOOSE or IPV6_RTHDR_STRICT.

     Upon an error the return value of the function is -1.

     Note: Addresses are indexed starting at 1, and flags starting at 0, to
     maintain consistency with the terminology and figures in RFC2460.

DIAGNOSTICS
     inet6_rthdr_space() returns 0 on errors.

     inet6_rthdr_add(), inet6_rthdr_lasthop() and inet6_rthdr_reverse() return
     0 on success, and returns -1 on error.
     inet6_rthdr_init() and inet6_rthdr_getaddr() return NULL on error.

     inet6_rthdr_segments() and inet6_rthdr_getflags() return -1 on error.

EXAMPLES
     RFC2292 gives comprehensive examples in chapter 8.

SEE ALSO
     W. Stevens and M. Thomas, Advanced Sockets API for IPv6, RFC2292,
     February 1998.

     S. Deering and R. Hinden, Internet Protocol, Version 6 (IPv6)
     Specification, RFC2460, December 1998.

HISTORY
     The implementation first appeared in KAME advanced networking kit.

STANDARDS
     The functions are documented in ``Advanced Sockets API for IPv6''
     (RFC2292).

BUGS
     The text was shamelessly copied from RFC2292.

     inet6_rthdr_reverse() is not implemented yet.

BSD                            December 10, 1999                           BSD
```

inet_net

```
INET_NET(3)              BSD Library Functions Manual              INET_NET(3)

NAME
     inet_net_ntop, inet_net_pton -- Internet network number manipulation rou-
     tines

SYNOPSIS
     #include <sys/socket.h>
     #include <netinet/in.h>
     #include <arpa/inet.h>

     char *
     inet_net_ntop(int af, const void *src, int bits, char *dst, size_t size);

     int
     inet_net_pton(int af, const char *src, void *dst, size_t size);

DESCRIPTION
     The inet_net_ntop() function converts an Internet network number from
     network format (usually a struct in_addr or some other binary form, in
     network byte order) to CIDR presentation format (suitable for external
     display purposes).  bits is the number of bits in src that are the net-
     work number.  It returns NULL if a system error occurs (in which case,
     errno will have been set), or it returns a pointer to the destination
     string.

     The inet_net_pton() function converts a presentation format Internet net-
     work number (that is, printable form as held in a character string) to
     network format (usually a struct in_addr or some other internal binary
     representation, in network byte order).  It returns the number of bits
     (either computed based on the class, or specified with /CIDR), or -1 if a
     failure occurred (in which case errno will have been set.  It will be set
     to ENOENT if the Internet network number was not valid).

     The only value for af currently supported is AF_INET.  size is the size
     of the result buffer dst.
```

```
NETWORK NUMBERS (IP VERSION 4)
     Internet network numbers may be specified in one of the following forms:

           a.b.c.d/bits
           a.b.c.d
           a.b.c
           a.b
           a

     When four parts are specified, each is interpreted as a byte of data and
     assigned, from left to right, to the four bytes of an Internet network
     number.  Note that when an Internet network number is viewed as a 32-bit
     integer quantity on a system that uses little-endian byte order (such as
     the Intel 386, 486, and Pentium processors) the bytes referred to above
     appear as ``d.c.b.a''.  That is, little-endian bytes are ordered from
     right to left.

     When a three part number is specified, the last part is interpreted as a
     16-bit quantity and placed in the rightmost two bytes of the Internet
     network number.  This makes the three part number format convenient for
     specifying Class B network numbers as ``128.net.host''.

     When a two part number is supplied, the last part is interpreted as a
     24-bit quantity and placed in the rightmost three bytes of the Internet
     network number.  This makes the two part number format convenient for
     specifying Class A network numbers as ``net.host''.

     When only one part is given, the value is stored directly in the Internet
     network number without any byte rearrangement.

     All numbers supplied as ``parts'' in a `.' notation may be decimal,
     octal, or hexadecimal, as specified in the C language (i.e., a leading 0x
     or 0X implies hexadecimal; otherwise, a leading 0 implies octal; other-
     wise, the number is interpreted as decimal).

SEE ALSO
     byteorder(3), inet(3), networks(5)

HISTORY
     The inet_net_ntop and inet_net_pton functions first appeared in BIND
     4.9.4.

BSD                             June 18, 1997                            BSD
```

ipx

```
IPX(3)                   BSD Library Functions Manual                   IPX(3)

NAME
     ipx_addr, ipx_ntoa -- IPX address conversion routines

SYNOPSIS
     #include <sys/types.h>
     #include <netipx/ipx.h>

     struct ipx_addr
     ipx_addr(const char *cp);

     char *
     ipx_ntoa(struct ipx_addr ipx);

DESCRIPTION
     The routine ipx_addr() interprets character strings representing IPX
     addresses, returning binary information suitable for use in system calls.
     The routine ipx_ntoa() takes IPX addresses and returns ASCII strings rep-
     resenting the address in a notation in common use:

           <network number>.<host number>.<port number>

     Trailing zero fields are suppressed, and each number is printed in hexa-
     decimal, in a format suitable for input to ipx_addr().  Any fields lack-
     ing super-decimal digits will have a trailing `H' appended.

     An effort has been made to ensure that ipx_addr() be compatible with most
     formats in common use.  It will first separate an address into 1 to 3
     fields using a single delimiter chosen from period (`.'), colon (`:'), or
     pound-sign (`#').  Each field is then examined for byte separators (colon
     or period).  If there are byte separators, each subfield separated is
     taken to be a small hexadecimal number, and the entirety is taken as a
     network-byte-ordered quantity to be zero extended in the high-network-
     order bytes.  Next, the field is inspected for hyphens, in which case the
     field is assumed to be a number in decimal notation with hyphens separat-
     ing the millenia.  Next, the field is assumed to be a number: It is
     interpreted as hexadecimal if there is a leading `0x' (as in C), a trail-
     ing `H' (as in Mesa), or there are any super-decimal digits present.  It
     is interpreted as octal is there is a leading `0' and there are no super-
     octal digits.  Otherwise, it is converted as a decimal number.

RETURN VALUES
     None.  (See BUGS.)

SEE ALSO
     ns(4), hosts(5), networks(5)

HISTORY
     The precursor ns_addr() and ns_ntoa() functions appeared in 4.3BSD.

BUGS
     The string returned by ipx_ntoa() resides in a static memory area.  The
     function ipx_addr() should diagnose improperly formed input, and there
     should be an unambiguous way to recognize this.

BSD                             June 4, 1993                             BSD
```

iso_addr

```
ISO_ADDR(3)              BSD Library Functions Manual              ISO_ADDR(3)

NAME
     iso_addr, iso_ntoa -- network address conversion routines for Open System
     Interconnection

SYNOPSIS
     #include <sys/types.h>
     #include <netiso/iso.h>

     struct iso_addr *
     iso_addr(char *cp);

     char *
     iso_ntoa(struct iso_addr *isoa);

DESCRIPTION
     The routine iso_addr() interprets character strings representing OSI
     addresses, returning binary information suitable for use in system calls.
     The routine iso_ntoa() takes OSI addresses and returns ASCII strings rep-
     resenting NSAPs (network service access points) in a notation inverse to
     that accepted by iso_addr().

     Unfortunately, no universal standard exists for representing OSI network
     addresses.
     The format employed by iso_addr() is a sequence of hexadecimal ``digits''
     (optionally separated by periods), of the form:

           <hex digits>.<hex digits>.<hex digits>

     Each pair of hexadecimal digits represents a byte with the leading digit
     indicating the higher-ordered bits.  A period following an even number of
     bytes has no effect (but may be used to increase legibility).  A period
     following an odd number of bytes has the effect of causing the byte of
     address being translated to have its higher order bits filled with zeros.

RETURN VALUES
     iso_ntoa() always returns a null terminated string.  iso_addr() always
     returns a pointer to a struct iso_addr.  (See BUGS.)

SEE ALSO
     iso(4)

HISTORY
     The iso_addr() and iso_ntoa() functions appeared in 4.3BSD-Reno.

BUGS
     The returned values reside in a static memory area.

     The function iso_addr() should diagnose improperly formed input, and
     there should be an unambiguous way to recognize this.

BSD                             June 4, 1993                             BSD
```

link_addr

```
LINK_ADDR(3)             BSD Library Functions Manual             LINK_ADDR(3)

NAME
     link_addr, link_ntoa -- elementary address specification routines for
     link level access

SYNOPSIS
     #include <sys/types.h>
     #include <sys/socket.h>
     #include <net/if_dl.h>

     void
     link_addr(const char *addr, struct sockaddr_dl *sdl);

     char *
     link_ntoa(const struct sockaddr_dl *sdl);

DESCRIPTION
     The link_addr() function interprets character strings representing link-
     level addresses, returning binary information suitable for use in system
     calls.  link_ntoa() takes a link-level address and returns an ASCII
     string representing some of the information present, including the link
     level address itself, and the interface name or number, if present.  This
     facility is experimental and is still subject to change.

     For link_addr(), the string addr may contain an optional network inter-
     face identifier of the form ``name unit-number'', suitable for the first
     argument to ifconfig(8), followed in all cases by a colon and an inter-
     face address in the form of groups of hexadecimal digits separated by
     periods.  Each group represents a byte of address; address bytes are
     filled left to right from low order bytes through high order bytes.

     Thus le0:8.0.9.13.d.30 represents an Ethernet address to be transmitted
     on the first Lance Ethernet interface.
```

```
RETURN VALUES
     link_ntoa() always returns a null-terminated string.  link_addr() has no
     return value.  (See BUGS.)

SEE ALSO
     iso(4), ifconfig(8)

HISTORY
     The link_addr() and link_ntoa() functions appeared in 4.3BSD-Reno.

BUGS
     The returned values for link_ntoa reside in a static memory area.

     The function link_addr() should diagnose improperly formed input, and
     there should be an unambiguous way to recognize this.

     If the sdl_len field of the link socket address sdl is 0, link_ntoa()
     will not insert a colon before the interface address bytes.  If this
     translated address is given to link_addr() without inserting an initial
     colon, the latter will not interpret it correctly.

BSD                              July 28, 1993                             BSD
```

net_addrcmp

```
NET_ADDRCMP(3)           BSD Library Functions Manual           NET_ADDRCMP(3)

NAME
     net_addrcmp -- compare socket address structures

SYNOPSIS
     #include <netdb.h>

     int
     net_addrcmp(struct sockaddr *sa1, struct sockaddr *sa2);

DESCRIPTION
     The net_addrcmp() function compares two socket address structures, sa1
     and sa2.

RETURN VALUES
     If sa1 and sa2 are for the same address, net_addrcmp() returns 0.

     The sa_len fields are compared first.  If they do not match,
     net_addrcmp() returns -1 or 1 if sa1->sa_len is less than or greater than
     sa2->sa_len, respectively.

     Next, the sa_family members are compared.  If they do not match,
     net_addrcmp() returns -1 or 1 if sa1->sa_family is less than or greater
     than sa2->sa_family, respectively.

     Lastly, if each socket address structure's sa_len and sa_family fields
     match, the protocol-specific data (the sa_data field) is compared.  If
     there's a match, both sa1 and sa2 must refer to the same address, and 0
     is returned; otherwise, a value >0 or <0 is returned.

HISTORY
     A net_addrcmp() function was added in OpenBSD 2.5.

BSD                              July 3, 1999                              BSD
```

ns

```
NS(3)                    BSD Library Functions Manual                    NS(3)
```

```
NAME
     ns_addr, ns_ntoa -- Xerox NS(tm) address conversion routines

SYNOPSIS
     #include <sys/types.h>
     #include <netns/ns.h>

     struct ns_addr
     ns_addr(char *cp);

     char *
     ns_ntoa(struct ns_addr ns);

DESCRIPTION
     The routine ns_addr() interprets character strings representing XNS
     addresses, returning binary information suitable for use in system calls.
     The routine ns_ntoa() takes XNS addresses and returns ASCII strings rep-
     resenting the address in a notation in common use in the Xerox Develop-
     ment Environment:

           <network number>.<host number>.<port number>

     Trailing zero fields are suppressed, and each number is printed in hexa-
     decimal, in a format suitable for input to ns_addr().  Any fields lacking
     super-decimal digits will have a trailing `H' appended.

     Unfortunately, no universal standard exists for representing XNS
     addresses.  An effort has been made to ensure that ns_addr() be compati-
     ble with most formats in common use.  It will first separate an address
     into 1 to 3 fields using a single delimiter chosen from period (`.'),
     colon (`:'), or pound-sign `#'.  Each field is then examined for byte
     separators (colon or period).  If there are byte separators, each sub-
     field separated is taken to be a small hexadecimal number, and the
     entirety is taken as a network-byte-ordered quantity to be zero extended
     in the high-network-order bytes.  Next, the field is inspected for
     hyphens, in which case the field is assumed to be a number in decimal
     notation with hyphens separating the millenia.  Next, the field is
     assumed to be a number: It is interpreted as hexadecimal if there is a
     leading `0x' (as in C), a trailing `H' (as in Mesa), or there are any
     super-decimal digits present.  It is interpreted as octal is there is a
     leading `0' and there are no super-octal digits.  Otherwise, it is con-
     verted as a decimal number.

RETURN VALUES
     None.  (See BUGS.)

SEE ALSO
     hosts(5), networks(5)

HISTORY
     The ns_addr() and ns_toa() functions appeared in 4.3BSD.

BUGS
     The string returned by ns_ntoa() resides in a static memory area.  The
     function ns_addr() should diagnose improperly formed input, and there
     should be an unambiguous way to recognize this.

BSD                             June 4, 1993                             BSD
```

resolver

```
RESOLVER(3)              BSD Library Functions Manual              RESOLVER(3)

NAME
     res_query, res_search, res_mkquery, res_send, res_init, dn_comp,
     dn_expand -- resolver routines

SYNOPSIS
     #include <sys/types.h>
     #include <netinet/in.h>
     #include <arpa/nameser.h>
     #include <resolv.h>

     int
     res_query(char *dname, int class, int type, u_char *answer, int anslen);

     int
     res_search(char *dname, int class, int type, u_char *answer, int anslen);

     int
     res_mkquery(int op, char *dname, int class, int type, char *data,
         int datalen, struct rrec *newrr, char *buf, int buflen);

     int
     res_send(char *msg, int msglen, char *answer, int anslen);

     int
     res_init(void);

     int
     dn_comp(char *exp_dn, char *comp_dn, int length, char **dnptrs,
         char **lastdnptr);

     int
     dn_expand(u_char *msg, u_char *eomorig, u_char *comp_dn, u_char *exp_dn,
         int length);

DESCRIPTION
     These routines are used for making, sending, and interpreting query and
     reply messages with Internet domain name servers.

     Global configuration and state information that is used by the resolver
     routines is kept in the structure _res.  Most of the values have reason-
     able defaults and can be ignored.  Options stored in _res.options are
     defined in <resolv.h> and are as follows.  Options are stored as a simple
     bit mask containing the bitwise OR of the options enabled.

     RES_INIT       True if the initial name server address and default domain
                    name are initialized (i.e., res_init() has been called).

     RES_DEBUG      Print debugging messages.

     RES_AAONLY     Accept authoritative answers only.  With this option,
                    res_send() should continue until it finds an authoritative
                    answer or finds an error.  Currently this is not imple-
                    mented.

     RES_USEVC      Use TCP connections for queries instead of UDP datagrams.

     RES_STAYOPEN   Used with RES_USEVC to keep the TCP connection open
                    between queries.  This is useful only in programs that
                    regularly do many queries.  UDP should be the normal mode
                    used.

     RES_IGNTC      Unused currently (ignore truncation errors, i.e., don't
                    retry with TCP).

     RES_RECURSE    Set the recursion-desired bit in queries.  This is the
                    default.  (res_send() does not do iterative queries and
                    expects the name server to handle recursion.)

     RES_DEFNAMES   If set, res_search() will append the default domain name
```

```
                    to single-component names (those that do not contain a
                    dot).  This option is enabled by default.

     RES_DNSRCH     If this option is set, res_search() will search for host
                    names in the current domain and in parent domains; see
                    hostname(7).  This is used by the standard host lookup
                    routine gethostbyname(3).  This option is enabled by
                    default.

     RES_USE_INET6  Enables support for IPv6-only applications.  This causes
                    IPv4 addresses to be returned as an IPv4 mapped address.
                    For example, 10.1.1.1 will be returned as ::ffff:10.1.1.1.
                    The option is not meaningful on OpenBSD.

     The res_init() routine reads the configuration file (if any; see
     resolv.conf(5)) to get the default domain name, search list, and the
     Internet address of the local name server(s).  If no server is config-
     ured, the host running the resolver is tried.  The current domain name is
     defined by the hostname if not specified in the configuration file; it
     can be overridden by the environment variable LOCALDOMAIN.  This environ-
     ment variable may contain several blank-separated tokens if you wish to
     override the search list on a per-process basis.  This is similar to the
     search command in the configuration file.  Another environment variable
     RES_OPTIONS can be set to override certain internal resolver options
     which are otherwise set by changing fields in the _res structure or are
     inherited from the configuration file's options command.  The syntax of
     the RES_OPTIONS environment variable is explained in resolv.conf(5).
     Initialization normally occurs on the first call to one of the following
     routines.

     The res_query() function provides an interface to the server query mecha-
     nism.  It constructs a query, sends it to the local server, awaits a
     response, and makes preliminary checks on the reply.  The query requests
     information of the specified type and class for the specified fully qual-
     ified domain name dname.  The reply message is left in the answer buffer
     with length anslen supplied by the caller.

     The res_search() routine makes a query and awaits a response like
     res_query(), but in addition, it implements the default and search rules
     controlled by the RES_DEFNAMES and RES_DNSRCH options.  It returns the
     first successful reply.

     The remaining routines are lower-level routines used by res_query().  The
     res_mkquery() function constructs a standard query message and places it
     in buf.  It returns the size of the query, or -1 if the query is larger
     than buflen.  The query type op is usually QUERY, but can be any of the
     query types defined in <arpa/nameser.h>.  The domain name for the query
     is given by dname.  newrr is currently unused but is intended for making
     update messages.

     The res_send() routine sends a pre-formatted query and returns an answer.
     It will call res_init() if RES_INIT is not set, send the query to the
     local name server, and handle timeouts and retries.  The length of the
     reply message is returned, or -1 if there were errors.

     The dn_comp() function compresses the domain name exp_dn and stores it in
     comp_dn.  The size of the compressed name is returned or -1 if there were
     errors.  The size of the array pointed to by comp_dn is given by length.
     The compression uses an array of pointers dnptrs to previously compressed
     names in the current message.  The first pointer points to the beginning
     of the message and the list ends with NULL.  The limit to the array is
     specified by lastdnptr.  A side effect of dn_comp() is to update the list
     of pointers for labels inserted into the message as the name is com-
     pressed.  If dnptr is NULL, names are not compressed.  If lastdnptr is
     NULL, the list of labels is not updated.

     The dn_expand() entry expands the compressed domain name comp_dn to a
     full domain name The compressed name is contained in a query or reply
     message; msg is a pointer to the beginning of the message.  The uncom-
     pressed name is placed in the buffer indicated by exp_dn which is of size
     length.  The size of compressed name is returned or -1 if there was an
     error.

FILES
     /etc/resolv.conf configuration file see resolv.conf(5).

SEE ALSO
     gethostbyname(3), resolv.conf(5), hostname(7), named(8)

     RFC1032, RFC1033, RFC1034, RFC1035, RFC1535, RFC974

     Name Server Operations Guide for BIND.

HISTORY
     The res_query function appeared in 4.3BSD.

BSD                              June 4, 1993                              BSD
```

accept

```
ACCEPT(2)                   BSD System Calls Manual                  ACCEPT(2)

NAME
     accept -- accept a connection on a socket

SYNOPSIS
     #include <sys/types.h>
     #include <sys/socket.h>

     int
     accept(int s, struct sockaddr *addr, socklen_t *addrlen);

DESCRIPTION
     The argument s is a socket that has been created with socket(2), bound to
     an address with bind(2), and is listening for connections after a
     listen(2).  The accept() argument extracts the first connection request
     on the queue of pending connections, creates a new socket with the same
     properties of s, and allocates a new file descriptor for the socket.  If
     no pending connections are present on the queue, and the socket is not
     marked as non-blocking, accept() blocks the caller until a connection is
     present.  If the socket is marked non-blocking and no pending connections
     are present on the queue, accept() returns an error as described below.
     The accepted socket may not be used to accept more connections.  The
     original socket s remains open.

     The argument addr is a result parameter that is filled in with the
     address of the connecting entity as known to the communications layer.
     The exact format of the addr parameter is determined by the domain in
     which the communication is occurring.  The addrlen is a value-result
     parameter; it should initially contain the amount of space pointed to by
     addr; on return it will contain the actual length (in bytes) of the
     address returned.  This call is used with connection-based socket types,
     currently with SOCK_STREAM.

     It is possible to select(2) or poll(2) a socket for the purposes of doing
     an accept() by selecting it for read.

     For certain protocols which require an explicit confirmation, such as ISO
     or DATAKIT, accept() can be thought of as merely dequeuing the next con-
     nection request and not implying confirmation.  Confirmation can be
     implied by a normal read or write on the new file descriptor, and rejec-
     tion can be implied by closing the new socket.
     One can obtain user connection request data without confirming the con-
     nection by issuing a recvmsg(2) call with an msg_iovlen of 0 and a non-
     zero msg_controllen, or by issuing a getsockopt(2) request.  Similarly,
     one can provide user connection rejection information by issuing a
     sendmsg(2) call with providing only the control information, or by call-
     ing setsockopt(2).

RETURN VALUES
     The call returns -1 on error.  If it succeeds, it returns a non-negative
     integer that is a descriptor for the accepted socket.

ERRORS
     The accept() will fail if:

     [EBADF]            The descriptor is invalid.

     [ENOTSOCK]         The descriptor references a file, not a socket.

     [EOPNOTSUPP]       The referenced socket is not of type SOCK_STREAM.

     [EINVAL]           The referenced socket is not listening for connections
                        (that is, listen(2) has not yet been called).

     [EFAULT]           The addr parameter is not in a writable part of the
                        user address space.

     [EWOULDBLOCK]      The socket is marked non-blocking and no connections
                        are present to be accepted.

     [EMFILE]           The per-process descriptor table is full.

     [ENFILE]           The system file table is full.

     [ECONNABORTED]     A connection has been aborted.

SEE ALSO
     bind(2), connect(2), listen(2), poll(2), select(2), poll(2), socket(2)

HISTORY
     The accept() function appeared in 4.2BSD.

BSD                            February 15, 1999                           BSD
```

bind

```
BIND(2)                     BSD System Calls Manual                    BIND(2)

NAME
     bind -- bind a name to a socket

SYNOPSIS
     #include <sys/types.h>
     #include <sys/socket.h>

     int
     bind(int s, const struct sockaddr *name, socklen_t namelen);

DESCRIPTION
     bind() assigns a name to an unnamed socket.  When a socket is created
     with socket(2) it exists in a name space (address family) but has no name
     assigned.  bind() requests that name be assigned to the socket.

NOTES
     Binding a name in the UNIX domain creates a socket in the file system
     that must be deleted by the caller when it is no longer needed (using
     unlink(2)).
     The rules used in name binding vary between communication domains.  Con-
     sult the manual entries in section 4 for detailed information.

RETURN VALUES
     If the bind is successful, a 0 value is returned.  A return value of -1
     indicates an error, which is further specified in the global errno.

ERRORS
     The bind() call will fail if:

     [EBADF]            S is not a valid descriptor.

     [ENOTSOCK]         S is not a socket.

     [EADDRNOTAVAIL]    The specified address is not available from the local
                        machine.

     [EADDRINUSE]       The specified address is already in use.

     [EINVAL]           The socket is already bound to an address.

     [EINVAL]           The family of the socket and that requested in
                        name->sa_family are not equivalent.

     [EACCES]           The requested address is protected, and the current
                        user has inadequate permission to access it.

     [EFAULT]           The name parameter is not in a valid part of the user
                        address space.

     The following errors are specific to binding names in the UNIX domain.

     [ENOTDIR]          A component of the path prefix is not a directory.

     [ENAMETOOLONG]     A component of a pathname exceeded {NAME_MAX} charac-
                        ters, or an entire path name exceeded {PATH_MAX} char-
                        acters.

     [ENOENT]           A prefix component of the path name does not exist.

     [ELOOP]            Too many symbolic links were encountered in translat-
                        ing the pathname.

     [EIO]              An I/O error occurred while making the directory entry
                        or allocating the inode.

     [EROFS]            The name would reside on a read-only file system.

     [EISDIR]           An empty pathname was specified.

SEE ALSO
     connect(2), getsockname(2), listen(2), socket(2)

HISTORY
     The bind() function call appeared in 4.2BSD.

BSD                             February 15, 1999                            BSD
```

connect

```
CONNECT(2)                   BSD System Calls Manual                  CONNECT(2)

NAME
     connect -- initiate a connection on a socket
```

```
SYNOPSIS
     #include <sys/types.h>
     #include <sys/socket.h>

     int
     connect(int s, const struct sockaddr *name, socklen_t namelen);

DESCRIPTION
     The parameter s is a socket.  If it is of type SOCK_DGRAM, this call
     specifies the peer with which the socket is to be associated; this
     address is that to which datagrams are to be sent, and the only address
     from which datagrams are to be received.  If the socket is of type
     SOCK_STREAM, this call attempts to make a connection to another socket.
     The other socket is specified by name, which is an address in the commu-
     nications space of the socket.  Each communications space interprets the
     name parameter in its own way.  Generally, stream sockets may success-
     fully connect() only once; datagram sockets may use connect() multiple
     times to change their association.  Datagram sockets may dissolve the
     association by connecting to an invalid address, such as a null address.

RETURN VALUES
     If the connection or binding succeeds, 0 is returned.  Otherwise a -1 is
     returned, and a more specific error code is stored in errno.

ERRORS
     The connect() call fails if:

     [EBADF]            S is not a valid descriptor.

     [ENOTSOCK]         S is a descriptor for a file, not a socket.

     [EADDRNOTAVAIL]    The specified address is not available on this
                        machine.

     [EAFNOSUPPORT]     Addresses in the specified address family cannot be
                        used with this socket.

     [EISCONN]          The socket is already connected.

     [ETIMEDOUT]        Connection establishment timed out without establish-
                        ing a connection.

     [EINVAL]           A TCP connection with a local broadcast, the all-ones
                        or a multicast address as the peer was attempted.

     [ECONNREFUSED]     The attempt to connect was forcefully rejected.

     [EINTR]            A connect was interrupted before it succeeded by the
                        delivery of a signal.

     [ENETUNREACH]      The network isn't reachable from this host.

     [EADDRINUSE]       The address is already in use.

     [EFAULT]           The name parameter specifies an area outside the
                        process address space.

     [EINPROGRESS]      The socket is non-blocking and the connection cannot
                        be completed immediately.  It is possible to select(2)
                        or poll(2) for completion by selecting the socket for
                        writing, and also use getsockopt(2) with SO_ERROR to
                        check for error conditions.

     [EALREADY]         The socket is non-blocking and a previous connection
                        attempt has not yet been completed.

     The following errors are specific to connecting names in the UNIX domain.
     These errors may not apply in future versions of the UNIX IPC domain.

     [ENOTDIR]          A component of the path prefix is not a directory.

     [ENAMETOOLONG]     A component of a pathname exceeded {NAME_MAX} charac-
                        ters, or an entire path name exceeded {PATH_MAX} char-
                        acters.

     [ENOENT]           The named socket does not exist.

     [EACCES]           Search permission is denied for a component of the
                        path prefix.

     [EACCES]           Write access to the named socket is denied.

     [ELOOP]            Too many symbolic links were encountered in translat-
                        ing the pathname.

SEE ALSO
     accept(2), getsockname(2), getsockopt(2), poll(2), select(2), socket(2)

HISTORY
     The connect() function call appeared in 4.2BSD.

BSD                            February 15, 1999                           BSD
```

getpeername

```
GETPEERNAME(2)              BSD System Calls Manual             GETPEERNAME(2)

NAME
     getpeername -- get name of connected peer

SYNOPSIS
     #include <sys/types.h>
     #include <sys/socket.h>

     int
     getpeername(int s, struct sockaddr *name, socklen_t *namelen);

DESCRIPTION
     getpeername() returns the address information of the peer connected to
     socket s.  One common use occurs when a process inherits an open socket,
     such as TCP servers forked from inetd(8).  In this scenario,
     getpeername() is used to determine the connecting client's IP address.

     getpeername() takes three parameters:

     s Contains the file descriptor of the socket whose peer should be looked
     up.

     name Points to a sockaddr structure that will hold the address informa-
     tion for the connected peer.  Normal use requires one to use a structure
     specific to the protocol family in use, such as sockaddr_in (IPv4) or
     sockaddr_in6 (IPv6), cast to a (struct sockaddr *).

     For greater portability, especially with the newer protocol families, the
     new struct sockaddr_storage should be used.  sockaddr_storage is large
     enough to hold any of the other sockaddr_* variants.  On return, it can
     be cast to the correct sockaddr type, based the protocol family contained
     in its ss_family field.

     namelen Indicates the amount of space pointed to by name, in bytes.

     If address information for the local end of the socket is required, the
     getsockname(2) function should be used instead.
     If name does not point to enough space to hold the entire socket address,
     the result will be truncated to namelen bytes.

RETURN VALUES
     If the call succeeds, a 0 is returned and namelen is set to the actual
     size of the socket address returned in name.  Otherwise, errno is set and
     a value of -1 is returned.

ERRORS
     On failure, errno is set to one of the following:

     [EBADF]            The argument s is not a valid descriptor.

     [ENOTSOCK]         The argument s is a file, not a socket.

     [ENOTCONN]         The socket is not connected.

     [ENOBUFS]          Insufficient resources were available in the system to
                        perform the operation.

     [EFAULT]           The name parameter points to memory not in a valid
                        part of the process address space.

SEE ALSO
     accept(2), bind(2), getsockname(2), getpeereid(2), socket(2)

HISTORY
     The getpeername() function call appeared in 4.2BSD.

BSD                             July 17, 1999                            BSD
```

getsockname

```
GETSOCKNAME(2)             BSD System Calls Manual            GETSOCKNAME(2)

NAME
     getsockname -- get socket name

SYNOPSIS
     #include <sys/types.h>
     #include <sys/socket.h>

     int
     getsockname(int s, struct sockaddr *name, socklen_t *namelen);

DESCRIPTION
     getsockname() returns the locally bound address information for a speci-
     fied socket.

     Common uses of this function are as follows:

     +   When bind(2) is called with a port number of 0 (indicating the kernel
         should pick an ephemeral port) getsockname() is used to retrieve the
         kernel-assigned port number.

     +   When a process calls bind(2) on a wildcard IP address, getsockname()
         is used to retrieve the local IP address for the connection.

     +   When a function wishes to know the address family of a socket,
         getsockname() can be used.

     getsockname() takes three parameters:

     s, Contains the file desriptor for the socket to be looked up.
     name points to a sockaddr structure which will hold the resulting address
     information.  Normal use requires one to use a structure specific to the
     protocol family in use, such as sockaddr_in (IPv4) or sockaddr_in6
     (IPv6), cast to a (struct sockaddr *).

     For greater portability (such as newer protocol families) the new struc-
     ture sockaddr_storage exists.  sockaddr_storage is large enough to hold
     any of the other sockaddr_* variants.  On return, it should be cast to
     the correct sockaddr type, according to the current protocol family.

     namelen Indicates the amount of space pointed to by name, in bytes.  Upon
     return, namelen is set to the actual size of the returned address infor-
     mation.

     If the address of the destination socket for a given socket connection is
     needed, the getpeername(2) function should be used instead.

     If name does not point to enough space to hold the entire socket address,
     the result will be truncated to namelen bytes.

RETURN VALUES
     On success, getsockname() returns a 0, and namelen is set to the actual
     size of the socket address returned in name.  Otherwise, errno is set,
     and a value of -1 is returned.

ERRORS
     If getsockname() fails, errno is set to one of the following:

     [EBADF]            The argument s is not a valid descriptor.

     [ENOTSOCK]         The argument s is a file, not a socket.

     [ENOBUFS]          Insufficient resources were available in the system to
                        perform the operation.

     [EFAULT]           The name parameter points to memory not in a valid
                        part of the process address space.

SEE ALSO
     accept(2), bind(2), getpeername(2), getpeereid(2), socket(2)

BUGS
     Names bound to sockets in the UNIX domain are inaccessible; getsockname
     returns a zero length name.

HISTORY
     The getsockname() function call appeared in 4.2BSD.

BSD                             July 17, 1999                            BSD
```

getsockopt

```
GETSOCKOPT(2)               BSD System Calls Manual              GETSOCKOPT(2)

NAME
     getsockopt, setsockopt -- get and set options on sockets

SYNOPSIS
     #include <sys/types.h>
     #include <sys/socket.h>

     int
     getsockopt(int s, int level, int optname, void *optval,
         socklen_t *optlen);

     int
     setsockopt(int s, int level, int optname, const void *optval,
         socklen_t optlen);

DESCRIPTION
     getsockopt() and setsockopt() manipulate the options associated with a
     socket.  Options may exist at multiple protocol levels; they are always
     present at the uppermost ``socket'' level.

     When manipulating socket options the level at which the option resides
     and the name of the option must be specified.  To manipulate options at
     the socket level, level is specified as SOL_SOCKET.  To manipulate
     options at any other level the protocol number of the appropriate proto-
     col controlling the option is supplied.  For example, to indicate that an
     option is to be interpreted by the TCP protocol, level should be set to
     the protocol number of TCP; see getprotoent(3).

     The parameters optval and optlen are used to access option values for
     setsockopt().  For getsockopt() they identify a buffer in which the value
     for the requested option(s) are to be returned.  For getsockopt(), optlen
     is a value-result parameter, initially containing the size of the buffer
     pointed to by optval, and modified on return to indicate the actual size
     of the value returned.  If no option value is to be supplied or returned,
     optval may be NULL.

     optname and any specified options are passed uninterpreted to the appro-
     priate protocol module for interpretation.  The include file
     <sys/socket.h> contains definitions for socket level options, described
     below.  Options at other protocol levels vary in format and name; consult
     the appropriate entries in section 4 of the manual.

     Most socket-level options utilize an int parameter for optval.  For
     setsockopt(), the parameter should be non-zero to enable a boolean
     option, or zero if the option is to be disabled.  SO_LINGER uses a struct
     linger parameter, defined in <sys/socket.h>, which specifies the desired
     state of the option and the linger interval (see below).  SO_SNDTIMEO and
     SO_RCVTIMEO use a struct timeval parameter, defined in <sys/time.h>.

     The following options are recognized at the socket level.  Except as
     noted, each may be examined with getsockopt() and set with setsockopt().

           SO_DEBUG        enables recording of debugging information
           SO_REUSEADDR    enables local address reuse
           SO_REUSEPORT    enables duplicate address and port bindings
           SO_KEEPALIVE    enables keep connections alive
           SO_DONTROUTE    enables routing bypass for outgoing messages
           SO_LINGER       linger on close if data present
           SO_BROADCAST    enables permission to transmit broadcast messages
           SO_OOBINLINE    enables reception of out-of-band data in band
           SO_SNDBUF       set buffer size for output
           SO_RCVBUF       set buffer size for input
           SO_SNDLOWAT     set minimum count for output
           SO_RCVLOWAT     set minimum count for input
           SO_SNDTIMEO     set timeout value for output
           SO_RCVTIMEO     set timeout value for input
           SO_TYPE         get the type of the socket (get only)
           SO_ERROR        get and clear error on the socket (get only)

     SO_DEBUG enables debugging in the underlying protocol modules.
     SO_REUSEADDR indicates that the rules used in validating addresses sup-
     plied in a bind(2) call should allow reuse of local addresses.
     SO_REUSEPORT allows completely duplicate bindings by multiple processes
     if they all set SO_REUSEPORT before binding the port.  This option per-
     mits multiple instances of a program to each receive UDP/IP multicast or
     broadcast datagrams destined for the bound port.  SO_KEEPALIVE enables
     the periodic transmission of messages on a connected socket.  Should the
     connected party fail to respond to these messages, the connection is con-
     sidered broken and processes using the socket are notified via a SIGPIPE
     signal when attempting to send data.  SO_DONTROUTE indicates that outgo-
     ing messages should bypass the standard routing facilities.  Instead,
     messages are directed to the appropriate network interface according to
     the network portion of the destination address.

     SO_LINGER controls the action taken when unsent messages are queued on
     socket and a close(2) is performed.  If the socket promises reliable
     delivery of data and SO_LINGER is set, the system will block the process
     on the close(2) attempt until it is able to transmit the data or until it
     decides it is unable to deliver the information (a timeout period mea-
     sured in seconds, termed the linger interval, is specified in the
     setsockopt() call when SO_LINGER is requested).  If SO_LINGER is disabled
     and a close(2) is issued, the system will process the close in a manner
     that allows the process to continue as quickly as possible.

     The option SO_BROADCAST requests permission to send broadcast datagrams
     on the socket.  Broadcast was a privileged operation in earlier versions
     of the system.  With protocols that support out-of-band data, the
     SO_OOBINLINE option requests that out-of-band data be placed in the nor-
     mal data input queue as received; it will then be accessible with recv(2)
     or read(2) calls without the MSG_OOB flag.  Some protocols always behave
     as if this option is set.  SO_SNDBUF and SO_RCVBUF are options to adjust
     the normal buffer sizes allocated for output and input buffers, respec-
     tively.  The buffer size may be increased for high-volume connections, or
     may be decreased to limit the possible backlog of incoming data.  The
     system places an absolute limit on these values.

     SO_SNDLOWAT is an option to set the minimum count for output operations.
     Most output operations process all of the data supplied by the call,
     delivering data to the protocol for transmission and blocking as neces-
     sary for flow control.  Nonblocking output operations will process as
     much data as permitted subject to flow control without blocking, but will
     process no data if flow control does not allow the smaller of the low
     water mark value or the entire request to be processed.  A select(2) or
     poll(2) operation testing the ability to write to a socket will return
     true only if the low water mark amount could be processed.  The default
     value for SO_SNDLOWAT is set to a convenient size for network efficiency,
     often 1024.  SO_RCVLOWAT is an option to set the minimum count for input
     operations.  In general, receive calls will block until any (non-zero)
     amount of data is received, then return with the smaller of the amount
     available or the amount requested.  The default value for SO_RCVLOWAT is
     1.  If SO_RCVLOWAT is set to a larger value, blocking receive calls nor-
     mally wait until they have received the smaller of the low water mark
     value or the requested amount.  Receive calls may still return less than
     the low water mark if an error occurs, a signal is caught, or the type of
     data next in the receive queue is different than that returned.

     SO_SNDTIMEO is an option to set a timeout value for output operations.
     It accepts a struct timeval parameter with the number of seconds and
     microseconds used to limit waits for output operations to complete.  If a
     send operation has blocked for this much time, it returns with a partial
     count or with the error EWOULDBLOCK if no data was sent.  In the current
     implementation, this timer is restarted each time additional data are
     delivered to the protocol, implying that the limit applies to output por-
     tions ranging in size from the low water mark to the high water mark for
     output.  SO_RCVTIMEO is an option to set a timeout value for input opera-
     tions.  It accepts a struct timeval parameter with the number of seconds
     and microseconds used to limit waits for input operations to complete.
     In the current implementation, this timer is restarted each time addi-
     tional data are received by the protocol, and thus the limit is in effect
     an inactivity timer.  If a receive operation has been blocked for this
     much time without receiving additional data, it returns with a short
     count or with the error EWOULDBLOCK if no data were received.

     Finally, SO_TYPE and SO_ERROR are options used only with getsockopt().
     SO_TYPE returns the type of the socket, such as SOCK_STREAM; it is useful
     for servers that inherit sockets on startup.  SO_ERROR returns any pend-
     ing error on the socket and clears the error status.  It may be used to
     check for asynchronous errors on connected datagram sockets or for other
     asynchronous errors.

RETURN VALUES
     A 0 is returned if the call succeeds, -1 if it fails.

ERRORS
     The call succeeds unless:

     [EBADF]            The argument s is not a valid descriptor.

     [ENOTSOCK]         The argument s is a file, not a socket.

     [ENOPROTOOPT]      The option is unknown at the level indicated.

     [EFAULT]           The address pointed to by optval is not in a valid
                        part of the process address space.  For getsockopt(),
                        this error may also be returned if optlen is not in a
                        valid part of the process address space.

SEE ALSO
     connect(2), ioctl(2), poll(2), select(2), poll(2), socket(2),
     getprotoent(3), protocols(5)

BUGS
     Several of the socket options should be handled at lower levels of the
     system.

HISTORY
     The getsockopt() system call appeared in 4.2BSD.

BSD                             February 15, 1999                            BSD
```

ioctl

```
IOCTL(2)                   BSD System Calls Manual                   IOCTL(2)

NAME
     ioctl -- control device

SYNOPSIS
     #include <sys/ioctl.h>

     int
     ioctl(int d, unsigned long request, …);

DESCRIPTION
     The ioctl() function manipulates the underlying device parameters of spe-
     cial files.  In particular, many operating characteristics of character
     special files (e.g., terminals) may be controlled with ioctl() requests.

     The argument d must be an open file descriptor. The third argument is
     called arg and contains additional information needed by this device to
     perform the requested function.  arg is either an int or a pointer to a
     device-specific data structure, depending upon the given request.

     An ioctl request has encoded in it whether the argument is an ``in''
     parameter or ``out'' parameter, and the size of the third argument (arg)
     in bytes.  Macros and defines used in specifying an ioctl request are
     located in the file <sys/ioctl.h>.

RETURN VALUES
     If an error has occurred, a value of -1 is returned and errno is set to
     indicate the error.
```

```
ERRORS
     ioctl() will fail if:

     [EBADF]            d is not a valid descriptor.

     [ENOTTY]           d is not associated with a character special device.

     [ENOTTY]           The specified request does not apply to the kind of
                        object that the descriptor d references.

     [EINVAL]           request or arg is not valid.

     [EFAULT]           arg points outside the process's allocated address
                        space.

SEE ALSO
     cdio(1), chio(1), mt(1), execve(2), fcntl(2), intro(4), tty(4)

HISTORY
     An ioctl() function call appeared in Version 7 AT&T UNIX.

BSD                            December 11, 1993                           BSD
```

listen

```
LISTEN(2)                   BSD System Calls Manual                  LISTEN(2)

NAME
     listen -- listen for connections on a socket

SYNOPSIS
     #include <sys/types.h>
     #include <sys/socket.h>

     int
     listen(int s, int backlog);

DESCRIPTION
     To accept connections, a socket is first created with socket(2), a will-
     ingness to accept incoming connections and a queue limit for incoming
     connections are specified with listen(), and then the connections are
     accepted with accept(2).  The listen() call applies only to sockets of
     type SOCK_STREAM or SOCK_SEQPACKET.

     The backlog parameter defines the maximum length the queue of pending
     connections may grow to.  If a connection request arrives with the queue
     full the client may receive an error with an indication of ECONNREFUSED,
     or, if the underlying protocol supports retransmission, the request may
     be ignored so that retries may succeed.

RETURN VALUES
     A 0 return value indicates success; -1 indicates an error.

ERRORS
     listen() will fail if:

     [EBADF]            The argument s is not a valid descriptor.

     [ENOTSOCK]         The argument s is not a socket.

     [EOPNOTSUPP]       The socket is not of a type that supports the opera-
                        tion listen().

SEE ALSO
     accept(2), connect(2), socket(2)
```

```
HISTORY
     The listen() function call appeared in 4.2BSD.

BUGS
     The backlog is currently limited (silently) to 128.

BSD                            December 11, 1993                           BSD
```

poll

```
POLL(2)                    BSD System Calls Manual                   POLL(2)

NAME
     poll -- synchronous I/O multiplexing

SYNOPSIS
     #include <poll.h>

     int
     poll(struct pollfd *fds, int nfds, int timeout);

DESCRIPTION
     poll() provides a mechanism for reporting I/O conditions across a set of
     file descriptors.

     The arguments are as follows:

     fds      Points to an array of pollfd structures, which are defined as:

                    struct pollfd {
                            int fd;
                            short events;
                            short revents;
                    };

              The fd member is an open file descriptor.  The events and
              revents members are bitmasks of conditions to monitor and condi-
              tions found, respectively.

     nfds     The number of pollfd structures in the array.

     timeout  Maximum interval to wait for the poll to complete, in millisec-
              onds.  If this value is 0, then poll() will return immediately.
              If this value is INFTIM (-1), poll() will block indefinitely
              until a condition is found.

     The calling process sets the events bitmask and poll() sets the revents
     bitmask.  Each call to poll() resets the revents bitmask for accuracy.
     The condition flags in the bitmasks are defined as:

     POLLIN      Data is available on the file descriptor for reading.

     POLLNORM    Same as POLLIN.

     POLLPRI     Same as POLLIN.

     POLLOUT     Data can be written to the file descriptor without blocking.

     POLLERR     This flag is not used in this implementation and is provided
                 only for source code compatibility.

     POLLHUP     The file descriptor was valid before the polling process and
                 invalid after.  Presumably, this means that the file descrip-
                 tor was closed sometime during the poll.

     POLLNVAL    The corresponding file descriptor is invalid.
     POLLRDNORM  Same as POLLIN.

     POLLRDBAND  Same as POLLIN.

     POLLWRNORM  Same as POLLOUT.

     POLLWRBAND  Same as POLLOUT.

     POLLMSG     This flag is not used in this implementation and is provided
                 only for source code compatibility.

     All flags except POLLIN, POLLOUT, and their synonyms are for use only in
     the revents member of the pollfd structure.  An attempt to set any of
     these flags in the events member will generate an error condition.

     In addition to I/O multiplexing, poll() can be used to generate simple
     timeouts.  This functionality may be achieved by passing a null pointer
     for fds.

WARNINGS
     The POLLHUP flag is only a close approximation and may not always be
     accurate.

RETURN VALUES
     Upon error, poll() returns a -1 and sets the global variable errno to
     indicate the error.  If the timeout interval was reached before any
     events occurred, a 0 is returned.  Otherwise, poll() returns the number
     of file descriptors for which revents is non-zero.

ERRORS
     poll() will fail if:

     [EINVAL]   nfds was either a negative number or greater than the number
                of available file descriptors.

     [EINVAL]   An invalid flags was set in the events member of the pollfd
                structure.

     [EINVAL]   The timeout passed to poll() was too large.

     [EAGAIN]   Resource allocation failed inside of poll().  Subsequent calls
                to poll() may succeed.

     [EINTR]    poll() caught a signal during the polling process.

SEE ALSO
     poll(2), select(2), sysconf(3)

HISTORY
     A poll() system call appeared in AT&T System V UNIX.

BSD                            December 13, 1994                           BSD
```

select

```
SELECT(2)                   BSD System Calls Manual                  SELECT(2)

NAME
     select -- synchronous I/O multiplexing

SYNOPSIS
     #include <sys/types.h>
     #include <sys/time.h>
     #include <unistd.h>
     int
     select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
         struct timeval *timeout);

     FD_SET(fd, &fdset);

     FD_CLR(fd, &fdset);

     FD_ISSET(fd, &fdset);

     FD_ZERO(&fdset);

DESCRIPTION
     select() examines the I/O descriptor sets whose addresses are passed in
     readfds, writefds, and exceptfds to see if some of their descriptors are
     ready for reading, are ready for writing, or have an exceptional condi-
     tion pending, respectively.  The first nfds descriptors are checked in
     each set; i.e., the descriptors from 0 through nfds-1 in the descriptor
     sets are examined.  On return, select() replaces the given descriptor
     sets with subsets consisting of those descriptors that are ready for the
     requested operation.  select() returns the total number of ready descrip-
     tors in all the sets.

     The descriptor sets are stored as bit fields in arrays of integers.  The
     following macros are provided for manipulating such descriptor sets:
     FD_ZERO(&fdset) initializes a descriptor set fdset to the null set.
     FD_SET(fd, &fdset) includes a particular descriptor fd in fdset.
     FD_CLR(fd, &fdset) removes fd from fdset.  FD_ISSET(fd, &fdset) is non-
     zero if fd is a member of fdset, zero otherwise.  The behavior of these
     macros is undefined if a descriptor value is less than zero or greater
     than or equal to FD_SETSIZE, which is normally at least equal to the max-
     imum number of descriptors supported by the system.

     If timeout is a non-null pointer, it specifies a maximum interval to wait
     for the selection to complete.  If timeout is a null pointer, the select
     blocks indefinitely.  To effect a poll, the timeout argument should be
     non-null, pointing to a zero-valued timeval structure.  timeout is not
     changed by select(), and may be reused on subsequent calls; however, it
     is good style to re-initialize it before each invocation of select().

     Any of readfds, writefds, and exceptfds may be given as null pointers if
     no descriptors are of interest.

RETURN VALUES
     select() returns the number of ready descriptors that are contained in
     the descriptor sets, or -1 is an error occurred.  If the time limit
     expires, select() returns 0.  If select() returns with an error, includ-
     ing one due to an interrupted call, the descriptor sets will be unmodi-
     fied.

ERRORS
     An error return from select() indicates:

     [EFAULT]           One or more of readfds, writefds, or exceptfds points
                        outside the process's allocated address space.

     [EBADF]            One of the descriptor sets specified an invalid
                        descriptor.

     [EINTR]            A signal was delivered before the time limit expired
                        and before any of the selected events occurred.

     [EINVAL]           The specified time limit is invalid.  One of its com-
                        ponents is negative or too large.

SEE ALSO
     accept(2), connect(2), gettimeofday(2), poll(2), read(2), recv(2),
     send(2), write(2), getdtablesize(3)

BUGS
     Although the provision of getdtablesize(3) was intended to allow user
     programs to be written independent of the kernel limit on the number of
     open files, the dimension of a sufficiently large bit field for select
     remains a problem.  The default bit size of fd_set is based on the symbol
     FD_SETSIZE (currently 256), but that is somewhat smaller than the current
     kernel limit to the number of open files.  However, in order to accommo-
     date programs which might potentially use a larger number of open files
     with select, it is possible to increase this size within a program by
     providing a larger definition of FD_SETSIZE before the inclusion of
     <sys/types.h>.  The kernel will cope, and the userland libraries provided
     with the system are also ready for large numbers of file descriptors.

     Alternatively, to be really safe, it is possible to allocate fd_set bit-
     arrays dynamically.  The idea is to permit a program to work properly
     even if it is execve(2)'d with 4000 file descriptors pre-allocated.  The
     following illustrates the technique which is used by userland libraries:

                   fd_set *fdsr;
                   int max = fd;

                   fdsr = (fd_set *)calloc(howmany(max+1, NFDBITS),
                       sizeof(fd_mask));
                   if (fdsr == NULL) {
                           …
                           return (-1);
                   }
                   FD_SET(fd, fdsr);
                   n = select(max+1, fdsr, NULL, NULL, &tv);
                   …
                   free(fdsr);

     Alternatively, it is possible to use the poll(2) interface.  poll(2) is
     more efficient when the size of select()'s fd_set bit-arrays are very
     large, and for fixed numbers of file descriptors one need not size and
     dynamically allocate a memory object.

     select() should probably have been designed to return the time remaining
     from the original timeout, if any, by modifying the time value in place.
     Even though some systems stupidly act in this different way, it is
     unlikely this semantic will ever be commonly implemented, as the change
     causes massive source code compatibility problems.  Furthermore, recent
     new standards have dictated the current behaviour.  In general, due to
     the existence of those brain-damaged non-conforming systems, it is unwise
     to assume that the timeout value will be unmodified by the select() call,
     and the caller should reinitialize it on each invocation.  Calculating
     the delta is easily done by calling gettimeofday(2) before and after the
     call to select(), and using timersub() (as described in getitimer(2)).

     Internally to the kernel, select() works poorly if multiple processes
     wait on the same file descriptor.  Given that, it is rather surprising to
     see that many daemons are written that way (i.e., httpd(8)).

HISTORY
     The select() function call appeared in 4.2BSD.

BSD                             March 25, 1994                             BSD
```

send

```
SEND(2)                     BSD System Calls Manual                    SEND(2)

NAME
     send, sendto, sendmsg -- send a message from a socket
```

```
SYNOPSIS
     #include <sys/types.h>
     #include <sys/socket.h>

     ssize_t
     send(int s, const void *msg, size_t len, int flags);

     ssize_t
     sendto(int s, const void *msg, size_t len, int flags,
         const struct sockaddr *to, socklen_t tolen);

     ssize_t
     sendmsg(int s, const struct msghdr *msg, int flags);

DESCRIPTION
     send(), sendto(), and sendmsg() are used to transmit a message to another
     socket.  send() may be used only when the socket is in a connected state,
     while sendto() and sendmsg() may be used at any time.

     The address of the target is given by to with tolen specifying its size.
     The length of the message is given by len.  If the message is too long to
     pass atomically through the underlying protocol, the error EMSGSIZE is
     returned, and the message is not transmitted.

     No indication of failure to deliver is implicit in a send().  Locally
     detected errors are indicated by a return value of -1.

     If no messages space is available at the socket to hold the message to be
     transmitted, then send() normally blocks, unless the socket has been
     placed in non-blocking I/O mode.  The select(2) or poll(2) system calls
     may be used to determine when it is possible to send more data.

     The flags parameter may include one or more of the following:

     #define MSG_OOB        0x1  /* process out-of-band data */
     #define MSG_DONTROUTE  0x4  /* bypass routing, use direct interface */

     The flag MSG_OOB is used to send ``out-of-band'' data on sockets that
     support this notion (e.g., SOCK_STREAM); the underlying protocol must
     also support ``out-of-band'' data.  MSG_DONTROUTE is usually used only by
     diagnostic or routing programs.

     See recv(2) for a description of the msghdr structure.

RETURN VALUES
     The call returns the number of characters sent, or -1 if an error
     occurred.

ERRORS
     send(), sendto(), and sendmsg() fail if:

     [EBADF]            An invalid descriptor was specified.

     [ENOTSOCK]         The argument s is not a socket.

     [EFAULT]           An invalid user space address was specified for a
                        parameter.

     [EMSGSIZE]         The socket requires that message be sent atomically,
                        and the size of the message to be sent made this
                        impossible.

     [EAGAIN]           The socket is marked non-blocking and the requested
                        operation would block.

     [ENOBUFS]          The system was unable to allocate an internal buffer.
```

```
                         The operation may succeed when buffers become avail-
                         able.

     [ENOBUFS]           The output queue for a network interface was full.
                         This generally indicates that the interface has
                         stopped sending, but may be caused by transient con-
                         gestion.

     [EACCES]            The SO_BROADCAST option is not set on the socket, and
                         a broadcast address was given as the destination.

     [EHOSTUNREACH]      The destination address specified an unreachable host.

     [EINVAL]            The flags parameter is invalid.

     [EHOSTDOWN]         The destination address specified a host that is down.

     [ENETDOWN]          The destination address specified a network that is
                         down.

     [ECONNREFUSED]      The destination host rejected the message (or a previ-
                         ous one).  This error can only be returned by con-
                         nected sockets.

     [ENOPROTOOPT]       There was a problem sending the message.  This error
                         can only be returned by connected sockets.

     [EDESTADDRREQ]      The socket is not connected, and no destination
                         address was specified.

     [EISCONN]           The socket is already connected, and a destination
                         address was specified.

     In addition, send() and sendto() may return the following error:

     [EINVAL]            len was larger than SSIZE_MAX.

     Also, sendmsg() may return the following errors:

     [EINVAL]            The sum of the iov_len values in the msg_iov array
                         overflowed an ssize_t.

     [EMSGSIZE]          The msg_iovlen member of msg was less than 0 or larger
                         than IOV_MAX.

     [EAFNOSUPPORT]      Addresses in the specified address family cannot be
                         used with this socket.

SEE ALSO
     fcntl(2), getsockopt(2), poll(2), recv(2), select(2), poll(2), socket(2),
     write(2)

HISTORY
     The send() function call appeared in 4.2BSD.

BSD                              July 28, 1998                             BSD
```

shutdown

```
SHUTDOWN(2)                 BSD System Calls Manual                SHUTDOWN(2)

NAME
     shutdown -- shut down part of a full-duplex connection

SYNOPSIS
     #include <sys/types.h>
     #include <sys/socket.h>

     int
     shutdown(int s, int how);

DESCRIPTION
     The shutdown() call causes all or part of a full-duplex connection on the
     socket associated with s to be shut down.  If how is SHUT_RD, further
     receives will be disallowed.  If how is SHUT_WR, further sends will be
     disallowed.  If how is SHUT_RDWR, further sends and receives will be dis-
     allowed.

RETURN VALUES
     A 0 is returned if the call succeeds, -1 if it fails.

ERRORS
     The call succeeds unless:

     [EINVAL]           how is not SHUT_RD, SHUT_WR, or SHUT_RDWR.

     [EBADF]            s is not a valid descriptor.

     [ENOTSOCK]         s is a file, not a socket.

     [ENOTCONN]         The specified socket is not connected.

SEE ALSO
     connect(2), socket(2)

HISTORY
     The shutdown() function call appeared in 4.2BSD.  The how arguments used
     to be simply 0, 1, and 2, but now have named values as specified by
     X/Open Portability Guide Issue 4 (``XPG4'').

BSD                              June 4, 1993                              BSD
```

socket

```
SOCKET(2)                   BSD System Calls Manual                  SOCKET(2)

NAME
     socket -- create an endpoint for communication

SYNOPSIS
     #include <sys/types.h>
     #include <sys/socket.h>

     int
     socket(int domain, int type, int protocol);

DESCRIPTION
     socket() creates an endpoint for communication and returns a descriptor.

     The domain parameter specifies a communications domain within which com-
     munication will take place; this selects the protocol family which should
     be used.  These families are defined in the include file <sys/socket.h>.
     The currently understood formats are

           AF_UNIX          (UNIX internal protocols),
           AF_INET          (ARPA Internet protocols),
           AF_INET6         (ARPA IPv6 protocols),
           AF_ISO           (ISO protocols),
           AF_NS            (Xerox Network Systems protocols),
           AF_IPX           (Internetwork Packet Exchange), and
           AF_IMPLINK       (IMP host at IMP link layer).
     The socket has the indicated type, which specifies the semantics of com-
     munication.  Currently defined types are:

           SOCK_STREAM
           SOCK_DGRAM
           SOCK_RAW
           SOCK_SEQPACKET
           SOCK_RDM

     A SOCK_STREAM type provides sequenced, reliable, two-way connection based
     byte streams.  An out-of-band data transmission mechanism may be sup-
     ported.  A SOCK_DGRAM socket supports datagrams (connectionless, unreli-
     able messages of a fixed (typically small) maximum length).  A
     SOCK_SEQPACKET socket may provide a sequenced, reliable, two-way connec-
     tion-based data transmission path for datagrams of fixed maximum length;
     a consumer may be required to read an entire packet with each read system
     call.  This facility is protocol specific, and presently implemented only
     for PF_NS.  SOCK_RAW sockets provide access to internal network protocols
     and interfaces.  The types SOCK_RAW, which is available only to the supe-
     ruser, and SOCK_RDM, which is planned, but not yet implemented, are not
     described here.

     The protocol specifies a particular protocol to be used with the socket.
     Normally only a single protocol exists to support a particular socket
     type within a given protocol family.  However, it is possible that many
     protocols may exist, in which case a particular protocol must be speci-
     fied in this manner.  The protocol number to use is particular to the
     communication domain in which communication is to take place; see
     protocols(5).  A value of 0 for protocol will let the system select an
     appropriate protocol for the requested socket type.

     Sockets of type SOCK_STREAM are full-duplex byte streams, similar to
     pipes.  A stream socket must be in a connected state before any data may
     be sent or received on it.  A connection to another socket is created
     with a connect(2) call.  Once connected, data may be transferred using
     read(2) and write(2) calls or some variant of the send(2) and recv(2)
     calls.  When a session has been completed a close(2) may be performed.
     Out-of-band data may also be transmitted as described in send(2) and
     received as described in recv(2).

     The communications protocols used to implement a SOCK_STREAM ensure that
     data is not lost or duplicated.  If a piece of data for which the peer
     protocol has buffer space cannot be successfully transmitted within a
     reasonable length of time, then the connection is considered broken and
     calls will indicate an error with -1 returns and with ETIMEDOUT as the
     specific code in the global variable errno.  The protocols optionally
     keep sockets ``warm'' by forcing transmissions roughly every minute in
     the absence of other activity.  An error is then indicated if no response
     can be elicited on an otherwise idle connection for a extended period
     (e.g., 5 minutes).  A SIGPIPE signal is raised if a process sends on a
     broken stream; this causes naive processes, which do not handle the sig-
     nal, to exit.

     SOCK_SEQPACKET sockets employ the same system calls as SOCK_STREAM sock-
     ets.  The only difference is that read(2) calls will return only the
     amount of data requested, and any remaining in the arriving packet will
     be discarded.

     SOCK_DGRAM and SOCK_RAW sockets allow sending of datagrams to correspon-
     dents named in send(2) calls.  Datagrams are generally received with
     recvfrom(2), which returns the next datagram with its return address.

     An fcntl(2) call can be used to specify a process group to receive a
     SIGURG signal when the out-of-band data arrives.  It may also enable non-
     blocking I/O and asynchronous notification of I/O events via SIGIO.

     The operation of sockets is controlled by socket level options.  These
     options are defined in the file <sys/socket.h>.  setsockopt(2) and
     getsockopt(2) are used to set and get options, respectively.

RETURN VALUES
     A -1 is returned if an error occurs, otherwise the return value is a
     descriptor referencing the socket.

ERRORS
     The socket() call fails if:

     [EPROTONOSUPPORT]  The protocol type or the specified protocol is not
                        supported within this domain.

     [EMFILE]           The per-process descriptor table is full.

     [ENFILE]           The system file table is full.

     [EACCES]           Permission to create a socket of the specified type
                        and/or protocol is denied.

     [ENOBUFS]          Insufficient buffer space is available.  The socket
                        cannot be created until sufficient resources are
                        freed.

SEE ALSO
     accept(2), bind(2), connect(2), getsockname(2), getsockopt(2), ioctl(2),
     listen(2), poll(2), read(2), recv(2), select(2), send(2), setsockopt(2),
     shutdown(2), socketpair(2), write(2), getprotoent(3), netintro(4)

     An Introductory 4.3 BSD Interprocess Communication Tutorial, reprinted in
     UNIX Programmer's Supplementary Documents Volume 1.

     BSD Interprocess Communication Tutorial, reprinted in UNIX Programmer's
     Supplementary Documents Volume 1.

HISTORY
     The socket() function call appeared in 4.2BSD.

BSD                              June 4, 1993                             BSD
```

Part XLII. FreeBSD TCP/IP Stack port for eCos

TCP/IP Networking for eCos now provides a complete TCP/IP networking stack, based on a recent snapshot of the FreeBSD code, released by the KAME project. The networking support is fully featured and well tested within the eCos environment.

Table of Contents

Chapter 150. Networking Stack Features

Since this networking package is based on BSD code, it is very complete and robust. The eCos implementation includes support for the following protocols:

- IPv4
- UDP
- TCP
- ICMP
- raw packet interface
- Multi-cast addressing
- IPv6 (including UDP, ICP, ICMP)

These additional features are also present in the package, but are not supported:

- Berkeley Packet Filter
- Uni-cast support
- Multi-cast routing

Chapter 151. Freebsd TCP/IP stack port

This document describes how to get started with the Freebsd TCP/IP network stack.

Targets

A number of ethernet devices may be supported. The default configuration supports two instances of the interface by default, and you will need to write your own driver instantiation code, and supplemental startup and initialization code, if you should add additional ones.

The target for your board will normally be supplied with an ethernet driver, in which case including the network stack and generic ethernet driver package to your build will automatically enable usage of the ethernet device driver. If your target is not supplied with an ethernet driver, you will need to use loopback (see the section called “Loopback tests”).

Building the Network Stack

Using the *Build->Packages* dialog, add the packages “Networking”, “Freebsd TCP/IP Stack” and “Common Ethernet Support” to your configuration. Their package names are CYGPKG_NET, CYGPKG_NET_FREEBSD_STACK and CYGPKG_NET_ETH_DRIVERS respectively.

A short-cut way to do this is by using the “net” *template* if it is available for your platform.

The platform-specific ethernet device driver for your platform will be added as part of the target selection (in the *Build->Templates* “Hardware” item), along with the PCI I/O subsystem (if relevent) and the appropriate serial device driver.

For example, the PowerPC MBX target selection adds the package PKG_NET_QUICC_ETH_DRIVERS, and the Cirrus Logic EDB7xxx target selection adds the package CYGPKG_NET_EDB7XXX_ETH_DRIVERS. After this, eCos and its tests can be built exactly as usual.

Note

By default, most of the network tests are not built. This is because some of them require manual intervention, i.e. they are to be run “by hand”, and are not suitable for automated testing. To build the full set of network tests, set the configuration option CYGPKG_NET_BUILD_HW_TESTS “Build hardware networking tests (demo programs)” within “Networking support build options”.

Chapter 152. APIs

Standard networking

The APIs for the standard networking calls such as `socket()`, `recv()` and so on, are in header files relative to the top-level include directory, within the standard subdirectories as conventionally found in `/usr/include`. For example:

```
install/include/arpa/tftp.h
install/include/netinet/tcpip.h
install/include/sys/socket.h
install/include/sys/socketvar.h
install/include/sys/sockio.h
```

`network.h` at the top level defines various extensions, for example the API `init_all_network_interfaces(void)` described above. We advise including `network.h` whether you use these features or not.

In general, using the networking code may require definition of two symbols: _KERNEL and __ECOS. _KERNEL is not normally required; __ECOS is normally required. So add this to your compile lines for files which use the network stack:

```
        -D__ECOS
```

To expand a little, it,s like this because this is a port of a standard distribution external to eCos. One goal is to perturb the sources as little as possible, so that upgrading and maintenance from the external distribution is simplified. The __ECOS symbol marks out the eCos additions in making the port. The _KERNEL symbol is traditional UNIX practice: it distinguishes a compilation which is to be linked into the kernel from one which is part of an application. eCos applications are fully linked, so this distinction does not apply. _KERNEL can however be used to control the visibility of the internals of the stack, so depending on what features your application uses, it may or may not be necessary.

The include file `network.h` undefines _KERNEL unconditionally, to provide an application-like compilation environment. If you were writing code which, for example, enumerates the stack,s internal structures, that is a kernel-like compilation environment, so you would need to define _KERNEL (in addition to __ECOS) and avoid including `network.h`.

Part XLIII. eCos PPP User Guide

This package provides support for PPP (Point-to-Point Protocol) in the eCos FreeBSD TCP/IP networking stack.

Table of Contents

Chapter 153. Features

The eCos PPP implementation provides the following features:

- PPP line protocol including VJ compression.
- LCP, IPCP and CCP control protocols.
- PAP and CHAP authentication.
- CHAT subset connection scripting.
- Modem control line support.

Chapter 154. Using PPP

Before going into detail, let's look at a simple example of how the eCos PPP package is used. Consider the following example:

```
static void ppp_up(void)
{
    cyg_ppp_options_t options;
    cyg_ppp_handle_t ppp_handle;

    // Bring up the TCP/IP network
    init_all_network_interfaces();

    // Initialize the options
    cyg_ppp_options_init( &options );

    // Start up PPP
    ppp_handle = cyg_ppp_up( "/dev/ser0", &options );

    // Wait for it to get running
    if( cyg_ppp_wait_up( ppp_handle ) == 0 )
    {
        // Make use of PPP
        use_ppp();

        // Bring PPP link down
        cyg_ppp_down( ppp_handle );

        // Wait for connection to go down.
        cyg_ppp_wait_down( ppp_handle );
    }
}
```

This is a simple example of how to bring up a simple PPP connection to another computer over a directly connected serial line. The other end is assumed to already be running PPP on the line and waiting for a connection.

The first thing this code does is to call `init_all_network_interfaces()` to bring up the TCP/IP stack and initialize any other network interfaces. It then calls `cyg_ppp_options_init()` to initialize the PPP options structure to the defaults. As it happens, the default options are exactly what we want for this example, so we don't need to make any further changes. We go straight on to bring the PPP interface up by calling `cyg_ppp_up()`. The arguments to this function give the name of the serial device to use, in this case `"/dev/ser0"`, and a pointer to the options.

When `cyg_ppp_up()` returns, it passes back a handle to the PPP connection which is to be used in other calls. The PPP link will not necessarily have been fully initialized at this time. There is a certain amount of negotiation that goes on between the ends of a PPP link before it is ready to pass packets. An application can wait until the link is ready by calling `cyg_ppp_wait_up()`, which returns zero if the link is up and running, or `-1` if it has gone down or failed to come up.

After a successful return from `cyg_ppp_wait_up()`, the application may make use of the PPP connection. This is represented here by the call to `use_ppp()` but it may, of course, be accessed by any thread. While the connection is up the application may use the standard socket calls to make or accept network connections and transfer data in the normal way.

Once the application has finished with the PPP link, it can bring it down by calling `cyg_ppp_down()`. As with bringing the connection up, this call is asynchronous, it simply informs the PPP subsystem to start bringing the link down. The application can wait for the link to go down fully by calling `cyg_ppp_wait_down()`.

That example showed how to use PPP to connect to a local peer. PPP is more often used to connect via a modem to a remote server, such as an ISP. The following example shows how this works:

```
static char *isp_script[] =
{
    "ABORT"             ,       "BUSY"                                  ,
    "ABORT"             ,       "NO CARRIER"                            ,
    "ABORT"             ,       "ERROR"                                 ,
    ""                  ,       "ATZ"                                   ,
    "OK"                ,       "AT S7=45 S0=0 L1 V1 X4 &C1 E1 Q0"      ,
    "OK"                ,       "ATD" CYGPKG_PPP_DEFAULT_DIALUP_NUMBER  ,
    "ogin:--ogin:"      ,       CYGPKG_PPP_AUTH_DEFAULT_USER            ,
    "assword:"          ,       CYGPKG_PPP_AUTH_DEFAULT_PASSWD          ,
    "otocol:"           ,       "ppp"                                   ,
    "HELLO"             ,       "\\c"                                   ,
    0
};

static void ppp_up(void)
{
    cyg_ppp_options_t options;
    cyg_ppp_handle_t ppp_handle;

    // Bring up the TCP/IP network
    init_all_network_interfaces();

    // Initialize the options
    cyg_ppp_options_init( &options );

    options.script = isp_script;
    options.modem  = 1;

    // Start up PPP
    ppp_handle = cyg_ppp_up( "/dev/ser0", &options );

    // Wait for it to get running
    if( cyg_ppp_wait_up( ppp_handle ) == 0 )
    {
        // Make use of PPP
        use_ppp();

        // Bring PPP link down
        cyg_ppp_down( ppp_handle );

        // Wait for connection to go down.
        cyg_ppp_wait_down( ppp_handle );
    }
}
```

The majority of this code is exactly the same as the previous example. The main difference is in the setting of a couple of options before calling `cyg_ppp_up()`. The *`script`* option is set to point to a CHAT script to manage the setup of the connection. The *`modem`* option is set to cause the PPP system to make use of the modem control lines.

During the PPP bring-up a call will be made to `cyg_ppp_chat()` to run the CHAT script (see Chapter 157, *CHAT Scripts*). In the example this script sets up various modem options and then dials a number supplied as part of the PPP package configuration (see Chapter 156, *Installing and Configuring PPP*). When the connection has been established, the script log on to the server, using a name and password also supplied by the configuration, and then starts PPP on the remote end. If this script succeeds the PPP connection will be brought up and will then function as expected.

The *`modem`* option causes the PPP system to make use of the modem control lines. In particular it waits for `Carrier Detect` to be asserted, and will bring the link down if it is lost. See cyg_ppp_options_init() for more details.

Chapter 155. PPP Interface

Name

cyg_ppp_options_init — Initialize PPP link options

Synopsis

#include <cyg/ppp/ppp.h>

`cyg_int32 **cyg_ppp_options_init**(*options);`

Description

This function initializes the PPP options, pointed to by the *options* parameter, to the default state. Once the defaults have been initialized, application code may adjust them by assigning new values to the the fields of the cyg_ppp_options_t structure.

This function returns zero if the options were initialized successfully. It returns -1 if the *options* argument is NULL, or the options could not be initialized.

The option fields, their functions and default values are as follows:

debug

If set to 1 this enables the reporting of debug messages from the PPP system. These will be generated using `diag_printf()` and will appear on the standard debug channel. Note that `diag_printf()` disables interrupts during output: this may cause the PPP link device to overrun and miss characters. It is quite possible for this option to cause errors and even make the PPP link fail completely. Consequently, this option should be used with care.

Default value: 0

kdebugflag

This five bit field enables low level debugging messages from the PPP device layer in the TCP/IP stack. As with the *debug* option, this may result in missed characters and cause errors. The bits of the field have the following meanings:

Bit	BSD Name	Description
0x01	SC_DEBUG	Enable debug messages
0x02	SC_LOG_INPKT	Log contents of good packets received
0x04	SC_LOG_OUTPKT	Log contents of packets sent
0x08	SC_LOG_RAWIN	Log all characters received
0x10	SC_LOG_FLUSH	Log all characters flushed

Default value: 0

default_route

If set to 1 this option causes the PPP subsystem to install a default route in the TCP/IP stack's routing tables using the peer as the gateway. This entry will be removed when the PPP link is broken. If there is already an existing working network connection, such as an ethernet device, then there may already be a default route established. If this is the case, then this option will have no effect.

Default value: 1

modem

If this option is set to 1, then the modem lines will be used during the connection. Specifically, the PPP subsystem will wait until the `carrier detect` signal is asserted before bringing up the PPP link, and will take the PPP link down if this signal is de-asserted.

Default value: 0

flowctl

This option is used to specify the mechanism used to control data flow across the serial line. It can take one of the following values:

`CYG_PPP_FLOWCTL_DEFAULT`

The flow control mechanism is not changed and is left at whatever value was set before bringing PPP up. This allows a non-standard flow control mechanism to be used, or for it to be chosen and set by some other means.

`CYG_PPP_FLOWCTL_NONE`

Flow control is turned off. It is not recommended that this option be used unless the baud rate is set low or the two communicating machines are particularly fast.

`CYG_PPP_FLOWCTL_HARDWARE`

Use hardware flow control via the RTS/CTS lines. This is the most effective flow control mechanism and should always be used if available. Availability of this mechanism depends on whether the serial device hardware has the ability to control these lines, whether they have been connected to the socket pins and whether the device driver has the necessary support.

`CYG_PPP_FLOWCTL_SOFTWARE`

Use software flow control by embedding XON/XOFF characters in the data stream. This is somewhat less effective that hardware flow control since it is subject to the propagation time of the serial cable and the latency of the communicating devices. Since it does not rely on any hardware support, this flow control mechanism is always available.

Default value: CYG_PPP_FLOWCTL_HARDWARE

refuse_pap

If this option is set to 1, then the PPP subsystem will not agree to authenticate itself to the peer with PAP. When dialling in to a remote server it is normal to authenticate the client. There are three ways this can be done, using a straightforward login mechanism via the CHAT script, with the Password Authentication Protocol (PAP), or with the Challenge Handshake Authentication Protocol (CHAP). For PAP to work the *`user`* and *`passwd`* options must be set to the expected values. If they are not, then this option should be set to force CHAP authentication.

Default value: 0

refuse_chap

If this option is set to 1, then the PPP subsystem will not agree to authenticate itself to the peer with CHAP. CHAP authentication will only work if the *`passwd`* option has been set to the required CHAP secret for the destination server. Otherwise this option should be disabled.

If both *`refuse_pap`* and *`refuse_chap`* are set, then either no authentication will be carried out, or it is the responsibility of the **chat** script to do it. If the peer does not require any authentication, then the setting of these options is irrelevant.

Default value: 0

baud

This option is set to the baud rate at which the serial connection should be run. The default value is the rate at which modems conventionally operate. This field is an instance of the cyg_serial_baud_rate_t enum defined in the `serialio.h` header and may only take one of the baud rate constants defined in there.

Default value: `CYGNUM_SERIAL_BAUD_115200` idle_time_limit

This is the number of seconds that the PPP connection may be idle before it is shut down automatically.

Default value: 60

maxconnect

This causes the connection to terminate when it has been up for this number of seconds. The default value of zero means that the connection will stay up indefinitely, until either end explicitly brings it down, or the link is lost.

Default value: 0

our_address

This is the IP address, in network byte order, to be attached to the local end of the PPP connection. The default value of `INADDR_ANY` causes the local address to be obtained from the peer.

Default value: `INADDR_ANY`

his_address

This is the IP address, in network byte order, to be attached to the remote end of the PPP connection. The default value of `INADDR_ANY` causes the remote address to be obtained from the peer.

Default value: `INADDR_ANY`

accept_local

This allows the behaviour described above for `our_address` to be modified. Normally, if `our_address` is set, then the PPPD will insist that this address be used. However, if this option is also set, the PPPD will accept a value supplied by the peer.

Default value: 0

accept_remote

This allows the behaviour described above for `his_address` to be modified. Normally, if `his_address` is set, then the PPPD will insist that this address be used. However, if this option is also set, the PPPD will accept a value supplied by the peer.

Default value: 0

script

This is a pointer to a CHAT script suitable for passing to `cyg_ppp_chat()`. See Chapter 157, *CHAT Scripts* for details of the format and contents of this script.

Default value: `NULL`

user

This array contains the user name to be used for PAP authentication. This field is not used for CHAP authentication. By default the value of this option is set from the `CYGPKG_PPP_AUTH_DEFAULT_USER` configuration option.

Default value: `CYGPKG_PPP_AUTH_DEFAULT_USER`

passwd

This array contains the password to be used for PAP authentication, or the secret to be used during CHAP authentication. By default the value of this option is set from the `CYGPKG_PPP_AUTH_DEFAULT_PASSWD` configuration option.

Default value: `CYGPKG_PPP_AUTH_DEFAULT_PASSWD`

Name

cyg_ppp_up — Bring PPP connection up

Synopsis

#include <cyg/ppp/ppp.h>

`cyg_ppp_handle_t` **`cyg_ppp_up`**`(`*`*devnam`*`, `*`*options`*`);`

Description

This function starts up a PPP connection. The *`devnam`* argument is the name of the device to be used for the connection, typically `"/dev/ser0"` or `"/dev/ser1"`. The *`options`* argument should point to an initialized cyg_ppp_options_t object.

The return value will either be zero, indicating a failure, or a cyg_ppp_handle_t object that may be used as an argument to other PPP functions.

Note

Although the PPP API is designed to permit several simultaneous connections to co-exist, at present only one PPP connection is actually implemented. Any attempt to create a second connection while there is already one open will fail.

Name

cyg_ppp_down — Bring PPP connection down

Synopsis

#include <cyg/ppp/ppp.h>

`cyg_int32 **cyg_ppp_down**(*handle*);`

Description

This function brings the PPP connection down. The *`handle`* argument is the result of a successful call to `cyg_ppp_up()`. This function only signals to the PPP subsystem that the link should be brought down. The link will be terminated asynchronously. If the application needs to wait for the link to terminate, then it should call `cyg_ppp_wait_down()` after calling `cyg_ppp_down()`.

The function returns zero if it was able to start the termination of the PPP connection successfully. It will return -1 if the connection is not running, or if it could not otherwise start the termination.

Name

cyg_ppp_wait_up — Wait for PPP connection to come up

Synopsis

#include <cyg/ppp/ppp.h>

```
cyg_int32 cyg_ppp_wait_up(handle);
```

Description

This function waits until the PPP connection is running and then returns. This is needed because the actual bring up of the connection happens mostly after the call to `cyg_ppp_up()` returns, and may take some time to complete, especially if dialling a remote server.

The result of this call will be zero when the connection is running, or -1 if the connection failed to start for some reason. If the connection is already running when this call is made it will return immediately with a zero result. If the connection is not in the process of coming up, or has failed, or has terminated, then a result of -1 will be returned immediately. Thus this function may also be used to test that the connection is still running at any point.

Name

cyg_ppp_wait_down — Wait for PPP connection to terminate

Synopsis

#include <cyg/ppp/ppp.h>

```
void cyg_ppp_wait_down(handle);
```

Description

This function waits for the PPP connection to terminate. The link may be terminated with a call to `cyg_ppp_down()`, by the remote end, or by the telephone line being dropped or lost.

This function has no return value. If the PPP connection is not running, or has terminated, it will return. Applications should use `cyg_ppp_wait_up()` to test the link state.

Name

cyg_ppp_chat — Execute chat script

Synopsis

#include <cyg/ppp/ppp.h>

```
cyg_int32 cyg_ppp_chat(*devname, *script[]);
```

Description

This function implements a subset of the automated conversational scripting as defined by the **chat** program. The first argument is the name of the serial device to be used, typically `"/dev/ser0"` or `"/dev/ser1"`. The *`script`* argument is a pointer to a zero terminated array of strings that comprise the chat script. See Chapter 154, *Using PPP* for an example script, and Chapter 157, *CHAT Scripts* for full detail of the script used.

The return value of this function will be zero if the chat script fails for any reason, such as an ABORT or a timeout. If the end of the script is reached, then the return value will be non-zero.

Under normal use this function is called from the PPP subsystem if the cyg_ppp_options_t *`script`* field is set to a non-`NULL` value. This function should only be used directly if the application needs to undertake special processing between running the chat script, and bringing up the PPP connections.

Chapter 156. Installing and Configuring PPP

Including PPP in a Configuration

PPP is contained entirely within a single eCos package. So to include PPP in a configuration all you need to do is add that package.

In the GUI configuration tool use the **Build->Packages** menu item, find the "PPP Support" package in the left-hand pane and use the **Add** button to add it to the list of packages in use in the right-hand pane.

In the command-line tool **ecosconfig**, you can use the following command during the configuration phase to add the PPP package:

```
$ ecosconfig add ppp
```

In addition to the PPP package you will also need to have the `"Network"` package and the `"Serial Device Drivers"` package in the configuration. The dependencies and requirements of the networking package are such that it is strongly recommended that you start with the `net` template.

See the eCos User Guide for full details on how to configure and build eCos.

Configuring PPP

The PPP package contains a number of configuration options that may be changed to affect its behaviour.

CYGNUM_PPP_PPPD_THREAD_PRIORITY

> The PPP system contains two threads, One is used for receiving data from the link and processing control packets. The other is used to transmit data asynchronously to the link when it cannot be completed synchronously. The receive thread runs at the priority given here, and the transmit thread runs at the next lower priority. The exact priority needed here depends on the importance of the PPP subsystem relative to the rest of the system. The default is to put it in the middle of the priority range to provide reasonable response without impacting genuine high priority threads.
>
> Default value: `CYGNUM_KERNEL_SCHED_PRIORITIES/2`

CYGPKG_PPP_DEBUG_WARN_ONLY

> The runtime `debug` option enables logging of high level debug messages. Too many of these can interfere with the PPP device and may result in missed messages. This is because these messages are emitted via the diag_printf() mechanism, which disables interrupts while it prints. By default, therefore, we only report errors and warnings, and not all events. Setting this option to zero will enable the logging of all events.
>
> Default value: `1`

CYGPKG_PPP_AUTH_DEFAULT_USER

> This option gives the default value for the user name used to initialize the *`user`* field in the PPP options.
>
> Default value: `"eCos"`

CYGPKG_PPP_AUTH_DEFAULT_PASSWD

> This option gives the default value for the password used to initialize the *`passwd`* field in the PPP options.

Default value: `"secret"`

CYGPKG_PPP_DEFAULT_DIALUP_NUMBER

This option provides a default dialup number for use in **chat** scripts. This value is not used anywhere in the PPP package, but is provided to complete the information needed, alongside the user name and password, for accessing a typical dialup server.

Default value: `"5551234"`

CYGPKG_PPP_PAP

This component enables the inclusion of PAP authentication support.

Default value: 1

CYGPKG_PPP_CHAP

This component enables the inclusion of CHAT authentication support.

Default value: 1

CYGPKG_PPP_COMPRESSION

This component provides control over PPP compression features. WARNING: at present there are problems with this option, and and in any case the compression code needs to allocate large amounts of memory. Hence this option is currently disabled and should remain so.

Default value: 0

PPP_BSDCOMP

This option enables inclusion of BSD compression into the PPP protocol.

Default value: 0

PPP_DEFLATE

This option enables inclusion of ZLIB compression into the PPP protocol.

Default value: 0

CYGPKG_PPP_CHAT

This component enables the inclusion of a simple scripting system to bring up PPP connections. It implements a subset of the **chat** scripting language.

Default value: 1

CYGNUM_PPP_CHAT_ABORTS_MAX

This option defines the maximum number of `ABORT` strings that the CHAT system will store.

Default value: 10

CYGNUM_PPP_CHAT_ABORTS_SIZE

This option defines the maximum size of each `ABORT` strings that the **chat** system will store.

Default value: 20

CYGNUM_PPP_CHAT_STRING_LENGTH

This option defines the maximum size of any expect or reply strings that the **chat** system will be given.

Default value: 256

CYGPKG_PPP_TEST_DEVICE

This option defines the serial device to be used for PPP test programs.

Default value: `"/dev/ser0"`

CYGPKG_PPP_TESTS_AUTOMATE

This option enables automated testing features in certain test programs. These programs will interact with a test server at the remote end of the serial link to run a variety of tests in different conditions. Without this option most tests default to running a single test instance and are suitable for being run by hand for debugging purposes.

Default value: 0

CYGDAT_PPP_TEST_BAUD_RATES

This option supplies a list of baud rates at which certain tests will run if the `CYGPKG_PPP_TESTS_AUTOMATE` option is set.

Default value: `"CYGNUM_SERIAL_BAUD_19200, CYGNUM_SERIAL_BAUD_38400, CYGNUM_SERIAL_BAUD_57600, CYGNUM_SERIAL_BAUD_115200"`

Chapter 157. CHAT Scripts

The automated conversational scripting supported by the eCos PPP package is a subset of the scripting language provided by the **chat** command found on most UNIX and Linux systems.

Unlike the **chat** command, the eCos `cyg_ppp_chat()` function takes as a parameter a zero-terminated array of pointers to strings. In most programs this will be defined by means of an initializer for a static array, although there is nothing to stop the application constructing it at runtime. A simple script would be defined like this:

```
static char *chat_script[] =
{
    "ABORT"        ,  "BUSY"        ,
    "ABORT"        ,  "NO CARRIER"  ,
    ""             ,  "ATD5551234"  ,
    "ogin:--ogin:" ,  "ppp"         ,
    "ssword:"      ,  "hithere"     ,
    0
};
```

The following sections have been abstracted from the public domain documentation for the **chat** command.

Chat Script

A script consists of one or more "expect-send" pairs of strings, separated by spaces, with an optional "subexpect- subsend" string pair, separated by a dash as in the following example:

```
    "ogin:--ogin:"     ,  "ppp"       ,
    "ssword:"          ,   "hello2u2" ,
    0
```

This script fragment indicates that the `cyg_ppp_chat()` function should expect the string "ogin:". If it fails to receive a login prompt within the time interval allotted, it is to send a carriage return to the remote and then expect the string "ogin:" again. If the first "ogin:" is received then the carriage return is not generated.

Once it received the login prompt the `cyg_ppp_chat()` function will send the string "ppp" and then expect the prompt "ssword:". When it receives the prompt for the password, it will send the password "hello2u2".

A carriage return is normally sent following the reply string. It is not expected in the "expect" string unless it is specifically requested by using the "\r" character sequence.

The expect sequence should contain only what is needed to identify the string. It should not contain variable information. It is generally not acceptable to look for time strings, network identification strings, or other variable pieces of data as an expect string.

To help correct for characters which may be corrupted during the initial sequence, look for the string "ogin:" rather than "login:". It is possible that the leading "l" character may be received in error and you may never find the string even though it was sent by the system. For this reason, scripts look for "ogin:" rather than "login:" and "ssword:" rather than "password:".

A very simple script might look like this:

```
    "ogin:"    , "ppp"       ,
    "ssword:"  , " hello2u2" ,
    0
```

In other words, expect "….ogin:", send "ppp", expect "…ssword:", send "hello2u2".

In actual practice, simple scripts are rare. At the very least, you should include sub-expect sequences should the original string not be received. For example, consider the following script:

```
    "ogin:--ogin:"  , "ppp"     ,
    "ssword:"       , "hello2u2",
    0
```

This would be a better script than the simple one used earlier. This would look for the same "login:" prompt, however, if one was not received, a single return sequence is sent and then it will look for "login:" again. Should line noise obscure the first login prompt then sending the empty line will usually generate a login prompt again.

ABORT Strings

Many modems will report the status of the call as a string. These strings may be CONNECTED or NO CARRIER or BUSY. It is often desirable to terminate the script should the modem fail to connect to the remote. The difficulty is that a script would not know exactly which modem string it may receive. On one attempt, it may receive BUSY while the next time it may receive NO CARRIER.

These "abort" strings may be specified in the script using the ABORT sequence. It is written in the script as in the following example:

```
    "ABORT"    , "BUSY"    ,
    "ABORT"    , "NO CARRIER"  ,
    ""         , "ATZ"         ,
    "OK"       , "ATDT5551212" ,
    "CONNECT"  , …
```

This sequence will expect nothing; and then send the string ATZ. The expected response to this is the string OK. When it receives OK, it sends the string ATDT5551212 to dial the telephone. The expected string is CONNECT. If the string CONNECT is received the remainder of the script is executed. However, should the modem find a busy telephone, it will send the string BUSY. This will cause the string to match the abort character sequence. The script will then fail because it found a match to the abort string. If it received the string NO CARRIER, it will abort for the same reason. Either string may be received. Either string will terminate the chat script.

TIMEOUT

The initial timeout value is 45 seconds. To change the timeout value for the next expect string, the following example may be used:

```
    ""               , "ATZ"          ,
    "OK"             , "ATDT5551212" ,
    "CONNECT"        , "\\c"          ,
    "TIMEOUT"        , "10"           ,
    "ogin:--ogin:"   , "ppp"          ,
    "TIMEOUT"        , "5"            ,
    "assword:"       , "hello2u2"     ,
    0
```

This will change the timeout to 10 seconds when it expects the login: prompt. The timeout is then changed to 5 seconds when it looks for the password prompt.

The timeout, once changed, remains in effect until it is changed again.

Sending EOT

The special reply string of EOT indicates that the chat program should send an EOT character to the remote. This is normally the End-of-file character sequence. A return character is not sent following the EOT. The EOT sequence may be embedded into the send string using the sequence "\x04" (i.e. a Control-D character).

Escape Sequences

Most standard **chat** escape sequences can be replaced with standard C string escapes such as '\r', '\n', '\t' etc. Additional escape sequences may be embedded in the expect or reply strings by introducing them with *two* backslashes.

\\c Suppresses the newline at the end of the reply string. This is the only method to send a string without a trailing return character. It must be at the end of the send string. For example, the sequence "hello\\c" will simply send the characters h, e, l, l, o. (not valid in expect strings.)

Chapter 158. PPP Enabled Device Drivers

For PPP to function fully over a serial device, its driver must implement certain features. At present not all eCos serial drivers implement these features. A driver indicates that it supports a certain feature by including an `"implements"` line in its CDL for the following interfaces:

`CYGINT_IO_SERIAL_FLOW_CONTROL_HW`

> This interface indicates that the driver implements hardware flow control using the RTS and CTS lines. When data is being transferred over high speed data lines, it is essential that flow control be used to prevent buffer overrun.
>
> The PPP subsystem functions best with hardware flow control. If this is not available, then it can be configured to use software flow control. Since software flow control is implemented by the device independent part of the serial device infrastructure, it is available for all serial devices. However, this will have an effect on the performance and reliability of the PPP link.

`CYGINT_IO_SERIAL_LINE_STATUS_HW`

> This interface indicates that the driver implements a callback interface for indicating the status of various RS232 control lines. Of particular interest here is the ability to detect changes in the Carrier Detect (CD) line. Not all drivers that implement this interface can indicate CD status.
>
> This functionality is only needed if it is important that the link be dropped immediately a telephone connection fails. Without it, a connection will only be dropped after it times out. This may be acceptable in many situations.

At the time of writing, the serial device drivers for the following platforms implement some or all of the required functionality:

- All drivers that use the generic 16x5x driver implement all functions:
  - ARM CerfPDA
  - ARM IQ80321
  - ARM PID
  - ARM IOP310
  - i386 PC
  - MIPS Atlas
  - MIPS Ref4955
  - SH3 SE77x9
- The following drivers implement flow control but either do not support line status callbacks, or do not report CD changes:
  - SH4 SCIF
  - A&M AdderI
  - A&M AdderII
- All other drivers can support software flow control only.

Chapter 159. Testing

Test Programs

There are a number of test programs supplied with the PPP subsystem. By default all of these tests use the device configured by `CYGPKG_PPP_TEST_DEVICE` as the PPP link device.

`ppp_up`

This test just brings up the PPP link on `CYGPKG_PPP_TEST_DEVICE` and waits until the remote end brings it back down. No modem lines are used and the program expects a PPP connection to be waiting on the other end of the line. Typically the remote end will test the link using **ping** or access the HTTP system monitor if it is present.

If `CYGPKG_PPP_TESTS_AUTOMATE` is set, then this test attempts to bring PPP up at each of the baud rates specified in `CYGDAT_PPP_TEST_BAUD_RATES`. If it is not set then it will just bring the connection up at 115200 baud.

`ppp_updown`

This test brings the PPP link up on `CYGPKG_PPP_TEST_DEVICE` and attempts to **ping** the remote end of the link. Once the pings have finished, the link is then brought down.

If `CYGPKG_PPP_TESTS_AUTOMATE` is set, then this test attempts to bring PPP up at each of the baud rates specified in `CYGDAT_PPP_TEST_BAUD_RATES`. If it is not set then it will just bring the connection up at 115200 baud.

`chat`

This test does not bring the PPP link up but simply executes a chat script. It expects a server at the remote end of the link to supply the correct responses.

This program expects the **test_server.sh** script to be running on the remote end and attempts several different tests, expecting a variety of different responses for each.

`ppp_auth`

This test attempts to bring up the PPP link under a variety of different authentication conditions. This includes checking that both PAP and CHAP authentication work, and that the connection is rejected when the incorrect authentication protcol or secrets are used.

This test expects the **test_server.sh** script to be running on the remote end. For this test to work the `/etc/ppp/pap-secrets` file on the remote end should contain the following two lines:

```
eCos       *          secret       *
eCosPAP    *          secretPAP    *
```

The `/etc/ppp/chap-secrets` file should contain:

```
eCos       *          secret       *
eCosCHAP   *          secretCHAP   *
```

`isp`

This test expects the serial test device to be connected to a Hayes compatible modem. The test dials the telephone number given in `CYGPKG_PPP_DEFAULT_DIALUP_NUMBER` and attempts to log on to an ISP using the user name and password supplied in `CYGPKG_PPP_AUTH_DEFAULT_USER` and `CYGPKG_PPP_AUTH_DEFAULT_PASSWD`. Once the PPP connection has been made, the program then attempts to ping a number of well known addresses.

Since this test is designed to interact with an ISP, it does not run within the automated testing system.

`tcp_echo`

This is a version of the standard network **tcp_echo** test that brings up the PPP connection before waiting for the **tcp_sink** and **tcp_source** programs to connect. It is expected that at least one of these programs will connect via the PPP link. However, if another network interface is present, such as an ethernet device, then one may connect via that interface.

While this test is supported by the **test_server.sh** script, it runs for such a long time that it should not normally be used during automated testing.

`nc_test_slave`

This is a version of the standard network **nc_test_slave** test that brings up the PPP connection before waiting for the **nc_test_master** program to connect. It is expected that the master will connect via the PPP link.

While this test is supported by the **test_server.sh** script, it runs for such a long time that it should not normally be used during automated testing.

Test Script

The PPP package additionally contains a shell script (**test_server.sh**) that may be used to operate the remote end of a PPP test link.

The script may be invoked with the following arguments:

`--dev=<devname>`

This mandatory option gives the name of the device to be used for the PPP link. Typically `"/dev/ttyS0"` or `"/dev/ttyS1"`.

`--myip=<ipaddress>`

This mandatory option gives the IP address to be attached to this end of the PPP link.

`--hisip=<ipaddress>`

This mandatory option gives the IP address to be attached to the remote (test target) end of the PPP link.

`--baud=<baud_rate>`

This option gives the baud rate at which the PPP link is to be run. If absent then the link will run at the value set for `--redboot-baud`.

`--redboot`

If this option is present then the script will look for a `"RedBoot>"` prompt between test runs. This is necessary if the serial device being used for testing is also used by RedBoot.

`--redboot-baud=<baud_rate>`

This option gives the baud rate at which the search for the RedBoot prompt will be made. If absent then the link will run at 38400 baud.

`--debug`

If this option is present, then the script will print out some additional debug messages while it runs.

This script operates as follows: If the `--redboot` option is set it sets the device baud rate to the RedBoot baud rate and waits until a `"RedBoot>"` prompt is encountered. It then sets the baud rate to the value given by the `--baud` option and reads lines from the device until a recognizable test announce string is read. It then executes an appropriate set of commands to satisfy the test. This usually means bringing up the PPP link by running **pppd** and maybe executing various commands. It then either terminates the link itself, or waits for the target to terminate it. It then goes back to looking for another test announce string. If a string of the form `"BAUD:XXX"` is received then the baud rate is changed depending on the `XXX` value. If a `"FINISH"` string is received it returns to waiting for a `"RedBoot>"` prompt. The script repeats this process until it is terminated with a signal.

Part XLIV. lwIP - the lightweight IP stack for eCosPro

Table of Contents

Chapter 160. lwIP overview

Introduction

lwIP, short for lightweight IP, is an implementation of a standard Internet Protocol v4 and v6 networking protocol stack designed to operate in a resource-constrained environment. It was created in 2001 by Adam Dunkels of the Swedish Institute of Computer Science for his Master's thesis. The core lwIP code was released publically under an open licence.

The lwIP stack supports the IP, TCP, UDP, ICMP, IGMP, ARP, DHCP, AutoIP, DNS, SNMP, SLIP and PPP protocols, and there is a selection of APIs which applications can use to interact with it. As well as being designed from the outset to have a low memory footprint, it also gains many of its lightweight properties from being highly configurable. This makes it an excellent choice for integration into eCos.

This documentation describes lwIP and properties specific to its port to eCosPro. The usage, configuration and tuning of lwIP will also be discussed. Many of the concepts discussed here will require some understanding of the inherent underlying properties of the TCP, UDP and IP protocols. This documentation cannot substitute for an introduction to TCP/IP stacks and protocols generally, and it is recommended that where needed the reader seeks out a good reference book, such as:

- *TCP/IP Illustrated, Volume 1: The Protocols*, W. Richard Stevens, published by Addison-Wesley Professional, ISBN-10: 0-201-63346-9, ISBN-13: 978-0-201-63346-7.
- *Internetworking with TCP/IP: volume 1*, Douglas E. Comer, published by Prentice-Hall, ISBN-10: 0-131-87671-6, ISBN-13: 978-0-131-87671-2.

or one of the many online guides:

- The TCP/IP Guide
- Network Sorcery RFC Sourcebook
- Wikipedia

lwIP sources and ports

lwIP is portable and by no means specific to eCos. It has an active development community and undergoes continuous development of its core code, focussed around its project page on the Savannah development site run by the Free Software Foundation.

In order to provide a robust, feature-rich, and commercially supportable solution for eCosPro, the eCos support has been overhauled by eCosCentric® to work with the latest lwIP releases.

This documentation corresponds solely to the eCosPro port of lwIP, and the usage, configuration system and operation differs in many regards from that in other code bases.

Warning

As detailed in the section called “Port status” the current eCosPro lwIP is using a much newer lwIP source base with substantial changes from previous eCosPro lwIP offerings. As such some CDL compatibility issues will arise if attempts are made to use old `.ecc` configuration files.

Either a fresh configuration can be created, and options re-selected as desired, or prior to switching to the newer source world, whilst still configured to use the older `ECOS_REPOSITORY` the eCos configtool can be used to export (`File->Export`) the configuration to a `.ecm` (minimal configuration) file. Then after switching to the new `ECOS_REPOSITORY` source tree the eCos configtool can be used to import (`File->Import`) the created `.ecm` file.

External documentation

A limited amount of publically available documentation is available for the lwIP project. Some of it has been incorporated into this manual. The following lists useful documentation known about at the time of writing:

- Adam Dunkel's Master's Thesis - the original description of lwIP design and operation, but now somewhat outdated.
- Report by Adam Dunkel into the design and implementation of lwIP, including a sequential API reference, and example code. Largely still applicable to current lwIP, albeit incomplete. A copy of the PDF version may be found in the `doc/` subdirectory of the lwIP package in the eCosPro source repository (`packages/net/lwip_tcpip/`*`VERSION`*`/doc/dunkels-lwip.pdf` relative to the base of the eCosPro installation).
- The lwIP Wiki site provides a good introduction to many lwIP features, and provides links to related documentation.
- Text description of the lwIP raw API. A copy of the version at time of writing may be found in the `doc/` subdirectory of the lwIP package in the eCosPro source repository (`packages/net/lwip_tcpip/`*`VERSION`*`/doc/rawapi.txt` relative to the base of the eCosPro installation).
- Text description of the sys_arch porting abstraction layer. A copy of the version at time of writing may be found in the `doc/` subdirectory of the lwIP package in the eCosPro source repository (`packages/net/lwip_tcpip/`*`VERSION`*`/doc/sys_arch.txt` relative to the base of the eCosPro installation).

Licensing

The lwIP core code is distributed under a 3 clause BSD-style license. Confirmation has been received from Adam Dunkels that the existing public lwIP documentation is also covered by this license.

The original public eCos port included elements distributed under the eCos license.

As a result of the changes made by eCosCentric, portions of the eCos port of lwIP in eCosPro are covered by the eCosPro License.

Chapter 161. Basic concepts

Structure

lwIP has been incorporated into eCos as a single package (`CYGPKG_NET_LWIP`) which contains all the core lwIP code and the bulk of the eCos port. The remaining elements that constitute the eCos port can be found in the generic Ethernet driver package (`CYGPKG_IO_ETH_DRIVERS`) and so is only relevant when using an Ethernet-based network card rather than SLIP or PPP. Support for SLIP and PPPoS (PPP-over-Serial) is layered over the standard eCos serial driver API.

The port to eCos has been constructed using the sys_arch porting abstraction within lwIP, and this allows the eCos port to be cleanly separated from the core lwIP code, although it still remains in the lwIP eCos package.

Application Programming Interfaces (APIs)

There are three different APIs which may be used by applications to interface with the stack: the raw API, the sequential API, or the BSD sockets compatibility API. Each one in turn builds on the functionality provided by the previous API. This allows users the flexibility of choosing a fairly bare implementation to squeeze the maximum out of the available resources; or to use a more powerful API to simplify application coding and reduce time-to-market. Note that despite the presence of the BSD sockets compatibility API, the lwIP stack implementation is not in any way related to the other BSD-derived TCP/IP stacks present in eCos.

The raw API provides an event-based interface with callbacks directly into the application in order to handle incoming/outgoing data and events. There is no inter-thread protection and can only operate with a single thread of execution.

The sequential API is a more traditional style of network interface API which provides functions that may be called synchronously to perform network operations, and where those operations can be considered complete (or will complete asynchronously with no further application interaction) when those functions return to the application.

When using the sequential API (or the BSD sockets API which is layered on top of it), lwIP maintains its own internal thread for network data processing and event management. This is usually referred to as lwIP's TCP/IP thread (even though that is a slight misnomer). This thread uses mailboxes to communicate with application threads, and semaphores to provide mutual exclusion protection.

The BSD sockets compatibility API included in lwIP provides a subset of the Berkeley sockets interface introduced in the BSD 4.2 operating system. The Berkeley sockets interface, recently standardised by ISO/IEC in POSIX 2003.1, will be familiar to those who have developed network applications on Linux, POSIX, UNIX or to a limited extent Windows with Winsock.

As the BSD sockets API provided as part of lwIP is only a subset of the full sockets, it should be considered only as an aid to development or for when porting existing code. It should not be considered as a drop-in replacement for applications written for a complete BSD network stack implementation which supports a wealth of features that do not exist in, and in many cases would be inappropriate for, a low footprint implementation such as lwIP.

Protocol implementations

lwIP implements a variety of protocols. Support for each protocol can be individually included in or excluded from the configuration, subject to dependency constraints. The protocol implementations are mostly compartmentalised into separate source modules. Support exists for TCP, UDP, UDP-Lite, IP (IPv4 and IPv6), ICMP, ICMP6, ARP when using Ethernet, IGMP, DHCP, AutoIP and Stateless AutoConfiguration, DNS, MLD, ND, SNMP, SNTP, TFTP, SLIP and PPP.

In most cases functionality has been intentionally restricted to avoid "bloat" (unnecessary features increasing resource use), or in some cases completely omitted. This is covered in slightly more detail in the section called “Limitations”.

Packet data buffers

lwIP does not only possess features allowing it by itself to maintain a small footprint, but also has design aspects which allow it to work with the application to reduce footprint. One important case of this is lwIP packet data buffers.

Packet data buffers in lwIP are termed *pbufs*. Pbufs can be chained together in fairly arbitrary ways to create a *pbuf chain*. The idea is that the application can pass the stack a pbuf of data to transmit, and the stack can prepend and possibly append other pbufs to encapsulate the data in protocol headers/footers without having to copy the data elsewhere, thus saving resources. In some cases, depending on precisely how the pbuf was allocated, the stack may even be able to fit protocol headers inside the pbuf passed to it. It also means that the application can itself provide data allocated in differing ways and from different locations, but assembled together as a pbuf chain. This will ensure that the data is treated as if it were all allocated contiguously. When using the sequential API, the underlying pbufs are wrapped in a *netbuf* construct in order to provide a simpler API to manipulate data in buffers; but the underlying functionality remains based on pbufs.

When a pbuf is created, it must be one of a variety of types:

PBUF_RAM This is a conventional buffer, which points to data allocated from a pool in RAM managed by lwIP. On creation the buffer size must be given.

PBUF_ROM This is a buffer pointing to immutable read-only data. This allows fixed literal data to be stored in ROM/Flash rather than using up precious RAM. Note that data pointed at by a PBUF_ROM pbuf does not literally have to point at read-only memory. All it means is that the data must not change, even if control has returned to the application. The pbuf data may still be being referenced as part of a packet waiting in a queue to be transmitted, or more often, waiting in a queue in case retransmission is necessary.

> ⚠ **Caution**
>
> Not all architectures will allow ethernet transfers direct from ROM, so the underlying hardware device driver may need to perform copying of data as required.

PBUF_REF This is a buffer pointing to mutable data, passed in by reference. This means data provided by the application allocated from its own resources, and which could change in the future. This differs from PBUF_RAM packets in that the data is allocated by the application, and not from lwIP's PBUF_RAM buffer memory pool. As the application could change the data after control is returned to it, if lwIP finds it must enqueue the pbuf, it will internally copy the data to a new PBUF_RAM. The benefits of this type of packet occur when the packet does not need to be enqueued, and so no PBUF_RAM pbuf needs to be allocated.

PBUF_POOL The buffer is allocated as a chain of fixed size pbufs from a fixed size pool.

Configurability

lwIP was designed from the outset to have a low resource footprint. One of the techniques it uses to achieve this goal is its high level of configurability.

lwIP allows both coarse- and fine-grained control of functionality. Large sections of potentially unused functionality can be selected to be removed by the user, including entire protocol stacks. Such examples of removable coarse-grained functionality include UDP, TCP, SLIP, PPP stacks, ethernet/ARP support, IP fragmentation and/or reassembly, or the sequential API.

It has a somewhat modular and layered design to assist with this. It is intentionally only somewhat modular: other TCP/IP stacks have strictly enforced interfaces and abstractions between protocol layers. These abstractions are frequently cumbersome and can result in unnecessary resource implications. lwIP deliberately violates some of these protocol interface layering abstractions where doing so could improve resource utilization. An example is reserving an estimated appropriate amount of space for protocol headers when constructing packets, where the choice of protocol dictates the amount of space.

Where lwIP really stands out is in its fine-grained control over the various pools of resources. Most resources are compartmentalised into fixed size memory pools to allow sizes to be constrained deterministically. The application designer will know, or can choose, the maximum number of network connections which are to be supported depending on application requirements. They also know the level of data throughput required for transmission or reception and can control the levels of the necessary resources appropriately, such as numbers of buffers (separately for incoming or outgoing packet data) and their sizes, numbers of protocol control blocks, TCP window sizes and more.

In this way, application designers can choose a configuration that maximises performance within the limitations of available memory. Clearly, the more constrained the memory, the greater the potential for adverse consequences for performance, or the number of supported connections. However, it should be realised that even with copious quantities of memory resources available to lwIP, it cannot be expected that a stack intentionally designed from the outset to be sparing with memory will perform as well as a stack intentionally designed from the outset for high performance. Nevertheless careful tuning of lwIP almost always results in significant performance improvements.

A simple real-world test application, from a target platform with only a total of 128K of RAM, would when performing a simple test transmitting multiple 1400-byte packets obtain a throughput of >1000k/s. However, the same platform configuration sending multiple 8192-byte packets would see the throughput drop to <100k/s since fragmentation and buffer availability now impact the lwIP performance. System designers need to consider how the application makes use of the available lwIP APIs in conjunction with the resources available to maximise application network throughput.

Limitations

As already mentioned, lwIP does not seek to provide a complete implementation of a TCP/IP stack providing the same level of functionality provided in large OSes such as Linux, Windows, *BSD, etc. While some aspects are controlled by configuration, in other cases functionality is intentionally limited to fit the design requirements of a compact footprint.

While a complete list of the limitations would be too numerous to enumerate, here are some of the most relevant ones to be aware of:

- Retransmission and windowing algorithms are implemented simply, at the expense of some performance.
- Routing is simplified - one gateway per interface. IP forwarding follows the same rules as the host itself.
- No support for NAT, nor packet filtering.
- The TCP, DHCP and IP protocols can contain options in their packets. Relatively few of these options are supported by lwIP.
- IPv4 Path MTU discovery (from RFC1191) is not supported. Ordinarily it is used to avoid fragmentation of packets resulting from the maximum MTU of an intermediate link between source and destination being smaller than the packet sizes actually transmitted. lwIP does however allow the TCP Maximum Segment Size (MSS) to be configured.
- No complex data structures, caches and search trees to optimise speed. Generally simple lists are used.
- Thread safety (for the sequential and BSD compatibility API) is implemented in a very simple form. Individual connections should not be operated on by multiple threads simultaneously. The mutual exclusion that is provided is at a very coarse grain - the network processing operations themselves are not multi-threaded.
- Most ICMP packet types are ignored.
- If IP fragmentation and reassembly support is enabled then a limited sequence of IP fragments can be reassembled at one time (controlled by lwIP configuration options). If the number of active sequences supported is exceeded then packet fragments for new sequences are simply dropped, with the hope that a subsequent retransmission may be successful. Received IP fragments are allowed to be reassembled out of order however.
- The BSD sockets compatibility API does not implement all socket options, API functions, nor API semantics.
- Error handling for application errors is frequently only handled with asserts - used only during debug builds during development, allowing for smaller production code in release builds.

- The TCP persist timer is not implemented. If a remote peer has filled its receive window and as a result lwIP stops sending, then when the remote peer processes more data it sends an ACK to update the window. However if that ACK is lost, then if data is entirely unidirectional (lwIP to remote host), the connection could stall. In practice, this has not been something people have experienced really.
- TCP data is not split in the unsent queue, resulting in somewhat inefficient use of receiver windows.
- The DNS client support only returns IPv4 addresses.
- The SNMP agent only provides traps to IPv4 addresses.
- The SNTP client provides a minimal `SNTPv4` implementation.
- The lwIP IPv6 implementation does not currently track router advertisement routeinfo information. The IPv6 routing simply uses the normal /64 prefix for matching destination addresses against acquired (source) addresses for each indivdual lwIP network interface. If a destination address cannot be matched against an acquired source /64 address then the routingerror (`ERR_RTE`) code is returned. This is not normally a limitation when using link-local or global addresses, but if an organisation is using unique-local addressing the lwIP stack by default will limit addressing to destinations on the same subnet (i.e. the matching /64 prefix). However, an eCos specific extension exists for supporting unique-local addresses, where the CDL option `CYGNUM_LWIP_IPV6_UNIQUELOCAL_MASK` can define the number of global prefix bits which are matched (from /48 to /64). This option can be configured to allow destination addresses for other unique-local subnets to be matched against the specific /64 interface unique-local address.

There are many more examples.

If a lwIP Direct device driver is being used then see the section called “GDB/RedBoot” regarding limitation of remote network GDB/RedBoot debugging.

Quick Start

Incorporating lwIP into your application is straightforward. The essential starting point is to incorporate the lwIP eCos package (`CYGPKG_NET_LWIP`) into your configuration.

This may be achieved directly using **ecosconfig add** on the command line, or the **Build->Packages…** menu item within the eCos Configuration Tool. If you wish to support Ethernet devices, you will also need to include the Common Ethernet Support (`CYGPKG_IO_ETH_DRIVERS`) eCos package. For SLIP/PPPoS support, you will need to enable the Hardware serial device drivers (`CYGPKG_IO_SERIAL_DEVICES`) configuration option within the Serial device drivers (`CYGPKG_IO_SERIAL`) eCos package.

Note

When using serial devices it is important to ensure the I/O driver configuration provides the necessary buffers and/or hardware flow-control to avoid the possibility of PPPoS/SLIP RX data loss.

Alternatively, as a convenience, configuration templates have been provided to permit an easy starting point for creating a configuration incorporating lwIP. Two templates are provided: `lwip_eth` for those intending to use lwIP with Ethernet; and `lwip_ppp` for those intending to use lwIP with PPP. These may be used either by providing the template name as an extra argument on the command line to **ecosconfig new**; or with the **Build->Templates…** menu item within the eCos Configuration Tool. Both these templates are basic, incorporating only those packages which are essential for lwIP operation.

At this stage it would be appropriate to tailor the lwIP package configuration to the application requirements. At a minimum it would be appropriate to consider whether a static IP address, or a dynamic IP address served from a DHCP server, is required. Note that if RedBoot is used on the target and incorporates network support, then you must not give lwIP and RedBoot the same IP address. For the same reason, you must not configure both lwIP and RedBoot to obtain an IP address via DHCP.

Prior to coding your application to perform lwIP stack operations using its APIs, the stack must be initialised. This does not happen automatically, and instead a C function must be called:

```
int cyg_lwip_init ();
```

The function declaration can be obtained by including the `network.h` header file:

```
#include <network.h>
```

`cyg_lwip_init` returns 0 on success and non-zero on failure. Note that 0 may be returned even if no network interfaces were successfully initialised. This is because in some cases interfaces are brought up asynchronously in any case, devaluing such an error indication; and because an interface not coming up may be expected. If the application needs to determine the status of interfaces, it should query the stack using the `netif_*` functions using the `<lwip/netif.h>` header file.

The `cyg_lwip_init` function **must** be called from a thread context. Raw API users need not call this function, although they instead will be required to perform their own stack initialization. Consult the raw API documentation for more information.

Note

The `cyg_lwip_init` function, depending on the configuration, may block for some time waiting for interfaces to acquire network addresses. Alternatively the:

```
int cyg_lwip_init_nowait ();
```

function can be called to perform just the necessary low-level initialisation, without the extra addresswait functionality. The `cyg_lwip_init` function itself uses the `cyg_lwip_init_nowait` routine prior to waiting for network addresses to be assigned.

Alternatively, the (weak) helper function:

```
void init_all_network_interfaces ();
```

is provided. By default it just calls the lwIP specific `cyg_lwip_init` initialisation, but it may be overridden by drivers or run-time support if alternative initialisation strategies are required.

Note

The `init_all_network_interfaces` name is the same as used by the alternative `CYGPKG_NET` BSD networking world.

If obtaining an address via DHCP it can be convenient to enable the network interface debugging configuration option within lwIP (`CYGDBG_LWIP_DEBUG_NETIF`). This will allow the IP address which was set to be viewed on the diagnostic output console. Similarly the helper function:

```
int cyg_lwip_netif_print_info (netif, pf);
```

can be called after `cyg_lwip_init` to output specific interface address information via the supplied `printf`-alike routine. For example using `diag_printf` for the *`pf`* parameter will display the address information on the diagnostic output console without having to enable the network interface debugging feature.

Chapter 162. Port

Port status

The eCos port of lwIP in eCosPro is based on the main lwIP Savannah git code base, with modifications consisting of both bug fixes and feature enhancements being made to the lwIP core code by eCosCentric.

The port requires the eCos kernel (`CYGPKG_KERNEL`) for now. The main reasons for this are because the ethernet driver and serial driver implementations have dependencies on interrupts and non-kernel interrupt support is tricky; and that it is only really feasible in the lwIP core code to avoid a multi-thread OS if solely using the raw API. And when using the raw API, the application would have to be responsible for polling the underlying device driver (e.g. Ethernet) in any case.

Some eCos Ethernet drivers may have alignment constraints on packet data. This is usually not a problem, however it can affect PBUF_ROM packets, whose alignment is dictated by the application. Therefore the application must ensure only appropriately aligned PBUF_ROM packets are passed to lwIP, as appropriate for the hardware-specific Ethernet device driver.

lwIP's BSD sockets compatibility API is completely separate from the socket and file descriptor interface provided by the eCos File I/O (`CYGPKG_IO_FILEIO`) package. As such, network packages which rely on semantics such as being able to read and write both files and sockets with that API, cannot work with lwIP at the present time. This includes the httpd, DNS, SNTP and FTP client packages. The NET-SNMP package uses BSD stack-specific APIs and so also cannot work with lwIP, though lwIP can be configured with its own internal SNMP agent providing MIB-2 support. Note that an example httpd server written using the lwIP raw API is included in the `tests/` subdirectory of the lwIP eCos package.

For convenience when using the BSD sockets compatibility API, including the `network.h` header file:

```
#include <network.h>
```

This allows access to the API. This also has the benefit of potentially allowing interchangeable application code if switching between the lwIP BSD socket compatibility API and the real BSD stack port in eCos.

lwIP does not attempt to provide a cleanly delineated namespace for lwIP functions. This could make it difficult to port legacy code where there is a chance of conflicting names and symbols, both functions and data. Care is required here.

Note

The serial-based SLIP and PPP protocols should be functional, however they have not been well tested, and so are not supported under the terms of incident support in eCosPro.

There is only convenient configuration for a single SLIP and/or PPP interface. Multiple interface support is planned for some future point.

Implementation

The following sections provide an overview of how the port is structured regarding the interface between eCos and the core lwIP implementation.

System Configuration

The normal lwIP approach of the user supplying a `lwipopts.h` header file that provides manifests to override the standard lwIP `opt.h` header file is used to configure the main stack features. For eCos the `lwipopts.h` is provided as part of the `CYGPKG_NET_LWIP` package along with the lwIP generic sources. Note: For eCos the `lwipopts.h` also contains definitions for some lwIP features that do not yet have defaults defined within `opt.h`.

The eCos `lwipopts.h` implementation itself sets the majority of the lwIP feature control options based on the standard eCos CDL (`.ecc`) configuration world.

System Source

The `CYGPKG_NET_LWIP` package provides some eCos specific functionality in the `src/ecos/` directory.

- `lwip_ecos_init.cxx`

  This source file provides two functions that are normally called by the application. The function cyg_lwip_init is needed to initialize the lwIP network stack, and cyg_lwip_netif_print_info can optionally be called to output network interface address information.

  If `CYGINT_IO_ETH_DRIVERS_PHY_EVENTS` is configured to provide PHY event notification support then the functions cyg_net_eth_phy_ctx_acquireand cyg_net_eth_phy_dsr are available for network device drivers to manage per-interface event notification between the driver and lwIP TCP/IP stack layers.

- `sys_arch.cxx`

  This source file implements the majority of the run-time support needed by lwIP to execute under eCos, which mainly covers:

  - Mailbox support, mapping lwIP sys_mbox_t to eCos Cyg_Mbox objects.
  - Semaphore support, mapping lwIP sys_sem_t to eCos Cyg_Counting_Semaphore objects.
  - Thread support, mapping lwIP sys_thread_t to eCos Cyg_Thread objects.
  - Mutex support, mapping lwIP sys_mutex_t to eCos Cyg_Mutex objects.
  - Timer conversion support for converting between real-time-clock and millisecond ticks. The cyg_lwip_tick_to_msecand cyg_lwip_msec_to_tick functions may be useful to applications.

- `sio.c`

  Some serial I/O utility routines for SLIP and PPPoS support.

  Only a single serial interface is support and is accessed via a named serial I/O device (either `CYGDAT_LWIP_PPP_DEV` for PPPoS, or `CYGDAT_LWIP_SLIP_DEV` for SLIP). The I/O device is configured with non-blocking RX, and blocking TX as per the requirements of the lwIP APIs. The lwIP package CDL does not enforce any specific serial configuration (due to the varied differences between architectures and driver feature sets), so the developer is responsible for ensuring a suitable serial I/O driver configuration.

Threads

The lwIP network stack is mostly thread-safe for sockets and the sequential API, but not for the raw API. The most important caveat is that even for the sequential API it is **NOT** thread-safe to access the **same** BSD-style socket, or netconn, from multiple threads.

The default for eCos is for the run-time support to provide the TCP/IP helper thread, which is enabled via the lwIP `NO_SYS=0` manifest definition. The lwIP API thread will be created even if the `CYGFUN_LWIP_SEQUENTIAL_API` option is not enabled, unless over-ridden by the `CYGFUN_LWIP_NO_SYS` option.

Note

Disabling the sequential API option does not disable the TCP/IP helper thread for backwards compatibility with previous eCos configurations where the helper thread is expected even if the option `CYGFUN_LWIP_SEQUENTIAL_API` is disabled.

When the TCP/IP helper thread is required, the eCos lwIP run-time support will call the lwIP `tcpip_init()` function as part of the initialization sequence. The created eCos thread is named with the configured `CYGDAT_LWIP_TCPIP_THREAD_NAME` value, and with the priority as configured by `CYGNUM_LWIP_NETWORK_THREAD_PRIORITY`.

Providing this thread allows simple raw API applications to interact with the eCos ethernet device drivers.

Note

The, mutually exclusive, SLIP and PPPoS features require the `CYGFUN_LWIP_SEQUENTIAL_API` support, and hence cannot be used with a `CYGFUN_LWIP_NO_SYS` configuration.

Caution

If the option `CYGFUN_LWIP_NO_SYS` is enabled then the system overhead required for suitably configured raw API applications will be minimised. It is the responsibility of the application to implement the necessary thread or polled model as desired for its specific, required, functionality.

If SLIP support is configured then a handler thread is also created during the system initialization. As with the main lwIP thread the name and the priority of the thread can be set in the eCos configuration.

The current PPPoS implementation does not use a seperate helper thread, with the required RX work being done as part of the normal TCP/IP helper thread.

Note: At some points within the lwIP network stack the eCos scheduler is locked. Whilst this is only for short sections of code, this could disrupt real-time behaviour.

Thread-safety considerations regarding lwIP:

- the network stack is thread-safe in general

  However, individual sockets should **not** be shared between different eCos threads simultaneously (e.g. two threads doing overlapping `recv()`s).

- uses semaphores

  So priority inversion is possible.

- coarse locking granularity

  The whole lwIP network stack can remain locked for a long time whilst an operation is processed. This could exacerbate any priority inversion issues.

- core lwIP code (and hence the raw API) is not thread-safe

- mailboxes are used extensively

  Mailboxes are used to funnel all threads' API requests into the lwIP context to avoid synchronisation problems, but this can cause priority inversion.

  Also the effects of `CYGNUM_KERNEL_SYNCH_MBOX_QUEUE_SIZE` should be considered, since a very shallow mailbox depth could lead to client threads blocking waiting to post requests.

For eCos the timer support has been extended to provide absolute timeout functionality to provide consistent timeout behaviour. The manifset setting `SYS_TIMEOUT_ENDTIME=1` is enforced for eCos builds. If the previous lwIP functionality is required then `SYS_TIMEOUT_ENDTIME=0` and `SYS_TIMEOUT_DELTA=1` should be manually configured in the `lwipopts.h` header file.

Caution

If the older SYS_TIMEOUT_DELTA=1 support is enabled then be aware that the default lwIP sources do not work on absolute times, and as such the lwIP idea of what time is will go wrong if the single lwIP thread is pre-empted for too long. In this case the lwIP code should really be executed in a high priority thread, but this is not enfored by default, as there may be application threads in the system which must run at a higher priority. This laxness of time management within lwIP can lead to problems with some sub-systems. For example, accuracy of DHCP lease-time renewal requests

Extensions

When creating a deeply-embedded network application some features of lwIP can be used by the general application code to save on the code footprint of duplicating similar support. One major area that can easily be made use of in such a way is the lwIP memory allocation support.

If the CDL option CYGFUN_LWIP_MEMP_USE_CUSTOM_POOLS is enabled then the user can supply the lwippools.h header file accessible to the lwIP build. This header file is included by lwIP when defining the memory pool structures, and can include application specific *custom* pools. The following example implementation provides a new application specific 16 entry memory pool containing 64-byte buffers:

```
/*> lwippools.h <*/

/* NOTE: We do NOT have the standard header file one-time inclusion conditional
   checks since this source file is referenced multiple-times, with different
   macro definitions, depending on the part of the memory pool support being
   instantiated. */

LWIP_MEMPOOL(APP_BUFFER, 16, 64, "APP_BUFFER")

/*> EOF lwippools.h <*/
```

The custom memory pool space is located within the configured lwIP memory pool space. The pool can then be accessed via the memp_malloc() and memp_free() as normal:

```
pointer = (someptr *)memp_malloc(MEMP_APP_BUFFER);
/* Do some work with the buffer … */
memp_free(MEMP_APP_BUFFER, pointer);
```

For eCos the memory pool allocation support is protected using the SYS_ARCH_PROTECT (DSR) serialization support code, allowing the calls to be made from any thread.

Similarly if lwIP is configured with CYGFUN_LWIP_MEM_USE_POOLS then it will use fixed size memory pools instead of a heap for the mem_malloc() calls. The lwippools.h header file can then be used to define the specific fixed size memory pools to be used by including suitable LWIP_MALLOC_MEMPOOL() macro calls. For example the following fragment will define a set of fixed size pools:

```
/* Define three pools with buffer sizes of 256, 512, and 1512 bytes  respectively. */
LWIP_MALLOC_MEMPOOL_START
LWIP_MALLOC_MEMPOOL(20,  256)
LWIP_MALLOC_MEMPOOL(10,  512)
LWIP_MALLOC_MEMPOOL( 5, 1512)
/* More pools can be added as required. */
LWIP_MALLOC_MEMPOOL_END
```

eCos API reference

Name

cyg_lwip_init — Initialise lwIP network stack

Synopsis

#include <arch/cc.h>

`int` **`cyg_lwip_init`** `();`

Description

This function should be called by the eCos application at startup. It performs the initialization of the lwIP network stack and any configured network device drivers.

If either of the IPv4 DHCP `CYGSEM_LWIP_DHCP_WAIT_DHCP_COMPLETE` or IPv6 stateless autoconfiguration `CYGSEM_LWIP_IPV6_AUTOCONFIG_WAIT_COMPLETE` options are enabled then this initialization routine will wait for up to the maximum of the configured timeouts for the network interfaces to acquire the relevant addresses. If the network services are slow in providing a required address then it is possible that this function will exit with the interface not yet reachable from the network, so applications should always validate interfaces after calling this initialization routine. The wait support is provided by this routine to ensure a common implementation. If required the function can be called again to re-wait for the configured timeouts, but it should be noted that subsequent calls do **not** re-initialize the lwIP stack.

Return value

The value 0 is returned on completion of the initialization. The value 1 is returned if the routine has previously been called, and is **not** an indication of a failure to initialise.

Name

cyg_lwip_netif_print_info — Output network interface address information

Synopsis

#include <arch/cc.h>

```
int cyg_lwip_netif_print_info (iface, pf);
```

Description

This helper function provides a common method for applications to output network address information for the specified *iface* interface. It can be used (see the section called "Quick Start") to provide user feedback of acquired addresses. It can easily be used with test programs by supplying `diag_printf()` as the *pf* output function. Supplying an application specific routine however can allow for the fixed format address string to be output via an alternative user interface.

The network interface pointer for passing to this function in the *iface* parameter can be obtained from the lwIP function:

```
struct netif *netif_find(char *name)
```

The *name* passed is the registered two-character interface name. Alternatively if the default network interface is required it can be directly referenced via the exported struct netif *netif_default variable.

Return value

The value 0 is returned on success. The value 1 is returned if there was a problem with the interface, e.g. the link is not up, or the address associated with the interface has not yet been set.

Name

cyg_net_eth_phy_ctx_acquire — Allocate PHY event context

Synopsis

#include <cyg/io/eth/eth_drv.h>

`void *`**`cyg_net_eth_phy_ctx_acquire`**` (`*`priv`*`);`

Description

To allow per-interface PHY notification, the client network driver will call this function to register the passed *`priv`* reference, and to be given an abstract handle which is subsequently passed to the cyg_net_eth_phy_dsr function when the driver needs to notify the lwIP stack of a PHY event. For lwIP this passed value **MUST** be a structnetif*netif (network interface) reference.

Return value

A `NULL` value indicates a system error, otherwise it is the abstract handle used in the driver->lwIP PHY event notification. The caller of this function does not need to interpret the returned value. Internal to the lwIP stack the value returned depends on whether the lwIP world is configured to use a helper thread. For `NO_SYS` (true raw) configurations without a helper thread this function will just return the passed *`netif`* parameter since that is all that is needed for the driver->lwIP support. When a helper thread is being used (e.g. sequential API) then it will be a pointer to the message used to communicate the PHY event information in a thread-safe manner.

Name

cyg_net_eth_phy_dsr — Notify lwIP stack of PHY event

Synopsis

#include <cyg/io/eth/eth_drv.h>

```
void cyg_net_eth_phy_dsr (ctx);
```

Description

This function is called when the underlying network device driver needs to notify the lwIP TCP/IP stack of a link (PHY) event. Primarily this would be used to report link up or down events as appropriate, since certain TCP/IP operations may need to occur when a network connection is re-established (e.g. ARP packets, address negotiation, etc.).

The *ctx* parameter should be the pointer returned by the cyg_net_eth_phy_ctx_acquire call.

Name

cyg_lwip_tick_to_msec — Convert eCos kernel clock ticks to millisecond count

Synopsis

#include <arch/sys_arch.h>

`u32_tcyg_lwip_tick_to_msec (ticks);`

Description

Convert eCos kernel clock *ticks* value to a millisecond count.

Return value

The number of milliseconds corresponding to the supplied *ticks* value.

Name

cyg_lwip_msec_to_tick — Convert millisecond count to eCos kernel clock ticks

Synopsis

#include <arch/sys_arch.h>

`cyg_tick_countcyg_lwip_msec_to_tick (msecs);`

Description

Convert millisecond count to eCos kernel clock ticks.

Return value

The eCos kernel clock ticks corresponding to the supplied *msecs* millisecond count.

Name

cyg_lwip_statistics — Statistics output

Synopsis

#include <arch/cc.h>

```
void cyg_lwip_statistics (outfn);
```

Description

This function is provided mainly for debugging and development tuning, since it may be useful to monitor lwIP resource usage over various test scenarios when tuning an eCos lwIP configuration.

The function is a NOP unless the lwIP statistics gathering support is configured. The main configuration option `CYGDBG_LWIP_STATS` controls the availability of the statistics feature, with sub-options then being available to control specific sub-section statistics gathering.

The *`outfn`* is the printf-style varargs routine used to output the human-readable information. For example, a simple usage would be to pass the `diag_printf` function so that the statistics are output on the same diagnostic channel as any other test/debug information.

Chapter 163. Configuration

This chapter shows how to include the lwIP networking support into an eCos configuration, and how to configure it once installed. This subject was briefly covered in the section called “Quick Start”.

Configuration Overview

The lwIP networking stack is contained in a single eCos package `CYGPKG_NET_LWIP`. However, it depends on the services of a collection of other packages for complete functionality:

CYGPKG_KERNEL

> The main eCos kernel package. This provides the main run-time infrastructure as needed by the lwIP stack. For example, support for threads, semaphores, mailboxes, etc.

CYGPKG_ISOINFRA

> ISO C and POSIX standards infrastructure package. This provides access to many run-time utility routines. For example `rand_r()`.

CYGPKG_ERROR

> Error package. This provides access to the common eCos error and status codes.

CYGPKG_LIBC_STDLIB

> General support library. This provides general ISO C utility functions to the lwIP system.

CYGPKG_LIBC_STRING

> Strings library. This provides the string and memory move and compare routines used by the lwIP system.

CYGPKG_LIBC_I18N

> Internationalization library. This provides character interpretation support routines.

CYGPKG_IO_ETH_DRIVERS

> The common ethernet device driver support package. This is only required when lwIP is being configured for ethernet support.

CYGPKG_IO_SERIAL_DEVICES

> The serial device driver support package. This is only required when lwIP is being configured for SLIP or PPPoS support.

To add the lwIP support to a configuration, it is necessary to add the packages listed above as appropriate. This is best done by using a template file. Two examples of such templates files are provided:

lwip_eth   Provides a `current.ect` file containing the packages necessary to add ethernet lwIP support to any configuration.

lwip_ppp   Provides a `current.ect` file containing the packages necessary to add lwIP serial-based interface support to any configuration.

In addition to the packages listed above, hardware-specific device driver packages will be needed for ethernet devices to be used. These device drivers are usually part of the target description in the eCos database and will be enabled if the `CYGPKG_IO_ETH_DRIVERS` package is included.

For some target platforms a choice of device driver will be available. The lwIP driver model section of the CYGPKG_IO_ETH_DRIVERS_LWIP option may allow either a Direct or Standard driver to be selected. Direct are lwIP only device drivers designed for better performance on lower resource systems, but with some limitations on features supported (for example remote network debugging as mentioned in the section called "GDB/RedBoot"). The Standard drivers use the eCos standard ethernet driver interface allowing the device driver to be used for configurations using other TCP/IP stacks (e.g. FreeBSD) as well as for lwIP configurations.

Note: If lwIP is being configured to provide POSIX-style names for some socket support operations then the eCos package CYGPKG_IO_FILEIO should not normally be enabled in the eCos configuration at the same time, since care needs to be taken to avoid name-space clashes.

Configuring the lwIP stack

Note: For a low-level brief overview of how the lwIP source accesses the configured features see the section called "System Configuration".

Once added to the configuration, the lwIP package has a large number of configuration options. There are too many configuration options to go into full detail in this section, though the major eCos port specific options and fundamental support options are detailed. The configuration tool can be used to examine the hierarchy of the complete set of lwIP configuration options.

Stack size for system threads (CYGNUM_LWIP_THREAD_STACK_SIZE)

> The eCos lwIP implementation uses this fixed value as the stack size for all the lwIP system threads.
>
> If this value is set too low then incorrect operation can result due to stack overflow. The value should be configured to be large enough to cover the target platform worst-case stack requirement. The Thread Information documentation provides an overview of the `cyg_thread_measure_stack_usage()` that can be used to monitor and tune the stack requirements of the network application.

Network thread priority (CYGNUM_LWIP_NETWORK_THREAD_PRIORITY)

> This value defines the main lwIP network thread priority, and also, if the feature is used, the default thread priority assigned to lwIP system threads.

Loop interface (CYGFUN_LWIP_LOOPIF)

> This option controls whether support is included for the standard loopback network interface. The interface is created with the IPv4 address `127.0.0.1` and the IPv6 address `::1`. The interface can be used for testing purposes, or where compatibility with existing code is required.

Ethernet support (CYGPKG_LWIP_ETH)

> This boolean option defines whether support for ethernet interfaces is enabled, and if enabled provides access to a variety of configuration options for the ethernet interfaces.
>
> By default the eCos configuration provides support for defining up to 3 ethernet network interfaces. If more interfaces are required then the CDL source in `cdl/lwip_net.cdl` will need to be manually edited. For the following descriptions the *n* suffix should one of `0`, `1` or `2`.
>
> Interface *n* config (CYGPKG_LWIP_ETH_DEV*n*)
>
> > If this boolean option is enabled then it provides the set of configuration options for the specific ethernet network interface. The following STATIC, DHCP, AUTOIP and MANUAL options are mutually exclusive, with only one being actively configured at any point.

Static IPv4 address (`CYGPKG_LWIP_ETH_DEV_ADDR_STATIC*n*`)

If this boolean option is enabled then the ethernet interface will be configured during initialization to use the supplied IPv4 addresses.

IP address (`CYGPKG_LWIP_ETH_DEV_ADDR_STATIC_IP*n*`)

This option provides the hard-coded IPv4 address for the interface.

Netmask (`CYGPKG_LWIP_ETH_DEV_ADDR_STATIC_NETMASK*n*`)

This option provides the hard-coded subnet IPv4 netmask for the interface.

Gateway (`CYGPKG_LWIP_ETH_DEV_ADDR_STATIC_GW*n*`)

This option provides the hard-coded gateway router IPv4 address to be used for the default route for packets sent via the interface that are not destined for the directly connected subnet.

IPv4 address from DHCP (`CYGPKG_LWIP_ETH_DEV_ADDR_DHCP*n*`)

If the lwIP DHCP (Dynamic Host Configuration Protocol) support is configured then this option can be enabled for the interface to obtain its IPv4 address, netmask and gateway values from a suitable DHCP server present on the network.

IPv4 address from AutoIP (`CYGPKG_LWIP_ETH_DEV_ADDR_AUTOIP*n*`)

This option if enabled configures the interface to obtain a link-local IPv4 address using the AutoIP feature.

Set address manually (`CYGPKG_LWIP_ETH_DEV_ADDR_MANUAL*n*`)

This option if enabled indicates that the application code will itself be calling the lwIP functions required to configure the interface addresses.

The application will **need** to supply a `cyg_lwip_eth_init_manual()` function implementation, which will be called from the common IO layer lwIP Ethernet initialisation.

Set as default interface (`CYGPKG_LWIP_ETH_IS_DEFAULT*n*`)

This option, if enabled, selects the respective interface as the default to be used when the network stack needs to communicate with an address which is not part of a network directly associated with a specific interface.

TCP (`CYGPKG_LWIP_TCP`)

This option controls whether the TCP protocol is supported by the lwIP configuration. If enabled, a set of configuration options are available to tune the lwIP TCP implementation.

IPv4 (`CYGFUN_LWIP_IPV4`)

This option enables the IPv4 support. If enabled, a set of configuration options are available to control IPv4 specific features.

IPv6 (`CYGFUN_LWIP_IPV6`)

This option enables the IPv6 support. If enabled, a set of configuration options are available to control IPv6 specific features.

UDP (`CYGPKG_LWIP_UDP`)

This option controls whether the UDP protocol is supported by the lwIP configuration. If enabled, a set of configuration options are available to tune the lwIP UDP implementation.

SNMP Agent (`CYGFUN_LWIP_SNMP`)

If enabled the lwIP world will provide a SNMP (Simple Network Management Protocol) MIB-II agent.

Due to the stated lightweight and simple nature of lwIP, with it mainly being targeted at resource limited embedded targets, the SNMP features available are constrained (e.g. lwIP has a limited notion of IP routing, only pre-compiled MIBs, etc.). Objects located above the `.iso.org.dod.internet` hierarchy are not supported. By default only the `.mgmt` sub-tree is available, though if the CDL option `CYGFUN_LWIP_SNMP_PRIVATE_MIB` is enabled then the `.private` sub-tree becomes available too via the application supplied `private_mib.h` header file.

The supplied `private_mib.h` must contain a struct mib_array_node mib_private definition which is referenced by the lwIP SNMP agent, and describes the private MIB hierarchy. As an example the main struct min_array_node mgmt provided in the source file `src/core/snmp/mib2.c` can be referenced.

Note: The SNMP agent has a sizeable code and data footprint, so may not be suitable for targets with limited resources.

SLIP (`CYGPKG_LWIP_SLIP`)

If enabled lwIP will provide support for the SLIP (Serial Line IP) subsystem. This will provide a network interface to encapsulate IP packets and to send and receive them to a remote system using eCos serial drivers. This option enables a set of SLIP specific configuration options. Note: Though basic functionality has been tested, the SLIP functionality is not supported under the terms of the incident support in eCosPro.

PPP (`CYGPKG_LWIP_PPP`)

If enabled lwIP will provide support for the PPP (Point-to-Point Protocol) subsystem. This option enables a set of PPP specific configuration options.

PPP-over-Ethernet (`CYGPKG_LWIP_PPPOE_SUPPORT`)

If this PPP sub-option is enabled then support for PPPoE (PPP-over-Ethernet) is provided. This provides support for encapsulating PPP frames inside ethernet frames, and is mainly used where a secure point-to-point connection is required, for example, to avoid IP, MAC and DHCP issues.

Note

Support for PPPoE is not yet tested or supported for eCosPro.

PPP-over-Serial (`CYGPKG_LWIP_PPPOS_SUPPORT`)

If this PPP sub-option is enabled then support is provided to encapsulate IP packets and to send and receive them to a remote system using eCos serial drivers.

Note: PPP is more sophisticated than SLIP, and is therefore larger. It does however provide extra features, such as authentication, better link management, option negotion and header compression.

Note: Though basic PPPoS functionality has been tested, the PPPoS functionality is not supported under the terms of the incident support in eCosPro.

RAW sockets (`CYGPKG_LWIP_RAW`)

This option enables support for raw sockets. These allow the transmission or reception of packets over IP but using protocols other than TCP or UDP; or in order to construct packets that cannot be constructed with the lwIP API directly. Raw sockets can be used by selecting a connection type of NETCONN_RAW with the lwIP sequential API. This support is also used by the BSD socket API when creating a socket of type SOCK_RAW

Provide sequential API (CYGPKG_LWIP_SEQUENTIAL_API)

> This option enables support for the lwIP sequential API (see the section called “Application Programming Interfaces (APIs)” for an overview).

Provide BSD-style socket API (CYGFUN_LWIP_COMPAT_SOCKETS)

> This option enables the lwIP support for BSD-style socket operations. This can be useful for adapting existing software to be able to use the lwIP stack.
>
> The socket functions in the API have the form `lwip_accept()`, `lwip_bind()`, `lwip_listen()` etc. Enabling this option causes macros to be defined to map these functions to the BSD function names (`accept()`, `bind()`, `listen()`, etc.). If this causes naming conflicts for the application, then you may wish to disable this option. Particular care is required if this option is enabled at the same time as the File I/O CYGPKG_IO_FILEIO package is used since a single source file will be unlikely to be able to use the File I/O APIs and the lwIP BSD compatible socket API.

Provide POSIX-style socket API (CYGFUN_LWIP_POSIX_SOCKETS_IO_NAMES)

> This option enables the lwIP support for POSIX-style socket operations, useful for adapting existing software to be able to use the lwIP stack. The socket functions in the API have the form `lwip_read()`, `lwip_write()`, etc. Enabling this option causes macros to be defined to map the POSIX function names (`read()`, `write()`, etc.) to these lwIP functions. If this causes naming conflicts for the application you may want to disable this option.

Generate *proto* checksums (CYGIMP_LWIP_CHECKSUM_GEN_*proto*)
Verify *proto* checksums (CYGIMP_LWIP_CHECKSUM_CHECK_*proto*)

> There are a set of configuration options to control checksum generation and calculation support. The IP suffix deals with the generic ethernet IP packet checksum, and the UDP and TCP suffixes with the specific protocol packet checksums. The ICMP6 suffix (CHECK only) performs verification of IPv6 ICMPv6 packets.
>
> See the section called “Checksums” for more information regarding the implications of these options.

Checksum on copy (CYGIMP_LWIP_CHECKSUM_ON_COPY)

> This option if enabled implements code to calculate checksums when copying data from application buffers to packet buffers.

Internal lwIP callback hook definition header (CYGBLD_LWIP_HOOK_H)

> This option allows a specific configuration (i.e. application) to provide its own optional lwIP hook callback definitions if required. Some hooks extend functionality (e.g. LWIP_HOOK_VLAN_SET) whilst others are useful for diagnostics or tracking (e.g. LWIP_HOOK_IP4_INPUT). If enabling this functionality the developer should be conversant with the internals of lwIP and understand how to declare the specific hook macros as well as how they will be called.
>
> One caveat to be aware of is that since the hooks are compiled-into the lwIP stack they may be called from the very beginning of the network stack startup, which may be before the main application code is fully initialised. Care should be taken in the called hooks to ensure a valid state exists. This also means that any hooks that are defined to call application code, the relevant functions need to be available to all code linked against the lwIP stack built with the hooks included.
>
> Any private state needed must be referenced via the `netif` descriptor. However we currently use the `netif->state` pointer internally for eCos. The exact interpretation of the `state` field depends on the actual eCos configuration. For standard driver worlds it is normally a reference to the driver `struct eth_drv_sc` descriptor.
>
> So for any other private application context that needs to be referenced we can use the `client_data` vector held in each `netif` descriptor. The lwIP stack internally has fixed slots allocated for its internal functionality, so we need to ensure that CYGNUM_LWIP_NETIF_CLIENT_DATA is configured with the number of extra slots needed by any application code. The application is responsible for managing the indexing from LWIP_NETIF_CLIENT_DATA_INDEX_MAX onwards.

Hardware driver override header (CYGBLD_LWIP_HW_DRIVER_OVERRIDE_HEADER)

> This option is not normally set by the user, but is provided to allow device drivers to specify a target specific header file that can be used to influence the lwIP configuration.
>
> For example, the lwIP direct ethernet drivers use a header file configured via this option to influence the way the lwIP packet buffer pool is created.

ALTCP abstraction layer (CYGFUN_LWIP_ALTCP)

> This option enables the TCP abstraction layer that replaces the internal TCP function references with indirect calls allowing the support for SSL/TLS or proxy-connect support to applications written against the lwIP TCP callback API without the application layer requiring knowledge of the underlying protocol details.
>
> When the ALTCP functionality is enabled the option CYGFUN_LWIP_ALTCP_TLS controls whether the ALTCP TLS API is available.
>
> For eCos, in conjunction with the Mbed TLS (CYGPKG_MBEDTLS) package, the CYGFUN_LWIP_MBEDTLS configuration option uses the lwIP Mbed TLS wrapper for the lwIP CYGFUN_LWIP_ALTCP_TLS controlled TLS API.

Note

If the CYGFUN_LWIP_ALTCP_TLS option is enabled, but the CYGFUN_LWIP_MBEDTLS option is **not** enabled, then the developer is responsible for providing the ALTCP wrapper functions required. The lwIP package sub-directory src/apps/altcp_tls/altcp_tls_mbedtls contains the Mbed TLS wrapper which can be used as a reference if required.

Memory pool sizes (CYGNUM_LWIP_MEMP_NUM_*pool*)

> The lwIP configuration contains the ability to set many memory related options. The major configuration being the number of pool entries for the different types of memory buffer and descriptors used within the various lwIP subsystems, and these are prefixed CYGNUM_LWIP_MEMP_NUM_ with a usage specific *pool* suffix.
>
> See the section called “Memory Footprint” for more information about tuning the lwIP memory footprint.

Normally the configuration options should be left at their default values unless you have a specific need to change them, e.g. memory requirements. Once the configuration has been created, it should be possible to compile eCos and link it with the application without any errors.

Performance and Footprint Tuning

Performance

There are many changes in configuration that can affect performance. For example, the number and size of buffers, how checksum calculations are implementated, etc.

The CYGDBG_LWIP_STATS option can be enabled to allow for a variety of statistics counts to be gathered during execution. The various options are all prefixed with CYGDBG_LWIP_STATS_, and a sub-system specific suffix.

These statistics can help with the tuning of the lwIP world during development, since monitoring the minimum and maximum usage counts of resources along with the error counts can indicate resource starvation issues. Note: Some error counts are indicative of a temporary inability to claim a resource, and are not necessarily a fatal error for the stack, just a potential slowdown.

In order to determine the number of resources used in practice, during development it is recommended that testing is performed under the expected maximum load expected to need to be handled, in order to understand the resource requirements at that load. To get useful information for this, temporarily configure lwIP with a higher number of resources than would be expected to be needed, memory permitting. Then the application should be tested under the expected network load, at the end of which, the statistics can be inspected, and attention paid to the "max" fields which show the maximum number of each resource used in practice in that sample scenario. This can then be used to inform decisions into the appropriate allocation of reduced resources set in the configuration of lwIP for the final product, without unduly compromising performance.

If CYGDBG_LWIP_STATS is enabled then the function:

#include <lwip/stats.h>

```
void stats_display ();
```

can be used to dump all of the statistics gathered via the output routine defined by the LWIP_PLATFORM_DIAG function wrapper (currently defined to use diag_printf() in the eCos specific arch/cc.h header file).

See the section called “Memory Footprint” for more information about tuning the lwIP memory footprint.

TCP

If the CYGPKG_LWIP_TCP option is configured then various TCP specific options are available for tuning the performance. The main options are covered in the subsections below.

Receive Window

The CYGNUM_LWIP_TCP_WND option defines the maximum TCP receive window size. This size is advertised to remote peers to indicate how much data they can send. While larger values are faster, you should not advertise more than you can receive, which means you must have sufficient capacity in the pbuf pool used for received data for **all** your connections.

Maximum Segment Size

The CYGNUM_LWIP_TCP_MSS option defines the Maximum Segment Size (MSS) advertised to peers to constrain the amount of TCP data they send in each packet. This is recommended not to be more than the interface MTU less 40 bytes. The 40 bytes are the sum of a TCP header and IP header, neither with any options. If any options are used regularly, this value should be reduced further.

If the MSS has been set too large, it will result in IP fragmentation and consequent inefficient network operation. If the MSS is too large and IP fragmentation has been disabled (CYGFUN_LWIP_IP_FRAG), incorrect stack operation will likely result including oversize packets never getting sent, or even a failure in the ethernet driver. The most common MTU size is 1500 bytes (leading to a recommended MSS of up to 1460 bytes) but is certainly not universal: some routers, and especially VPNs, can have lower MTUs and will in turn fragment packets leading to lower efficiency. For best resource utilisation by lwIP, it is a good idea for the MSS to be set so that incoming packets can fit into a whole number of pbufs from the packet buffer pool. As such the default MSS is that of the pbuf pool packet buffer size (CYGNUM_LWIP_PBUF_POOL_BUFSIZE), less 40 bytes to allow room for TCP and IP headers without options.

Sending Data

The CYGNUM_LWIP_TCP_SND_BUF option defines the amount of buffer space in bytes allowed for outstanding (unacked) sent data for each TCP connection. This option is complementary to CYGNUM_LWIP_TCP_SND_QUEUELEN which defines the number of packet buffers allowed for outstanding (unacked) sent data for each TCP connection. The TCP layer will refuse to queue a buffer to be sent if either the total quantity of data in bytes waiting to be sent would then exceed CYGNUM_LWIP_TCP_SND_BUF, or there are already at least CYGNUM_LWIP_TCP_SND_QUEUELEN buffers in the queue waiting to be sent.

Optimizations

The following sections detail some optimization hints that could be useful on certain target platforms to maximise lwIP data throughput.

Checksums

A major performance bottle-neck for lwIP is the software checksum code, since it is executed frequently. If the underlying ethernet device driver provides hardware checksum support then the appropriate CHECKSUM_GEN_* and CHECKSUM_CHECK_* options can be disabled. However if software checksums are needed then you may want to override the standard checksum implementation. This can be achieved by adding a LWIP_CHKSUM definition to a header file included by lwIP, e.g. adding the following to lwipopts.h:

```
#define LWIP_CHKSUM your_checksum_routine
```

The standard lwip_standard_chksum() implementations from src/core/inet_chksum.c provide some C examples, though you might want to craft an assembly function for this specific case. RFC#1071 is a good introduction to this subject. A highly optimized assembler routine will provide the greatest improvement in overall lwIP performance for software checksum based systems.

If the CYGIMP_LWIP_CHECKSUM_ON_COPY functionality is enabled then support for calculating checksums when data is copied into the stack (from application buffers into packet buffers) and can result in fewer checksum calculations if a packet buffer is going to be used multiple times, or if pre-calculated checksums are available for pre-built packets.

The memcpy()-alike function:

```
u16_t lwip_chksum_copy (dest, src, len);
```

can be used to copy data, and return the checksum of the data copied. The extra TCP TF_SEG_DATA_CHECKSUMMED flag is used internally by the lwIP TCP support to track whether a checksum has been set on the payload data.

Network-vs-Host

Since network byte order is big-endian, other significant improvements can be made by supplying assembly or inline replacements for htons() and htonl() if you're using a little-endian architecture.

```
#define LWIP_PLATFORM_BYTESWAP 1
#define LWIP_PLATFORM_HTONS(x) your_htons
#define LWIP_PLATFORM_HTONL(x) your_htonl
```

If the lwIP CYGIMP_LWIP_HAL_BYTESWAP configuration option is enabled then lwIP will use the HAL supplied support. The CYGIMP_LWIP_HAL_BYTESWAP option is enabled by default if the architecture indicates that optimised byte-swap implementations are available, otherwise the option is disabled by default and for little-endian architectures lwIP will provide byte-swap functions.

Device Driver

The ethernet MAC device driver should ideally use interrupts and DMA to avoid busy loops wherever possible. Hardware support for scatter-gather DMA should be used if available, since multiple packet buffers can then be used to hold the different sections of a frame, allowing for zero-copy of payload data.

Release Builds

For a production release it is highly recommended to disable CYGDBG_LWIP_STATS.

Memory Footprint

The setting of the CYGNUM_LWIP_THREAD_STACK_SIZE configuration option and the memory configuration options described in the section called “Performance” will all affect the overall RAM footprint required by lwIP.

However, as long as the option to use the standard run-time allocator (`CYGFUN_LWIP_MEM_LIB_MALLOC`) is **NOT** enabled, the memory footprint of lwIP is deterministic and fixed by the selected configuration.

The major memory configuration options are listed below. Setting these configuration values is usually a compromise between the amount of physical RAM available on the target platform, and the lwIP throughput (performance) requirements.

Heap size (`CYGNUM_LWIP_MEM_SIZE`)

> This option defines the size of the heap that lwIP maintains separate from the system heap so that the resource requirements of one do not affect the other. It is primarily (although not exclusively) used as the memory pool from which packet buffers for transmission are allocated, when the data to be sent needs to be copied (type PBUF_RAM). It is also used to allocate space for dynamically created messages boxes and semaphores. This option can be increased to improve performance when sending large amounts of data.

Packet buffer size (`CYGNUM_LWIP_PBUF_POOL_BUFSIZE`)

> This option specifies the maximum size of data which a single packet buffer (pbuf) allocated from the packet buffer pool for incoming packets can contain. The overall memory footprint of each packet buffer is slightly larger to account for metadata. Incoming packets larger than this size are chained together, using additional packet buffers. If only short packets are usually received, memory efficiency may be improved by reducing the packet buffer size, even if this is accompanied by an increase in the number of packets in the pool using the `CYGNUM_LWIP_PBUF_POOL_SIZE` option. If larger packets tend to be received, the converse is true.
>
> Note: Some network drivers set constraints on the value of this option, in order to better integrate with hardware properties.

Incoming packet messages (`CYGNUM_LWIP_MEMP_NUM_TCPIP_MSG`)
API messages (`CYGNUM_LWIP_MEMP_NUM_API_MSG`)

> When using the sequential API these options define the simultaneous number of, respectively, the packet input and API messages. These messages are used for communicating between external threads and the core lwIP network stack.

Netbufs (`CYGNUM_LWIP_MEMP_NUM_NETBUF`)

> This option defines the maximum number of netbuf structures which may be in use simultaneously with the sequential API (which in turn are used by the BSD sockets API). Each netbuf structure corresponds to a chain of packet buffers to be used for sending or receiving data. This option may be set to 0 if the application will only be using the raw API.

Netconns (`CYGNUM_LWIP_MEMP_NUM_NETCONNS`)

> This option defines the maximum number of netconn structures which may be in use simultaneously with the sequential API. Each netconn structure corresponds to a connection, whether active or inactive. This option may be set to 0 if the application will only be using the raw API.

Packet buffer pool size (`CYGNUM_LWIP_PBUF_POOL_SIZE`)

> This option specifies the number of packet buffers (pbufs) present in the packet buffer pool. This pool is used to provide space for incoming data packets, and so this option limits the number of incoming data packets being processed, or pending (including those not yet read out from the stack by the application). It is also used to hold packet fragments if the option `CYGFUN_LWIP_IP_REASS` is enabled, and so must be large enough to cover the `CYGNUM_LWIP_IP_REASS_MAX_PBUFS` requirement. Note that additional buffers are used in a chain when incoming packets are received which exceed the maximum size of each packet buffer. This option may be adjusted depending on the anticipated peak network traffic. Incoming packets are dropped when the pool is depleted.

Number of memp packet buffers (`CYGNUM_LWIP_MEMP_NUM_PBUF`)

> The lwIP API allows packets to be transmitted which only contain a reference to the data being sent, instead of copying the data into a separate buffer. This can be useful when sending a lot of data out of ROM (or other static memory). This option

specifies the number of such packets that can be used simultaneously. You may wish to increase the value of this option if the application sends a lot of such data, or reduce if not sending any of this form. These buffers are also used when IP fragmentation support is enabled, but a static buffer is not used (`CYGIMP_LWIP_IP_FRAG_USES_STATIC_BUF` disabled), so may also need increasing if fragmentation is common.

RAW protocol control blocks (`CYGNUM_LWIP_MEMP_NUM_RAW_PCB`)

This option defines the number of RAW protocol control blocks that may be used simultaneously. One is required for each active RAW connection.

UDP control blocks (`CYGNUM_LWIP_MEMP_NUM_UDP_PCB`)

This option defines the number of UDP protocol control blocks that may be used simultaneously. One is required for each active UDP connection.

TCP control blocks (`CYGNUM_LWIP_MEMP_NUM_TCP_PCB`)

This option defines the number of TCP protocol control blocks that may be used simultaneously. One is required for each TCP connection. Hence this option defines the maximum number of TCP connections that may be open simultaneously. Increase the value of this option if more simultaneous TCP connections are required.

Listening TCP control blocks (`CYGNUM_LWIP_MEMP_NUM_TCP_PCB_LISTEN`)

This option defines the number of protocol control blocks dedicated to listening for incoming TCP connection requests. This corresponds to the maximum number of TCP ports which may be simultaneously listened on.

Queued TCP segments (`CYGNUM_LWIP_MEMP_NUM_TCP_SEG`)

This option defines the maximum number of TCP segments which may be simultaneously queued. This option may need to be adjusted if the stack reports memory failure errors when attempting to send large quantities of data through TCP connections simultaneously, or when individual TCP writes are so large that the number of MSS-sized segments exceeds the value of this option. If the option to allow out-of-order incoming packets (`CYGIMP_LWIP_TCP_QUEUE_OOSEQ`) is enabled, then such segments may also be dropped if the maximum number of TCP segments specified in this option has been reached.

Queued packets for ARP resolve (`CYGNUM_LWIP_MEMP_NUM_ARP_QUEUE`)

The number of simultaneously queued outgoing packet buffers that are waiting for an ARP request to finish to resolve their destination address.

Queued IP reassembly packets (`CYGNUM_LWIP_MEMP_NUM_REASSDATA`)
Simultaneous IP fragments (`CYGNUM_LWIP_MEMP_NUM_FRAG_PBUF`)

These options provide respectively the number of packets that can simultaneously be queued for reassembly, and the number of fragments (not packets) that can be simultaneously queued for sending.

System timeouts (`CYGNUM_LWIP_MEMP_NUM_INTERNAL_TIMEOUTS`)
User timeouts (`CYGNUM_LWIP_MEMP_NUM_USER_TIMEOUTS`)

The `INTERNAL` value is the number of timeout objects required to support the configured lwIP features. The `USER` value defines the maximum number of user timeouts that may be pending simultaneously. The value of this option may need to be increased if there are more threads using the raw API, or if there are more threads calling the `select()` BSD compatibility function.

Multicast group members (`CYGNUM_LWIP_MEMP_NUM_IGMP_GROUP`)

This option defines the number of multicast groups whose network interfaces can be members at the same time. This value must be at least twice the number of active network interfaces active in the configuration.

Active `lwip_addrinfo()` calls (`CYGNUM_LWIP_MEMP_NUM_NETDB`)
Local host list entries (`CYGNUM_LWIP_MEMP_NUM_LOCALHOSTLIST`)

> If DNS support is enabled then these options respectively control the number of concurrent `lwip_addrinfo()` calls supported, and the number of host entries in the dynamic local host list.

Simultaneous PPP connections (`CYGNUM_LWIP_MEMP_NUM_PPP_PCB`)
Concurrent PPPoE interfaces (`CYGNUM_LWIP_MEMP_NUM_PPPOE_INTERFACES`)

> These options respectively control the number of simultaneously active PPP connections, and the number of concurrently active PPPoE connections.

lwIP Footprint

The following size information was gathered from a CortexM3 targeted configuration using the eCosCentric GNU tools (version 4.4.5c) with gcc *-O2* optimization selected. The byte sizes are provided to give an *example* overview of the lwIP footprint that can be expected, and are purely for informational purposes.

In the following builds *Basic* refers to a sequential API configuration with UDP and TCP support, but with most options disabled (no fragmentation or reassembly support, static address, no SNMP agent, no IGMP, etc.). The builds marked *Reassembly* refers to the addition of fragmented packet reassembly code to the *Basic* builds. The *Full* entry is a configuration with all the lwIP ethernet features enabled (excluding SNMP, SLIP and PPP) to give an idea of the upper footprint for a fully-featured ethernet build.

The values given are for the complete lwIP library package, so specific application linkage (due to the eCos use of *-ffunction-sections*) means that not all of the code and data measured in the sizes given below may actually be included in the final executable. The footprint can be made even smaller by explicit use of the raw API.

Note: The *bss* values below do **NOT** include the stack requirement for the sequential API thread, nor the main configurable lwIP heap space. This is because the aim is to present an example of the base lwIP requirement, independent of the configured heap and stack space required for a particular application or target environment.

CortexM3 (STM32F2xx)	text + rodata	data	bss
Basic IPv4 static	40224	16	516
Basic IPv4 AutoIP	41660	16	516
Basic IPv4 DHCP	46712	16	520
Basic IPv4 IPv6	58680	24	613
Reassembly IPv4 static	41928	16	526
Reassembly IPv4 IPv6	60488	24	627
Full IPv4 IPv6	80512	24	1843

Note: Configurations built with the options `CYGDBG_LWIP_DEBUG`, `CYGDBG_LWIP_ASSERTS` or `CYGDBG_LWIP_STATS` enabled will have a significantly larger code footprint. Similarly configurations built with the `CYGPKG_INFRA_DEBUG` option or the compiler *-O0* optimisation flag will also have a significant effect on the footprint.

Example "small" footprint

The example described in this section targets the STM3220G-EVAL platform, but similar figures have also been obtained for other platforms (e.g. AT91SAM7XEK).

With careful tuning it is possible to implement a simple raw API webserver using the httpd2 test example in ~32K of ROM and ~10K of RAM. This is for the complete application, thread stacks, network buffers, etc.

Even though httpd2 is a simple application it does provide a real-world useful working data point for a minimal footprint system. Note: For this example build the `httpd2.c` source was modified to use the minimal `STACK_SIZE` definition.

The `small_rom_stm3220g_httpd2.ecm` example template used is provided in the lwIP package `doc` directory. The steps needed to build the minimal example binary are:

```
$ mkdir small_httpd2
$ cd small_httpd2
$ ecosconfig new stm3220g_eval
[  ecosconfig output elided  ]
$ ecosconfig import $ECOS_REPOSITORY/net/lwip_tcpip/VERSION/doc/small_rom_stm3220g_httpd2.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make tests
[  make output elided  ]
$ arm-eabi-objcopy -O binary install/tests/net/lwip_tcpip/VERSION/tests/httpd2 httpd2.bin
```

The produced `httpd2.bin` binary can then be loaded into the flash of the STM3220G-EVAL at address `0x08000000`.

Chapter 164. Sequential API

Overview

As described earlier, the lwIP sequential API provides a straightforward and easy-to-use method of interfacing to the stack. Unlike the raw API, which requires event-driven callbacks, an application can simply call the API functions as needed to perform stack operations such as sending data, receiving data, or manipulating packet buffers or connections. While the raw API may allow for more efficient operation, the sequential API typically allows for simpler application design.

Comparison with BSD sockets

In design, it is not unlike the BSD sockets API. Some of the terminology differs however: in the sequential API, the term *connection* is used for any communication link between network peers, and the handle for a connection is termed a *netconn*. A netconn can be considered analogous to a socket, albeit specific to networking - BSD sockets traditionally represent both network connections and files.

The main reason for superiority over the socket API occurs with buffer management. The BSD socket API was designed to manage the fact that the user and the operating system kernel operate in different address spaces and data must always be copied regardless. This results in not only decreased performance, but also increased footprint as buffers must be allocated to hold the copied data.

BSD API Restrictions

By default the eCos lwIP configurations enable the option `CYGFUN_LWIP_COMPAT_SOCKETS` which means normally you cannot include both the lwIP headers `<lwip/sockets.h>` or `<network.h>`, and the `CYGPKG_IO_FILEIO` package (i.e. POSIX-alike) headers from the same file.

Netbufs

Instead of the BSD approach of generic buffers, the sequential API uses *netbufs*, which are based on pbufs. This allows users to manage buffers directly, including even allowing data to come from ROM. Since pbufs, and hence netbufs, can be chained, this also allows the application and lwIP to avoid the need for large regions of entirely contiguous memory in order to hold data. Instead data can be constructed in chunks, and chained together.

When the application wishes to send data, it can send a netbuf directly with UDP. TCP is different as it is intrinsically a buffering, streaming protocol, which requires data to be kept aside to allow for retransmissions. As a result data is sent using just a pointer to memory and a length. However since TCP data can also reside in ROM, it is possible to indicate that the data does not need copying, and so will persist even if the stack needs to queue the data. This can lead to huge savings of memory. For example, static web page content can reside in ROM, and never need to be copied to RAM.

For both TCP and UDP, incoming data is passed to the application as netbufs. The application can use API functions to extract the data from the netbufs - care must be taken as the received data may in fact be a chain. A convenience function exists to copy out the entirety of data across the whole chain into a single contiguous region of memory. Otherwise the application can process data in each netbuf in the chain in turn. The functions `netbuf_first()` and `netbuf_next()` can be used to iterate throught the chain.

TCP/IP thread

When interacting with the network stack using the sequential API, all operations are not handled by the calling thread, but instead are passed to the lwIP network processing (TCP/IP) thread. Inter-thread communication is used inside lwIP to ensure that at the point the API function returns, operation is either complete, or for asynchronous operations, under way.

For example, to register a timeout callback the `tcpip_timeout()` function can be used from client threads to cross the thread boundary into the sequential TCP/IP thread. However, since the actual timeout callback handler registered will be executed within the sequential TCP/IP thread context, it can subsequently directly call the lwIP internal `sys_timeout()` if it needs to re-schedule its callback.

Usage

API declarations

Declarations for all sequential API types and functions may be obtained by including the `<lwip/api.h>` header file:

```
#include <lwip/api.h>
```

Types

Objects of type struct netconn and struct netbuf are intended to be used as opaque types and the structure contents are intended to be maintained and viewed only by lwIP itself. User applications accessing internal members do so at their own risk, and future API compatibility is not guaranteed, nor is thread synchronization since lwIP is entitled to change structure contents at any time.

IP address representation

Depending on the lwIP configuration some API functions take an IP address, which can either be an IPv4 or an IPv6 address.

The IPv4 type struct ip_addr may be accessed as if it has the following structure:

```
struct ip_addr {
  u32_t addr;
};
```

The IPv6 type struct ip6_addr may be accessed as if it has the following structure:

```
struct ip6_addr {
  u32_t addr[4];
};
```

⚠ **Caution**

API users must use the declarations of these structures from the header file `<lwip/ip_addr.h>` which is included implicitly by `<lwip/api.h>`. These types must not be declared by the application itself.

To make it easier to work with either IPv4 or IPv6 addresses the type ipX_addr_t is provided. This is a union of the IPv4 and IPv6 address structures, and may be accessed as if it has the following structure:

```
typedef union {
  ip_addr_t ip4;
  ip6_addr_t ip6;
} ipX_addr_t;
```

See the section called “ipX Helpers” for an overview of the IP version neutral address support. As with the caveat regarding the declarations of the specific IPv4 and IPv6 address structures, the `ipX` declarations should be accessed via including the `<lwip/ip_addr.h>` header file.

IPv4 Addresses

For convenience, predefined struct ip_addr instances are provided for the special cases of "any" IP address (0.0.0.0), and the global broadcast address (255.255.255.255). These instances can be accessed with the macro defines `IP_ADDR_ANY` and `IP_ADDR_BROADCAST` which return values of type struct ip_addr *.

The *`addr`* field is a 32-bit integral value representing the IP address in network byte order (not host byte order).

A variety of convenience function-like macros exist for manipulation or evaluation of IP addresses:

`IP_ADDR_ANY`

> This macro evaluates to an expression of type struct ip_addr * identifying an IP address structure which can be used to represent the special "any" IP address `0.0.0.0`.

`IP_ADDR_BROADCAST`

> This macro evaluates to an expression of type struct ip_addr * identifying an IP address structure which can be used to represent the special global IP address `255.255.255.255`.

`IN_CLASSA(a)`

> An expression which evaluates to non-zero if *`a`* (of type u32_t and in host byte order) is a class A internet address.

`IN_CLASSB(a)`

> An expression which evaluates to non-zero if *`a`* (of type u32_t and in host byte order) is a class B internet address.

`IN_CLASSC(a)`

> An expression which evaluates to non-zero if *`a`* (of type u32_t and in host byte order) is a class C internet address.

`IN_CLASSD(a)`

> An expression which evaluates to non-zero if *`a`* (of type u32_t and in host byte order) is a class D internet address.

`IP4_ADDR(ipaddr, a, b, c, d)`

> Sets ipaddr (of type struct ip_addr *) to the internet address a.b.c.d. For example:

```
struct ip_addr host;
    …
    IP4_ADDR(host, 192, 168, 1, 1);
```

`ip_addr_cmp(addr1, addr2)`

> Returns non-zero if the arguments *`addr1`* and *`addr2`*, both of type struct ip_addr * are identical. Zero if they differ.

`ip_addr_netcmp(addr1, addr2, mask)`

> Returns non-zero if the arguments *`addr1`* and *`addr2`*, both of type struct ip_addr * are on the same network, as indicated by the network mask *`mask`* which is itself also of type struct ip_addr *. Zero if they are on different networks.

`htons(s)`

> Portably converts *`s`* of type u16_t from host byte order to a u16_t in network byte order.

`ntohs(s)`

> Portably converts *`s`* of type u16_t from network byte order to a u16_t in host byte order.

`htonl(l)`

> Portably converts *`l`* of type u32_t from host byte order to a u32_t in network byte order.

`ntohl(l)`

Portably converts *`l`* of type u32_t from network byte order to a u32_t in host byte order.

Some further potentially useful macro definitions can be viewed in `<lwip/ip_addr.h>`.

IPv6 Addresses

The header file `<lwip/ip6_addr.h>` (which is included by default from `<lwip/api.h>`) contains definitions for many IPv6 address convenience function-like macros, as well as utility function prototypes.

The following is not an exhaustive list, so the reader is recommended to inspect the header file to get a complete overview of the IPv6 address support macros and functions.

`IP6_ADDR_ANY`

This macro evaluates to an expression of type struct ip6_addr * identifying an IP address structure which can be used to represent the special "any" IPv6 address `::/128`. It actually just returns the address of the exported `ip6_addr_any` variable.

`ip6_addr_copy(dest, src)`

This implements a fast (no `NULL` check) address copy.

`ip6_addr_set(dest, src)`

Set the *`dest`* address from the supplied *`src`*. If *`src`* is `NULL` the destination is written with zeroes.

`ip6_addr_set_zero(ip6addr)`

Sets the *`ip6addr`* address to all zeroes.

`ip6_addr_set_any(ip6addr)`

This explicitly sets the `IP6_ADDR_ANY` address value.

`ip6_addr_set_loopback(ip6addr)`

This sets the destination *`ip6addr`* parameters to the `::1` loopback address.

`ip6_addr_set_hton(dest, src)`

Copy the *`src`* address to the *`dest`* address converting from host to network byte order.

`ip6_addr_netcmp(addr1, addr2)`

An expression which evaluates to non-zero if the supplied *`addr1`* and *`addr2`* parameters are on the same network, by comparing the most-significant 64-bits of the addresses.

`ip6_addr_cmp(addr1, addr2)`

An expression which evaluates to non-zero if there is an exact match between the supplied *`addr1`* and *`addr2`* parameters.

`ip6_get_subnet_id(ip6addr)`

This returns in host byte order the 16-bit subnet identifier.

`ip6_addr_isany(ip6addr)`

An expression which evaluates to non-zero if *`ip6addr`* matches the `IP6_ADDR_ANY` address (all zeroes).

`ip6_addr_isglobal(ip6addr)`

An expression which evaluates to non-zero if the supplied *`ip6addr`* is a valid global address.

`ip6_addr_islinklocal(ip6addr)`

An expression which evaluates to non-zero if the *`ip6addr`* parameter is a valid link-local address.

`ip6_addr_issitelocal(ip6addr)`

An expression which evaluates to non-zero if the *`ip6addr`* parameter is a valid site-local address.

`ip6_addr_isuniquelocal(ip6addr)`

An expression which evaluates to non-zero if the *`ip6addr`* parameter is a valid unique link-local address.

`ip6_addr_ismulticast(ip6addr)`

An expression which evaluates to non-zero if the supplied *`ip6addr`* is a valid multicast address.

There are various other function-like macros provided to further decode whether the multicast address is a loopback, link-local, admin-local, global, etc. address. The definitions for these variants can be found by inspecting the `<lwip/ip6_addr.h>` header file.

`ip6_addr_isallnodes_iflocal(ip6addr)`

An expression which evaluates to non-zero if the supplied *`ip6addr`* matches the IPv6 `ff01::1` loopback "all nodes" address.

`ip6_addr_isallnodes_linklocal(ip6addr)`

An expression which evaluates to non-zero if the supplied *`ip6addr`* matches the IPv6 link-local "all nodes" address.

`ip6_addr_isallrouters_linklocal(ip6addr)`

An expression which evaluates to non-zero if the supplied *`ip6addr`* matches the IPv6 link-local "all routers" address.

`ip6_addr_set_allnodes_linklocal(ip6addr)`

Sets the given *`ip6addr`* to the `ff02::1` link-local "all nodes" multicast address.

`ip6_addr_set_allrouters_linklocal(ip6addr)`

Sets the given *`ip6addr`* to the `ff02::2` link-local "all routers" multicast address.

`ip6_addr_isinvalid(addr_state)`

An expression which evaluates to non-zero if the supplied *`addr_state`* parameter is `IP6_ADDR_INVALID`.

`ip6_addr_isvalid(addr_state)`

An expression which evaluates to non-zero if the supplied *`addr_state`* parameter denotes a valid address state, where the `IP6_ADDR_VALID` bitmask is set.

`ip6_addr_istentative(addr_state)`

An expression which evaluates to non-zero if the supplied *`addr_state`* parameter has the `IP6_ADDR_TENTATIVE` bitmask set.

`ip6_addr_ispreferred(addr_state)`

An expression which evaluates to non-zero if the supplied *`addr_state`* parameter is `IP6_ADDR_PREFERRED`.

`ip6_addr_isdeprecated(addr_state)`

An expression which evaluates to non-zero if the supplied *`addr_state`* parameter is `IP6_ADDR_DEPRECATED`.

`int ip6addr_aton(const char *cp, ip6_addr_t *addr)`

Checks whether *`cp`* is a valid ASCII representation of an IPv6 address, and if valid converts it to the binary IPv6 address destination *`addr`*. The function returns 1 on successful conversion, or 0 on failure.

`char *ip6addr_ntoa_r(const ip6_addr_t *addr, char *buf, int buflen)`

Converts the binary IPv6 address *`addr`* into an ASCII representation written into the supplied *`buf`* buffer of *`buflen`* bytes.

Returns the *`buf`* parameter on success if the buffer has been updated to hold the ASCII address representation, or `NULL` if the buffer was too small.

ipX Helpers

Instead of directly referencing the IPv4 or IPv6 versions of the utility routines, applications should ideally use the common `ipX_*` variants. These functions, and function-like macros, take as their first parameter a boolean *`is_ipv6`* value denoting when zero that the referenced addresses are IPv4 structures, or when non-zero that the referenced addresses are IPv6 structures. Note: If IPv6 support is not enabled for lwIP then the `ipX_*` implementations default to only using IPv4 addresses.

`ipX_addr_copy(is_ipv6, dest, src)`

This implements a fast (no `NULL` check) address copy.

`ipX_addr_set(is_ipv6, dest, src)`

Set the *`dest`* address from the supplied *`src`*. If *`src`* is `NULL` the destination is written with zeroes.

`ipX_addr_set_ipaddr(is_ipv6, dest, src)`

Sets the *`dest`* address parameter from the specified *`src`*. If *`src`* is `NULL` the destination is written with zeroes.

`ipX_addr_set_zero(is_ipv6, ipaddr)`

Sets the address to all zeroes. This is normally the `ANY` address.

`ipX_addr_set_any(is_ipv6, ipaddr)`

This explicitly sets the `ANY` address.

`ipX_addr_set_loopback(is_ipv6, ipaddr)`

This sets the destination *`ipaddr`* to the respective loopback interface address. For IPv4 this is `127.0.0.1` and `::1` for IPv6.

`ipX_addr_set_hton(is_ipv6, dest, src)`

Copy the *`src`* address to the *`dest`* address converting from host to network byte order.

`ipX_addr_cmp(is_ipv6, addr1, addr2)`

An expression which evaluates to non-zero if there is an exact match between the supplied *`addr1`* and *`addr2`* parameters.

`ipX_addr_isany(is_ipv6, ipaddr)`

An expression which evaluates to non-zero if *ipaddr* is NULL or points at the ANY address value.

`ipX_addr_ismulticast(is_ipv6, ipaddr)`

An expression which evaluates to non-zero if *ipaddr* references a valid multicast address value.

`ipX_addr_debug_print(is_ipv6, debug, ipaddr)`

This debugging helper macro will output the raw IPv4 or IPv6 address via the printf-alike debug support calls if the relevant lwIP debugging option specified by the *debug* parameter is enabled.

Error codes

While the BSD sockets API uses POSIX standard error codes (ENOMEM, EINVAL, etc.) the lwIP sequential API has its own separate set of error code definitions.

These error definitions are used by any API function that returns a value of type err_t. The following table indicates possible error code values and their meaning:

Table 164.1. lwIP sequential API error codes

Code	Meaning
ERR_OK	No error, operation successful.
ERR_MEM	Out of memory error.
ERR_BUF	Buffer error.
ERR_ABRT	Connection aborted.
ERR_RST	Connection reset.
ERR_CLSD	Connection closed.
ERR_CONN	Not connected.
ERR_VAL	Illegal value.
ERR_ARG	Illegal argument.
ERR_RTE	Routing problem.
ERR_USE	Address in use.
ERR_IF	Low-level network interface error.
ERR_ISCONN	Already connected.
ERR_TIMEOUT	Timeout.
ERR_INPROGRESS	Operation in progress.
ERR_WOULDBLOCK	Operation would block.

API reference

Name

netbuf_new() — Allocate a netbuf structure

Synopsis

```
struct netbuf *netbuf_new ();
```

Description

Allocates a netbuf structure. No buffer space is allocated when doing this, only the top level structure. After use, the netbuf must be deallocated with `netbuf_delete()`.

Name

netbuf_delete() — Deallocate a netbuf structure

Synopsis

```
void netbuf_delete ();
```

Description

Deallocates a netbuf structure previously allocated by a call to the `netbuf_new()` function. Any buffer memory allocated to the netbuf by calls to `netbuf_alloc()` is also deallocated.

Example

Example 164.1. This example shows the basic mechanisms for using netbufs.

```
int
main()
{
    struct netbuf *buf;
    buf = netbuf_new();     /* create a new netbuf */
    netbuf_alloc(buf, 100); /* allocate 100 bytes of buffer */

    /* do something with the netbuf */
    /* […] */
    netbuf_delete(buf); /* deallocate netbuf */
}
```

Name

netbuf_alloc() — Allocate space in a netbuf

Synopsis

```
void *netbuf_alloc (buf, size);
```

Description

Allocates buffer memory with *size* number of bytes for the netbuf *buf*. The function returns a pointer to the allocated memory. Any memory previously allocated to the netbuf *buf* is deallocated. The allocated memory can later be deallocated with the `netbuf_free()` function. Since protocol headers are expected to precede the data when it should be sent, the function allocates memory for protocol headers as well as for the actual data.

Name

netbuf_free() — Deallocate buffer memory associated with a netbuf

Synopsis

```
void netbuf_free (buf);
```

Description

Deallocates the buffer memory associated with the netbuf *buf*. If no buffer memory has been allocated for the netbuf, this function does nothing.

Name

netbuf_ref() — Associate a data pointer with a netbuf

Synopsis

```
err_t netbuf_ref (buf, data, size);
```

Description

Associates the external memory pointed to by the *data* pointer with the netbuf *buf*. The size of the external memory is given by *size*. Any memory previously allocated to the netbuf is deallocated. The difference between allocating memory for the netbuf with `netbuf_alloc()` and allocating memory using, e.g., `malloc()` and referencing it with `netbuf_ref()` is that in the former case, space for protocol headers is allocated as well which makes processing and sending the buffer faster.

The result returned will be `ERR_OK` if the data is referenced successfully, or the error `ERR_MEM` if the data could not be referenced due to lack of memory.

Example

Example 164.2. This example shows a simple use of the `netbuf_ref()`

```
int
main()
{
    struct netbuf *buf;
    char string[] = "A string";

    /* create a new netbuf */
    buf = netbuf_new();

    /* reference the string */
    if (netbuf_ref(buf, string, sizeof(string)) == ERR_OK) {
        /* do something with the netbuf */
        /* […] */
    }

    /* deallocate netbuf */
    netbuf_delete(buf);
}
```

Name

netbuf_len() — Obtain the total length of a netbuf

Synopsis

```
u16_t netbuf_len (buf);
```

Description

Returns the total length of the data in the netbuf *buf*, even if the netbuf is fragmented. For a fragmented netbuf, the value obtained by calling this function is not the same as the size of the first fragment in the netbuf.

Name

netbuf_data() — Obtain a pointer to netbuf data

Synopsis

```
err_t netbuf_data (buf, data, len);
```

Description

This function is used to obtain a pointer to and the length of a block of data in the netbuf *buf*. The arguments *data* and *len* are result parameters that will be filled with a pointer to the data and the length of the data pointed to. If the netbuf is fragmented, this function gives a pointer to one of the fragments in the netbuf. The application program must use the fragment handling functions `netbuf_first()` and `netbuf_next()` in order to reach all data in the netbuf. See the example under `netbuf_next()` for an example of how use `netbuf_data()`.

Name

netbuf_next() — Traverse internal fragments in a netbuf

Synopsis

`s8_t` **`netbuf_next`** `(`*`buf`*`);`

Description

This function updates the internal fragment pointer in the netbuf *`buf`* so that it points to the next fragment in the netbuf. The return value is zero if there are more fragments in the netbuf, > 0 if the fragment pointer now points to the last fragment in the netbuf, and < 0 if the fragment pointer already pointed to the last fragment.

Example

Example 164.3. This example shows how to use the `netbuf_next()` function

We assume that this is in the middle of a function and that the variable `buf` is a netbuf.

```
/* […] */
do {
    char *data;
    int len;

    /* obtain a pointer to the data in the fragment */
    netbuf_data(buf, data, len);

    /* do something with the data */
    do_something(data, len);
} while(netbuf_next(buf) >= 0);
/* […] */
```

Name

netbuf_first() — Reset fragment pointer to start of netbuf

Synopsis

```
void netbuf_first (buf);
```

Description

Resets the internal fragment pointer in the netbuf *buf* so that it points to the first fragment.

Name

netbuf_copy() — Copy all netbuf data to memory pointer

Synopsis

`void` **`netbuf_copy`** `(`*`buf`*`,` *`data`*`,` *`len`*`);`

Description

Copies all of the data from the netbuf *`buf`* into the memory pointed to by *`data`* even if the netbuf buf is fragmented. The *`len`* parameter is an upper bound of how much data that will be copied into the memory pointed to by data.

Example

Example 164.4. This example shows a simple use of `netbuf_copy()`

Here, 200 bytes of memory is allocated on the stack to hold data. Even if the netbuf buf has more data that 200 bytes, only 200 bytes are copied into `data`.

```
void
example_function(struct netbuf *buf)
{
    char data[200];
    netbuf_copy(buf, data, 200);

    /* do something with the data */
}
```

Name

netbuf_copy_partial() — Copy some netbuf data to memory pointer

Synopsis

```
void netbuf_copy_partial (buf, data, len, offset);
```

Description

This function is similar to `netbuf_copy()` except that it takes an extra parameter, *`offset`*, which can be used to set an offset from the start of the packet to start copying the *`len`* bytes.

Name

netbuf_chain() — Chain two netbufs together

Synopsis

```
void netbuf_chain (head, tail);
```

Description

Chains the two netbufs *head* and *tail* together so that the data in tail will become the last fragment(s) in *head*. The netbuf *tail* is deallocated and should not be used after the call to this function.

Name

netbuf_fromaddr() — Obtain the sender's IPv4 address for a netbuf

Synopsis

```
struct ip_addr *netbuf_fromaddr (buf);
```

Description

Returns the IPv4 address of the host the netbuf *buf* was received from. If the netbuf has not been received from the network, the return value of this function is undefined. The function `netbuf_fromport()` can be used to obtain the port number of the remote host.

Name

netbuf_fromaddr_ip6() — Obtain the sender's IPv6 address for a netbuf

Synopsis

```
struct ip6_addr *netbuf_fromaddr (buf);
```

Description

Returns the IPv6 address of the host the netbuf *buf* was received from. If the netbuf has not been received from the network, the return value of this function is undefined. The function `netbuf_fromport()` can be used to obtain the port number of the remote host.

Name

netbuf_fromport() — Obtain the sender's port number for a netbuf

Synopsis

```
u16_t netbuf_fromport (buf);
```

Description

Returns the port number of the host the netbuf *buf* was received from. If the netbuf has not been received from the network, the return value of this function is undefined. The function `netbuf_fromaddr()` can be used to obtain the IP address of the remote host.

Name

netconn_new() — Create a new connection structure

Synopsis

```
struct netconn *netconn_new (type);
```

Description

Creates a new connection abstraction structure. The argument usually one of either `NETCONN_TCP` or `NETCONN_UDP`, yielding either a TCP or a UDP connection. No connection is established by the call to this function and no data is sent over the network.

For more advanced use, it is also possible to specify different connection types: `NETCONN_UDPLITE`, `NETCONN_UDPNOCHKSUM` or `NETCONN_RAW`.

If IPv6 support is configured then the type can be suffixed with `_IPV6` to specify an IPv6 connection. e.g. `NETCONN_TCP_IPV6`.

Name

netconn_new_with_callback() — Create a new connection structure with a callback

Synopsis

```
struct netconn *netconn_new_with_callback (type, (*callback));
```

Description

This function is similar to `netconn_new()` except that an additional function pointer *`callback`* is passed. The function pointed to by *`callback`* will be called when data is sent or received. Specifically, the *`netconn_evt`* parameter to the callback is used to indicate the event type. This enum can have the following values:

NETCONN_EVT_RCVPLUS

> Used when new incoming data from a remote peer arrives. The amount of data received is passed in *`len`*. If *`len`* is 0 then a connection event has occurred: this may be an error, the acceptance of a connection for a listening connection (called for the listening connection), or deletion of the connection.

NETCONN_EVT_RCVMINUS

> Used when new incoming data from a remote peer has been received and accepted by higher layers. The amount of data accepted is passed in *`len`*. If *`len`* is 0 then this indicates the acceptance of a connection as a result of a listening port (called for the newly created accepted connection).

NETCONN_EVT_SENDPLUS

> Used when data has been sent to a remote peer and received by it. This only occurs for TCP connections, and specifically is only triggered when, as a consequence of TCP acknowledgements from the remote peer, the free TCP send buffer size now exceeds the configured send buffer low water mark (configured with the `CYGNUM_LWIP_TCP_SNDLOWAT` CDL configuration option). The amount of data sent in the most recent transaction is passed in *`len`*. If *`len`* is 0 then this indicates the connection has been deleted.

NETCONN_EVT_SENDMINUS

> This is only used for TCP connections, and is triggered when a sufficient amount of data has been sent on the connection that the amount of free send buffer space is now under the send buffer low water mark (configured with the `CYGNUM_LWIP_TCP_SNDLOWAT` CDL configuration option). The amount of data sent in the most recent transaction is passed in *`len`*.

NETCONN_EVT_ERROR

> This is only used for TCP connections, and is triggered when an error has occurred or a connection is being forced closed. It is used to signal `select()`.

Name

netconn_new_with_proto_and_callback() — Create a new connection structure with a callback for a specific protocol

Synopsis

```
struct netconn *netconn_new_with_callback(type, proto, (*callback));
```

Description

This function is similar to `netconn_new_with_callback()` except that an additional parameter *`proto`* may be used to indicate the IP protocol number to use. If *`proto`* is non-zero, it must only be used with the *`type`* set to `NETCONN_RAW`.

The most common use of this function is the creation of connections suitable for generating ICMP packets.

Name

netconn_delete() — Deallocate a netconn

Synopsis

```
err_t netconn_delete (conn);
```

Description

Deallocates the netconn *conn*. If the connection is open, it is closed as a result of this call.

Name

netconn_type() — Obtain the type of netconn

Synopsis

```
enum netconn_type netconn_type (conn);
```

Description

Returns the type of the connection *conn*. This is the same type that is given as an argument to netconn_new() (and its variants) and can be one of `NETCONN_TCP`, `NETCONN_UDP`, `NETCONN_UDPLITE`, `NETCONN_UDPNOCHKSUM` or `NETCONN_RAW`.

If IPv6 support is configured then it can also be one of `NETCONN_TCP_IPV6`, `NETCONN_UDP_IPV6`, `NETCONN_UDPLITE_IPV6` or `NETCONN_UDPNOCHKSUM_IPV6`.

Name

netconn_peer() — Obtain the remote host IP address/port of a netconn

Synopsis

```
err_t netconn_peer (conn, addr, port);
```

Description

This function is used to obtain the IPv4 address and port of the remote end of the connection indicated by *conn*. The parameters *addr* and *port* are result parameters that are set by the function. If the connection *conn* is not connected to any remote host, the results are undefined.

Name

netconn_addr() — Obtain the local host IPv4 address/port of a netconn

Synopsis

```
err_t netconn_addr (conn, addr, port);
```

Description

This function is used to obtain the local IPv4 address and port number of the connection *conn*.

Name

netconn_bind() — Set local IP address/port of a netconn

Synopsis

```
err_t netconn_bind (conn, addr, port);
```

Description

Binds the connection *conn* to the local IP address *addr* and TCP or UDP port *port*. If *addr* is `NULL`, the local IP address is determined by the networking system.

The *addr* is defined as a pointer to an IPv4 address, though the routine will accept a suitably cast IPv6 address structure pointer. However the `netconn_bind_ip6()` function provides explicit IPv6 address support.

Name

netconn_bind_ip6() — Set local IPv6 address/port of a netconn

Synopsis

```
err_t netconn_bind (conn, addr, port);
```

Description

Binds the connection `conn` to the local IPv6 address `addr` and TCP or UDP port `port`. If `addr` is `NULL`, the local IP address is determined by the networking system.

Name

netconn_connect() — Connect netconn to remote peer

Synopsis

```
err_t netconn_connect (conn, remote_addr, remote_port);
```

Description

In case of UDP, sets the remote receiver as given by *remote_addr* and *remote_port* of UDP messages sent over the connection. For TCP, `netconn_connect()` opens a connection with the remote host.

Solely for UDP, it is possible to call `netconn_connect()` repeatedly to set a new remote destination to use for UDP packets, rather than having to create and delete netconns for each destination.

Name

netconn_connect_ip6() — Connect netconn to remote peer

Synopsis

```
err_t netconn_connect (conn, remote_addr, remote_port);
```

Description

If IPv6 is configured then this provides similar functionality to the IPv4 function `netconn_connect()`, but with the *`remote_addr`* referencing an IPv6 addresss.

Name

netconn_disconnect() — Disconnect UDP connection

Synopsis

```
err_t netconn_disconnect (conn);
```

Description

This function is only relevant for UDP connections. It unsets any previously set (using `netconn_connect()`) remote peer address and port associated with connection *`conn`*.

Name

netconn_listen() — Make a listening TCP netconn

Synopsis

```
err_t netconn_listen (conn);
```

Description

Puts the TCP connection *conn* into the TCP LISTEN state. This means its purpose will become listening for incoming connections from remote peers. `netconn_accept()` is required to establish a connection resulting from incoming connection requests.

Name

netconn_accept() — Wait for incoming connections

Synopsis

`err_t` **`netconn_accept`** `(`*`conn`*`, `*`new_conn`*`);`

Description

This function blocks the process until a connection request from a remote host arrives on the TCP connection *`conn`*. The connection must be in the LISTEN state so `netconn_listen()` must be called prior to `netconn_accept()`. When a connection is established with the remote host, a new connection structure is returned in the *`new_conn`* parameter.

Example

Example 164.5. This example shows how to open a TCP server on port 2000 [*]

```
int
main()
{
    struct netconn *conn, *newconn;

    /* create a connection structure */
    conn = netconn_new(NETCONN_TCP);

    /* bind the connection to port 2000 on any local IP address */
    netconn_bind(conn, NULL, 2000);

    /* tell the connection to listen for incoming connection requests */
    netconn_listen(conn);

    /* block until we get an incoming connection */
    if (netconn_accept(conn, newconn) == ERR_OK) {
        /* do something with the connection */
        process_connection(newconn);

        /* deallocate both connections */
        netconn_delete(newconn);
    }
    netconn_delete(conn);
}
```

[*] This is only an example for illustrative purposes, and a complete version should perform comprehensive error checking.

Name

netconn_recv() — Wait for data

Synopsis

`err_t` **`netconn_recv`** `(`*`conn`*`,` *`new_buf`*`);`

Description

This function blocks the process while waiting for data to arrive on the connection *`conn`*. The return value will be `ERR_OK` on success. On error, for example if the connection has been closed by the remote host, NULL is returned in *`new_buf`*, otherwise a netbuf containing the received data is returned in *`new_buf`*.

Example

Example 164.6. This example demonstrates usage of the `netconn_recv()` function

In the following code, we assume that a connection has been established before the call to `example_function()`.

```
voidexample_function(struct netconn *conn)
{
    struct netbuf *buf;
    err_t err;

    /* receive data until the other host closes the connection */
    while((err = netconn_recv(conn, buf)) == ERR_OK) {
        do_something(buf);
    }

    /* the connection has now been closed by the other end, so we close our end */
    netconn_close(conn);
}
```

Name

netconn_recv_tcp_pbuf() — Wait for data

Synopsis

```
err_t netconn_recv_tcp_pbuf (conn, new_buf);
```

Description

This function is similar to the `netconn_recv()` function, with the difference that the received data is placed in a pbuf instead of a netbuf.

Name

netconn_recved() — Update receive window

Synopsis

`void` **`netconn_recved`** `(`*`conn`*`, `*`length`*`);`

Description

The application can call this function to notify the stack that it has processed the received data and is able to accept new data.

Warning

This function is primarily for use with sockets, and should be used with care. It can only be used when `netconn_set_noautorecved(conn,1)` has been used to disable the automatic receive window updating.

Name

netconn_write() — Send data on TCP connection

Synopsis

```
err_t netconn_write (conn, data, len, copy);
```

Description

This function is only used for TCP connections. It puts the data pointed to by *data* on the output queue for the TCP connection *conn*. The length of the data is given by *len*. There is no restriction on the length of the data. This function does not require the application to explicitly allocate buffers, as this is taken care of by the stack. The *copy* parameter is a combination of the following bitmask flags:

```
#define NETCONN_NOFLAG 0x00
#define NETCONN_COPY 0x01
#define NETCONN_MORE 0x02
#define NETCONN_DONTBLOCK 0x04
```

When passed the flag `NETCONN_COPY` the data is copied into internal buffers which are allocated for the data. This allows the data to be modified directly after the call, but is inefficient both in terms of execution time and memory usage. If the flag is not set then the data is not copied but rather referenced, and the `NETCONN_NOCOPY` manifest is provided for backwards compatibality. The data must not be modified after the call, since the data can be put on the retransmission queue for the connection, and stay there for an indeterminate amount of time. This is useful when sending data that is located in ROM and therefore is immutable. If greater control over the modifiability of the data is needed, a combination of copied and non-copied data can be used, as seen in the example below.

The flag `NETCONN_MORE` can be used for TCP connections and indicates that the PSH (push) flag will be set on the last segment sent. The flag `NETCONN_DONTBLOCK` tells the stack to only write the data if all the data can be written at once.

Example

Example 164.7. This example demonstrates basic usage of the `netconn_write()` function [*]

Here, the variable data is assumed to be modified later in the program, and is therefore copied into the internal bufiers by passing the flag `NETCONN_COPY` to `netconn_write()`. The text variable contains a string that will not be modified and can therefore be sent using references instead of copying.

```
int
main()
{
    struct netconn *conn;
    char data[10];
    char text[] = "Static text";
    int i;

    /* set up the connection conn */
    /* […] */

    /* create some arbitrary data */
    for(i = 0; i < 10; i++)
        data[i] = i;

    netconn_write(conn, data, 10, NETCONN_COPY);
    netconn_write(conn, text, sizeof(text), NETCONN_NOFLAG);
```

[*] This is only an example for illustrative purposes, and a complete version should perform comprehensive error checking.

```
    /* the data can be modified */
    for(i = 0; i < 10; i++)
        data[i] = 10 - i;

    /* take down the connection conn */
    netconn_close(conn);
}
```

Name

netconn_send() — Send data on UDP connection

Synopsis

`err_t **netconn_send**(*conn*, *buf*);`

Description

Send the data in the netbuf *`buf`* on the UDP connection *`conn`*. The data in the netbuf should not be too large if IP fragmentation support is disabled. If IP fragmentation support is disabled, the data should not be larger than the maximum transmission unit (MTU) of the outgoing network interface, less the space required for link layer, IP and UDP headers. No checking is necessarily made of whether the data is sufficiently small and sending very large netbufs might give undefined results.

Example

Example 164.8. This example demonstrates basic usage of the `netconn_send()` function [*]

This example shows how to send some UDP data to UDP port 7000 on a remote host with IP address 10.0.0.1.

```
int
main()
{
    struct netconn *conn;
    struct netbuf *buf;
    struct ip_addr addr;
    char *data;
    char text[] = "A static text";
    int i;

    /* create a new connection */
    conn = netconn_new(NETCONN_UDP);

    /* set up the IP address of the remote host */
    addr.addr = htonl(0x0a000001);

    /* connect the connection to the remote host */
    netconn_connect(conn, addr, 7000);

    /* create a new netbuf */
    buf = netbuf_new();
    data = netbuf_alloc(buf, 10);

    /* create some arbitrary data */
    for(i = 0; i < 10; i++)
        data[i] = i;

    /* send the arbitrary data */
    netconn_send(conn, buf);

    /* reference the text into the netbuf */
    netbuf_ref(buf, text, sizeof(text));

    /* send the text */
    netconn_send(conn, buf);

    /* deallocate connection and netbuf */
    netconn_delete(conn);
```

[*] This is only an example for illustrative purposes, and a complete version should perform comprehensive error checking.

```
    netconn_delete(buf);
}
```

Name

netconn_close() — Close a connection

Synopsis

```
err_t netconn_close (conn);
```

Description

Close the connection *conn*.

Name

netconn_shutdown() — Shutdown a connection

Synopsis

```
err_t netconn_shutdown (conn, shut_rx, shut_tx);
```

Description

Shut down one, or both, sides of a TCP connection, but without deleting the connection. The *shut_rx* and *shut_tx* parameters are treated as boolean values with non-zero values indicating that the respective read or write side should be closed. Specifying that both RX and TX are to be shut is the same as closing the connection via calling `netconn_close()`.

Name

netconn_set_noautorecved() — Set the connection no-auto-recved state

Synopsis

```
void netconn_set_noautorecved (conn, val);
```

Description

If *val* equates to `true` then the `NETCONN_FLAG_NO_AUTO_RECVED` state is set for the connection, otherwise the flag is cleared.

Name

netconn_get_noautorecved() — Get the connection no-auto-recved state

Synopsis

```
cyg_bool netconn_get_noautorecved (conn);
```

Description

Returns a boolean state indicating the current `NETCONN_FLAG_NO_AUTO_RECVED` flag state for the connection.

Name

netconn_err() — Obtain connection error status

Synopsis

```
err_t netconn_err (conn);
```

Description

Obtain the stored error status of connection *conn*.

Chapter 165. Raw API

Much of the information in this chapter has been derived from lwIP's own raw API documentation, although additions, modifications and adaptations for eCos have been made.

Overview

While the high level lwIP sequential API is good for programs that are themselves sequential and can benefit from the blocking open-read-write-close paradigm, lwIP itself is event based by nature. If an application can be written with an event-based approach, then it becomes possible to integrate directly with the event-based design of the core lwIP code.

The *raw TCP/IP API* allows the application program to integrate better with the TCP/IP code. Program execution is event based by having callback functions being called from within the TCP/IP code. The TCP/IP code and the application program both run in the same thread. The sequential API has a much higher overhead and is not very well suited for small systems since it forces a multithreaded paradigm on the application.

The raw TCP/IP interface is not only faster in terms of code execution time but is also less memory intensive. The drawback is that program development is somewhat harder and application programs written for the raw TCP/IP interface are more difficult to understand. Still, this is the preferred way of writing applications that should be small in code size and memory usage.

Both APIs can be used simultaneously by different application programs. In fact, the sequential API is implemented as an application program using the raw TCP/IP interface.

An example of an application using the raw API can be found in the `tests/` subdirectory of the lwIP eCos package. This httpd2 test is built when the CDL configuration option `CYGBLD_NET_LWIP_BUILD_MANUAL_TESTS` is enabled. This raw API application acts as a simple HTTP server. For more information see the section called “httpd2”.

Usage

The raw API is a very direct interface, and is close to the metal. If the `CYGFUN_LWIP_NO_SYS` option is enabled then there still needs to be a single lwIP owner thread but an application can be constructed where the main processing loop of that thread performs lwIP support as well as other application event processing as required so that only a single stack footprint is required. The trueraw application is built when `CYGFUN_LWIP_NO_SYS` is configured, and the `CYGBLD_NET_LWIP_BUILD_MANUAL_TESTS` option is enabled. This provides a simple example of an application using the raw API without the overhead of the TCP/IP helper thread.

For true raw API applications the `cyg_lwip_init()` function can be used to initialise the lwIP stack (as for sequential or BSD API applications), but there is no support for waiting for the network to be brought up within that function call, since when using a true raw world the caller of the `cyg_lwip_init()` is also responsible for processing network packets that may be needed to bring up the network interface up. If required an application can perform its own lwIP stack initialization, and does not need to use the eCos default support.

Note that if you do decide to use `cyg_lwip_init()` with the configuration option `CYGFUN_LWIP_SEQUENTIAL_API` disabled, so that solely the raw API is available, bbut with the configuration option `CYGFUN_LWIP_NO_SYS` also disabled, then the application will need to provide its own alternative to the tcpip_input() function which had previously been used to inject received packets into the stack. This function must be declared as follows:

```
err_t tcpip_input (, );
```

See the section called “System initialization” for further details on initialization.

Declarations for the API functions are found in header files within the lwIP include tree. The TCP functions are found in `<lwip/tcp.h>`, and UDP in `<lwip/udp.h>`.

The raw API uses many of the same types and definitions used in the sequential API. In particular the raw API functions use struct ip_addr and err_t error codes.

Callbacks

The configuration option `CYGFUN_LWIP_EVENT_CALLBACK` defaults to enabled. If enabled then program execution is driven by callbacks. Each callback is an ordinary C function that is called from within the TCP/IP code. Every callback function is passed the current TCP or UDP connection state as an argument. Also, in order to be able to keep program specific state, the callback functions are called with a program specified argument that is independent of the TCP/IP state.

If the `CYGFUN_LWIP_EVENT_CALLBACK` option is disabled then a common user-supplied function is called from within the TCP/IP code instead of the respective callback routine:

```
err_t lwip_tcp_event (arg, pcb, lwip_event, p, size, err);
```

For the individual callbacks or the shared `lwip_tcp_event()` the `tcp_arg()` function is used for setting the private `arg` application connection state.

Name

tcp_arg() — Set the application connection state

Synopsis

```
void tcp_arg (pcb, arg);
```

Description

The `tcp_arg()` function specifies the program specific state that should be passed to all other callback functions, or if configured the `lwip_tcp_event()` function. The "pcb" argument is the current TCP connection control block, and the "arg" argument is the argument that will be passed to the callbacks.

TCP connection setup

The functions used for setting up connections are similar to those of the sequential API and of the BSD socket API. A new TCP connection identifier (i.e., a protocol control block - PCB) is created with the `tcp_new()` function. This PCB can then be either set to listen for new incoming connections or be explicitly connected to another host.

Name

tcp_new() — Create a new TCP PCB

Synopsis

```
struct tcp_pcb *tcp_new ();
```

Description

Creates a new TCP connection identifier (PCB).

Return value

Returns the new PCB. If memory is not available for creating the new PCB, `NULL` is returned.

Name

tcp_bind() — Bind PCB to local IP address and port

Synopsis

```
err_t tcp_bind (pcb, ipaddr, port);
```

Description

Binds `pcb` to a local IP address and port number. The IP address can be specified as `IP_ADDR_ANY` in order to bind the connection to all local IP addresses.

Return value

If another connection is bound to the same port, the function will return `ERR_USE`, otherwise `ERR_OK` is returned.

Name

tcp_listen() — Make PCB listen for incoming connections

Synopsis

```
struct tcp_pcb *tcp_listen (pcb);
```

Description

Commands *pcb* to start listening for incoming connections. When an incoming connection is accepted, the function specified with the `tcp_accept()` function will be called. *pcb* must have been bound to a local port with the `tcp_bind()` function.

Return value

The `tcp_listen()` function returns a new connection identifier, and the one passed as an argument to the function will be deallocated. The reason for this behavior is that less memory is needed for a connection that is listening, so `tcp_listen()` will reclaim the memory needed for the original connection and allocate a new smaller memory block for the listening connection.

`tcp_listen()` may return `NULL` if no memory was available for the listening connection. If so, the memory associated with *pcb* will not be deallocated.

Name

tcp_accept() — Set callback used for new incoming connections

Synopsis

```
void tcp_accept (pcb, (*accept));
```

Description

Specify the callback function that should be called when a new connection arrives for a listening TCP PCB.

Name

tcp_connect() — Open connection to remote host

Synopsis

```
err_t tcp_connect (pcb, ipaddr, port, (*connected));
```

Description

Sets up *pcb* to connect to the remote host indicated by *ipaddr* on port *port* and sends the initial SYN segment which opens the connection.

The tcp_connect() function returns immediately; it does not wait for the connection to be properly set up. Instead, it will call the *connected()* function specified as the fourth argument when the connection is established. If the connection could not be properly established, either because the other host refused the connection or because the other host didn't answer, the *connected()* function will be called with its *err* argument set accordingly.

Return value

The tcp_connect() function can return ERR_MEM if no memory is available for enqueueing the SYN segment. If the SYN indeed was enqueued successfully, the tcp_connect() function returns ERR_OK.

Sending TCP data

TCP data is sent by enqueueing the data with a call to tcp_write(). When the data is successfully transmitted to the remote host, the application will be notified with a call to a specified callback function.

Name

tcp_write() — Enqueue data for transmission

Synopsis

```
err_t tcp_write (pcb, dataptr, len, copy);
```

Description

Enqueues the data pointed to by *dataptr*. The length of the data is passed in *len*. The argument *copy* may be either 0 or 1 and indicates whether the new memory should be allocated for the data to be copied into. If the argument is 0, no new memory should be allocated and the data should only be referenced by pointer.

Return value

The `tcp_write()` function will fail and return `ERR_MEM` if the length of the data exceeds the current send buffer size (as defined by the `CYGNUM_LWIP_TCP_SND_BUF` CDL configuration option) or if the length of the queue of outgoing segment is larger than the upper limit defined by the `CYGNUM_LWIP_TCP_SND_QUEUELEN` CDL configuration option. The number of bytes available in the output queue can be retrieved with the `tcp_sndbuf()` function:

```
u16_t tcp_sndbuf (pcb);
```

The proper way to use this function is to call the function with at most `tcp_sndbuf()` bytes of data. If the function returns `ERR_MEM`, the application should wait until some of the currently enqueued data has been successfully received by the other host and try again. This can be achieved with a callback function previously provided to `tcp_sent()`.

Name

tcp_sent() — Set callback for successful transmission

Synopsis

```
void tcp_sent (pcb, (*sent));
```

Description

Specifies the callback function that should be called when data has successfully been received (i.e. acknowledged) by the remote host. The *len* argument passed to the *sent* callback function gives the number of bytes that were acknowledged by the last acknowledgment.

Receiving TCP data

TCP data reception is callback based - an application specified callback function is called when new data arrives. When the application has taken the data, it has to call the `tcp_recved()` function to indicate that TCP can advertise an increase in the receive window.

Name

tcp_recv() — Set callback for incoming data

Synopsis

```
void tcp_recv (pcb, (*recv));
```

Description

Sets the callback function that will be called when new data arrives on the connection associated with *pcb*. The callback function will be passed a `NULL` pbuf to indicate that the remote host has closed the connection.

Name

tcp_recved() — Indicate receipt of data

Synopsis

```
void tcp_recved (pcb, len);
```

Description

This function must be called when the application has received the data. *len* indicates the length of the received data.

Application polling

When a connection is idle (i.e., no data is either transmitted or received), lwIP will repeatedly poll the application by calling a specified callback function. This can be used either as a watchdog timer for killing connections that have stayed idle for too long, or as a method of waiting for memory to become available. For instance, if a call to `tcp_write()` has failed because memory wasn't available, the application may use the polling functionality to call `tcp_write()` again when the connection has been idle for a while.

Name

tcp_poll() — Set application poll callback

Synopsis

```
void tcp_poll (pcb, interval, (*poll));
```

Description

Specifies the polling interval and the callback function that should be called to poll the application. The interval is specified in number of TCP coarse grained timer shots, which typically occurs twice a second. An interval of 10 means that the application would be polled every 5 seconds.

Closing connections, aborting connections and errors

Name

tcp_close() — Close the connection

Synopsis

```
err_t tcp_close (pcb);
```

Description

Closes the connection. The pcb is deallocated by the TCP code after a call to tcp_close().

Return value

The function may return `ERR_MEM` if no memory was available for closing the connection. If so, the application should wait and try again either by using the acknowledgment callback or the polling functionality. If the close succeeds, the function returns `ERR_OK`.

Name

tcp_abort() — Abort the connection

Synopsis

```
void tcp_abort (pcb);
```

Description

Aborts the connection by sending a RST (reset) segment to the remote host. *pcb* is deallocated. This function never fails.

If a connection is aborted because of an error, the application is alerted of this event by the callback previously registered with `tcp_err()`. Errors that might abort a connection are when there is a shortage of memory.

Name

tcp_err() — Set callback for errors

Synopsis

```
void tcp_err (pcb, (*err));
```

Description

Set callback function to be used on connection errors. The error callback function does not get the connection's pcb passed to it as a parameter since the pcb may already have been deallocated.

Lower layer TCP interface

TCP provides a simple interface to the lower layers of the system. During system initialization, the function `tcp_init()` has to be called before any other TCP function is called. When the system is running, the two timer functions `tcp_fasttmr()` and `tcp_slowtmr()` must be called at regular intervals. The `tcp_fasttmr()` should be called every `TCP_FAST_INTERVAL` milliseconds (defined in tcp.h, and currently 250ms) and `tcp_slowtmr()` should be called every `TCP_SLOW_INTERVAL` milliseconds, currently 500ms.

UDP interface

The UDP interface is similar to that of TCP, but due to the lower level of complexity of UDP, the interface is significantly simpler.

Name

udp_new() — Create a new UDP pcb

Synopsis

```
struct udp_pcb *udp_new ();
```

Description

Creates a new connection identifier (PCB) which can be used for UDP communication. The PCB is not active until it has either been bound to a local address or connected to a remote address.

Return value

Returns the new PCB. If memory is not available for creating the new PCB, `NULL` is returned.

Name

udp_remove() — Remove a UDP pcb

Synopsis

```
void udp_remove (pcb);
```

Description

Removes and deallocates *pcb*.

Name

udp_bind() — Bind PCB to local IP address and port

Synopsis

```
err_t udp_bind (pcb, ipaddr, port);
```

Description

Binds *pcb* to the local address indicated by *ipaddr* and port indicated by *port*. *ipaddr* can be IP_ADDR_ANY to indicate that it should listen to any local IP address. Port may be 0 for any port.

Return value

This function can return ERR_USE if all usable UDP dynamic ports are used (only relevant if *port* is 0. Otherwise udp_bind() will always return ERR_OK.

Name

udp_connect() — Set remote UDP peer

Synopsis

```
err_t udp_connect (pcb, ipaddr, port);
```

Description

Sets the remote end of *pcb*. This function does not generate any network traffic, but only sets the remote address of the pcb.

Return value

This function can return `ERR_USE` if all usable UDP dynamic ports are used. Otherwise `udp_connect()` will always return `ERR_OK`.

Name

udp_disconnect() — Set remote UDP peer

Synopsis

```
void udp_disconnect (pcb);
```

Description

Remove the remote end of *pcb*. This function does not generate any network traffic, but only removes the remote address of the pcb.

Name

udp_send() — Send UDP packet

Synopsis

```
err_t udp_send (pcb, p);
```

Description

Sends the pbuf *p* to the remote host associated with *pcb*. The pbuf is not deallocated.

Return value

This function returns `ERR_OK` on success; but may return `ERR_MEM` if there is insufficient memory to prepend a UDP header, or `ERR_RTE` if no suitable outgoing network interface could be found to route the packet on.

Name

udp_recv() — Set callback for incoming UDP data

Synopsis

`void` **`udp_recv`** `(`*`pcb`*`, `*`(*recv)`*`, `*`recv_arg`*`);`

Description

Registers a callback function *`recv`* with the PCB *`pcb`* so that when a UDP datagram is received, the callback is invoked. The callback argument *`arg`* is set as the argument *`recv_arg`* to `udp_recv()`. The received datagram packet buffer is held in *`p`*. The source address of the datagram is provided in *`addr`*, and the source port in *`port`*. The callback is expected to free the packet.

System initialization

When performing manual initialization of lwIP for use with the raw API, the function `lwip_init()` can be called to perform the core setup. Depending on the actual `lwipopts.h` configuration `lwip_init()` will call the necessary routines to initialize the required lwIP sub-systems.

In this example, these functions must be called in the order of appearance:

`lwip_init()`

Calls the individual, as configured, low-level lwIP module initialization routines.

If `LWIP_ARP` is defined then `etharp_tmr()` must be called at the regular `ARP_TMR_INTERVAL` interval (default 5 seconds) after the system has been initialized by this call.

Similarly if `LWIP_TCP` is defined then you must ensure that `tcp_fasttmr()` and `tcp_slowtmr()` are called at the predefined regular intervals.

`struct netif *netif_add(struct netif *netif, struct ip_addr *ipaddr, struct ip_addr *netmask, struct ip_addr *gw, void *state, err_t (* init)(struct netif *netif), err_t (* input)(struct pbuf *p, struct netif *netif))`

Adds your network interface to the `netif_list`. Allocate a struct netif and pass a pointer to this structure as the first argument. Give pointers to cleared struct ip_addr structures when using DHCP, or fill them with sane numbers otherwise. The state pointer may be NULL.

The *`init`* function pointer must point to an initialization function for your ethernet netif interface. The following code illustrates an example use:

```
err_t netif_if_init(struct netif *netif)
{
    u8_t i;

    for(i = 0; i < 6; i++)
        netif->hwaddr[i] = some_eth_addr[i];
    init_my_eth_device();
    return ERR_OK;
}
```

Normally for ethernet devices the *`input`* function must point to the lwIP function `ethernet_input()`.

`netif_set_default(struct netif *netif)`

Registers *`netif`* as the default network interface.

`netif_set_up(struct netif *netif)`

When *`netif`* is fully configured, this function must be called to allow it to be used.

`dhcp_start(struct netif *netif)`

If `LWIP_DHCP` is configured then this function creates a new DHCP client for this interface the first time the routine is called. Note: you must call `dhcp_fine_tmr()` and `dhcp_coarse_tmr()` at the predefined regular intervals after starting the client.

You can peek in the `netif->dhcp` struct for the actual DHCP status.

Initialization detail

If required the manual raw API initialization could directly call the required lwIP sub-system module initialization functions (rather then calling the `lwip_init()` function).

The calls should be performed in the following order:

`stats_init()`

Clears the structure where runtime statistics are gathered.

Note: The statistics support is only included if `LWIP_STATS` is configured, and then some of the statistics code is only present if `LWIP_DEBUG` is also defined.

`sys_init()`

Not generally used with raw API, but can be called for ease of compatibility if using sequential API in addition, initialised manually. The `lwip_init()` implementation only calls this function if `NO_SYS` is **NOT** defined.

`mem_init()`

Initializes the dynamic memory heap defined by the CDL configuration option `CYGNUM_LWIP_MEM_SIZE`.

`memp_init()`

Initializes the memory pools defined by the CDL configuration options `CYGNUM_LWIP_MEMP_NUM_*`.

`pbuf_init()`

Initializes the pbuf (packet buffer) memory pool defined by the CDL configuration option `CYGNUM_LWIP_PBUF_POOL_SIZE`.

`netif_init()`

This function will call `netif_add()` as appropriate to create the `LWIP_HAVE_LOOPIF` configured loopback network interface.

`lwip_socket_init()`

If `LWIP_SOCKET` is configured then this function is called to initialise the BSD-alike API module. It does not do much at present, but it should be called to handle future changes.

`ip_init()`

This function does not do much at present, but it should be called to handle future changes.

`etharp_init()`

Called if `LWIP_ARP` is configured to initialize the ARP table and queue.

Note: you must regularly call the `etharp_tmr` function at the `ARP_TMR_INTERVAL` (default 5 seconds) interval after this initialization.

`raw_init()`

If `LWIP_RAW` is configured then this function is called. It does not do much at present, but it should be called to handle future changes.

`udp_init()`

If `LWIP_UDP` is configured then this function is called to initialize the required UDP support state.

`tcp_init()`

If `LWIP_TCP` is configured then this function is called to initialise the required TCP support state.

Note: you must call `tcp_fasttmr()` and `tcp_slowtmr()` at the predefined regular intervals after this initialization.

`snmp_init()`

If `LWIP_SNMP` is configured then this function is called to start the SNMP agent support. It allocates a UDP pcb and binds it to `IP_ADDR_ANY` for the `SNMP_IN_PORT` (default `161`) configured port, listening for SNMP The routine will also generate a SNMP coldstart trap if configured appropriately.

`autoip_init()`

If `LWIP_AUTOIP` is configured then this function is called for the IPv4 AutoIP support. It does not do much at present, but it should be called to handle future changes.

`igmp_init()`

If `LWIP_IGMP` is configured then this function is called for the IPv4 IGMP support. It configures the `allsystems` and `allroutes` multicast addresses.

`dns_init()`

If `LWIP_DNS` is configured then this function is called to allocate the UDP pcb for the client and initialise the default DNS server address.

`ip6_init()`

If `LWIP_IPV6` is configured then this function is called. It does not do much at present, but it should be called to handle future changes.

`nd6_init()`

If `LWIP_IPV6` is configured then this function is called. It does not do much at present, but it should be called to handle future changes.

`mld6_init()`

If `LWIP_IPV6` and `LWIP_IPV6_MLD` are configured then this function is called. It does not do much at present, but it should be called to handle future changes.

`sys_timeouts_init()`

If `LWIP_TIMERS` is configured then this function is called. It uses the `sys_timeout()` to register timeout callbacks for the configured lwIP features. Normally the raw API will not be providing the `sys_timeout` functionality, and will, as mentioned above, have to manually ensure the relevant timeout functions are called. e.g. ARP, TCP, etc.

Chapter 166. Debug and Test

Debugging

Some explicit lwIP configuration items exist to aid with debugging problems. These, along with some other suggestions, are documented in the sections below.

Asserts

An initial starting point for checking valid operation is to enable lwIP asserts. The `CYGDBG_LWIP_ASSERTS` option turns on run-time asserts that can usually detect problems before they reach a fatal target/platform exception.

The lwIP asserts are based on the standard eCos assertion support, so will normally stop the code in a busy loop if triggered. Normally when debugging it is usual to set a breakpoint on the entry to the `cyg_assert_fail()` function so that debugger access to application state can be performed.

Memory Allocations

Run-time validation of the memory pools can be enabled in the lwIP configuration by setting one or more of the following options:

Sanity check memory pools (`CYGDBG_LWIP_MEMP_SANITY_CHECK`)

> If enabled lwIP will perform extra sanity checking of the memory pools every time an item is released.

Memory pool overflow checks (`CYGDBG_LWIP_MEMP_OVERFLOW_CHECK`)

> This configuration option can currently be set to the value 0, 1 or 2.
>
> If set to 0 then the feature is disabled (the default configuration), with the non-zero values enabling code to perform MEMP under-/over-flow checking.
>
> If set to 1 then a buffer boundary check is performed when an item is released.
>
> If set to 2 then the code performs the buffer boundary check on every item, in every pool, every time an allocation or release operation is performed. This, obviously, will be slow. However, it will normally provide quicker detection of buffer problems.
>
> For the non-zero configurations the options `CYGDBG_LWIP_MEMP_SANITY_REGION_BEFORE` and `CYGDBG_LWIP_MEMP_SANITY_REGION_AFTER` respectively define the size (in bytes) of the *catch* areas placed before and after all allocations.

Statistics

The lwIP stack provides support for tracking statistics via enabling the Trafficstatistics `CYGDBG_LWIP_STATS` option. These statistics have been briefly covered in the section called “Performance”. For debugging the error count (normally the `err` field of the relevant statistics structure) can be useful in indicating resource issues, some of which will result in the stack failing to operate correctly.

GDB/RedBoot

Some standard platforms may, by default, provide the RedBoot debug monitor, which in turn may be configured to allow remote network GDB debugging connections. It should be noted that a limitation exists where an eCos application configured to use a lwIP Direct driver **CANNOT** be debugged via such a remote network GDB connection due to interaction between the RedBoot use of a Standard device driver and the application Direct device driver models. Using GDB via a UART or hardware debug connection is not affected.

In practice this is not normally an issue since low-level debugging, when developing for such low resource platforms that require the use of the lwIP Direct device driver, is normally performed via a hardware debug interface (e.g. JTAG).

Host Tools

An invaluable tool to aid debugging of both network protocol stack problems and application level network interaction is WireShark. It can be used to log packets, and to trace connection streams, whilst providing human readable data dumps.

The eCos synthetic ethernet target support can be a useful aid in separating application level networking problems from issues with the under-lying network transport stack (e.g. lwIP). It can usefully be used to debug higher-level network applications in a resource rich environment, before tuning the code for the resource-restricted lwIP target platform.

Testing

Some test applications are built if the active eCos lwIP configuration is suitable:

- **lwipsnmp**
- **lwipsntp**
- **lwiperf**
- **unitwrap**

If the configuration option `CYGBLD_NET_LWIP_BUILD_MANUAL_TESTS` is enabled then a further set of simple tests are built. Note: The option is disabled by default. These manually executed are just basic verification tests and are not designed to be an exhaustive test of all lwIP or TCP/IP networking features.

For the manual tests frag and udpecho the host tool nc (netcat) can be used to interface with the test. In the following sections any examples given of using the tests assume that the Unit Under Test (UUT) is at the local network IPv4 address `192.168.1.200`. The actual local addresses for the UUT should be ascertained and substituted accordingly.

Note

Most of the manual tests are currently limited to accepting IPv4 connections. The exceptions are socket and tcpecho which will accept IPv4 or IPv6 connections.

lwipsnmp

This is a simple test to exercise the SNMP agent when enabled via the `CYGFUN_LWIP_SNMP` option. It relies on the `hostsnmp.sh` script being executed on a host system to exercise a set of SNMP operations agsinst the target executing this application.

lwipsntp

This application tests the use of the lwIP SNTP client implementation when enabled via the `CYGFUN_LWIP_SNTP` option. It initialises the SNTP client code and then waits to receive a valid time.

lwiperf

This test can be used to exercise the iPerf2 server provided by lwIP when it is enabled in the configuration using the `CYGFUN_LWIP_LWIPERF` option. It requests the test host to exercise a **iperf-c** against the target executing this application as a simple demonstration of measuring the achievable network bandwidth.

unitwrap

If the eCos lwip configuration meets the requirements (re. memory configuration, TCP options, etc.) for the standard lwIP check-framework unit tests then this test application is built. The application is a wrapper to support the lwIP unit tests and allows verification of some lwIP features.

socket

This is a very simple TCP protocol test using the BSD-alike socket API. The test will listen for two IPv4 or IPV6 connections on port 7. For each connection established the test will continue to echo the data received on the TCP stream until the particular connection is closed.

The nc utility can be used to communicate with the test program.

```
nc 192.168.1.200 7
```

After starting nc the UUT will acknowledge the connection by displaying:

```
PASS:<Received connection OK>
```

As this point it will wait for a line of text to be input and completed by pressing the **Enter** key. Multiple lines of text can be entered, and should be echoed back to indicate that the UUT has received and responded OK. Entering **Ctrl-D** will terminate the nc connection.

Another execution of nc as above will complete the test.

tcpecho

This is a very simple TCP protocol test. See the section called “socket” for a description of the test, since it has identical requirements to that example.

udpecho

This is a very simple UDP protocol test, listening for two connections on port 7 and echo-ing back the data received.

The nc utility can be used to communicate with the test program.

```
nc -u 192.168.1.200 7
```

After starting nc it will wait for a line of text to be input and completed by pressing the **Enter** key. The nc will then perform the necessary UDP connection to the specified port, transmit the data and output any reply. Another line of text can then be entered to complete the test. Unlike the tcpecho a single nc execution can be used for the test, since each line transmitted counts as a single UDP connection test.

frag

This is a simple test that should result in fragmented TCP packets being transmitted. The test can be exercised like the section called “udpecho” since it listens for two UDP connections on port 7.

The major difference being that after echoing the received data the UUT will transmit a large amount of data to the host nc.

nc_test_slave

This test provides a client for use with the host side `CYGPKG_NET` nc_test_master application. The host side test code must be manually built within the `packages/net/common/`*`vsn`*`/tests` directory.

When the UUT is started it will initially perform some slave performance calculations before it will start listening for connections.

httpd

This is a very simple HTTP daemon that will listen for two `HTTP GET` connections on port 80. Currently only IPv4 socket connections are listened for. The test when started will display some information on the diagnostic channel, including:

```
PASS:<Listening on TCP port 80>
INFO:<Will wait for two HTTP connections>
```

When it is has displayed the Will wait … message the UUT is ready to be accessed from the test host. The following example uses the host tool wget to perform such a page fetch, and can be executed twice to perform the test. e.g.

```
wget http://192.168.7.165/
```

After the second `GET` operation the test will exit.

httpd2

This is a slightly more realistic HTTP daemon test, that will execute indefinitely. This test is a useful example of using the raw API, and could form the basis of a simple, lightweight, webserver.

Currently it will only accept IPv4 socket connections on port 80. On startup it will not display any diagnostic output other than the `cyg_lwip_netif_print_info( netif_default, diag_printf )` output displaying the default network interface address information.

A standard web browser can be used to access the pages served by the daemon, returning a simple demonstration page as the root document. The test has been designed to be extendible to easily support multiple pages.

lookup

If IPv4 DNS support is configured then this test performs some simple verification using some known lookups against the locally configured DNS address (e.g. obtained via DHCP), and then again using a known fixed DNS server.

sys_timeout

This is a simple stand-alone test of the lwIP internal timeout handling. No external interaction is required.

lwiphttpd

This is a build of the standard lwIP example HTTPD application provided when the option `CYGFUN_LWIP_HTTPD` is enabled. It provides a static (built-in) web-page.

If the lwIP configuration option `CYGFUN_LWIP_MBEDTLS` is enabled then the test also provides a TLS daemon (port 443) using a built-in self-signed certificate as a basic example of the lwIP ALTCP TLS support.

Note

For eCos the `CYGFUN_LWIP_MBEDTLS` option is only available when the `CYGFUN_LWIP_ALTCP_TLS` option is enabled, in turn controlled by the support for the lwIP ALTCP being enabled by the `CYGFUN_LWIP_ALTCP` feature. The developer should be aware that you need to provide your own ALTCP wrapper functions if `LWIP_ALTCP_TLS` is defined but not the `CYGPKG_MBEDTLS` integration option `LWIP_ALTCP_TLS_MBEDTLS`.

Part XLV. Ethernet Device Support

Documentation for drivers of this type is often integrated into the eCos board support documentation. You should review the documentation for your target board for details. Standalone and more generic drivers are documented in the following sections.

Table of Contents

Chapter 167. Writing Ethernet Device Drivers

Generic Ethernet API

This section provides a simple description of how to write a low-level, hardware dependent ethernet driver. In eCos this is known as a “standard” driver.

There is a high-level driver (which is only code — with no state of its own) that is part of the stack. There will be one or more low-level drivers tied to the actual network hardware. Each of these drivers contains one or more driver instances. The intent is that the low-level drivers know nothing of the details of the stack that will be using them. Thus, the same driver can be used by the eCos supported TCP/IP stack, RedBoot, or any other, with no changes.

A driver instance is contained within a struct eth_drv_sc:

```
struct eth_hwr_funs {
    // Initialize hardware (including startup)
    void (*start)(struct eth_drv_sc *sc,
                  unsigned char *enaddr,
                  int flags);
    // Shut down hardware
    void (*stop)(struct eth_drv_sc *sc);
    // Device control (ioctl pass-thru)
    int  (*control)(struct eth_drv_sc *sc,
                    unsigned long key,
                    void *data,
                    int   data_length);
    // Query - can a packet be sent?
    int  (*can_send)(struct eth_drv_sc *sc);
    // Send a packet of data
    void (*send)(struct eth_drv_sc *sc,
                 struct eth_drv_sg *sg_list,
                 int sg_len,
                 int total_len,
                 unsigned long key);
    // Receive [unload] a packet of data
    void (*recv)(struct eth_drv_sc *sc,
                 struct eth_drv_sg *sg_list,
                 int sg_len);
    // Deliver data to/from device from/to stack memory space
    // (moves lots of memcpy()s out of DSRs into thread)
    void (*deliver)(struct eth_drv_sc *sc);
    // Poll for interrupts/device service
    void (*poll)(struct eth_drv_sc *sc);
    // Get interrupt information from hardware driver
    int (*int_vector)(struct eth_drv_sc *sc);
    // Logical driver interface
    struct eth_drv_funs *eth_drv, *eth_drv_old;
};

struct eth_drv_sc {
    struct eth_hwr_funs *funs;
    void                *driver_private;
    const char          *dev_name;
    int                  state;
    struct arpcom        sc_arpcom; /* ethernet common */
};
```

Note

If you have two instances of the same hardware, you only need one struct eth_hwr_funs shared between them.

There is another structure which is used to communicate with the rest of the stack:

```
struct eth_drv_funs {
    // Logical driver - initialization
    void (*init)(struct eth_drv_sc *sc,
                 unsigned char *enaddr);
    // Logical driver - incoming packet notifier
    void (*recv)(struct eth_drv_sc *sc,
                 int total_len);
    // Logical driver - outgoing packet notifier
    void (*tx_done)(struct eth_drv_sc *sc,
                    CYG_ADDRESS key,
                    int status);
};
```

Your driver does *not* create an instance of this structure. It is provided for driver code to use in the eth_drv member of the function record. Its usage is described below in the section called “Upper Layer Functions”

One more function completes the API with which your driver communicates with the rest of the stack:

```
extern void eth_drv_dsr(cyg_vector_t vector,
                        cyg_ucount32 count,
                        cyg_addrword_t data);
```

This function is designed so that it can be registered as the DSR for your interrupt handler. It will awaken the “Network Delivery Thread” to call your deliver routine. See the section called “Deliver function”.

You create an instance of struct eth_drv_sc using the `ETH_DRV_SC()` macro which sets up the structure, including the prototypes for the functions, etc. By doing things this way, if the internal design of the ethernet drivers changes (e.g. we need to add a new low-level implementation function), existing drivers will no longer compile until updated. This is much better than to have all of the definitions in the low-level drivers themselves and have them be (quietly) broken if the interfaces change.

The “magic” which gets the drivers started (and indeed, linked) is similar to what is used for the I/O subsystem. This is done using the `NETDEVTAB_ENTRY()` macro, which defines an initialization function and the basic data structures for the low-level driver.

```
  typedef struct cyg_netdevtab_entry {
      const char        *name;
      bool             (*init)(struct cyg_netdevtab_entry *tab);
      void              *device_instance;
      unsigned long     status;
  } cyg_netdevtab_entry_t;
```

The `device_instance` entry here would point to the struct eth_drv_sc entry previously defined. This allows the network driver setup to work with any class of driver, not just ethernet drivers. In the future, there will surely be serial PPP drivers, etc. These will use the `NETDEVTAB_ENTRY()` setup to create the basic driver, but they will most likely be built on top of other high-level device driver layers.

To instantiate itself, and connect it to the system, a hardware driver will have a template (boilerplate) which looks something like this:

```
#include <cyg/infra/cyg_type.h>
#include <cyg/hal/hal_arch.h>
#include <cyg/infra/diag.h>
#include <cyg/hal/drv_api.h>
#include <cyg/io/eth/netdev.h>
#include <cyg/io/eth/eth_drv.h>

```

```
ETH_DRV_SC(DRV_sc,
           0,             // No driver specific data needed
           "eth0",        // Name for this interface
           HRDWR_start,
           HRDWR_stop,
           HRDWR_control,
           HRDWR_can_send
           HRDWR_send,
           HRDWR_recv,
           HRDWR_deliver,
           HRDWR_poll,
           HRDWR_int_vector
);

NETDEVTAB_ENTRY(DRV_netdev,
                "DRV",
                DRV_HRDWR_init,
                &DRV_sc);
```

This, along with the referenced functions, completely define the driver.

Note

If one needed the same low-level driver to handle multiple similar hardware interfaces, you would need multiple invocations of the `ETH_DRV_SC()`/`NETDEVTAB_ENTRY()` macros. You would add a pointer to some instance specific data, e.g. containing base addresses, interrupt numbers, etc, where the

```
        0, // No driver specific data
```

is currently.

Review of the functions

Now a brief review of the functions. This discussion will use generic names for the functions — your driver should use hardware-specific names to maintain uniqueness against any other drivers.

Init function

```
static bool DRV_HDWR_init(struct cyg_netdevtab_entry *tab)
```

This function is called as part of system initialization. Its primary function is to decide if the hardware (as indicated via tab->device_instance) is working and if the interface needs to be made available in the system. If this is the case, this function needs to finish with a call to the ethernet driver function:

```
    struct eth_drv_sc *sc = (struct eth_drv_sc *)tab->device_instance;
    ….initialization code….
    // Initialize upper level driver
    (sc->funs->eth_drv->init)( sc, unsigned char *enaddr );
```

where *`enaddr`* is a pointer to the ethernet station address for this unit, to inform the stack of this device's readiness and availability.

Note

The ethernet station address (ESA) is supposed to be a world-unique, 48 bit address for this particular ethernet interface. Typically it is provided by the board/hardware manufacturer in ROM.

In many packages it is possible for the ESA to be set from RedBoot, (perhaps from 'fconfig' data), hard-coded from CDL, or from an EPROM. A driver should choose a run-time specified ESA (e.g. from RedBoot) preferentially, otherwise (in order) it should use a CDL specified ESA if one has been set, otherwise an EPROM set ESA, or otherwise fail. See the `cl/cs8900a` ethernet driver for an example.

Start function

```
static void
HRDWR_start(struct eth_drv_sc *sc, unsigned char *enaddr, int flags)
```

This function is called, perhaps much later than system initialization time, when the system (an application) is ready for the interface to become active. The purpose of this function is to set up the hardware interface to start accepting packets from the network and be able to send packets out. The receiver hardware should not be enabled prior to this call.

Notes:

- This function will be called whenever the up/down state of the logical interface changes, e.g. when the IP address changes, or when promiscuous mode is selected by means of an `ioctl()` call in the application. This may occur more than once, so this function needs to be prepared for that case.
- In future, the *`flags`* field (currently unused) may be used to tell the function how to start up, e.g. whether interrupts will be used, alternate means of selecting promiscuous mode etc.

Stop function

```
static void HRDWR_stop(struct eth_drv_sc *sc)
```

This function is the inverse of “start.” It should shut down the hardware, disable the receiver, and keep it from interacting with the physical network.

Control function

```
static int
HRDWR_control(
        struct eth_drv_sc *sc, unsigned long key,
        void *data, int len)
```

This function is used to perform low-level “control” operations on the interface. These operations would typically be initiated via `ioctl()` calls in the BSD stack, and would be anything that might require the hardware setup to change (i.e. cannot be performed totally by the platform-independent layers).

The *`key`* parameter selects the operation, and the *`data`* and *`len`* params point describe, as required, some data for the operation in question.

Warning

Debugging of applications or execution of tests that use low-level filtering is strongly discouraged when connecting over an ethernet connection to RedBoot.

In such instances the ethernet device is shared between eCos and RedBoot. Low-level “control” operations instructing the device to filter ethernet packets by IP address, port or VLAN can filter ethernet packets destined to or from RedBoot.

Where these is no alternative, the developer must ensure that their application does not filter away ethernet packets to or from RedBoot by adjusting the filters accordingly. e.g. Ensure that the RedBoot TCP port (default 9000) and address are never filtered out.

Certain network tests (e.g. `control`) will detect when such a connection is made and either report that the test is NOTAPPLICABLE or skip over the filtering portion of the test.

Available Operations:

ETH_DRV_SET_MAC_ADDRESS

This operation sets the ethernet station address (ESA or MAC) for the device. Normally this address is kept in non-volatile memory and is unique in the world. This function must at least set the interface to use the new address. It may also update the NVM as appropriate.

ETH_DRV_GET_IF_STATS_UD
ETH_DRV_GET_IF_STATS

These acquire a set of statistical counters from the interface, and write the information into the memory pointed to by *data*. The "UD" variant explicitly instructs the driver to acquire up-to-date values. This is a separate option because doing so may take some time, depending on the hardware.

The definition of the data structure is in `cyg/io/eth/eth_drv_stats.h`.

This call is typically made by SNMP.

ETH_DRV_SET_MC_LIST

This entry instructs the device to set up multicast packet filtering to receive only packets addressed to the multicast ESAs in the list pointed to by *data*.

The format of the data is a 32-bit count of the ESAs in the list, followed by packed bytes which are the ESAs themselves, thus:

```
struct eth_drv_mc_list {
    int len;
    unsigned char addrs[CYGNUM_IO_ETH_DRIVERS_FILTER_LIST_SIZE][ETHER_ADDR_LEN];
};
```

Pass an empty list (*len*=0) to clear any existing multicast filters.

Some driver/hardware combinations can support a large number of ESAs, which can lead to a very large `struct eth_drv_mc_list` object if all the available address slots are supported. The `CYGNUM_IO_ETH_DRIVERS_FILTER_LIST_SIZE` CDL option can be tuned to reflect the upper limit required by an application configuration to minimise the overhead of passing unnecessarily large `struct eth_drv_mc_list` objects around.

ETH_DRV_SET_MC_ALL

This entry instructs the device to receive all multicast packets, and delete any explicit filtering which had been set up.

ETH_DRV_SET_DA_LIST

This entry allows a list of unicast-DA (Destination Address) values to be supplied, and any perfect filtering supported by the underlying driver to be configured appropriately.

The `eth_drv_filter_list_t` structure is used to provide the unicast-DA addresses to replace any existing DA filtering in place.

Pass an empty list (*len*=0) to clear any existing unicast-DA filters.

ETH_DRV_SET_SA_LIST

This entry allows a `eth_drv_filter_list_t` supplied list of SA (Source Address) filters to be specified.

Pass an empty list (*len*=0) to clear any existing SA filters.

ETH_DRV_FILTER_OPTIONS

This entry is provided as a single API for get/set of multiple filtering options (minimising the number of calls and the code required). It uses a standard AND/EOR approach to provide a single get/set interface.

For example, assuming the variable declaration:

```
struct eth_drv_options fo;
```

then the following will perform a GET without changing any flag state:

```
fo.u.mand = 0xFFFFFFFF;
fo.eor = 0x00000000;
```

To set an explicit value then the corresponding flag bit can be 0 for the AND. e.g.

```
fo.u.mand = 0x00000000;
fo.eor = 0x12345678;
```

To set an explicit value then the corresponding flag bit can be 0 for the AND. e.g. to set flag bit-0 to regardless of the current state:

```
fo.u.mand = 0xFFFFFFFE;
fo.eor = 0x00000001;
```

Toggling bits can also be supported. e.g. to toggle bit-2 and bit-4:

```
fo.u.mand = 0xFFFFFFFF;
fo.eor = 0x00000014;
```

After a successful request the `eth_drv_options` field `u.val` is updated to reflect the current driver option flag state after any changes that may have been requested.

Currently the following flags are defined, but not all drivers will necessarily support all the features:

The `ETH_DRV_FILTER_OPT_PROMISC` flag is provided as an alternative to the existing `ETH_DRV_SET_PROMISC` *`key`* option, just so that the control of the feature can be managed along with the other flags. It controls promiscuous mode.

The `ETH_DRV_FILTER_OPT_BLOCK_BCAST` flag controls whether all broadcast frames are dropped.

The `ETH_DRV_FILTER_OPT_INVERSE_DA` controls whether any enabled unicast DA (Destination Address) or multicast filtering (as set via `ETH_DRV_SET_DA_LIST` or `ETH_DRV_SET_MC_LIST`) operates in inverse filtering mode where matches are dropped, and non-matching frames are allowed through.

The `ETH_DRV_FILTER_OPT_INVERSE_SA` controls whether any enabled SA (Source Address) filtering (set by `ETH_DRV_SET_SA_LIST`) operates in inverse filtering mode where SA matches are dropped and non-SA matches allowed.

The `ETH_DRV_FILTER_OPT_L4_TCPUDP_ONLY` flag controls whether any enabled L4 filtering will drop all non-TCP and non-UDP packets. e.g. ICMP.

ETH_DRV_SET_FILTER_L3L4

This entry allows a L3 and/or L4 filter to be added. The `struct eth_drv_filter_l3l4` descriptor provides the filter configuration settings. The port numbers, IPv4 and IPv6 addresses must all be provided in network byte order.

The `flags` field is a combination of binary (boolean) flags describing the filter to be applied:

`ETH_DRV_L3L4_L3SRC` is set when the supplied L3 source address (SA) should be used for the match.

`ETH_DRV_L3L4_L3SRC_IPV6` is used to distinguish the type of SA supplied: unset (0) for IPv4, and set (1) for IPv6.

`ETH_DRV_L3L4_L3SRC_INV` is set when an inverted SA match should be configured.

`ETH_DRV_L3L4_L3SRC_MASK` is set when the `l3src_mb` SA bitmask should be applied.

`ETH_DRV_L3L4_L3DST` is set when the supplied L3 destination address (DA) should be used for the match.

`ETH_DRV_L3L4_L3DST_IPV6` is used to distinguish the type of DA supplied: unset (0) for IPv4, and set (1) for IPv6.

`ETH_DRV_L3L4_L3DST_INV` is set when an inverted DA match should be configured.

`ETH_DRV_L3L4_L3DST_MASK` is set when the `l3dst_mb` DA bitmask should be applied.

`ETH_DRV_L3L4_L4SRC` is set when a L4 source port filter should be applied as supplied in `l4_src`.

`ETH_DRV_L3L4_L4SRC_UDP` is set for L4 source UDP match, and unset for TCP.

`ETH_DRV_L3L4_L4SRC_INV` is set when an inverted L4 source match should be applied.

`ETH_DRV_L3L4_L4DST` is set when a L4 destination port filter should be applied as supplied in `l4_dst`.

`ETH_DRV_L3L4_L4DST_UDP` is set for L4 destination UDP match, and unset for TCP.

`ETH_DRV_L3L4_L4DST_INV` is set when an inverted L4 destination match should be applied.

Note

Not all drivers may support all of the L3/L4 filtering options available in this API. The developer should be aware of the features and limitations of the underlying Ethernet hardware MAC interface (and driver) in use.

ETH_DRV_CLR_FILTER_L3L4

This entry allows a L3/L4 filter to be removed. The `eth_drv_filter_l3l4` structure should be populated as per the original `ETH_DRV_SET_FILTER_L3L4` call.

ETH_DRV_SET_VLANTAG

This operation instructs the device to configure a single, perfect, VLAN Tag filter.

The passed `eth_drv_vlantag` structure defines the VLAN Tag to be set for the filter, along with control flags that can affect the operation. The following values can be ORed into the `flags` field to control the filter:

`ETH_DRV_VLANTAG_FLG_INVERSE` if set configures the filter as an inverted match; where only packets matching the VLAN Tag are dropped.

`ETH_DRV_VLANTAG_FLG_12BIT` configures the driver to only match against the least-significant 12-bits of the supplied `vt` field.

`ETH_DRV_VLANTAG_FLG_SVLAN` configures the driver to also accept the S-VLAN Tag (`0x88A8`) as a valid match.

The special flag `ETH_DRV_VLANTAG_FLG_DISABLE` is used to disable the VLAN Tag driver feature. The other `flags` settings are ignored, as-is the `vt` value.

ETH_DRV_GET_VLANTAG

This call will return the current VLAN Tag filter setting in the supplied `eth_drv_vlantag` structure. The extra flag `ETH_DRV_VLANTAG_FLG_VALID` in the returned `flags` field indicates whether a valid VLAN Tag filter has been set, and whether the contents of the structure can be interpreted.

ETH_DRV_OPTIONS

This entry allows control of the operation of the underlying device driver. It is provided as a single API for get/set of multiple options using a standard AND/EOR approach, though currently only the RX interrupt-vs-polled option is provided.

See `ETH_DRV_FILTER_OPTIONS` for more detail regarding using AND/EOR for get and set operations.

The RX operation mode is controlled by multiple bits covered by the `ETH_DRV_OPTION_RX_MODE_MASK`.

The `ETH_DRV_OPTION_RX_INT` mode selects interrupt driven RX mode and is the default driver mode.

The `ETH_DRV_OPTION_RX_POLL` mode, where supported by the underlying driver, selects a RX polled mode of operation. Normally this would not be desirable, but for some applications the (undefined) interrupt overhead of a high rate of RX activity may adversely affect the performance of other subsystems; such that limiting RX reception (at the cost of increased missed packets) is desired.

Note

The polled operation is less efficient with CPU bandwidth than the normal interrupt driven driver mode so throughput will be lower when polled mode is selected.

Care should be taken with the driver specific poll-period selected since high-frequency polling when a high-priority networking stack control thread is in use can be just as "bad" as an interrupt storm in denying other threads CPU time.

The use of the `ETH_DRV_OPTION_RX_POLL` mode should be viewed as an option in extremis. Suitable selection of the network thread priority levels, in conjunction with the driver and network stack buffering options, should allow for correct application operation in a well constructed application when present on a RX saturated network. If supported by the driver then H/W filtering options can further reduce the S/W load of the system hopefully avoiding the need to switch to polled RX.

The `ETH_DRV_OPTION_RX_AUTO` mode, where supported by the underlying driver, selects a mode of operation where (under driver specific configuration) the driver will switch between interrupt and polled modes depending on the RX activity. This can be used to ensure that if the driver and network stack are receiving high volumes of data that the RX interrupt load of the system can be reduced by throttling RX reception using the polled mode.

This function should return zero if the specified operation was completed successfully. It should return non-zero if the operation could not be performed, for any reason.

Can-send function

```
static int HRDWR_can_send(struct eth_drv_sc *sc)
```

This function is called to determine if it is possible to start the transmission of a packet on the interface. Some interfaces will allow multiple packets to be "queued" and this function allows for the highest possible utilization of that mode.

Return the number of packets which could be accepted at this time, zero implies that the interface is saturated/busy.

Send function

```
struct eth_drv_sg {
    CYG_ADDRESS  buf;
    CYG_ADDRWORD len;
};

static void
HRDWR_send(
        struct eth_drv_sc *sc,
        struct eth_drv_sg *sg_list, int sg_len,
        int total_len, unsigned long key)
```

This function is used to send a packet of data to the network. It is the responsibility of this function to somehow hand the data over to the hardware interface. This will most likely require copying, but just the address/length values could be used by smart hardware.

Note

All data in/out of the driver is specified via a “scatter-gather” list. This is just an array of address/length pairs which describe sections of data to move (in the order given by the array), as in the struct eth_drv_sg defined above and pointed to by *`sg_list`*.

Once the data has been successfully sent by the interface (or if an error occurs), the driver should call `(sc->funs->eth_drv->tx_done)()` (see the section called “Callback Tx-Done function”) using the specified *`key`*. Only then will the upper layers release the resources for that packet and start another transmission.

Note

In future, this function may be extended so that the data need not be copied by having the function return a “disposition” code (done, send pending, etc). At this point, you should move the data to some “safe” location before returning.

Deliver function

```
static void
HRDWR_deliver(struct eth_drv_sc *sc)
```

This function is called from the “Network Delivery Thread” in order to let the device driver do the time-consuming work associated with receiving a packet — usually copying the entire packet from the hardware or a special memory location into the network stack's memory.

After handling any outstanding incoming packets or pending transmission status, it can unmask the device's interrupts, and free any relevant resources so it can process further packets.

It will be called when the interrupt handler for the network device has called

```
    eth_drv_dsr( vector, count, (cyg_addrword_t)sc );
```

to alert the system that “something requires attention.” This `eth_drv_dsr()` call must occur from within the interrupt handler's DSR (not the ISR) or actually *be* the DSR, whenever it is determined that the device needs attention from the foreground. The third parameter (*`data`* in the prototype of `eth_drv_dsr()` *must* be a valid struct eth_drv_sc pointer `sc`.

The reason for this slightly convoluted train of events is to keep the DSR (and ISR) execution time as short as possible, so that other activities of higher priority than network servicing are not denied the CPU by network traffic.

To deliver a newly-received packet into the network stack, the deliver routine must call the following which will in turn call the receive function, which we talk about next.

```
(sc->funs->eth_drv->recv)(sc, len);
```

See also the section called “Callback Receive function” below.

Receive function

```
static void
HRDWR_recv(
        struct eth_drv_sc *sc,
        struct eth_drv_sg *sg_list, int sg_len)
```

This function is a call back, only invoked after the upper-level function

```
(sc->funs->eth_drv->recv)(struct eth_drv_sc *sc, int total_len)
```

has been called itself from your deliver function when it knows that a packet of data is available on the interface. The `(sc->funs->eth_drv->recv)()` function then arranges network buffers and structures for the data and then calls `HRDWR_recv()` to actually move the data from the interface.

A scatter-gather list (struct eth_drv_sg) is used once more, just like in the send case.

Poll function

```
static void
HRDWR_poll(struct eth_drv_sc *sc)
```

This function is used when in a non-interrupt driven system, e.g. when interrupts are completely disabled. This allows the driver time to check whether anything needs doing either for transmission, or to check if anything has been received, or if any other processing needs doing.

It is perfectly correct and acceptable for the poll function to look like this:

```
static void
HRDWR_poll(struct eth_drv_sc *sc)
{
   my_interrupt_ISR(sc);
   HRDWR_deliver(struct eth_drv_sc *sc);
}
```

provided that both the ISR and the deliver functions are idempotent and harmless if called when there is no attention needed by the hardware. Some devices might not need a call to the ISR here if the deliver function contains all the “intelligence.”

Interrupt-vector function

```
static int
HRDWR_int_vector(struct eth_drv_sc *sc)
```

This function returns the interrupt vector number used for receive interrupts. This is so that the common GDB stubs can detect when to check for incoming “CTRL-C” packets (used to asynchronously halt the application) when debugging over ethernet. The GDB stubs need to know which interrupt the ethernet device uses so that they can mask or unmask that interrupt as required.

Upper Layer Functions

Upper layer functions are called by drivers to deliver received packets or transmission completion status back up into the network stack.

These functions are defined by the hardware independent upper layers of the networking driver support. They are present to hide the interfaces to the actual networking stack so that the hardware drivers may be used by different network stack implementations without change.

These functions require a pointer to a struct eth_drv_sc which describes the interface at a logical level. It is assumed that the low level hardware driver will keep track of this pointer so it may be passed “up” as appropriate.

Callback Init function

```
void (sc->funs->eth_drv->init)(
                struct eth_drv_sc *sc, unsigned char *enaddr)
```

This function establishes the device at initialization time. It should be called once per device instance only, from the initialization function, if all is well (see the section called “Init function”). The hardware should be totally initialized (*not* “started”) when this function is called.

Callback Tx-Done function

```
void (sc->funs->eth_drv->tx_done)(
                struct eth_drv_sc *sc,
                unsigned long key, int status)
```

This function is called when a packet completes transmission on the interface. The *key* value must be one of the keys provided to *HRDWR*_send() above. The value *status* should be non-zero (details currently undefined) to indicate that an error occurred during the transmission, and zero if all was well.

It should be called from the deliver function (see the section called “Deliver function”) or poll function (see the section called “Poll function”).

Callback Receive function

```
void (sc->funs->eth_drv->recv)(struct eth_drv_sc *sc, int len)
```

This function is called to indicate that a packet of length *len* has arrived at the interface. The callback *HRDWR*_recv() function described above will be used to actually unload the data from the interface into buffers used by the device independent layers.

It should be called from the deliver function (see the section called “Deliver function”) or poll function (see the section called “Poll function”).

Calling graph for Transmission and Reception

It may be worth clarifying further the flow of control in the transmit and receive cases, where the hardware driver does use interrupts and so DSRs to tell the “foreground” when something asynchronous has occurred.

Transmission

1. Some foreground task such as the application, SNMP “daemon”, DHCP management thread or whatever, calls into network stack to send a packet, or the stack decides to send a packet in response to incoming traffic such as a “ping” or ARP request.

2. The driver calls the *HRDWR*_can_send() function in the hardware driver.

3. *HRDWR*_can_send() returns the number of available "slots" in which it can store a pending transmit packet. If it cannot send at this time, the packet is queued outside the hardware driver for later; in this case, the hardware is already busy transmitting, so expect an interrupt as described below for completion of the packet currently outgoing.

4. If it can send right now, *HRDWR*_send() is called. *HRDWR*_send() copies the data into special hardware buffers, or instructs the hardware to “send that.” It also remembers the key that is associated with this tx request.

5. These calls return … time passes …

6. Asynchronously, the hardware makes an interrupt to say “transmit is done.” The ISR quietens the interrupt source in the hardware and requests that the associated DSR be run.

7. The DSR calls (or *is*) the eth_drv_dsr() function in the generic driver.

8. eth_drv_dsr() in the generic driver awakens the “Network Delivery Thread” which calls the deliver function *HRDWR*_deliver() in the driver.

9. The deliver function realizes that a transmit request has completed, and calls the callback tx-done function `(sc->funs->eth_drv->tx_done)()` with the same key that it remembered for this tx.

10. The callback tx-done function uses the key to find the resources associated with this transmit request; thus the stack knows that the transmit has completed and its resources can be freed.

11. The callback tx-done function also enquires whether *`HRDWR`*_can_send() now says “yes, we can send” and if so, dequeues a further transmit request which may have been queued as described above. If so, then *`HRDWR`*_send() copies the data into the hardware buffers, or instructs the hardware to "send that" and remembers the new key, as above. These calls then all return to the “Network Delivery Thread” which then sleeps, awaiting the next asynchronous event.

12. All done …

Receive

1. Asynchronously, the hardware makes an interrupt to say “there is ready data in a receive buffer.” The ISR quietens the interrupt source in the hardware and requests that the associated DSR be run.

2. The DSR calls (or *is*) the `eth_drv_dsr()` function in the generic driver.

3. `eth_drv_dsr()` in the generic driver awakens the “Network Delivery Thread” which calls the deliver function *`HRDWR`*_deliver() in the driver.

4. The deliver function realizes that there is data ready and calls the callback receive function `(sc->funs->eth_drv->recv)()` to tell it how many bytes to prepare for.

5. The callback receive function allocates memory within the stack (eg. MBUFs in BSD/Unix style stacks) and prepares a set of scatter-gather buffers that can accommodate the packet.

6. It then calls back into the hardware driver routine *`HRDWR`*_recv(). *`HRDWR`*_recv() must copy the data from the hardware's buffers into the scatter-gather buffers provided, and return.

7. The network stack now has the data in-hand, and does with it what it will. This might include recursive calls to transmit a response packet. When this all is done, these calls return, and the “Network Delivery Thread” sleeps once more, awaiting the next asynchronous event.

Chapter 168. lwIP Direct Ethernet Device Driver

Introduction

This chapter provides a simple description of the basic requirements for a low-level, hardware specific, lwIP-direct ethernet driver.

Using a lwIP-direct driver provides benefits in performance and smaller code- and memory-footprints. It also allows for the potential for zero-copy UDP support and reduced (single) copy TCP support depending on the hardware available. The main disadvantage over the standard ethernet driver world is the lack of RedBoot network debugging support.

The high-level driver implemented by this package (which is only code, without state of its own) is used to provide a common interface for lwIP to either a lwIP-specific direct driver (as described in this chapter), or via a wrapper interface to a standard generic ethernet driver (covered by the section called “Generic Ethernet API”).

Unlike the generic ethernet (standard) device driver support the lwIP device driver interface uses a fixed namespace between the lwIP and driver layers. Normally only a single driver instance exists for a lwIP configured world, so the use of a fixed namespace is, in reality, not an issue since lwIP is designed for lightweight, low resource, deeply-embedded systems. If a target platform really does provide more than one distinct ethernet hardware implementation, requiring completely different hardware drivers, then a wrapper layer conforming to the “direct” driver interface is provided when the option `CYGFUN_IO_ETH_DRIVERS_LWIP_DRIVER_DIRECT_MULTI` is configured. This implements a per-driver descriptor interface between the individual low-level hardware interfaces for the platform and this common Ethernet I/O package.

Normally a direct driver implementation will also provide a driver specific header file which is referenced from the lwIP CDL option `CYGBLD_LWIP_HW_DRIVER_OVERRIDE_HEADER`. The CDL covering the direct driver package should explicitly set the value to the required header file name. Similarly when support is configured for multiple direct drivers, the CDL option `CYGBLD_LWIP_VARIANT_OVERRIDE_HEADER` can be used to reference a header providing any needed platform/variant/driver specific features.

These header files can be used to provide access to prototypes and manifests needed to support specific lwIP features as required. For example, if the hardware driver uses DMA, and requires timely support for re-using PBUFs once lwIP has finished processing them, then the `ECOS_LWIP_PBUF_POOL_FREE_HOOK` manifest can be defined to reference a callback function (See *`DRV_HDWR`*_pbuf_pool_free_hook()).

The following sections give an overview of the small set of functions that the driver needs to provide to be usable by this package. When the multiple direct driver support is being used then these named functions are provided by this common `CYGPKG_IO_ETH_DRIVERS` I/O Ethernet package, with a per-driver descriptor structure used to reference the specific driver implementations (See the section called “Multiple direct drivers”).

API reference

The following function definitions document the namespace used by the eCos lwIP TCP/IP stack to interact with hardware drivers.

Name

cyg_lwip_eth_ecos_init() — Initialize the hardware driver

Synopsis

#include <cyg/io/eth/eth_drv.h>

```
void cyg_lwip_eth_ecos_init();
```

Description

This function should perform the necessary hardware initialization, along with attaching any required eCos ISR and DSR support. As part of the initialization the upper-layer (generic) ethernet driver routine `cyg_lwip_eth_drv_init_netif()` should be called.

The direct driver DSR should call the lwIP routine `cyg_lwip_eth_dsr()` on completion of its DSR handling to ensure the lwIP delivery mechanism is notified.

Name

cyg_lwip_eth_low_level_output() — Transmit a packet

Synopsis

```
#include <cyg/io/eth/eth_drv.h>
```

```
err_t cyg_lwip_eth_low_level_output(netif, p);
```

Description

This function is called by higher layers to perform the actual transmission of the data packet referenced by *p*. The passed pointer may be a chain of linked struct pbuf descriptors containing the data of the single packet, and is not a chain of packets.

Return value

This function returns a standard error code, as defined in `<lwip/err.h>`, with ERR_OK being returned on success.

Name

cyg_lwip_eth_run_deliveries() — Packet buffer house-keeping

Synopsis

#include <cyg/io/eth/eth_drv.h>

```
void cyg_lwip_eth_run_deliveries();
```

Description

This function is called from the lwIP thread context if the DSR has indicated that an ethernet delivery event needs to happen. It can be used by the device driver to re-fill transmission buffers, or to pass pending receptions to the higher layers as required by the hardware state. For example, if a received packet is available it should pass it into the stack via the common driver routine `cyg_lwip_eth_drv_ecosif_input()`.

Name

cyg_lwip_eth_ioctl() — Control interface

Synopsis

#include <cyg/io/eth/eth_drv.h>

```
int cyg_lwip_eth_ioctl(netif, key, data, data_length);
```

Description

This function is used to perform low-level “control” operations on the specified lwIP network interface. It provides an interface to the hardware where the function cannot be performed totally by the platform-independent layers.

The *`key`* parameter selects the operation, and the *`data`* and *`data_length`* parameters describe, as required, some data for the specified operation.

Available Operations:

ETH_DRV_SET_MAC_ADDRESS

This operation sets the ethernet station address (ESA or MAC) for the specified network interface. Normally this address would be kept in non-volatile memory and is unique in the world. This function must at minimum set the hardware interface to use the supplied address, but (if required) it may also update the non-volatile memory as appropriate.

ETH_DRV_SET_MC_LIST

Configure the driver with the given list of multicast filters so that only received packets with a matching filter are accepted and passed onto the lwIP stack.

The common ethernet support currently defines a fixed size vector for holding the “list” of multicast filter addresses:

```
struct eth_drv_mc_list {
    int len;
    unsigned char addrs[CYGNUM_IO_ETH_DRIVERS_FILTER_LIST_SIZE][ETHER_ADDR_LEN];
};
```

For the lwIP ethernet driver support the passed list may have unused entries (marked by the first byte having the multicast flag bit 0 clear). This is an optimisation, and still allows for users to pass a front-filled *`len`* count of used entries, or for the driver to avoid having to copy-down data when list entries are removed by marking individual passed list entries as unused.

For simple, lightweight, drivers it is common for the list support to simply perform the equivalent of the `ETH_DRV_SET_MC_ALL` option, where the lwIP stack filters unwanted multicast packets instead of the driver having pre-filtered based on the specified list.

ETH_DRV_SET_MC_ALL

Configure the driver to accept all multicast packets.

ETH_DRV_GET_IF_STATS, ETH_DRV_GET_IF_STATS_UD

These options acquire a set of statistical counters from the interface, and write the information into the memory referenced by *`data`*. The calls to these options are typically made by SNMP agents. The “UD” variant explicitly instructs the driver to acquire up-to-date values. This is a separate option because doing so may take some time, depending on the hardware.

The definition of the data structure can be found in the header file `cyg/io/eth/eth_drv_stats.h`.

ETH_DRV_SET_DA_LIST
ETH_DRV_SET_SA_LIST
ETH_DRV_SET_FILTER_OPTIONS
ETH_DRV_SET_FILTER_L3L4
ETH_DRV_CLR_FILTER_L3L4
ETH_DRV_SET_VLANTAG
ETH_DRV_GET_VLANTAG
ETH_DRV_SET_OPMODE_RX

Refer to the section called “Control function” for the documentation for these filtering control options.

Return value

Successful completion of the operation is indicated by a result of 0 being returned. If a specific *key* operation is not supported by the driver, or there is an error processing the requested operation then a result of 1 is returned to indicate failure.

Name

*DRV_HDWR*_pbuf_pool_free_hook() — PBUF free hook callback

Synopsis

#include <cyg/io/eth/eth_drv.h>

u8_t ***DRV_HDWR*_pbuf_pool_free_hook**(*p*);

Description

If the driver specific header file defines ECOS_LWIP_PBUF_POOL_FREE_HOOK then the driver should implement a hardware-specific function matching this defined prototype.

The function is called when lwIP is releasing a packet buffer, allowing the low-level device driver access to the (now unused) packet buffer. This avoids the overhead of lwIP having to complete the free operation, and for the driver having to make a subsequent allocation call.

If the driver does want to make use of the packet buffer descriptor (for example, to replace a DMA buffer slot) then it must call the lwIP routine pbuf_pool_reinit() to ensure a valid descriptor state prior to reuse.

Return value

If the driver does not make use of the referenced struct pbuf then it returns a value of 1 to indicate that the packet buffer has not been reused.

If the driver does claim the packet buffer then it should return 0 to indicate that the packet buffer descriptor has been reused.

Multiple direct drivers

When support for multiple direct drivers is configured then a driver instance is contained within a cyg_lwip_eth_t structure:

```
typedef struct cyg_lwip_eth {
  const char *name; // NUL terminated ASCII human-readable name
  void (*init)(struct cyg_lwip_eth *drvdesc);
  void (*run_deliveries)(void *instance);
  err_t (*ll_output)(struct netif *netif,struct pbuf *p);
  int (*pbuf_free_hook)(void *instance,struct pbuf *p);
  void (*phy_event)(struct netif *netif);
  int (*ioctl)(struct netif *netif,unsigned long key,void *data,int data_length);
  void *instance;
  cyg_uint32 flags;
} CYG_HAL_TABLE_TYPE cyg_lwip_eth_t;
```

This CYGPKG_IO_ETH_DRIVERS package will implement the wrapper namespace to support lwIP, calling the relevant individual device driver registered functions as required.

You create an instance of cyg_lwip_eth_t using the CYG_LWIP_DRIVER macro, which sets up the structure. Using this macro ensures that if the internal design changes then existing source will fail to compile until updated to reflect the changed functionality. This is better than having definitions within the low-level drivers themselves, with the possibility of them building successfully but then failing at run-time.

The individual hardware drivers are initialised automatically via the wrapper provided cyg_lwip_eth_ecos_init() function, which iterates over the __LWIPDEVTAB__ vector containing the driver instance descriptors as required.

Note

When lwIP direct drivers are written to support `CYGFUN_IO_ETH_DRIVERS_LWIP_DRIVER_DIRECT_MULTI` configurations they *MUST* reference their cyg_lwip_eth_t descriptor via the `state` field of the struct netif describing the lwIP network interface. The `instance` field of the cyg_lwip_eth_t can be used to hold driver specific instance data.

The function pointers referenced from the cyg_lwip_eth_t descriptor closely match the raw namespace, with the exception that initialisation is passed the cyg_lwip_eth_t driver descriptor pointer, and the `run_deliveries` and `pbuf_free_hook` implementations are passed the private `instance` pointer. This ensures that the individual driver implementation can access the necessary state as would be the case for a single driver configuration.

Note

For the `pbuf_free_hook` support we should ideally pass the `pbuf` back to the original driver instance that allocated that specific `pbuf`. However, for the moment, the code just offers the `pbuf` to each configured driver in turn (the alternative would introduce complexity into the driver model for minimal gains).

This “do you want this `pbuf`” approach does not affect the behaviour, only the performance, of the driver when used in a multi-driver configuration. If the developer needs to ensure that a particular driver instance is “higher priority” than other lwIP Ethernet drivers for `pbuf` re-use then they should enforce a mechanism for ensuring the ordering of the `__LWIPDEVTAB__`device table.

lwIP MANUAL initialisation

Normally lwIP will default to `DHCP` for network interface address acquisition, but alternative methods can be configured (`AUTOIP`, `STATIC` or `MANUAL`). The relevant configuration specific interface initialisation code is actually performed in this common IO Ethernet package by the `cyg_lwip_eth_drv_init_netif()` function. When configured to use fixed `STATIC` addresses those are held in the eCos configuration file for the build. The `MANUAL` option, however, allows for the application code to manually supply address information and perform the interface initialisation.

When `MANUAL` address configuration is selected for an lwIP interface then an explicitly named function **must** be supplied by the application run-time, with the prototype:

```
char cyg_lwip_eth_init_manual(struct netif *netif, char inum, unsigned char *enaddr);
```

The *`netif`* parameter references the underlying lwIP network interface descriptor, with the parameter *`inum`* being the logical (indexed from 0) interface number. The *`enaddr`* references the IEEE MAC address for the interface.

It is expected that the application supplied routine will set the address configuration et al., before adding the interface, based on some per-device stored/calculated values.

It is expected that if manual application interface initialisation is being used that the developer **has** a reasonable understanding of lwIP and its internal requirements, and is au fait with the eCos network source base.

The following is a simple example implementation of the basic operations that need to be performed by the application to provide `MANUAL` interface support:

```
char cyg_lwip_eth_init_manual(struct netif *netif, char inum, unsigned char *enaddr)
{
    ip4_addr_t ipaddr;
    ip4_addr_t netmask;
    ip4_addr_t gw;

    application_code_to_fill_addresses_for_interface_number(inum, &ipaddr, &netmask, &gw);

    char ok = (NULL != netif_add((netif),
                                 &ipaddr,
                                 &netmask,
                                 &gw,
                                 (netif)->state,
                                 cyg_lwip_eth_netif_init,
                                 ethernet_input));

    if (ok) {
#if LWIP_CHECKSUM_CTRL_PER_NETIF // per-interface checksum offload control
        // Set following as desired for the application configuration, or the
        // target H/W driver feature support:
        (netif)->chksum_flags = NETIF_CHECKSUM_DISABLE_ALL;
#endif // LWIP_CHECKSUM_CTRL_PER_NETIF
        netif_set_up(netif);
    }

    return ok;
}
```

Chapter 169. CDC-EEM Target USB driver

Introduction

eCosPro-CDCEEM is eCosCentric's commercial name for the USB peripheral device CYGPKG_DEVS_ETH_USB_CDCEEM package. The package is not included as standard in eCosPro Developer's Kit releases, but is available as a separate add-on package.

The CYGPKG_DEVS_ETH_USB_CDCEEM package implements a USB peripheral device CDC-EEM transport driver. The current implementation makes use of the generic Ethernet driver package CYGPKG_DEVS_ETH_GENERIC_DIRECT to integrate with the lwIP TCP/IP stack.

The CDC-EEM peripheral driver is currently limited to use with the lwIP network stack, and is not available for the BSD network stacks. This is a limitation of the parent CYGPKG_DEVS_ETH_GENERIC_DIRECT package, and not explicitly a limitation of this CDC-EEM peripheral driver.

Normally the eCos lwIP network interface should be configured to use AutoIP, so that a link-local network address is assigned. This ensures that when connected to hosts that do not provide a DHCP daemon, or support for routing to manual or application set network addresses, an automatic connection is still configured.

API

There is no “user” API as such, since the cyg_eth_drv_generic_transport_cdceem structure is exported via the __ETH_TRANSPORT_TAB__ table constructed at build-time, and referenced from the generic Ethernet device driver. The CDC-EEM driver just provides a transport driver for the generic Ethernet world.

The exported CDC-EEM device features are controlled by the CDL for the package.

Configuration

This section shows how to include the CDC-EEM support into an eCos configuration, and how to configure it once installed.

Configuration Overview

The CDC-EEM driver is contained in a single eCos package CYGPKG_DEVS_ETH_USB_CDCEEM. However, it depends on the services of a collection of other packages for complete functionality. Currently the CDC-EEM implementation is tightly bound with the generic Ethernet driver package CYGPKG_DEVS_ETH_GENERIC_DIRECT.

Incorporating the CDC-EEM driver into your application is straightforward. The essential starting point is to incorporate the CDC-EEM eCos package (CYGPKG_DEVS_ETH_USB_CDCEEM) into your configuration.

This may be achieved directly using **ecosconfig add** on the command line, or the **Build->Packages…** menu item within the eCos Configuration Tool.

Configuring the CDC-EEM driver

Once added to the eCos configuration, the CDC-EEM package has a number of configuration options.

CYGPKG_DEVS_ETH_USB_CDCEEM_VID

The device VendorID. The VendorID number space is managed by the USB organisation, www.usb.org, and a unique ID must be formally obtained.

In conjunction with the `CYGPKG_DEVS_ETH_USB_CDCEEM_PID` value this is used to uniquely identify a specific peripheral product to the host O/S environment.

Note

The VID is normally expressed as a 16-bit hexadecimal number, but the eCos graphical configurarion tool will normally display the value as a decimal.

`CYGPKG_DEVS_ETH_USB_CDCEEM_PID`

The device ProductID. The ProductID number space is managed by the vendor. This ID is sometimes used to uniquely identify specific devices as regards the host device driver needed to communicate with the target device. It is the responsibility of the developer to manage this internal (company) number space.

Note

The PID is normally expressed as a 16-bit hexadecimal number, but the eCos graphical configurarion tool will normally display the value as a decimal.

`CYGPKG_DEVS_ETH_USB_CDCEEM_MANUFACTURER`

A human-readable device manufacturer identification string, that is returned as part of the device USB description. The string may be used by the host O/S in its description of the product presented to end-users.

`CYGPKG_DEVS_ETH_USB_CDCEEM_PRODUCT`

A human-readable product identification string, that is returned as part of the device USB description. Like the manufacturer string this may be used on the host when presenting a device to the user.

`CYGPKG_DEVS_ETH_USB_CDCEEM_SERIAL_FIXED`

Depending on the product requirements the serial number returned as part of the USB descriptor can either be supplied at run-time by the application HAL or defined by the CDL and fixed for a binary build.

The former approach relies on the HAL having a method of obtaining a unique identifier from the hardware from which to construct a unique serial number string. This is normally the preferred approach to providing per-device unique identification, and is used when this option is *disabled*. When this option is *enabled* the build uses the string defined by this option as the value returned in the device USB description. This latter approach is less flexible if different physical devices need a unique ID since the CDL will need to be modified and a unique binary constructed for each specific device. If the devices do not need to present a unique identity then the same serial number can be configured into the binary build with the same value being used across *all* target devices.

`CYGPKG_DEVS_ETH_USB_CDCEEM_POWERED`

This option defines how the device declares its power state to the host, and should be configured to match the hardware implementation supporting the CDC-EEM target driver. When configured as `Bus` powered then a further configuration option is made available:

`CYGPKG_DEVS_ETH_USB_CDCEEM_MAXPOWER`

When bus-powered this option specifies the maximum power consumption of the device.

`CYGIMP_CDCEEM_CRC_RX`

When enabled verify EEM packet CRC on reception. If disabled then the CRC is ignored and all received packets are passed to the parent Ethernet driver.

CYGIMP_CDCEEM_CRC_TX

If enabled then a CRC is calculated for every EEM packet transmitted. When disabled the special 0xDEADBEEF CRC sentinel is used. NOTE: It is recommended to leave this option disabled currently, since problems have been seen where Linux host drivers will incorrectly calculate the CRC and drop packets."

CYGDBG_CDCEEM_DIAGNOSTICS

When enabled this option allows diagnostic output to be generated for different subsystems within the CDC-EEM driver, and a set of further options are made available for configuration. This information is primarily for internal driver development, and is not normally needed when debugging applications using the USB CDC-EEM network driver. The debug output is sent to the diagnostic console channel as configured for the application.

Debug and Test

Debugging

Asserts

If the target platform resources allow the first step in debugging should be to enable ASSERTs. The inclusion of assert checking will increase the code footprint and lower the performance, but do allow the code to catch internal errors from unexpected data values. e.g. when the application/client is not able to guarantee the validity of data passed into the CDC-EEM layer.

The CDC-EEM transport driver asserts are controlled via the standard eCos Infrastructure CYGPKG_INFRA package CYGDBG_USE_ASSERTS option. If enabled then run-time assertion checks are performed by the CDC-EEM driver.

If assertions are enabled, and a debugger is being used it is normally worth-while setting a breakpoint on the cyg_assert_fail symbol so that the debugger will stop prior to entering the default busy-loop processing.

Diagnostic Output

In conjuction with the CYGDBG_CDCEEM_DIAGNOSTICS CDL configuration setting, the source-file src/cdceem.c implements the CDC-EEM specific diagnostics control.

When CYGDBG_CDCEEM_DIAGNOSTICS is enabled a set of individually selectable sub-systems are available to control the diagnostic output generated.

However, when developing or debugging the CDC-EEM driver implementation it may be simpler (with less build side-effects) to control the debugging output via uncommenting the necessary manifests at the head of the src/cdceem.c source file than re-configuring the complete eCos configuration via the CDL. That way only the CDC-EEM package will be re-built.

Note

Some diagnostic output if enabled may adversely affect the operation of the CDC-EEM driver as seen by 3rd-party code. For example, “slow” serial diagnostic output of the packet parsing and response generation could mean that a significant amount of time passes, such that the CDC-EEM driver no longer adheres to the timings required by the USB host driver.

Chapter 170. RNDIS Target USB driver

Introduction

eCosPro-RNDIS is eCosCentric's commercial name for the USB peripheral device CYGPKG_DEVS_ETH_USB_RNDIS package. The package is not included as standard in eCosPro Developer's Kit releases, but is available as a separate add-on package.

The CYGPKG_DEVS_ETH_USB_RNDIS package implements a USB peripheral device Remote NDIS transport driver. The current implementation makes use of the generic Ethernet driver package CYGPKG_DEVS_ETH_GENERIC_DIRECT to integrate with the lwIP TCP/IP stack.

This driver has been tested against a range of host operating systems, including:

- Linux

  Most modern Linux distributions will, by default, have support for RNDIS USB devices. For example, Ubuntu 12, CentOS 6, etc. The target driver has been explicitly tested against 2.6 and 3.8 kernel based hosts.

- Mac OS X

  The 3rd-party, open-source, HoRNDIS driver needs to be installed on the host. The eCos RNDIS driver has been explicitly tested against Mac OS X versions 10.8.5 and 10.9, though earlier versions of Mac OS X should present no problems assuming available HoRNDIS support.

- Windows

  Windows XP (SP2), 7 (SP1), 8 and 8.1 have been testing using the standard Windows RNDIS host driver support.

The RNDIS peripheral driver is currently limited to use with the lwIP network stack, and is not available for the BSD network stacks. This is a limitation of the parent CYGPKG_DEVS_ETH_GENERIC_DIRECT package, and not explicitly a limitation of this RNDIS peripheral driver.

Normally the eCos lwIP network interface should be configured to use AutoIP, so that a link-local network address is assigned. This ensures that when connected to hosts that do not provide a DHCP daemon, or support for routing to manual or application set network addresses, an automatic connection is still configured.

One side-effect of the RNDIS networking model (as opposed to CDC-EEM for example) is that two network interfaces exist; the host-end network interface created by the host O/S, and the peripheral lwIP interface providing the target application networking. This means that each device configured to use the RNDIS USB target driver needs to provide two IEEE MAC addresses. The platform HAL support supplying the MAC address to this driver, in conjunction with the developer/manufacturer build world, must be aware of the requirements for managing the “unique identity” 24-bit MAC space in conjunction with the 24-bit IEEE OUI space specific to the device manufacturer.

API

There is no “user” API as such, since the cyg_eth_drv_generic_transport_rndis structure is exported via the __ETH_TRANSPORT_TAB__ table constructed at build-time, and referenced from the generic Ethernet device driver. The RNDIS driver just provides a transport driver for the generic Ethernet world.

The exported RNDIS device features are controlled by the CDL for the package.

Configuration

This section shows how to include the RNDIS support into an eCos configuration, and how to configure it once installed.

Configuration Overview

The RNDIS driver is contained in a single eCos package `CYGPKG_DEVS_ETH_USB_RNDIS`. However, it depends on the services of a collection of other packages for complete functionality. Currently the RNDIS implementation is tightly bound with the generic Ethernet driver package `CYGPKG_DEVS_ETH_GENERIC_DIRECT`.

Incorporating the RNDIS driver into your application is straightforward. The essential starting point is to incorporate the RNDIS eCos package (`CYGPKG_DEVS_ETH_USB_RNDIS`) into your configuration.

This may be achieved directly using **ecosconfig add** on the command line, or the **Build->Packages…** menu item within the eCos Configuration Tool.

Configuring the RNDIS driver

Once added to the eCos configuration, the RNDIS package has a number of configuration options.

`CYGPKG_DEVS_ETH_USB_RNDIS_VID`

The device VendorID. The VendorID number space is managed by the USB organisation, www.usb.org, and a unique ID must be formally obtained.

In conjunction with the `CYGPKG_DEVS_ETH_USB_RNDIS_PID` value this is used to uniquely identify a specific peripheral product to the host O/S environment.

Note

The VID is normally expressed as a 16-bit hexadecimal number, but the eCos graphical configurarion tool will normally display the value as a decimal.

`CYGPKG_DEVS_ETH_USB_RNDIS_PID`

The device ProductID. The ProductID number space is managed by the vendor. This ID is sometimes used to uniquely identify specific devices as regards the host device driver needed to communicate with the target device. It is the responsibility of the developer to manage this internal (company) number space.

Note

The PID is normally expressed as a 16-bit hexadecimal number, but the eCos graphical configurarion tool will normally display the value as a decimal.

`CYGPKG_DEVS_ETH_USB_RNDIS_MANUFACTURER`

A human-readable device manufacturer identification string, that is returned as part of the device USB description. The string may be used by the host O/S in its description of the product presented to end-users.

`CYGPKG_DEVS_ETH_USB_RNDIS_PRODUCT`

A human-readable product identification string, that is returned as part of the device USB description. Like the manufacturer string this may be used on the host when presenting a device to the user.

`CYGPKG_DEVS_ETH_USB_RNDIS_SERIAL_FIXED`

Depending on the product requirements the serial number returned as part of the USB descriptor can either be supplied at run-time by the application HAL or defined by the CDL and fixed for a binary build.

The former approach relies on the HAL having a method of obtaining a unique identifier from the hardware from which to construct a unique serial number string. This is normally the preferred approach to providing per-device unique identification, and is used when this option is *disabled*. When this option is *enabled* the build uses the string defined by this option as the value returned in the device USB description. This latter approach is less flexible if different physical devices need a unique ID since the CDL will need to be modified and a unique binary constructed for each specific device. If the devices do not need to present a unique identity then the same serial number can be configured into the binary build with the same value being used across *all* target devices.

CYGPKG_DEVS_ETH_USB_RNDIS_POWERED

This option defines how the device declares its power state to the host, and should be configured to match the hardware implementation supporting the RNDIS target driver. When configured as Bus powered then a further configuration option is made available:

CYGPKG_DEVS_ETH_USB_RNDIS_MAXPOWER

When bus-powered this option specifies the maximum power consumption of the device.

CYGDBG_RNDIS_DIAGNOSTICS

When enabled this option allows diagnostic output to be generated for different subsystems within the RNDIS driver, and a set of further options are made available for configuration. This information is primarily for internal driver development, and is not normally needed when debugging applications using the USB RNDIS network driver. The debug output is sent to the diagnostic console channel as configured for the application.

Debug and Test

Debugging

Asserts

If the target platform resources allow the first step in debugging should be to enable ASSERTs. The inclusion of assert checking will increase the code footprint and lower the performance, but do allow the code to catch internal errors from unexpected data values. e.g. when the application/client is not able to guarantee the validity of data passed into the RNDIS layer.

The RNDIS transport driver asserts are controlled via the standard eCos Infrastructure CYGPKG_INFRA package CYGDBG_USE_ASSERTS option. If enabled then run-time assertion checks are performed by the RNDIS driver.

If assertions are enabled, and a debugger is being used it is normally worth-while setting a breakpoint on the cyg_assert_fail symbol so that the debugger will stop prior to entering the default busy-loop processing.

Diagnostic Output

In conjuction with the CYGDBG_RNDIS_DIAGNOSTICS CDL configuration setting, the source-file src/rndis.c implements the RNDIS specific diagnostics control.

When CYGDBG_RNDIS_DIAGNOSTICS is enabled a set of individually selectable sub-systems are available to control the diagnostic output generated.

However, when developing or debugging the RNDIS driver implementation it may be simpler (with less build side-effects) to control the debugging output via uncommenting the necessary manifests at the head of the src/rndis.c source file than re-configuring the complete eCos configuration via the CDL. That way only the RNDIS package will be re-built.

Note

Some diagnostic output if enabled may adversely affect the operation of the RNDIS driver as seen by 3rd-party code. For example, “slow” serial diagnostic output of the packet parsing and response generation could mean that a significant amount of time passes, such that the RNDIS driver no longer adheres to the timings required by the USB host driver.

Chapter 171. Ethernet PHY Device Support

Ethernet PHY Device API

Modern ethernet subsystems are often separated into two pieces, the media access controller (sometimes known as a MAC) and the physical device or line interface (often referred to as a PHY). In this case, the MAC handles generating and parsing physical frames and the PHY handles how this data is actually moved to/from the wire. The MAC and PHY communicate via a special protocol, known as MII. This MII protocol can handle control over the PHY which allows for selection of such transmission criteria as line speed, duplex mode, etc.

In most cases, ethernet drivers only need to bother with the PHY during system initialization. Since the details of the PHY are separate from the MAC, there are different drivers for each. The drivers for the PHY are described by a set of exported functions which are commonly used by the MAC. The primary use of these functions currently is to initialize the PHY and determine the status of the line connection.

The connection between the MAC and the PHY differs from MAC to MAC, so the actual routines to manipulate this data channel are a property of the MAC instance. Furthermore, there are many PHY devices each with their own internal operations. A complete MAC/PHY driver setup will be comprised of the MAC MII access functions and the PHY internal driver.

A driver instance is contained within a eth_phy_access_t:

```
#define PHY_BIT_LEVEL_ACCESS_TYPE 0
#define PHY_REG_LEVEL_ACCESS_TYPE 1

typedef struct {
    int ops_type;  // 0 => bit level, 1 => register level
    bool init_done;
    void (*init)(void);
    void (*reset)(void);
    union {
        struct {
            void (*set_data)(int);
            int  (*get_data)(void);
            void (*set_clock)(int);
            void (*set_dir)(int);
        } bit_level_ops;
        struct {
            void (*put_reg)(int reg, int unit, unsigned short data);
            bool (*get_reg)(int reg, int unit, unsigned short *data);
        } reg_level_ops;
    } ops;
    int phy_addr;
    struct _eth_phy_dev_entry *dev;  // Chip access functions
} eth_phy_access_t;

struct _eth_phy_dev_entry {
    char          *name;
    cyg_uint32    id;
    bool         (*stat)(eth_phy_access_t *f, int *stat);
    unsigned int (*event)(eth_phy_access_t *f, unsigned int bitmask); // Configuration option
    cyg_uint32    idmask; // Masked with id to determine if there's a match
};
```

The `dev` element points to the PHY specific support functions. Currently, the only function which must be defined is `stat()`.

The MAC-MII-PHY interface is a narrow connection, with commands and status moving between the MAC and PHY using a bit-serial protocol. Some MAC devices contain the intelligence to run this protocol, exposing a mechanism to access PHY registers one at a time. Other MAC devices may only provide access to the MII data lines (or even still, this may be considered completely separate from the MAC). In these cases, the PHY support layer must handle the serial protocol. The choice between the access methods is in the `ops_type` field. If it has the value `PHY_BIT_LEVEL_ACCESS_TYPE`, then the PHY device layer will run the protocol, using the access functions `set_data()`, `get_data()`, `set_clock()`, `set_dir()` are used to control the MII signals and run the protocol. If `ops_type` has the value `PHY_REG_LEVEL_ACCESS_TYPE`, then the routines `put_reg()`, and `get_reg()` are used to access the PHY registers.

Two additional functions may be defined. These are `init()`, and `reset()`. The purpose of these functions is for gross-level management of the MII interface. The `init()` function will be called once, at system initialization time. It should do whatever operations are necessary to prepare the MII channel. In the case of `PHY_BIT_LEVEL_ACCESS_TYPE` devices, `init()` should prepare the signals for use, i.e. set up the appropriate parallel port registers, etc. The `reset()` function may be called by a driver to cause the PHY device to be reset to a known state. Not all drivers will require this and this function may not even be possible, so it's use and behavior is somewhat target specific.

Currently, the only function required of device specific drivers is `stat()`. This routine should query appropriate registers in the PHY and return a status bitmap indicating the state of the physical connection. In the case where the PHY can auto-negotiate a line speed and condition, this information may be useful to the MAC to indicate what speed it should provide data, etc. The status bitmask contains these bits:

```
#define ETH_PHY_STAT_LINK  0x0001   // Link up/down
#define ETH_PHY_STAT_100MB 0x0002   // Connection is 100Mb/10Mb
#define ETH_PHY_STAT_FDX   0x0004   // Connection is full/half duplex
```

Note: the usage here is that if the bit is set, then the condition exists. For example, if the `ETH_PHY_STAT_LINK` is set, then a physical link has been established.

For platforms capable of supporting asynchronous PHY event notification the `event()` function can be implemented. The CDL for the specific PHY and Ethernet driver combination defines whether the `CYGINT_DEVS_ETH_PHY_PLF_IF_EVENTS` controlled feature is actually included. The `event()` function, for simplicity, provides both the event control and status support depending on the *`bitmask`* setting passed to the function. This function provides the following functionality:

- configure the PHY for the events we are interested in receiving asynchronous notification for
- ascertain which events have occurred when an event is triggered
- clear any pending event (interrupt) status on the PHY
- ascertain the current status of the PHY

All of the above functionality is rolled into the single function to avoid the need for a separate `stat()` call to be made to the PHY when processing a PHY interrupt at the Ethernet driver layer. Also the `event()` function should never block, unlike the `stat()` implementation which may block depending on the PHY driver requirements.

In addition to the status bitmask bits defined for `stat()` (as listed above) extra status and control bits are defined. The status bit:

```
#define ETH_PHY_STAT_ANC  0x0008   // Auto-Negotiation Completed
```

is used to reflect whether Auto-Negotiation has completed. When requesting enable/disable control, or detecting a change in state indicated in the function result, the extra bits:

```
#define ETH_PHY_EVENT_LINK    (1 << 16) // Link up/down change
#define ETH_PHY_EVENT_SPEED   (1 << 17) // Speed (e.g. 10-/100-Mb/s) change
#define ETH_PHY_EVENT_DUPLEX  (1 << 18) // Duplex (half/full) change
#define ETH_PHY_EVENT_AUTONEG (1 << 19) // Auto-Negotiation completed
```

are available. When passing the *`bitmask`* to the function the bit:

```
#define ETH_PHY_EVENT_UPDATE  (1 << 31) // Update enabled events
```

is used to control both the enabling and disabling of specific PHY events. If `ETH_PHY_EVENT_UPDATE` if set then the `ETH_PHY_EVENT_` bit setting is used to control the enable (bit is set) or disable (bit is clear) state of the corresponding PHY event. If this `ETH_PHY_EVENT_UPDATE` bit is not set then the PHY event configuration is not changed, allowing the function call to be used purely for the clearing of pending events and ascertaing the event status and current PHY state. The returned result bitmask will have the bit:

```
#define ETH_PHY_EVENT_STATUS  (1 << 30) // Valid EVENT status flags
```

set if the event status information returned is valid. This is used to distinguish from the error value 0, used when the underlying PHY operations are either not available (PHY event support not actually included) or an error has occurred.

Note

The default starting state for PHY drivers is that all PHY events should be disabled. The Ethernet driver then requires an explicit call to enable PHY event support. For example to enable the LINK up/down event, and check that the PHY actually supports the functionality, a driver could make the call:

```
    if (_eth_phy_event(eth->phy, (ETH_PHY_EVENT_LINK | ETH_PHY_EVENT_UPDATE)) & ETH_PHY_EVENT_STATUS) {
      // create, attach and enable platform specific PHY interrupt handler
    }
```

Chapter 172. Synopsys DesignWare Ethernet GMAC Driver

Name

CYGPKG_DEVS_ETH_DWC_GMAC — eCos Support for Synopsys DesignWare Ethernet GMAC Devices

Description

The CYGPKG_DEVS_ETH_DWC_GMAC package only implements the standard eCos driver interface. When used with the lwIP TCP/IP network stack it provides implementations of the io/eth extended filtering options, and also provides support for automatically throttling RX frame processing to limit the system overhead when used on a saturated network. See the section called “Control function”.

Configuration Options

This Ethernet package should be loaded automatically when selecting a target containing a DWC GMAC controller, and it should never be necessary to load it explicitly. If the application does not actually require Ethernet functionality then the package is inactive and the final executable will not suffer any overheads from unused functionality. This is determined by the presence of the generic Ethernet I/O package CYGPKG_IO_ETH_DRIVERS. Typically the choice of eCos template causes the right thing to happen. For example the default template does not include any TCP/IP stack so CYGPKG_IO_ETH_DRIVERS is not included, but the net, redboot and lwip_eth templates do include a TCP/IP stack so will specify that package and hence enable the Ethernet driver.

All eCos network devices need a unique name. By default the first Ethernet device is assigned the name eth0. The platform specific package providing the DWC GMAC network device descriptor also normally provides the interface name in its struct eth_drv_sc structure instance.

The hardware requires that incoming Ethernet frames are received into one of a small number of buffers, arranged in a ring. Once a frame has been received and its size is known the driver will pass it up to higher-level code for further processing. The number of these buffers is provided in the platform specific package providing the dwc_gmac_priv structure instance.

In the standard Ethernet driver, each receive buffer requires 1528 bytes of memory. The package header file if_dwc_gmac.h defines the manifest DWC_GMAC_RX_BUFF_SIZE which is used as the size of the receive descriptor ring buffers. A smaller number of buffers increases the probability that incoming Ethernet frames have to be discarded. TCP/IP stacks are designed to cope with the occasional lost packet, but if too many frames are discarded then this will greatly affect performance. A key issue here is that passing the incoming frames up to higher-level code happens at thread level and hence the system behaviour is defined in large part by the priority of various threads running in the TCP/IP stack. If application code has high-priority threads that take up much of the available CPU time and the TCP/IP stack gets little chance to run then there will be little opportunity to pass received frames up the stack. Similarly the priority of the TCP/IP network stack threads may affect the CPU bandwidth available for other lower-priority application threads in a saturated network. Balancing out the various thread priorities and the number of receive buffers is the responsibility of the application developer.

By default the Ethernet driver will raise interrupts using a low priority level. The exact value will depend on the processor being used, and is held in the vector_pri field supplied by the platform specific package dwc_gmac_common structure definition. The driver does very little at interrupt level, instead the real work is done via threads inside the TCP/IP stack. Hence the interrupt priority has little or no effect on the system's behaviour. The RX interrupts are disabled whilst RX processing by the thread level TCP/IP stack is pending; and only re-enabled once the thread level code has processed the RX ring buffer.

RX throttling

CYGIMP_DEVS_ETH_DWC_GMAC_NET_RX_AUTO

By default the driver uses interrupt driven RX frame handling, but allows for manual control of whether interrupt or polled RX operation is used (via the ETH_DRV_OPTIONS interface). If CYGIMP_DEVS_ETH_DWC_GMAC_NET_RX_AUTO is enabled then the driver will also support automatic switching between modes. Enabling this feature does not affect the run-time operation of the driver by itself. It is the responsibility of the application to manually select ETH_DRV_OPTION_RX_AUTO mode via the ETH_DRV_OPTIONS control API if required.

```
// Simplistic example of setting driver AUTO-throttle option

struct netif *p_nif = netif_default;
struct eth_drv_sc *p_sc = (struct eth_drv_sc *)(netif->state);

if (p_sc->funs->control) {
    cyg_uint32 mode = ETH_DRV_OPTION_RX_AUTO;
    struct eth_drv_options drvopt;

    drvopt.u.mand = ~ETH_DRV_OPTION_RX_MODE_MASK;
    drvopt.eor = mode;

    int res = (p_sc->funs->control)(p_sc,ETH_DRV_OPTIONS,&drvopt,sizeof(drvopt));
    // Check mode is configured as requested:
    if (0 == res) {
        if (mode == (drvopt.u.val & ETH_DRV_OPTION_RX_MODE_MASK)) {
            success = true;
        } else {
            res = -1;
        }
    }
}
```

When AUTO-throttling is enabled other configuration options are made available to control the behaviour of the AUTO-throttle support.

CYGNUM_DEVS_ETH_DWC_GMAC_RUNAVG_SAMPLES

This option configures the number of samples used for the running averages. A smaller value will be coarser, but result in quicker transitions on large deltas. A greater number of samples value will result in a smoother transition over a longer period of time.

CYGSEM_DEVS_ETH_DWC_GMAC_NET_RX_AUTO_INTAVG

This option enables the AUTO support tracking of the RX load when in INT (interrupt) state. Since this has an impact on the performance of the RX path the feature can be disabled by deselecting this option. When this option is disabled the INT-to-POLL transition will only occur on network buffer exhaustion. When enabled the code will also track the active RX load when in AUTO-INT mode and switch to polled mode when the CYGNUM_DEVS_ETH_DWC_GMAC_WM_INT2POLL_PPMS configured watermark is exceeded.

CYGNUM_DEVS_ETH_DWC_GMAC_WM_INT2POLL_PPMS

When the AUTO-throttle feature is enabled this option specifies the load threshold (packets-per-millisecond) over which we will switch from INT mode into POLL mode. This watermark is only used when AUTO mode is enabled, and AUTO-INT state is active, and triggers a switch to POLL mode When the load average rises above this threshold. It should be tuned appropriately.

CYGNUM_DEVS_ETH_DWC_GMAC_WM_POLL2INT_PPMS

When the AUTO-throttle feature is enabled this option specifies the load threshold (packets-per-millisecond) under which we switch from POLL mode back to INT mode when the load average drops below this threshold. It should be tuned appropriately.

CYGNUM_DEVS_ETH_DWC_GMAC_NET_RX_POLL_PERIOD

When the driver is configured for RX polled operation, this value is the number of milliseconds used between polled receiver calls.

Note

Care should be taken when setting a short poll period; since on networks where the driver will accepts large numbers of RX packets, and a large amount of packet buffer space is allocated allowing the system to hold a large number of pending RX packets, then depending on the relative priority of the network stack control thread other application threads can be denied bandwidth.

MAC Address

All Ethernet devices should have a unique address which has to be provided from somewhere. There are a number of possibilities:

1. The platform supplied driver instance `dwc_gmac_priv` structure provides the `enaddr` field which can be pre-initialised by the platform specific world. The MAC address supplied in the referenced structure is used if neither of the run-time options detailed below are provided. For example, the `cyclone5_sx` platform provides the CDL variable `CYGDAT_DEVS_ETH_CYCLONE5_SX_MACADDR_ETH0` which is used to initialise the descriptor `enaddr` field for the eth0 instance.

2. The platform HAL can provide the address. For example the target board may have a small serial EPROM or similar which is initialized during board manufacture. The platform HAL can read the serial EPROM during system startup and provide the information to the Ethernet driver. If this is the case then the platform HAL should provide a macro `CYGHWR_DEVS_ETH_DWC_GMAC_GET_ESA` in the exported header `cyg/hal/plf_arch.h`.

3. If the target hardware boots via RedBoot and uses a block of flash to hold configuration variables then one of these variables will be the MAC address. This is normally indicated by `CYGSEM_HAL_VIRTUAL_VECTOR_SUPPORT` being defined, in which case the driver will attempt to access the MAC address via the `CYGNUM_CALL_IF_FLASH_CFG_GET` interface to read the `CYGNUM_FLASH_CFG_TYPE_CONFIG_ESA` option. The MAC address to use can be manipulated at the RedBoot prompt using the **fconfig** command, thus giving each board a unique address. An eCos application containing the Ethernet driver will automatically pick up this address.

When designing a new target board it is recommended that the board comes with a unique network address supported by the platform HAL, rather than relying on users to change the address. The latter approach can be error-prone and will lead to failures that are difficult to track down.

Platform-specific PHY

The Ethernet GMAC hardware relies on an external media independent interface (MII), also known as a PHY chip. This separate chip handles the low-level details of Ethernet communication, for example negotiating a link speed with the hub. In most scenarios the PHY chip simply does the right thing and needs no support from the Ethernet driver. If there are special requirements, for example if the board has to be hardwired to communicate at 10Mbps rather than autonegotiate the link speed, then usually this is handled by fixed logic levels on some of the PHY pins or by using jumpers.

The driver supports asynchronous reporting of PHY events when the CDL option `CYGSEM_DEVS_ETH_DWC_GMAC_PHY_EVENT` is configured. Currently this event support is only available when using the lwIP TCP/IP networking stack.

Chapter 173. Freescale ColdFire Ethernet Driver

Name

CYGPKG_DEVS_ETH_MCFxxxx — eCos Support for Freescale ColdFire On-chip Ethernet Devices

Description

Some members of the Freescale ColdFire family of processors come with an on-chip ethernet device. This package provides an eCos driver for that device. The driver supports both polled mode for use by RedBoot and interrupt-driven mode for use by a full TCP/IP stack.

The original version of the driver was written specifically for the MCF5272 processor. It has since been made to work on other members of the ColdFire family.

There are in fact two driver implementations within this driver package, one standard driver suitable for use with various TCP/IP stacks including at least RedBoot, BSD and lwIP; and one specific to lwIP. The lwIP-specific driver is a streamlined efficient version designed for very low RAM overhead. As a result it is implemented intentionally at the expense of features such as multiple network device support, and network debugging under RedBoot, but has improvements such as zero-copy reception as well as zero-copy transmission if certain constraints are met by the data packet to be transmitted.

Configuration Options

This ethernet package should be loaded automatically when selecting a target containing a ColdFire processor with on-chip ethernet, and it should never be necessary to load it explicitly. If the application does not actually require ethernet functionality then the package is inactive and the final executable will not suffer any overheads from unused functionality. This is determined by the presence of the generic ethernet I/O package CYGPKG_IO_ETH_DRIVERS. Typically the choice of eCos template causes the right thing to happen. For example the default template does not include any TCP/IP stack so CYGPKG_IO_ETH_DRIVERS is not included, but the net, redboot and lwip_eth templates do include a TCP/IP stack so will specify that package and hence enable the ethernet driver.

The choice between using the standard driver, or the lwIP-specific driver is not made within this package, but is instead made in the generic ethernet I/O package CYGPKG_IO_ETH_DRIVERS using the options within the lwIP driver model component (CYGIMP_IO_ETH_DRIVERS_LWIP_DRIVER_MODEL). The standard driver is the default.

All eCos network devices need a unique name. By default the on-chip ethernet device is assigned the name eth0 but can be changed through the configuration option CYGDAT_DEVS_ETH_MCFxxxx_NAME. This is useful if for example the target hardware includes a number of additional off-chip ethernet devices. This setting is unused for the lwIP-specific driver.

The hardware requires that incoming ethernet frames are received into one of a small number of buffers, arranged in a ring. Once a frame has been received and its size is known the driver will pass it up to higher-level code for further processing. The number of these buffers is configurable via the option CYGNUM_DEVS_ETH_MCFxxxx_RXBUFFERS. In the standard ethernet driver, each receive buffer requires 1528 bytes of memory; with the lwIP-specific driver, the size of each buffer is set with the lwIP option CYGNUM_LWIP_PBUF_POOL_BUFSIZE, and multiple buffers are chained if needed to fulfil the requirements of incoming frames. A smaller number of buffers increases the probability that incoming ethernet frames have to be discarded. TCP/IP stacks are designed to cope with the occasional lost packet, but if too many frames are discarded then this will greatly affect performance. A key issue here is that passing the incoming frames up to higher-level code typically happens at thread level and hence the system behaviour is defined in large part by the priority of various threads running in the TCP/IP stack. If application code has high-priority threads that take up much of the available cpu time and the TCP/IP stack gets little chance to run then there will be little opportunity to pass received frames up the stack. Balancing out the various thread priorities and the number of receive buffers is the responsibility of the application developer.

By default the ethernet driver will raise interrupts using a low priority level. The exact value will depend on the processor being used, for example the MCF5282 interrupt controllers impose specific constraints on interrupt priorities. The driver does very little at interrupt level, instead the real work is done via threads inside the TCP/IP stack. Hence the interrupt priority has little or no effect on the system's behaviour. If the default priorities are inappropriate for some reason then they can be changed through the configuration options CYGNUM_DEVS_ETH_MCFxxxx_ISR_RX_PRIORITY and CYGNUM_DEVS_ETH_MCFxxxx_ISR_TX_PRIORITY.

There is an option related to the default network MAC address, CYGDAT_DEVS_ETH_MCFxxxx_PLATFORM_MAC. This is discussed in more detail below.

Optionally the ethernet driver can maintain statistics about the number of incoming and transmitted ethernet frames, receive overruns, collisions, and other conditions. Maintaining and providing these statistics involves some overhead, and is controlled by the configuration option CYGFUN_DEVS_ETH_MCFxxxx_STATISTICS. Typically these statistics are only accessed through SNMP, so by default statistics gathering is enabled if the configuration includes CYGPKG_SNMPAGENT and disabled otherwise.

MAC Address

The ColdFire processors do not have a built-in unique network MAC address since that would require slightly different manufacturing for each chip. All ethernet devices should have a unique address so this has to come from elsewhere. There are a number of possibilities:

1. The platform HAL can provide the address. For example the target board may have a small serial EPROM or similar which is initialized during board manufacture. The platform HAL can read the serial EPROM during system startup and provide the information to the ethernet driver. If this is the case then the platform HAL should implement the CDL interface CYGINT_DEVS_ETH_MCFxxxx_PLATFORM_MAC and provide a macro HAL_MCFxxxx_ETH_GET_MAC_ADDRESS in the exported header cyg/hal/plf_arch.h.

2. There is a configuration option CYGDAT_DEVS_ETH_MCFxxxx_PLATFORM_MAC which specifies the default MAC address. Manipulating this option is fine if the configuration will only be used on a single board. However if multiple boards run applications with the same configuration then they would all have the same MAC address, and the resulting behaviour is undefined.

3. If the target hardware boots via RedBoot and uses a block of flash to hold configuration variables then one of these variables will be the MAC address. It can be manipulated at the RedBoot prompt using the **fconfig** command, thus giving each board a unique address. An eCos application containing the ethernet driver will automatically pick up this address.

When designing a new target board it is recommended that the board comes with a unique network address supported by the platform HAL, rather than relying on users to change the address. The latter approach can be error-prone and will lead to failures that are difficult to track down.

Platform-specific PHY

The on-chip ethernet hardware relies on an external media independent interface (MII), also known as a PHY chip. This separate chip handles the low-level details of ethernet communication, for example negotiating a link speed with the hub. In most scenarios the PHY chip simply does the right thing and needs no support from the ethernet driver. If there are special requirements, for example if the board has to be hardwired to communicate at 10Mbps rather than autonegotiate the link speed, then usually this is handled by fixed logic levels on some of the PHY pins or by using jumpers.

The eCos ethernet driver assumes that the PHY is already fully operational and does not interact with it in any way. If the target hardware does require software initialization of the PHY chip then usually this will be done in the platform HAL, because the choice of PHY chip is a characteristic of the platform.

Chapter 174. Nios II Triple Speed Ethernet Driver

Name

`CYGPKG_DEVS_ETH_NIOS2_TSE` — eCos Support for Nios II Triple-Speed Ethernet Devices

Description

A Nios II hardware design can include one or more triple speed ethernet devices or TSEs. The package `CYGPKG_DEVS_ETH_NIOS2_TSE` provides an eCos driver for a single TSE device. It supports both polled mode for use by RedBoot and interrupt-driven mode for use by a full TCP/IP stack.

Configuration Options

The Nios II TSE driver package should be loaded automatically when creating an eCos configuration for a hardware design which includes the required devices, and it should never be necessary to load the package explicitly. If the application does not actually require ethernet functionality then the package is inactive and the final executable will not suffer any overheads from unused functionality. This is determined by the presence of the generic ethernet I/O package `CYGPKG_IO_ETH_DRIVERS`. Typically the choice of eCos template causes the right thing to happen. For example the default template does not include any TCP/IP stack so `CYGPKG_IO_ETH_DRIVERS` is not included, but both the net and redboot templates do include a TCP/IP stack so will specify that package and hence enable the ethernet driver.

The driver package will only instantiate the support for a single TSE device. If the hardware design involves multiple TSE devices then support for the additional ones can be instantiated by application code. This can be achieved largely by cloning file `src/tse0.c` in this package.

There are two configuration options related to the device instantiation. `CYGDAT_DEVS_ETH_NIOS2_TSE0_NAME` sets the device name, defaulting to “eth0”. Typically this only needs to be changed if the hardware design includes other types of ethernet device and their drivers also attempt to create a device “eth0”. `CYGDAT_DEVS_ETH_NIOS2_TSE_ETH0_MAC` specifies the fallback ethernet station address or MAC address. In typical eCos systems the MAC address is provided via a RedBoot fconfig option, allowing each board to have its own address. However if the fconfig functionality is unavailable, for example when debugging via jtag, then the fallback address will be used instead. Note that each board on a network must have a unique MAC address, so if there are several boards on the network using the fallback address and the same eCos configuration then network communication can be expected to fail. Applications can also change the MAC address at run-time using a `SIOCSIFHWADDR` `ioctl`. However this `ioctl` should not be used when debugging over ethernet because it will break the debug channel.

Porting

Each triple speed ethernet device requires three units in the hardware design: the tse_mac unit itself, and sgdma_rx and sgdma_tx scatter-gather DMA controllers. Typically the hardware design will also include a bank of on-chip RAM to hold the DMA descriptors. The settings for these units are best cloned from a reference hardware design such as the TSE_SGDMA examples in the Nios II Embedded Design Suite, or the eCosPro_TSEplus design. The h/w design HAL package should provide address and interrupt vector definitions for the various units.

Chapter 175. SMSC LAN9118 Ethernet Driver

Name

CYGPKG_DEVS_ETH_SMSC_LAN9118 — eCos Support for SMSC LAN9118 Ethernet Devices

Description

The SMSC LAN9118 chip is a high performance single chip ethernet controller which can be interfaced to a variety of embedded processors. This package provides an eCos driver for that device. The driver supports both polled mode for use by RedBoot and interrupt-driven mode for use by a full TCP/IP stack.

The exact interface between the LAN9118 chip and the main processor is determined by the platform HAL. On some platforms there may even be multiple LAN9118 chips. This package only provides the platform-independent code. It is up to the platform HAL to instantiate one or more device instances and to provide information such as the base address and interrupt vector. There is also no explicit support for features like auto-negotiation or advanced flow control. These are left to the platform HAL or to the application, although usually the default settings will be acceptable for most applications.

Configuration Options

This package should be loaded automatically when selecting a target equipped with a LAN9118 ethernet chip. and it should never be necessary to load it explicitly. If the application does not actually require ethernet functionality then the package is inactive and the final executable will not suffer any overheads from unused functionality. This is determined by the presence of the generic ethernet I/O package CYGPKG_IO_ETH_DRIVERS. Typically the choice of eCos template causes the right thing to happen. For example the default template does not include any TCP/IP stack so CYGPKG_IO_ETH_DRIVERS is not included, but both the net and redboot templates do include a TCP/IP stack so will specify that package and hence enable the ethernet driver.

Optionally the ethernet driver can maintain statistics about the number of incoming and transmitted ethernet frames, receive overruns, collisions, and other conditions. Maintaining and providing these statistics involves some overhead, and is controlled by the configuration option CYGFUN_DEVS_ETH_SMSC_LAN9118_STATISTICS. Typically these statistics are only accessed through SNMP, so by default statistics gathering is enabled if the configuration includes CYGPKG_SNMPAGENT and disabled otherwise.

Porting the Driver to New Hardware

It is the responsibility of the platform HAL to instantiate one or more devices, depending on the number of LAN9118 chips present. Typically this involves a separate file in the platform HAL sources:

```
#include <cyg/io/lan9118.h>

LAN9118_INSTANCE(alaia, 0, "eth0", alaia_eth_init);

static bool
alaia_eth_init(struct cyg_netdevtab_entry* tab)
{
    …

    return cyg_lan9118_eth_init(tab);
}
```

The first two arguments to the LAN9118_INSTANCE macro identify the platform and the device instance, and are used to construct unique variable names. The third argument gives the device name, and the final argument is a platform-specific initialization function. The platform HAL should also contain suitable CDL to build this file:

```
cdl_component CYGHWR_HAL_ALAIA_ETH {
    display   "External ethernet support"
    parent    CYGPKG_IO_ETH_DRIVERS
    flavor    none
    active_if CYGPKG_IO_ETH_DRIVERS
    implements  CYGNUM_DEVS_ETH_SMSC_LAN9118_COUNT
    compile   -library=libextras.a alaia_eth.c
    description "
              The Alaia board comes with a single LAN9118 ethernet device."

    cdl_option CYGNUM_HAL_ALAIA_ETH_ISR_PRIORITY {
        …
    }
    …
}
```

If the configuration does not include the generic ethernet support then this component will be inactive. Otherwise the file alaia_eth.c will get built into libextras.a, ensuring the device instance does not get eliminated by linker garbage collection. The interface CYGNUM_DEVS_ETH_SMSC_LAN9118_COUNT should be implemented once per LAN9118 chip. If additional configuration options are needed, for example to control the MAC address or the interrupt priority, then these can go inside the component.

The driver needs to know where to access the device. If there is a single LAN9118 chip then the required information can be supplied via #define's in the plf_io.h header:

```
#define HAL_LAN9118_BASE            0xBA000000
#define HAL_LAN9118_ISRVEC          CYGNUM_HAL_ISR_LAN9118
#define HAL_LAN9118_ISRPRI          CYGNUM_HAL_ALAIA_ETH_ISR_PRIORITY
```

Otherwise the platform-specific initialization function should put this information in fields in the LAN9118 instance structure:

```
static bool
alaia_eth_init(struct cyg_netdevtab_entry* tab)
{
    LAN9118_INSTANCE_NAME(alaia, 0).lan9118_base   = 0xBA000000;
    LAN9118_INSTANCE_NAME(alaia, 0).lan9118_isrvec = CYGNUM_HAL_ISR_PIO4;
    LAN9118_INSTANCE_NAME(alaia, 0).lan9118_isrpri = 1;
    LAN9118_INSTANCE_NAME(alaia, 1).lan9118_base   = 0xBB000000;
    LAN9118_INSTANCE_NAME(alaia, 1).lan9118_isrvec = CYGNUM_HAL_ISR_PIO5;
    LAN9118_INSTANCE_NAME(alaia, 1).lan9118_isrpri = 1;
    …
    return cyg_lan9118_eth_init(tab);
}
```

The initialization function should ensure that the processor's bus interface is set up correctly for talking to the ethernet chip, and that the interrupt vector has been configured correctly for level vs. edge interrupts. This must happen before calling the driver init function. Also the *lan9118_hw_flags* should be set correctly as per the flags in lan9118.h, for example:

```
static bool
alaia_eth_init(struct cyg_netdevtab_entry* tab)
{
    …
    LAN9118_INSTANCE_NAME(alaia, 0).lan9118_hw_flags  =
        (LAN9118_HW_FLAGS_IRQ_POL_ACTIVE_HIGH    |
         LAN9118_HW_FLAGS_IRQ_PUSH_PULL          |
         LAN9118_HW_FLAGS_HAS_LED1               |
         LAN9118_HW_FLAGS_HAS_LED2);
    …
    return cyg_lan9118_eth_init(tab);
}
```

The LAN9118 can be accessed via either a 16-bit or 32-bit bus, and from big-endian or little-endian processors. This gives a number of combinations. The chip is inherently little-endian, so on a little-endian processor there should be no problems. On a big-endian processor there are two possibilities. If the LAN9118 is interfaced in the obvious way then it will be necessary to swap the data of all incoming and outgoing packets, which imposes a significant performance penalty. On a 16-bit bus the LAN9118_HW_FLAGS_16BIT_BE flag should be set. Alternatively the bytes on the bus can be swapped, either by the hardware or by programming the processor's bus interface. This means no swapping is needed for data, but all accesses to the LAN9118's command and status registers need swapping instead. However most of that swapping can be done at compile-time so has no overhead. Defining `HAL_LAN9118_SWAP_COMMANDS` in `plf_io.h` sets up this mode. If there are multiple ethernet chips then the driver assumes they are all wired the same way. For further details consult the driver's source code.

All ethernet devices require a unique MAC address. Ideally this will be provided by a serial EEPROM or similar, and if such a device is present and attached to the LAN9118 then it will be used automatically by the ethernet chip to set the MAC address. If the platform does not have a suitable EEPROM then the MAC address must come from elsewhere, for example a RedBoot fconfig option, and the platform-specific initialization function should fill in the instance's *`lan9118_mac`* field.

Chapter 176. Synthetic Target Ethernet Driver

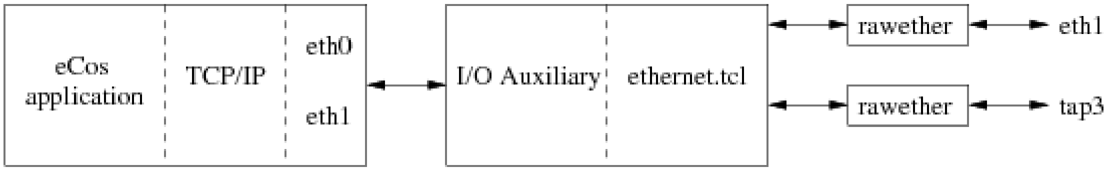
Name

Synthetic Target Ethernet Support — Allow synthetic target applications to perform ethernet I/O

Overview

The synthetic target ethernet package can provide up to four network devices, `eth0` to `eth3`. These can be used directly by the eCos application or, more commonly, by a TCP/IP stack that is linked with the eCos application. Each eCos device can be mapped on to a real Linux network device. For example, if the Linux PC has two ethernet cards and `eth1` is not currently being used by Linux itself, then one of the eCos devices can be mapped on to this Linux device. Alternatively, it is possible to map some or all of the eCos devices on to the ethertap support provided by the Linux kernel.

The ethernet package depends on the I/O auxiliary provided by the synthetic target architectural HAL package. During initialization the eCos application will attempt to instantiate the desired devices, by sending a request to the auxiliary. This will load a Tcl script `ethernet.tcl` that is responsible for handling the instantiation request and subsequent I/O operations, for example transmitting an ethernet packet. However, some of the low-level I/O operations cannot conveniently be done by a Tcl script so `ethernet.tcl` will actually run a separate program **rawether** to interact with the Linux network device.



On the target-side there are configuration options to control which network devices should be present. For many applications a single device will be sufficient, but if the final eCos application is something like a network bridge then the package can support multiple devices. On the host-side each eCos network device needs to be mapped on to a Linux one, either a real ethernet device or an ethertap device. This is handled by an entry in the target definition file:

```
synth_device ethernet {
    eth0 real eth1
    eth1 ethertap tap3 00:01:02:03:FE:05
    …
}
```

The ethernet package also comes with support for packet logging, and provides various facilities for use by user Tcl scripts.

Installation

Before a synthetic target eCos application can access ethernet devices it is necessary to build and install host-side support. The relevant code resides in the `host` subdirectory of the synthetic target ethernet package, and building it involves the standard **configure**, **make** and **make install** steps. The build involves a new executable **rawether** which must be able to access a raw Linux network device. This is achieved by installing it suid root, so the **make install** step has to be run with superuser privileges.

Caution

Installing **rawether** suid root introduces a potential security problem. Although normally **rawether** is executed only by the I/O auxiliary, theoretically it can be run by any program. Effectively it gives any user the ability to monitor all ethernet traffic and to inject arbitrary packets into the network. Also, as with any suid root programs there may be as yet undiscovered exploits. Users and system administrators should consider the risks before running **make install**.

There are two main ways of building the host-side software. It is possible to build both the generic host-side software and all package-specific host-side software, including the ethernet support, in a single build tree. This involves using the **configure** script at the toplevel of the eCos repository. For more information on this, see the `README.host` file at the top of the repository. Note that if you have an existing build tree which does not include the synthetic target ethernet support then it will be necessary to rerun the toplevel configure script: the search for appropriate packages happens at configure time.

The alternative is to build just the host-side for this package. This requires a separate build directory, building directly in the source tree is disallowed. The **configure** options are much the same as for a build from the toplevel, and the `README.host` file can be consulted for more details. It is essential that the ethernet support be configured with the same `--prefix` option as other eCos host-side software, especially the I/O auxiliary provided by the architectural synthetic target HAL package, otherwise the I/O auxiliary will be unable to locate the ethernet support.

Target-side Configuration Options

The target-side code can be configured to support up to four ethernet devices, `eth0` to `eth3`. By default `eth0` is enabled if the configuration includes a TCP/IP stack, otherwise it is disabled. The other three devices are always disabled by default. If any of the devices are enabled then there will also be the usual configuration options related to building this package. Other options related to network devices, for example whether or not to use DHCP, are provided by the generic network device package.

Real Ethernet

One obvious way of providing a synthetic target eCos application with ethernet I/O is to use a real ethernet device in the PC: transmitted packets go out on a real network, and packets on the network addressed to the right MAC address are passed on to eCos. This way synthetic target networking behaves just like networking on a real target with ethernet hardware. For example, if there is a DHCP server anywhere on the network then eCos will be able to contact it during networking startup and get hold of IP address information.

Configuring the ethernet support to use a real ethernet device requires a simple entry in the target definition file:

```
synth_device ethernet {
    <eCos device> real <linux device>
    …
}
```

For example, to map the eCos network device `eth0` to the Linux device `eth1`:

```
synth_device ethernet {
    eth0 real eth1
    …
}
```

It is not possible for an ethernet device to be shared by both the eCos TCP/IP stack and the Linux one: there would be no simple way to work out which stack incoming packets are intended for. In theory it might be possible to do some demultiplexing using distinct IP addresses, but it would be impossible to support some functionality such as DHCP. Therefore the **rawether** program will refuse to access any ethernet device already in use. On a typical Linux system `eth0` will be used for Linux networking, and the PC will have to be equipped with additional ethernet devices for use by eCos.

The **rawether** program will access the hardware via the appropriate Linux device driver, so it is important that the system is set up such that the relevant module will be automatically loaded or is already loaded. The details of this will depend on the installed distribution and version, but typically it will involve an entry in `/etc/modules.conf`.

Ethertap

The Linux kernel's ethertap facility provides a virtual network interface. A Linux application, for example the **rawether** program, can open a special character device `/dev/net/tun`, perform various `ioctl` calls, and then `write` and `read` ethernet packets. When the device is opened the Linux kernel automatically creates a new network interface, for example `tap0`. The Linux TCP/IP stack can be made to use this network interface like any other interface, receiving and transmitting ethernet packets. The net effect is a virtual network connecting just the Linux and eCos TCP/IP stacks, with no other nodes attached. By default all traffic remains inside this virtual network and is never forwarded to a real network.

Support for the ethertap facility may or may not be provided automatically, depending on your Linux distribution and version. If your system does not have a device `/dev/net/tun` or a module `tun.o` then the appropriate kernel documentation should be consulted, for example /usr/src/linux-2.4/Documentation/networking/tuntap.txt. If you are using an old Linux kernel then the ethertap functionality may be missing completely. When the **rawether** program is configured and built, the **configure** script will check for a file /usr/include/linux/if_tun.h. If that file is missing then **rawether** will be built without ethertap functionality, and only real ethernet interfaces will be supported.

The target definition file is used to map eCos network devices on to ethertap devices. The simplest usage is:

```
synth_device ethernet {
    eth0 ethertap
    …
}
```

The Linux kernel will automatically allocate the next available tap network interface. Usually this will be tap0 but if other software is using the ethertap facility, for example to implement a VPN, then a different number may be allocated. Usually it will be better to specify the particular tap device that should be used for each eCos device, for example:

```
synth_device ethernet {
    eth0 ethertap tap3
    eth1 ethertap tap4
    …
}
```

The user now knows exactly which eCos device is mapped onto which Linux device, avoiding much potential confusion. Because the virtual devices are emulated ethernet devices, they require MAC addresses. There is no physical hardware to provide these addresses, so normally MAC addresses will be invented. That means that each time the eCos application is run it will have different MAC addresses, which makes it more difficult to compare the results of different runs. To get more deterministic behaviour it is possible to specify the MAC addresses in the target definition file:

```
synth_device ethernet {
    eth0 ethertap tap3 00:01:02:03:FE:05
    eth1 ethertap tap4 00:01:02:03:FE:06
    …
}
```

During the initialization phase the eCos application will instantiate the various network devices. This will cause the I/O auxiliary to load the ethernet.tcl script and spawn **rawether** processes, which in turn will open /dev/net/tun and perform the appropriate ioctl calls. On the Linux side there will now be new network interfaces such as tap3, and these can be configured like any other network interface using commands such as **ifconfig**. In addition, if the Linux system is set up with hotplug support then it may be possible to arrange for the network interface to become active automatically. On a Red Hat Linux system this would require files such as /etc/sysconfig/network-scripts/ifcfg-tap3, containing data like:

```
DEVICE="tap3"
BOOTPROTO="none"
BROADCAST=10.2.2.255
IPADDR="10.2.2.1"
NETMASK="255.255.255.0"
NETWORK=10.2.2.0
ONBOOT="no"
```

This gives the Linux interface the address 10.2.2.1 on the network 10.2.2.0. The eCos network device should be configured with a compatible address. One way of doing this would be to enable CYGHWR_NET_DRIVER_ETH0_ADDRS, set CYGHWR_NET_DRIVER_ETH0_ADDRS_IP to 10.2.2.2, and similarly update the NETMASK, BROADCAST, GATEWAY and SERVER configuration options.

It should be noted that the ethertap facility provides a virtual network, and any packets transmitted by the eCos application will not appear on a real network. Therefore usually there will no accessible DHCP server, and eCos cannot use DHCP or BOOTP to obtain IP address information. Instead the eCos configuration should use manual or static addresses.

When **rawether** exits, the tap interface is removed by the kernel. By adding the parameter persistent **rawether** will set the persistent flag on the tap device.

```
synth_device ethernet {
    eth0 ethertap tap3 00:01:02:03:FE:05
    eth1 ethertap tap4 00:01:02:03:FE:06 persistent
    …
}
```

With this flag set the kernel will not remove the interface when **rawether** exits. This means applications such as **dhcpd**, **radvd**, and **tcpdump** will continue to run on the interface between invocations of synthetic targets. As a result the target can dynamically obtain its IP addresses from these daemons. Note it is a good idea to specify a MAC address otherwise a different random MAC address will be used each time and the dhcpd daemon will not be able to reissue the same IP address.

Host daemons like dhcpd, ntpd, radvd etc are started at boot time. Since the tap device does not exists at this point in time it is not possible for these daemons to bind to the tap device. A simple solution is to use the program **install/bin/mktap**. This takes one parameter, the name of the tap device it should create. eg, `tap3`.

An alternative approach would be to set up the Linux box as a network bridge, using commands like **brctl** to connect the virtual network interface `tap3` to a physical network interface such as `eth0`. Any packets sent by the eCos application will get forwarded automatically to the real network, and some packets on the real network will get forwarded over the virtual network to the eCos application. Note that the eCos application might also get some packets that were not intended for it, but usually those will just be discarded by the eCos TCP/IP stack. The exact details of setting up a network bridge are left as an exercise to the reader.

Packet Logging

The ethernet support comes with support for logging the various packets that are transferred, including a simple protocol analyser. This generates simple text output using the filter mechanisms provided by the I/O auxiliary, so it is possible to control the appearance and visibility of different types of output. For example the user might want to see IPv4 headers and all ICMPv4 and ARP operations, but not TCP headers or any of the packet data.

The protocol analyser is not intended to be a fully functional analyser with knowledge of many different TCP/IP protocols, advanced search facilities, graphical traffic displays, and so on. Functionality like that is already provided by other tools such as ethereal and tcpdump. Achieving similar levels of functionality would require a lot of work, for very little gain. It is still useful to have some protocol analysis functionality available because the output will be interleaved with other output, for example `printf` calls from the application. That may make it easier to understand the sequence of events.

One problem with logging ethernet traffic is that it can involve very large amounts of data. If the application is expected to run for a long time or is very I/O intensive then it is easy to end up with many megabytes. When running in graphical mode all the logging data will be held in memory, even data that is not currently visible. At some point the system will begin to run low on memory and performance will suffer. To avoid problems, the ethernet script maintains a flag that controls whether or not packet logging is active. The default is to run with logging disabled, but this can be changed in the target definition file:

```
synth_device ethernet {
    …
    logging 1
}
```

The ethernet script will add a toolbar button that allows this flag to be changed at run-time, allowing the user to capture traffic for certain periods of time while the application continues running.

The target definition file can contain the following entries for the various packet logging filters:

```
synth_device ethernet {
    …
    filter ether  -hide 0 -background LightBlue -foreground "#000080"
    filter arp    -hide 0 -background LightBlue -foreground "#000050"
    filter ipv4   -hide 0 -background LightBlue -foreground "#000040"
    filter ipv6   -hide 1 -background LightBlue -foreground "#000040"
    filter icmpv4 -hide 0 -background LightBlue -foreground "#000070"
    filter icmpv6 -hide 1 -background LightBlue -foreground "#000070"
    filter udp    -hide 0 -background LightBlue -foreground "#000030"
    filter tcp    -hide 0 -background LightBlue -foreground "#000020"
    filter hexdata   -hide 1 -background LightBlue -foreground "#000080"
    filter asciidata -hide 1 -background LightBlue -foreground "#000080"
}
```

All output will show the eCos network device, for example `eth0`, and the direction relative to the eCos application. Some of the filters will show packet headers, for example `ether` gives details of the ethernet packet header and `tcp` gives information about TCP headers such as whether or not the SYN flag is set. The TCP and UDP filters will also show source and destination addresses, using numerical addresses and if possible host names. However, host names will only be shown if the host appears in `/etc/hosts`: doing full DNS lookups while the data is being captured would add significantly to complexity and overhead. The `hexdata` and `asciidata` filters show the remainder of the packets after the ethernet, IP and TCP or UDP headers have been stripped.

Some of the filters will provide raw dumps of some of the packet data. Showing up to 1500 bytes of data for each packet would be expensive, and often the most interesting information is near the start of the packet. Therefore it is possible to set a limit on the number of bytes that will be shown using the target definition file. The default limit is 64 bytes.

```
synth_device ethernet {
    …
    max_show 128
}
```

User Interface Additions

When running in graphical mode the ethernet script extends the user interface in two ways: a button is added to the toolbar so that users can enable or disable packet logging; and an entry is added to the Help menu for the ethernet-specific documentation.

Command Line Arguments

The synthetic target ethernet support does not use any command line arguments. All configuration is handled through the target definition file.

Hooks

The ethernet support defines two hooks that can be used by other scripts, especially user scripts: `ethernet_tx` and `ethernet_rx`. The tx hook is called whenever eCos tries to transmit a packet. The rx hook is called whenever an incoming packet is passed to the eCos application. Note that this may be a little bit after the packet was actually received by the I/O auxiliary since it can buffer some packets. Both hooks are called with two arguments, the name of the network device and the packet being transferred. Typical usage might look like:

```
  proc my_tx_hook { arg_list } {
    set dev [lindex $arg_list 0]
    incr ::my_ethernet_tx_packets($dev)
    incr ::my_ethernet_tx_bytes($dev) [string length [lindex $arg_list 1]]
  }
  proc my_rx_hook { arg_list } {
    set dev [lindex $arg_list 0]
    incr ::my_ethernet_rx_packets($dev)
    incr ::my_ethernet_rx_bytes($dev) [string length [lindex $arg_list 1]]
  }
  synth::hook_add "ethernet_tx" my_tx_hook
  synth::hook_add "ethernet_rx" my_rx_hook
```

The global arrays `my_ethernet_tx_packets` etc. will now be updated whenever there is ethernet traffic. Other code, probably running at regular intervals by use of the Tcl **after** procedure, can then use this information to update a graphical monitor of some sort.

Additional Tcl Procedures

The ethernet support provides one additional Tcl procedure that can be used by other scripts;

```
ethernet::devices_get_list
```

This procedure returns a list of the ethernet devices that have been instantiated, for example `{eth0 eth1}`.

Part XLVI. DNS for eCos and RedBoot

eCos and RedBoot can both use the DNS package to perform network name lookups.

Table of Contents

Chapter 177. DNS

DNS API

The DNS client uses the normal BSD API for performing lookups: `gethostbyname()`, `gethostbyaddr()`, `getaddrinfo()`, `getnameinfo()`.

There are a few restrictions:

- If the DNS server returns multiple authoritive records for a host name to `gethostbyname`, the hostent will only contain a record for the first entry. If multiple records are desired, use `getaddrinfo`, which will return multiple results.
- The code has been made thread safe. ie multiple threads may call `gethostbyname()` without causing problems to the hostent structure returned. What is not safe is one thread using both `gethostbyname()` and `gethostbyaddr()`. A call to one will destroy the results from the previous call to the other function. `getaddrinfo()` and `getnameinfo()` are thread safe and so these are the preferred interfaces. They are also address family independent so making it easier to port code to IPv6.
- The DNS client will only return IPv4 addresses to RedBoot. At the moment this is not really a limitation, since RedBoot only supports IPv4 and not IPv6.

To initialise the DNS client the following function must be called:

```
#include <network.h>
int cyg_dns_res_start(char * dns_server)
```

Where dns_server is the address of the DNS server. The address must be in numeric form and can be either an IPv4 or an IPv6 address.

There also exists a deprecated function to start the DNS client:

```
int cyg_dns_res_init(struct in_addr *dns_server)
```

where dns_server is the address of the DNS server the client should query. The address should be in network order and can only be an IPv4 address.

On error both this function returns -1, otherwise 0 for success. If lookups are attemped before this function has been called, they will fail and return NULL, unless numeric host addresses are passed. In this cause, the address will be converted and returned without the need for a lookup.

A default, hard coded, server may be specified in the CDL option `CYGDAT_NS_DNS_DEFAULT_SERVER`. The use of this is controlled by `CYGPKG_NS_DNS_DEFAULT`. If this is enabled, `init_all_network_interfaces()` will initialize the resolver with the hard coded address. The DHCP client or user code my override this address by calling `cyg_dns_res_init` again.

The DNS client understands the concepts of the target being in a domain. By default no domain will be used. Host name lookups should be for fully qualified names. The domain name can be set and retrieved using the functions:

`int` **`getdomainname`**`(`*`name`*`, `*`len`*`);`

`int` **`setdomainname`**`(`*`name`*`, `*`len`*`);`

Alternatively, a hard coded domain name can be set using CDL. The boolean `CYGPKG_NS_DNS_DOMAINNAME` enables this and the domain name is taken from `CYGPKG_NS_DNS_DOMAINNAME_NAME`.

Once set, the DNS client will use some simple heuristics when deciding how to use the domainname. If the name given to the client ends with a "." it is assumed to be a FQDN and the domain name will not be used. If the name contains a "." somewhere within it, first a lookup will be performed without the domainname. If that fails the domainname will be appended and looked up. If the name does not contain a ".", the domainname is appended and used for the first query. If that fails, the unadorned name is lookup.

The `getaddrinfo` will return both IPv4 and IPv6 addresses for a given host name, when IPv6 is enabled in the eCos configuration. The CDL option `CYGOPT_NS_DNS_FIRST_FAMILY` controls the order IPv6 and IPv4 addresses are returned in the linked list of `addrinfo` structures. If the value `AF_INET` is used, the IPv4 addresses will be first. If the value `AF_INET6`, which is the default, is used, IPv6 address will be first. This ordering will control how clients attempt to connect to servers, ie using IPv6 or IPv4 first.

DNS Client Testing

The DNS client has a test program, dns1.c, which tests many of the features of the DNS client and the functions `gethostbyname()`, `gethostbyaddr()`, `getaddrinfo()`, `getnameinfo()`.

In order for this test to work, a DNS server must be configured with a number of names and addresses. The following is an example forward address resolution database for bind v9, which explains the requirements.

```
@         1D IN SOA       @ hostmaster.ecoscentric.com. (
                        2017022501      ; serial
                        3H              ; refresh
                        2H              ; retry
                        2W              ; expiry
                        1D )            ; minimum

          1D IN NS        ns0
          1D IN NS        ns1
          1D IN NS        dns1.zoneedit.com.
          1D IN NS        dns2.zoneedit.com.

albus     1D IN A         212.13.207.200
barn      1D IN A         87.127.120.188
farm      1D IN A         88.97.17.238
fawkes    1D IN A         212.13.207.202
www       1D IN CNAME     albus
www2      1D IN CNAME     fawkes
```

The actual names and addresses do not matter, since they are configurable in the test. What is important is the relationship between the names and the addresses and there family. ie hostnamev4 should map to one IPv4 address. hostnamev46 should map to both an IPv4 and an IPv6 address. cnamev4 should be a CNAME record for hostname4. Reverse lookup information is also needed by the test.

The information placed into the DNS server is also need in the test case. A structure is defined to hold this information:

```
struct test_info_s {
    char * dns_server_v4;
    char * dns_server_v6;
    char * domain_name;
    char * hostname_v4;
    char * cname_v4;
    char * ip_addr_v4;
    char * hostname_v6;
    char * cname_v6;
    char * ip_addr_v6;
    char * hostname_v46;
    char * cname_v46;
    char * ip_addr_v46_v4;
    char * ip_addr_v46_v6;
};
```

The test program may hold a number of such structures for different DNS server. The test will use each structure in turn to perform the tests. If IPv6 is not enabled in the eCos configuration, the entries which use IPv6 may be assigned to NULL.

Part XLVII. eCosPro-SecureSockets

Important

eCosPro-SecureSockets is an optional add-on package and may not be included in your release of eCosPro. If this package is not listed in either the graphical or command line eCos Configuration tool, please contact eCosCentric for availability and pricing.

Table of Contents

Chapter 178. OpenSSL eCos Support

Introduction

The eCosPro-SecureSockets package is a port of OpenSSL to eCos. It currently comprises a port of version 1.0.1u, but eCosCentric will issue updates from time to time after new releases become avaliable.

Licensing, Copyrights and Patents

OpenSSL is distributed under a BSD-style Open Source license. The user is referred to the file `LICENSE` in the OpenSSL release for details, and is responsible for complying with the conditions therein.

The following text on the subject of patents is adapted from text in the `README` file in the OpenSSL release:

> Various companies hold various patents for various algorithms in various locations around the world. *YOU* are responsible for ensuring that your use of any algorithms is legal by checking if there are any patents in your country. The following are some of the patents that we know about or are rumoured to exist. This is not a definitive list.
>
> - RSA Security holds software patents on the RC5 algorithm. If you intend to use this cipher, you must contact RSA Security for licensing conditions. Their web page is http://www.rsasecurity.com/.
> - RC4 is a trademark of RSA Security, so use of this label should perhaps only be used with RSA Security's permission.
> - The IDEA algorithm used to be patented by Ascom, but as of 2012, there are no longer any valid patents remaining so it may now be used patent-free.
> - NTT and Mitsubishi have patents and pending patents on the Camellia algorithm, but allow use at no charge without requiring an explicit licensing agreement: http://info.isl.ntt.co.jp/crypt/eng/info/chiteki.html.

To ensure that these patents are not accidentally violated, these algorithms are disabled by default and must be enabled explicitly by the user to be included.

Part of the conditions of using OpenSSL is that the following acknowledgment be displayed and applies not only to us, here, now, but to *you* as well:

> This product includes cryptographic software written by Eric Young (eay@cryptsoft.com).

Configuration

OpenSSL is designed to work in large, fully featured operating systems. It expects to find a filesystem, a networking stack and a full C library. All of these things are available in eCos. However, a full configuration of this sort can be very large, both in terms of the link library generated and the size of executables. This may not be appropriate for an embedded system with limited memory availability. To mitigate these effects, the eCos port for OpenSSL has been adapted to work in three basic configurations. Users can then adapt these further to their own needs.

Full Configuration

Accessing the complete functionality of OpenSSL requires a fully configured version of eCos. This should be based on the `net` template together with a number of additional packages and configurations. These additions are contained in the `openssl_full.ecm` file in the package `misc` directory. This configuration may be built using the following sequence of shell commands.

```
$ mkdir openssl_full
$ cd openssl_full
$ ecosconfig new TARGET net
$ ecosconfig import $ECOS_REPOSITORY/services/openssl/VERSION/misc/openssl_full.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

The same effect can be achieved from the graphical **configtool** by selecting the net template and then importing the ECM file.

Default Configuration

If no networking is available, openssl may still be built, but only the cryptographic and filesystem based functions may be used. This configuration is based on the `default` template which included file I/O functions and the C library, but no networking, plus the addition of an ECM file. This configuration may be built with the following sequence of commands, or the equivalent in the **configtool**:

```
$ mkdir openssl_default
$ cd openssl_default
$ ecosconfig new TARGET default
$ ecosconfig import $ECOS_REPOSITORY/services/openssl/VERSION/misc/openssl_default.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

Kernel Configuration

OpenSSL will also build in a minimal kernel only configuration. This will be based on the `kernel` template plus an ECM file. This may be built with the following commands, or the equivalent in the **configtool**:

```
$ mkdir openssl_kernel
$ cd openssl_kernel
$ ecosconfig new TARGET kernel
$ ecosconfig import $ECOS_REPOSITORY/services/openssl/VERSION/misc/openssl_kernel.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

This configuration is very basic, with no networking or file I/O, only the basic cryptographic functions will be available. The ECM file adds only STDLIB and Internationalization, mainly so that `qsort()` can be include; without this ASN1 support, and therefore most cryptographic components, cannot be built.

Serial Line Support

Some parts of the OpenSSL library read data from standard input; in particular some functions read passwords with echoing disabled. They achieve this by using TERMIOS functions on the device. For this to work correctly it is necessary to configure the standard I/O to use a serial driver, and to enable TERMIOS support on it. If you intend to use serial line 0 for this, then the following ECM fragment will set up the serial device correctly:

```
cdl_configuration eCos {
    package CYGPKG_IO_SERIAL current ;
};

# Enable serial device support
cdl_component CYGPKG_IO_SERIAL_DEVICES {
    user_value 1 ;
};

# Enable general TERMIOS support
cdl_component CYGPKG_IO_SERIAL_TERMIOS {
    user_value 1;
};
```

```
# Enable TERMIOS on serial 0
cdl_component CYGPKG_IO_SERIAL_TERMIOS_TERMIOS0 {
    user_value 1;
};

# Switch standard I/O to TERMIOS device 0
cdl_option CYGDAT_LIBC_STDIO_DEFAULT_CONSOLE {
    user_value "\"/dev/termios0\"";
};
```

Substitute a different serial device number for 0 in the above where necessary.

File System Dependencies

Much of OpenSSL can function without access to a filesystem. However, there are parts that expect to load or store data to or from files. If your application already uses a filesystem for other purposes (for example JFFS2 or YAFFS), then is should be easy to store OpenSSL's files there.

Where no external file store is available, the RAM filesystem can be used for temporary storage. The RAM filesystem can be mounted on system startup, and can be populated with files from data stored in memory. The following code shows how this might be done:

```
#define SERVER_CERT "/ram/server.pem"
static const char server_pem[] {
    ….
};

void init_ramfs( void )
{
    int fd;
    int err;
    size_t done;

    // Mount the RAM filesystem
    err = mount( "", "/ram", "ramfs" );
    if( err != 0 )
        diag_printf("RAMFS mount failed\n");

    // Write server.pem to RAM filesystem
    fd = open( SERVER_CERT, O_WRONLY|O_CREAT );
    done = write( fd, server_pem, sizeof(server_pem) );
    if( done != sizeof(server_pem) )
        diag_printf("server_pem write failed\n");
    close(fd);
}
```

Constant files can also be stored in the ROM filesystem, using **mkromfs** to create a file tree that can then be mounted and read.

Configuring OpenSSL

In addition to configuring eCos for OpenSSL, OpenSSL also contains a number of configuration points. The OpenSSL sources have been separated into a number of components, mainly corresponding to specific cryptographic algorithms and other components. Each of these is controlled by its own CDL option. By default only a subset of components are enabled. Other components may be enabled individually and alternatively all components may be enabled by setting `CYGPKG_OPENSSL_ALL`.

Various components of OpenSSL depend on different sets of operating system functionality, such as networking or file I/O. OpenSSL has internal configuration options to control the inclusion of different functional elements. In general we try to use those to control the build process. We have also encoded some dependencies in the CDL, both internal and external, to control the inclusion of entire components. However, the OpenSSL code is not normally compiled in systems with missing functionality, so even if the CDL and OpenSSL dependencies are correct, it is still possible for builds to fail with compile or link errors.

openssl Command Tool

The OpenSSL package contains a command line tool, **openssl**, that can be used to test the openssl package and to generate keys and certificates. This command can either execute a single command at a time, or run in interactive mode where successive commands are issued to a prompt. Under eCos it only runs in interactive mode, taking commands and issuing responses to the serial console.

The **openssl** tool will only build and run in the full configuration and additionally needs the RAM filesystem. If passwords are to be supplied, the serial line support described earlier should be enabled.

The command executable is created and saved in the `INSTALL_DIR/bin` directory. Both the original ELF file and an SREC file, `openssl.srec` are saved here. To run the command transfer the SREC file to your TFTP server and download and run it under RedBoot. You should see something similar to the following:

```
RedBoot> load openssl.srec
Using default protocol (TFTP)
Entry point: 0x20040040, address range: 0x20040000-0x201e76ec
RedBoot> go
[cyg_net_init] Init: mbinit(0x00000000)
[cyg_net_init] Init: cyg_net_init_devs(0x00000000)
Init device 'dm9000_eth0'
[cyg_net_init] Init: loopattach(0x00000000)
[cyg_net_init] Init: ifinit(0x00000000)
[cyg_net_init] Init: domaininit(0x00000000)
[cyg_net_init] Init: cyg_net_add_domain(0x201e51b0)
New domain internet at 0x00000000
[cyg_net_init] Init: cyg_net_add_domain(0x201e2c4c)
New domain route at 0x00000000
[cyg_net_init] Init: call_route_init(0x00000000)
[cyg_net_init] Done
mount /ram
set current directory to /ram
load openssl.cnf into /ram/openssl.cnf
initialise network interfaces
BOOTP[eth0] op: REPLY
       htype: Ethernet
        hlen: 6
        hops: 0
         xid: 0x0
        secs: 0
       flags: 0x0
       hw_addr: 00:03:47:df:32:a8
     client IP: 192.168.7.20
         my IP: 192.168.7.20
     server IP: 192.168.7.22
    gateway IP: 192.168.0.1
  options:
        subnet mask: 255.255.0.0
       IP broadcast: 192.168.255.255
            gateway: 192.168.0.1
[eth_drv_ioctl] Warning: Driver can't set multi-cast mode
[eth_drv_ioctl] Warning: Driver can't set multi-cast mode
[eth_drv_ioctl] Warning: Driver can't set multi-cast mode
Start OpenSSL
OpenSSL> version
OpenSSL 1.0.0c 2 Dec 2010
OpenSSL>
```

The RAM filesystem is mounted and `/ram` is set as the current directory. It is therefore possible to test the generation of keys and certificates into files:

```
OpenSSL> req -x509 -nodes -days 36500 \
            -subj "/C=GB/ST=England/L=Cambridge/O=eCosCentric/CN=ecoscentric.com" \
            -newkey rsa:1024 -keyout mycert.pem -out mycert.pem
Generating a 1024 bit RSA private key
```

```
…………........++++++
……..++++++
writing new private key to 'mycert.pem'
-----
OpenSSL>
```

If you want to enter passwords without reflection, you need to enable the TERMIOS support described above.

The eCos hosted **openssl** command serves as a test for OpenSSL functionality, and is a good check that the library is complete. However, it is of little practical use and has some limitations. While is possible to generate key files and certificates, it is not then easy to get then off the board for future use, unless they are stored to an external medium such as an SD card. It is recommended, instead, that a host based version of **openssl** be used to do this. Files may then be imported via removable media, or written to the RAM filesystem as described above. Another limitation is that if you run the **s_server** command, you cannot terminate it. Under Unix/Linux this command relies on catching the signal generated by a Ctrl-C to terminate; there is no support for this under eCos and the only way to terminate this command is to reboot and reload **openssl**.

Thread Safety

The OpenSSL library does not directly contain support for thread safe code. Instead it relies on application code to register some callbacks to perform the locking required. Under eCos there are two ways of doing this: through the POSIX compatibility package and directly using eCos APIs.

POSIX locking support is already available in OpenSSL. Example code is available in `src/crypto/threads/th-lock.c`, and similar code is tested in the `mttest` test program. However, this code will only work in threads that have been created using `pthread_create()`, or in the `main()` application thread.

eCos locking support uses lower level primitives and can be used from any kind of thread. In order to provide locking, an application must register two callbacks with the library, and set up an array of locks that the library will request it to lock and unlock. The following code does this:

```
// Forward definitions for callback functions.
void ecos_locking_callback(int mode, int type, char *file, int line);
unsigned long ecos_thread_id_callback(void);

// Pointer to array of locks.
static cyg_mutex_t *lock_cs;

// This function allocates and initializes the lock array
// and registers the callbacks. This should be called
// after the OpenSSL library has been initialized and
// before any new threads are created.
void thread_setup(void)
{
    int i;

    // Allocate lock array according to OpenSSL's requirements
    lock_cs=OPENSSL_malloc(CRYPTO_num_locks() * sizeof(cyg_mutex_t));

    // Initialize the locks
    for (i=0; i<CRYPTO_num_locks(); i++)
    {
        cyg_mutex_init(&(lock_cs[i]));
    }

    // Register callbacks
    CRYPTO_set_id_callback((unsigned long (*)())ecos_thread_id_callback);
    CRYPTO_set_locking_callback((void (*)())ecos_locking_callback);
}

// This function deallocates the lock array and deregisters the
// callbacks. It should be called after all threads have
// terminated.
```

```
void thread_cleanup(void)
{
    int i;

    // Deregister locking callback. No real need to
    // deregister id callback.
    CRYPTO_set_locking_callback(NULL);

    // Destroy the locks
    for (i=0; i<CRYPTO_num_locks(); i++)
    {
        cyg_mutex_destroy(&(lock_cs[i]));
    }

    // Release the lock array.
    OPENSSL_free(lock_cs);
}

// Locking callback. The type, file and line arguments are
// ignored. The file and line may be used to identify the site of the
// call in the OpenSSL library for diagnostic purposes if required.
void ecos_locking_callback(int mode, int type, char *file, int line)
{
    if (mode & CRYPTO_LOCK)
    {
        cyg_mutex_lock(&(lock_cs[type]));
    }
    else
    {
        cyg_mutex_unlock(&(lock_cs[type]));
    }
}

// Thread id callback.
unsigned long ecos_thread_id_callback(void)
{
    return (unsigned long)cyg_thread_get_id(cyg_thread_self());
}
```

Example code similar to this can be found in the `mttest_ecos` test program.

eCos Customization

The eCos port of OpenSSL contains a number of customizations to adapt OpenSSL to the eCos environment.

Random Number Support

To function correctly, OpenSSL requires a source of cryptographically strong random numbers. These are usually sourced either from operating system level entropy gathering or from a hardware random number generator. At present eCos does not have any entropy gathering mechanism so the only viable source is a hardware RNG. Without entropy gathering or hardware RNG use, some forms of encrypted data may be more vulnerable to attack. Contact eCosCentric if a solution is required for this.

OpenSSL gathers random numbers by calling `RAND_poll()` when necessary. This function is responsible for calling `RAND_add()` to mix new random data into OpenSSL's PRNG state. Application code can also call `RAND_add()` directly to add entropy from any source.

The source file `src/ecos/rand_ecos.c` contains an implementation of `RAND_poll()` that adds data from a static table whenever called. This is clearly not cryptographically strong, since the same random data will be added each time an application starts. This implementation is adequate for testing the library only and should not be used for real applications. The eCos port of OpenSSL does not automatically use a hardware RNG if present, and so application code is responsible for calling `RAND_add()` to incorporate random entropy from a hardware RNG into OpenSSL's PRNG.

BIO_diag

OpenSSL implements a general purpose data source/sink/filter object called a BIO. These may be attached to various sources and sinks such as C library FILEs, file descriptors and sockets. Many functions that need to output messages take a pointer to a BIO as an argument, which is typically attached to stdout or stderr. In certain eCos configurations these streams are not present, but we still want to use these functions and supply a BIO for output.

eCos implements a new BIO type, BIO_diag which outputs any data on the eCos diagnostic channel. It can be created using the new function BIO *BIO_new_diag(void), and can subsequently be used in place of any other output-only BIO. It may be freed in the usual way with BIO_free().

Tests

OpenSSL contains a number of test programs that validate the correctness of various cryptographic algorithms and test the functioning of the OpenSSL library. Most of these tests have been ported directly to eCos with only minor changes to allow them to function in the eCos test environment. Since all these tests expect to use C library standard I/O (STDIO) for output, they will only be built if that package is configured in.

A small number of encryption algorithm tests have been further adapted to function without needing STDIO. This has mainly involved replacing (f)printf with diag_printf, but also includes spawning a thread to run the test with sufficient stack. These tests are all named the same as the base OpenSSL test from which they were derived with the addition of _ecos to the file name.

The test program ssltest1 is the only purpose written SSL test program. It is a simple client that attempts to contact an SSL server and send it some data. The server it contacts is defined by the configuration options CYGDAT_OPENSSL_TESTS_SERVER_IP and CYGDAT_OPENSSL_TESTS_SERVER_PORT which are used to set IP address and port number of the server to contact. A suitable server can be run using the **s_server** command of the **openssl** tool on the host with the configured IP address. The shell script in misc/runtest within the eCos OpenSSL package in the package repository should do this correctly for any Linux host.

The test program openssl1 is a version of the **openssl** command line tool that runs a sequence of predefined commands to test the library as a whole. One of the commands that this test runs is a timing test against the same server that ssltest1 uses.

Limitations

OpenSSL includes files crypto/asn1/a_utctm.c and src/crypto/asn1/a_time.c which have hard-coded year limits of 2050. Use beyond that year is at present unsupported, although it is expected that upstream OpenSSL will resolve this at some point.

Chapter 179. OpenSSL Manual

openssl Command Line Tool

Name

openssl — OpenSSL command line tool

Synopsis

```
opensslcommand [ command_opts ] [ command_args ]
```

```
openssl [ list-standard-commands | list-message-digest-commands |
    list-cipher-commands | list-cipher-algorithms | list-message-digest-algorithms |
    list-public-key-algorithms]
```

```
opensslno-XXX [ arbitrary options ]
```

DESCRIPTION

OpenSSL is a cryptography toolkit implementing the Secure Sockets Layer (SSL v2/v3) and Transport Layer Security (TLS v1) network protocols and related cryptography standards required by them.

The **openssl** program is a command line tool for using the various cryptography functions of OpenSSL's **crypto** library from the shell. It can be used for

```
o  Creation and management of private keys, public keys and parameters
o  Public key cryptographic operations
o  Creation of X.509 certificates, CSRs and CRLs
o  Calculation of Message Digests
o  Encryption and Decryption with Ciphers
o  SSL/TLS Client and Server Tests
o  Handling of S/MIME signed or encrypted mail
o  Time Stamp requests, generation and verification
```

COMMAND SUMMARY

The **openssl** program provides a rich variety of commands (*command* in the SYNOPSIS above), each of which often has a wealth of options and arguments (*command_opts* and *command_args* in the SYNOPSIS).

The pseudo-commands **list-standard-commands**, **list-message-digest-commands**, and **list-cipher-commands** output a list (one entry per line) of the names of all standard commands, message digest commands, or cipher commands, respectively, that are available in the present **openssl** utility.

The pseudo-commands **list-cipher-algorithms** and **list-message-digest-algorithms** list all cipher and message digest names, one entry per line. Aliases are listed as:

```
from => to
```

The pseudo-command **list-public-key-algorithms** lists all supported public key algorithms.

The pseudo-command **no-***XXX* tests whether a command of the specified name is available. If no command named *XXX* exists, it returns 0 (success) and prints **no-***XXX*; otherwise it returns 1 and prints *XXX*. In both cases, the output goes to **stdout** and nothing is printed to **stderr**. Additional command line arguments are always ignored. Since for each cipher there is a command of the same name, this provides an easy way for shell scripts to test for the availability of ciphers in the **openssl** program. (**no-***XXX* is not able to detect pseudo-commands such as **quit**, **list-**…**-commands** or **no-***XXX* itself.)

STANDARD COMMANDS

asn1parse

Parse an ASN.1 sequence.

ca

Certificate Authority (CA) Management.

ciphers

Cipher Suite Description Determination.

cms

CMS (Cryptographic Message Syntax) utility

crl

Certificate Revocation List (CRL) Management.

crl2pkcs7

CRL to PKCS#7 Conversion.

dgst

Message Digest Calculation.

dh

Diffie-Hellman Parameter Management. Obsoleted by **dhparam**.

dhparam

Generation and Management of Diffie-Hellman Parameters. Superseded by **genpkey** and **pkeyparam**

dsa

DSA Data Management.

dsaparam

DSA Parameter Generation and Management. Superseded by **genpkey** and **pkeyparam**

ec

EC (Elliptic curve) key processing

ecparam

EC parameter manipulation and generation

enc

Encoding with Ciphers.

engine

Engine (loadble module) information and manipulation.

errstr

Error Number to Error String Conversion.

gendh

Generation of Diffie-Hellman Parameters. Obsoleted by **dhparam**.

gendsa

Generation of DSA Private Key from Parameters. Superseded by **genpkey** and **pkey**

genpkey

Generation of Private Key or Parameters.

genrsa

Generation of RSA Private Key. Superceded by **genpkey**.

nseq

Create or examine a netscape certificate sequence

ocsp

Online Certificate Status Protocol utility.

passwd

Generation of hashed passwords.

pkcs12

PKCS#12 Data Management.

pkcs7

PKCS#7 Data Management.

pkey

Public and private key management.

pkeyparam

Public key algorithm parameter management.

pkeyutl

Public key algorithm cryptographic operation utility.

rand

Generate pseudo-random bytes.

req

PKCS#10 X.509 Certificate Signing Request (CSR) Management.

rsa

RSA key management.

rsautl

RSA utility for signing, verification, encryption, and decryption. Superseded by **pkeyutl**

s_client

This implements a generic SSL/TLS client which can establish a transparent connection to a remote server speaking SSL/TLS. It's intended for testing purposes only and provides only rudimentary interface functionality but internally uses mostly all functionality of the OpenSSL **ssl** library.

s_server

This implements a generic SSL/TLS server which accepts connections from remote clients speaking SSL/TLS. It's intended for testing purposes only and provides only rudimentary interface functionality but internally uses mostly all functionality of the OpenSSL **ssl** library. It provides both an own command line oriented protocol for testing SSL functions and a simple HTTP response facility to emulate an SSL/TLS-aware webserver.

s_time

SSL Connection Timer.

sess_id

SSL Session Data Management.

smime

S/MIME mail processing.

speed

Algorithm Speed Measurement.

spkac

SPKAC printing and generating utility

ts

Time Stamping Authority tool (client/server)

verify

X.509 Certificate Verification.

version

OpenSSL Version Information.

x509

X.509 Certificate Data Management.

MESSAGE DIGEST COMMANDS

md2

MD2 Digest

md5

MD5 Digest

mdc2

MDC2 Digest

rmd160

RMD-160 Digest

sha

SHA Digest

sha1

SHA-1 Digest

sha224

SHA-224 Digest

sha256

SHA-256 Digest

sha384

SHA-384 Digest

sha512

SHA-512 Digest

ENCODING AND CIPHER COMMANDS

base64

Base64 Encoding

bf bf-cbc bf-cfb bf-ecb bf-ofb

Blowfish Cipher

cast cast-cbc

CAST Cipher

cast5-cbc cast5-cfb cast5-ecb cast5-ofb

CAST5 Cipher

des des-cbc des-cfb des-ecb des-ede des-ede-cbc des-ede-cfb des-ede-ofb des-ofb

DES Cipher

des3 desx des-ede3 des-ede3-cbc des-ede3-cfb des-ede3-ofb

Triple-DES Cipher

idea idea-cbc idea-cfb idea-ecb idea-ofb

IDEA Cipher

rc2 rc2-cbc rc2-cfb rc2-ecb rc2-ofb

RC2 Cipher

rc4

RC4 Cipher

rc5 rc5-cbc rc5-cfb rc5-ecb rc5-ofb

RC5 Cipher

PASS PHRASE ARGUMENTS

Several commands accept password arguments, typically using **-passin** and **-passout** for input and output passwords respectively. These allow the password to be obtained from a variety of sources. Both of these options take a single argument whose format is described below. If no password argument is given and a password is required then the user is prompted to enter one: this will typically be read from the current terminal with echoing turned off.

pass:password

the actual password is **password**. Since the password is visible to utilities (like 'ps' under Unix) this form should only be used where security is not important.

env:var

obtain the password from the environment variable **var**. Since the environment of other processes is visible on certain platforms (e.g. ps under certain Unix OSes) this option should be used with caution.

file:pathname

the first line of **pathname** is the password. If the same **pathname** argument is supplied to **-passin** and **-passout** arguments then the first line will be used for the input password and the next line for the output password. **pathname** need not refer to a regular file: it could for example refer to a device or named pipe.

fd:number

read the password from the file descriptor **number**. This can be used to send the data via a pipe for example.

stdin

read the password from standard input.

SEE ALSO

asn1parse(1), ca(1), config(5), crl(1), crl2pkcs7(1), dgst(1), dhparam(1), dsa(1), dsaparam(1), enc(1), gendsa(1), genpkey(1), genrsa(1), nseq(1), openssl(1), passwd(1), pkcs12(1), pkcs7(1), pkcs8(1), rand(1), req(1), rsa(1), rsautl(1), s_client(1), s_server(1), s_time(1), smime(1), spkac(1), verify(1), version(1), x509(1), crypto(3), ssl(3), x509v3_config(5)

HISTORY

The openssl(1) document appeared in OpenSSL 0.9.2. The **list-*XXX*-commands** pseudo-commands were added in OpenSSL 0.9.3; The **list-*XXX*-algorithms** pseudo-commands were added in OpenSSL 1.0.0; the **no-*XXX*** pseudo-commands were added in OpenSSL 0.9.5a. For notes on the availability of other commands, see their individual manual pages.

Name

asn1parse — ASN.1 parsing tool

Synopsis

```
opensslasn1parse
[-inform PEM|DER]
[-in filename]
[-out filename]
[-noout]
[-offset number]
[-length number]
[-i]
[-oid filename]
[-dump]
[-dlimit num]
[-strparse offset]
[-genstr string]
[-genconf file]
```

DESCRIPTION

The **asn1parse** command is a diagnostic utility that can parse ASN.1 structures. It can also be used to extract data from ASN.1 formatted data.

OPTIONS

-inform DER|PEM

the input format. **DER** is binary format and **PEM** (the default) is base64 encoded.

-in filename

the input file, default is standard input

-out filename

output file to place the DER encoded data into. If this option is not present then no data will be output. This is most useful when combined with the **-strparse** option.

-noout

don't output the parsed version of the input file.

-offset number

starting offset to begin parsing, default is start of file.

-length number

number of bytes to parse, default is until end of file.

-i

indents the output according to the "depth" of the structures.

-oid filename

a file containing additional OBJECT IDENTIFIERs (OIDs). The format of this file is described in the NOTES section below.

-dump

dump unknown data in hex format.

-dlimit num

like **-dump**, but only the first **num** bytes are output.

-strparse offset

parse the contents octets of the ASN.1 object starting at **offset**. This option can be used multiple times to "drill down" into a nested structure.

-genstr string, **-genconf file**

generate encoded data based on **string**, **file** or both using ASN1_generate_nconf(3) format. If **file** only is present then the string is obtained from the default section using the name **asn1**. The encoded data is passed through the ASN1 parser and printed out as though it came from a file, the contents can thus be examined and written to a file using the **out** option.

OUTPUT

The output will typically contain lines like this:

```
0:d=0  hl=4 l= 681 cons: SEQUENCE
```

…

```
229:d=3  hl=3 l= 141 prim: BIT STRING
373:d=2  hl=3 l= 162 cons: cont [ 3 ]
376:d=3  hl=3 l= 159 cons: SEQUENCE
379:d=4  hl=2 l=  29 cons: SEQUENCE
381:d=5  hl=2 l=   3 prim: OBJECT            :X509v3 Subject Key Identifier
386:d=5  hl=2 l=  22 prim: OCTET STRING
410:d=4  hl=2 l= 112 cons: SEQUENCE
412:d=5  hl=2 l=   3 prim: OBJECT            :X509v3 Authority Key Identifier
417:d=5  hl=2 l= 105 prim: OCTET STRING
524:d=4  hl=2 l=  12 cons: SEQUENCE
```

…..

This example is part of a self signed certificate. Each line starts with the offset in decimal. **d=XX** specifies the current depth. The depth is increased within the scope of any SET or SEQUENCE. **hl=XX** gives the header length (tag and length octets) of the current type. **l=XX** gives the length of the contents octets.

The **-i** option can be used to make the output more readable.

Some knowledge of the ASN.1 structure is needed to interpret the output.

In this example the BIT STRING at offset 229 is the certificate public key. The contents octets of this will contain the public key information. This can be examined using the option **-strparse 229** to yield:

```
  0:d=0  hl=3 l= 137 cons: SEQUENCE
  3:d=1  hl=3 l= 129 prim: INTEGER         :E5D21E1F5C8D208EA7A2166C7FAF9F6BDF2059669C60876DDB70840F1A5AAFA59699FE471F379F1D \
                                            D6A487E7D5409AB6A88D4A9746E24B91D8CF55DB3521015460C8EDE44EE8A4189F7A7BE77D6CD3A9 \
                                            AF2696F486855CF58BF0EDF2B4068058C7A947F52548DDF7E15E96B385F86422BEA9064A3EE9E115 \
                                            8A56E4A6F47E5897
135:d=1  hl=2 l=   3 prim: INTEGER         :010001
```

NOTES

If an OID is not part of OpenSSL's internal table it will be represented in numerical form (for example 1.2.3.4). The file passed to the **-oid** option allows additional OIDs to be included. Each line consists of three columns, the first column is the OID in numerical format and should be followed by white space. The second column is the "short name" which is a single word followed by white space. The final column is the rest of the line and is the "long name". **asn1parse** displays the long name. Example:

```
1.2.3.4 shortName A long name
```

EXAMPLES

Parse a file:

```
openssl asn1parse -in file.pem
```

Parse a DER file:

```
openssl asn1parse -inform DER -in file.der
```

Generate a simple UTF8String:

```
openssl asn1parse -genstr 'UTF8:Hello World'
```

Generate and write out a UTF8String, don't print parsed output:

```
openssl asn1parse -genstr 'UTF8:Hello World' -noout -out utf8.der
```

Generate using a config file:

```
openssl asn1parse -genconf asn1.cnf -noout -out asn1.der
```

Example config file:

```
asn1=SEQUENCE:seq_sect

[seq_sect]

field1=BOOL:TRUE
field2=EXP:0, UTF8:some random string
```

BUGS

There should be options to change the format of output lines. The output of some ASN.1 types is not well handled (if at all).

SEE ALSO

ASN1_generate_nconf(3)

Name

ca — sample minimal CA application

Synopsis

```
opensslca
[-verbose]
[-config filename]
[-name section]
[-gencrl]
[-revoke file]
[-status serial]
[-updatedb]
[-crl_reason reason]
[-crl_hold instruction]
[-crl_compromise time]
[-crl_CA_compromise time]
[-crldays days]
[-crlhours hours]
[-crlexts section]
[-startdate date]
[-enddate date]
[-days arg]
[-md arg]
[-policy arg]
[-keyfile arg]
[-keyform PEM|DER]
[-key arg]
[-passin arg]
[-cert file]
[-selfsign]
[-in file]
[-out file]
[-notext]
[-outdir dir]
[-infiles]
[-spkac file]
[-ss_cert file]
[-preserveDN]
[-noemailDN]
[-batch]
[-msie_hack]
[-extensions section]
[-extfile section]
[-engine id]
[-subj arg]
[-utf8]
[-multivalue-rdn]
```

DESCRIPTION

The **ca** command is a minimal CA application. It can be used to sign certificate requests in a variety of forms and generate CRLs it also maintains a text database of issued certificates and their status.

The options descriptions will be divided into each purpose.

CA OPTIONS

-config filename

specifies the configuration file to use.

-name section

specifies the configuration file section to use (overrides **default_ca** in the **ca** section).

-in filename

an input filename containing a single certificate request to be signed by the CA.

-ss_cert filename

a single self signed certificate to be signed by the CA.

-spkac filename

a file containing a single Netscape signed public key and challenge and additional field values to be signed by the CA. See the **SPKAC FORMAT** section for information on the required input and output format.

-infiles

if present this should be the last option, all subsequent arguments are assumed to the the names of files containing certificate requests.

-out filename

the output file to output certificates to. The default is standard output. The certificate details will also be printed out to this file in PEM format (except that **-spkac** outputs DER format).

-outdir directory

the directory to output certificates to. The certificate will be written to a filename consisting of the serial number in hex with ".pem" appended.

-cert

the CA certificate file.

-keyfile filename

the private key to sign requests with.

-keyform PEM|DER

the format of the data in the private key file. The default is PEM.

-key password

the password used to encrypt the private key. Since on some systems the command line arguments are visible (e.g. Unix with the 'ps' utility) this option should be used with caution.

-selfsign

indicates the issued certificates are to be signed with the key the certificate requests were signed with (given with **-keyfile**). Cerificate requests signed with a different key are ignored. If **-spkac**, **-ss_cert** or **-gencrl** are given, **-selfsign** is ignored.

A consequence of using **-selfsign** is that the self-signed certificate appears among the entries in the certificate database (see the configuration option **database**), and uses the same serial number counter as all other certificates sign with the self-signed certificate.

-passin arg

the key password source. For more information about the format of **arg** see the **PASS PHRASE ARGUMENTS** section in openssl(1).

-verbose

this prints extra details about the operations being performed.

-notext

don't output the text form of a certificate to the output file.

-startdate date

this allows the start date to be explicitly set. The format of the date is YYMMDDHHMMSSZ (the same as an ASN1 UTCTime structure).

-enddate date

this allows the expiry date to be explicitly set. The format of the date is YYMMDDHHMMSSZ (the same as an ASN1 UTCTime structure).

-days arg

the number of days to certify the certificate for.

-md alg

the message digest to use. Possible values include md5, sha1 and mdc2. This option also applies to CRLs.

-policy arg

this option defines the CA "policy" to use. This is a section in the configuration file which decides which fields should be mandatory or match the CA certificate. Check out the **POLICY FORMAT** section for more information.

-msie_hack

this is a legacy option to make **ca** work with very old versions of the IE certificate enrollment control "certenr3". It used UniversalStrings for almost everything. Since the old control has various security bugs its use is strongly discouraged. The newer control "Xenroll" does not need this option.

-preserveDN

Normally the DN order of a certificate is the same as the order of the fields in the relevant policy section. When this option is set the order is the same as the request. This is largely for compatibility with the older IE enrollment control which would only accept certificates if their DNs match the order of the request. This is not needed for Xenroll.

-noemailDN

The DN of a certificate can contain the EMAIL field if present in the request DN, however it is good policy just having the e-mail set into the altName extension of the certificate. When this option is set the EMAIL field is removed from the certificate' subject and set only in the, eventually present, extensions. The **email_in_dn** keyword can be used in the configuration file to enable this behaviour.

-batch

this sets the batch mode. In this mode no questions will be asked and all certificates will be certified automatically.

-extensions section

the section of the configuration file containing certificate extensions to be added when a certificate is issued (defaults to **x509_extensions** unless the **-extfile** option is used). If no extension section is present then, a V1 certificate is created. If the extension section is present (even if it is empty), then a V3 certificate is created. See the:w x509v3_config(5) manual page for details of the extension section format.

-extfile file

an additional configuration file to read certificate extensions from (using the default section unless the **-extensions** option is also used).

-engine id

specifying an engine (by its unique **id** string) will cause **ca** to attempt to obtain a functional reference to the specified engine, thus initialising it if needed. The engine will then be set as the default for all available algorithms.

-subj arg

supersedes subject name given in the request. The arg must be formatted as */type0=value0/type1=value1/type2=...*, characters may be escaped by \ (backslash), no spaces are skipped.

-utf8

this option causes field values to be interpreted as UTF8 strings, by default they are interpreted as ASCII. This means that the field values, whether prompted from a terminal or obtained from a configuration file, must be valid UTF8 strings.

-multivalue-rdn

this option causes the -subj argument to be interpretedt with full support for multivalued RDNs. Example:

/DC=org/DC=OpenSSL/DC=users/UID=123456+CN=John Doe

If -multi-rdn is not used then the UID value is *123456+CN=John Doe*.

CRL OPTIONS

-gencrl

this option generates a CRL based on information in the index file.

-crldays num

the number of days before the next CRL is due. That is the days from now to place in the CRL nextUpdate field.

-crlhours num

the number of hours before the next CRL is due.

-revoke filename

a filename containing a certificate to revoke.

-status serial

displays the revocation status of the certificate with the specified serial number and exits.

-updatedb

Updates the database index to purge expired certificates.

-crl_reason reason

revocation reason, where **reason** is one of: **unspecified**, **keyCompromise**, **CACompromise**, **affiliationChanged**, **superseded**, **cessationOfOperation**, **certificateHold** or **removeFromCRL**. The matching of **reason** is case insensitive. Setting any revocation reason will make the CRL v2.

In practive **removeFromCRL** is not particularly useful because it is only used in delta CRLs which are not currently implemented.

-crl_hold instruction

This sets the CRL revocation reason code to **certificateHold** and the hold instruction to **instruction** which must be an OID. Although any OID can be used only **holdInstructionNone** (the use of which is discouraged by RFC2459) **holdInstructionCallIssuer** or **holdInstructionReject** will normally be used.

-crl_compromise time

This sets the revocation reason to **keyCompromise** and the compromise time to **time**. **time** should be in GeneralizedTime format that is **YYYYMMDDHHMMSSZ**.

-crl_CA_compromise time

This is the same as **crl_compromise** except the revocation reason is set to **CACompromise**.

-crlexts section

the section of the configuration file containing CRL extensions to include. If no CRL extension section is present then a V1 CRL is created, if the CRL extension section is present (even if it is empty) then a V2 CRL is created. The CRL extensions specified are CRL extensions and **not** CRL entry extensions. It should be noted that some software (for example Netscape) can't handle V2 CRLs. See x509v3_config(5) manual page for details of the extension section format.

CONFIGURATION FILE OPTIONS

The section of the configuration file containing options for **ca** is found as follows: If the **-name** command line option is used, then it names the section to be used. Otherwise the section to be used must be named in the **default_ca** option of the **ca** section of the configuration file (or in the default section of the configuration file). Besides **default_ca**, the following options are read directly from the **ca** section: RANDFILE preserve msie_hack With the exception of **RANDFILE**, this is probably a bug and may change in future releases.

Many of the configuration file options are identical to command line options. Where the option is present in the configuration file and the command line the command line value is used. Where an option is described as mandatory then it must be present in the configuration file or the command line equivalent (if any) used.

oid_file

This specifies a file containing additional **OBJECT IDENTIFIERS**. Each line of the file should consist of the numerical form of the object identifier followed by white space then the short name followed by white space and finally the long name.

oid_section

This specifies a section in the configuration file containing extra object identifiers. Each line should consist of the short name of the object identifier followed by = and the numerical form. The short and long names are the same when this option is used.

new_certs_dir

the same as the **-outdir** command line option. It specifies the directory where new certificates will be placed. Mandatory.

certificate

the same as **-cert**. It gives the file containing the CA certificate. Mandatory.

private_key

same as the **-keyfile** option. The file containing the CA private key. Mandatory.

RANDFILE

a file used to read and write random number seed information, or an EGD socket (see RAND_egd(3)).

default_days

the same as the **-days** option. The number of days to certify a certificate for.

default_startdate

the same as the **-startdate** option. The start date to certify a certificate for. If not set the current time is used.

default_enddate

the same as the **-enddate** option. Either this option or **default_days** (or the command line equivalents) must be present.

default_crl_hours default_crl_days

the same as the **-crlhours** and the **-crldays** options. These will only be used if neither command line option is present. At least one of these must be present to generate a CRL.

default_md

the same as the **-md** option. The message digest to use. Mandatory.

database

the text database file to use. Mandatory. This file must be present though initially it will be empty.

unique_subject

if the value **yes** is given, the valid certificate entries in the database must have unique subjects. if the value **no** is given, several valid certificate entries may have the exact same subject. The default value is **yes**, to be compatible with older (pre 0.9.8) versions of OpenSSL. However, to make CA certificate roll-over easier, it's recommended to use the value **no**, especially if combined with the **-selfsign** command line option.

serial

a text file containing the next serial number to use in hex. Mandatory. This file must be present and contain a valid serial number.

crlnumber

a text file containing the next CRL number to use in hex. The crl number will be inserted in the CRLs only if this file exists. If this file is present, it must contain a valid CRL number.

x509_extensions

the same as **-extensions**.

crl_extensions

the same as **-crlexts**.

preserve

the same as **-preserveDN**

email_in_dn

the same as **-noemailDN**. If you want the EMAIL field to be removed from the DN of the certificate simply set this to 'no'. If not present the default is to allow for the EMAIL filed in the certificate's DN.

msie_hack

the same as **-msie_hack**

policy

the same as **-policy**. Mandatory. See the **POLICY FORMAT** section for more information.

name_opt, **cert_opt**

these options allow the format used to display the certificate details when asking the user to confirm signing. All the options supported by the **x509** utilities **-nameopt** and **-certopt** switches can be used here, except the **no_signame** and **no_sigdump** are permanently set and cannot be disabled (this is because the certificate signature cannot be displayed because the certificate has not been signed at this point).

For convenience the values **ca_default** are accepted by both to produce a reasonable output.

If neither option is present the format used in earlier versions of OpenSSL is used. Use of the old format is **strongly** discouraged because it only displays fields mentioned in the **policy** section, mishandles multicharacter string types and does not display extensions.

copy_extensions

determines how extensions in certificate requests should be handled. If set to **none** or this option is not present then extensions are ignored and not copied to the certificate. If set to **copy** then any extensions present in the request that are not already present are copied to the certificate. If set to **copyall** then all extensions in the request are copied to the certificate: if the extension is already present in the certificate it is deleted first. See the **WARNINGS** section before using this option.

The main use of this option is to allow a certificate request to supply values for certain extensions such as subjectAltName.

POLICY FORMAT

The policy section consists of a set of variables corresponding to certificate DN fields. If the value is "match" then the field value must match the same field in the CA certificate. If the value is "supplied" then it must be present. If the value is "optional" then it may be present. Any fields not mentioned in the policy section are silently deleted, unless the **-preserveDN** option is set but this can be regarded more of a quirk than intended behaviour.

SPKAC FORMAT

The input to the **-spkac** command line option is a Netscape signed public key and challenge. This will usually come from the **KEYGEN** tag in an HTML form to create a new private key. It is however possible to create SPKACs using the **spkac** utility.

The file should contain the variable SPKAC set to the value of the SPKAC and also the required DN components as name value pairs. If you need to include the same component twice then it can be preceded by a number and a '.'.

When processing SPKAC format, the output is DER if the **-out** flag is used, but PEM format if sending to stdout or the **-outdir** flag is used.

EXAMPLES

Note: these examples assume that the **ca** directory structure is already set up and the relevant files already exist. This usually involves creating a CA certificate and private key with **req**, a serial number file and an empty index file and placing them in the relevant directories.

To use the sample configuration file below the directories demoCA, demoCA/private and demoCA/newcerts would be created. The CA certificate would be copied to demoCA/cacert.pem and its private key to demoCA/private/cakey.pem. A file demoCA/serial would be created containing for example "01" and the empty index file demoCA/index.txt.

Sign a certificate request:

```
openssl ca -in req.pem -out newcert.pem
```

Sign a certificate request, using CA extensions:

```
openssl ca -in req.pem -extensions v3_ca -out newcert.pem
```

Generate a CRL

```
openssl ca -gencrl -out crl.pem
```

Sign several requests:

```
openssl ca -infiles req1.pem req2.pem req3.pem
```

Certify a Netscape SPKAC:

```
openssl ca -spkac spkac.txt
```

A sample SPKAC file (the SPKAC line has been truncated for clarity):

```
SPKAC=MIG0MGAwXDANBgkqhkiG9w0BAQEFAANLADBIAkEAn7PDhCeV/xIxUg8V70YRxK2A5
CN=Steve Test
emailAddress=steve@openssl.org
0.OU=OpenSSL Group
1.OU=Another Group
```

A sample configuration file with the relevant sections for **ca**:

```
[ ca ]
default_ca      = CA_default            # The default ca section

[ CA_default ]

dir            = ./demoCA              # top dir
database       = $dir/index.txt        # index file.
new_certs_dir  = $dir/newcerts         # new certs dir

certificate    = $dir/cacert.pem       # The CA cert
serial         = $dir/serial           # serial no file
private_key    = $dir/private/cakey.pem# CA private key
RANDFILE       = $dir/private/.rand    # random number file

default_days   = 365                   # how long to certify for
default_crl_days= 30                   # how long before next CRL
default_md     = md5                   # md to use

policy         = policy_any            # default policy
email_in_dn    = no                    # Don't add the email into cert DN
```

```
name_opt       = ca_default            # Subject name display option
cert_opt       = ca_default            # Certificate display option
copy_extensions = none                 # Don't copy extensions from request

[ policy_any ]
countryName            = supplied
stateOrProvinceName    = optional
organizationName       = optional
organizationalUnitName = optional
commonName             = supplied
emailAddress           = optional
```

FILES

Note: the location of all files can change either by compile time options, configuration file entries, environment variables or command line options. The values below reflect the default values.

```
/usr/local/ssl/lib/openssl.cnf - master configuration file
./demoCA                       - main CA directory
./demoCA/cacert.pem            - CA certificate
./demoCA/private/cakey.pem     - CA private key
./demoCA/serial                - CA serial number file
./demoCA/serial.old            - CA serial number backup file
./demoCA/index.txt             - CA text database file
./demoCA/index.txt.old         - CA text database backup file
./demoCA/certs                 - certificate output file
./demoCA/.rnd                  - CA random seed information
```

ENVIRONMENT VARIABLES

OPENSSL_CONF reflects the location of master configuration file it can be overridden by the **-config** command line option.

RESTRICTIONS

The text database index file is a critical part of the process and if corrupted it can be difficult to fix. It is theoretically possible to rebuild the index file from all the issued certificates and a current CRL: however there is no option to do this.

V2 CRL features like delta CRLs are not currently supported.

Although several requests can be input and handled at once it is only possible to include one SPKAC or self signed certificate.

BUGS

The use of an in memory text database can cause problems when large numbers of certificates are present because, as the name implies the database has to be kept in memory.

The **ca** command really needs rewriting or the required functionality exposed at either a command or interface level so a more friendly utility (perl script or GUI) can handle things properly. The scripts **CA.sh** and **CA.pl** help a little but not very much.

Any fields in a request that are not present in a policy are silently deleted. This does not happen if the **-preserveDN** option is used. To enforce the absence of the EMAIL field within the DN, as suggested by RFCs, regardless the contents of the request' subject the **-noemailDN** option can be used. The behaviour should be more friendly and configurable.

Cancelling some commands by refusing to certify a certificate can create an empty file.

WARNINGS

The **ca** command is quirky and at times downright unfriendly.

The **ca** utility was originally meant as an example of how to do things in a CA. It was not supposed to be used as a full blown CA itself: nevertheless some people are using it for this purpose.

The **ca** command is effectively a single user command: no locking is done on the various files and attempts to run more than one **ca** command on the same database can have unpredictable results.

The **copy_extensions** option should be used with caution. If care is not taken then it can be a security risk. For example if a certificate request contains a basicConstraints extension with CA:TRUE and the **copy_extensions** value is set to **copyall** and the user does not spot this when the certificate is displayed then this will hand the requestor a valid CA certificate.

This situation can be avoided by setting **copy_extensions** to **copy** and including basicConstraints with CA:FALSE in the configuration file. Then if the request contains a basicConstraints extension it will be ignored.

It is advisable to also include values for other extensions such as **keyUsage** to prevent a request supplying its own values.

Additional restrictions can be placed on the CA certificate itself. For example if the CA certificate has:

```
basicConstraints = CA:TRUE, pathlen:0
```

then even if a certificate is issued with CA:TRUE it will not be valid.

SEE ALSO

req(1), spkac(1), x509(1), config(5), x509v3_config(5)

Name

ciphers — SSL cipher display and cipher list tool.

Synopsis

```
opensslciphers
[-v]
[-V]
[-ssl2]
[-ssl3]
[-tls1]
[cipherlist]
```

DESCRIPTION

The **ciphers** command converts textual OpenSSL cipher lists into ordered SSL cipher preference lists. It can be used as a test tool to determine the appropriate cipherlist.

COMMAND OPTIONS

-v

Verbose option. List ciphers with a complete description of protocol version (SSLv2 or SSLv3; the latter includes TLS), key exchange, authentication, encryption and mac algorithms used along with any key size restrictions and whether the algorithm is classed as an "export" cipher. Note that without the **-v** option, ciphers may seem to appear twice in a cipher list; this is when similar ciphers are available for SSL v2 and for SSL v3/TLS v1.

-V

Like **-v**, but include cipher suite codes in output (hex format).

-ssl3, **-tls1**

This lists ciphers compatible with any of SSLv3, TLSv1, TLSv1.1 or TLSv1.2.

-ssl2

Only include SSLv2 ciphers.

-h, **-?**

Print a brief usage message.

cipherlist

A cipher list to convert to a cipher preference list. If it is not included then the default cipher list will be used. The format is described below.

CIPHER LIST FORMAT

The cipher list consists of one or more *cipher strings* separated by colons. Commas or spaces are also acceptable separators but colons are normally used.

The actual cipher string can take several different forms.

It can consist of a single cipher suite such as **RC4-SHA**.

It can represent a list of cipher suites containing a certain algorithm, or cipher suites of a certain type. For example **SHA1** represents all ciphers suites using the digest algorithm SHA1 and **SSLv3** represents all SSL v3 algorithms.

Lists of cipher suites can be combined in a single cipher string using the **+** character. This is used as a logical **and** operation. For example **SHA1+DES** represents all cipher suites containing the SHA1 **and** the DES algorithms.

Each cipher string can be optionally preceded by the characters **!**, **-** or **+**.

If **!** is used then the ciphers are permanently deleted from the list. The ciphers deleted can never reappear in the list even if they are explicitly stated.

If **-** is used then the ciphers are deleted from the list, but some or all of the ciphers can be added again by later options.

If **+** is used then the ciphers are moved to the end of the list. This option doesn't add any new ciphers it just moves matching existing ones.

If none of these characters is present then the string is just interpreted as a list of ciphers to be appended to the current preference list. If the list includes any ciphers already present they will be ignored: that is they will not moved to the end of the list.

Additionally the cipher string **@STRENGTH** can be used at any point to sort the current cipher list in order of encryption algorithm key length.

CIPHER STRINGS

The following is a list of all permitted cipher strings and their meanings.

DEFAULT

The default cipher list. This is determined at compile time and is normally **ALL:!EXPORT:!LOW:!aNULL:!eNULL:!SSLv2**. When used, this must be the first cipherstring specified.

COMPLEMENTOFDEFAULT

the ciphers included in **ALL**, but not enabled by default. Currently this is **ADH** and **AECDH**. Note that this rule does not cover **eNULL**, which is not included by **ALL** (use **COMPLEMENTOFALL** if necessary).

ALL

all cipher suites except the **eNULL** ciphers which must be explicitly enabled; as of OpenSSL, the **ALL** cipher suites are reasonably ordered by default

COMPLEMENTOFALL

the cipher suites not enabled by **ALL**, currently being **eNULL**.

HIGH

"high" encryption cipher suites. This currently means those with key lengths larger than 128 bits, and some cipher suites with 128-bit keys.

MEDIUM

"medium" encryption cipher suites, currently some of those using 128 bit encryption.

LOW

Low strength encryption cipher suites, currently those using 64 or 56 bit encryption algorithms but excluding export cipher suites. As of OpenSSL 1.0.1s, these are disabled in default builds.

EXP, **EXPORT**

Export strength encryption algorithms. Including 40 and 56 bits algorithms. As of OpenSSL 1.0.1s, these are disabled in default builds.

EXPORT40

40-bit export encryption algorithms As of OpenSSL 1.0.1s, these are disabled in default builds.

EXPORT56

56-bit export encryption algorithms. In OpenSSL 0.9.8c and later the set of 56 bit export ciphers is empty unless OpenSSL has been explicitly configured with support for experimental ciphers. As of OpenSSL 1.0.1s, these are disabled in default builds.

eNULL, **NULL**

The "NULL" ciphers that is those offering no encryption. Because these offer no encryption at all and are a security risk they are not enabled via either the **DEFAULT** or **ALL** cipher strings. Be careful when building cipherlists out of lower-level primitives such as **kRSA** or **aECDSA** as these do overlap with the **eNULL** ciphers. When in doubt, include **!eNULL** in your cipherlist.

aNULL

The cipher suites offering no authentication. This is currently the anonymous DH algorithms and anonymous ECDH algorithms. These cipher suites are vulnerable to a "man in the middle" attack and so their use is normally discouraged. These are excluded from the **DEFAULT** ciphers, but included in the **ALL** ciphers. Be careful when building cipherlists out of lower-level primitives such as **kDHE** or **AES** as these do overlap with the **aNULL** ciphers. When in doubt, include **!aNULL** in your cipherlist.

kRSA, **RSA**

cipher suites using RSA key exchange.

kDHr, **kDHd**, **kDH**

cipher suites using DH key agreement and DH certificates signed by CAs with RSA and DSS keys or either respectively. Not implemented.

kEDH

cipher suites using ephemeral DH key agreement, including anonymous cipher suites.

EDH

cipher suites using authenticated ephemeral DH key agreement.

ADH

anonymous DH cipher suites, note that this does not include anonymous Elliptic Curve DH (ECDH) cipher suites.

DH

cipher suites using DH, including anonymous DH, ephemeral DH and fixed DH.

kECDHr, **kECDHe**, **kECDH**

cipher suites using fixed ECDH key agreement signed by CAs with RSA and ECDSA keys or either respectively.

kEECDH

cipher suites using ephemeral ECDH key agreement, including anonymous cipher suites.

EECDH

cipher suites using authenticated ephemeral ECDH key agreement.

AECDH

anonymous Elliptic Curve Diffie Hellman cipher suites.

ECDH

cipher suites using ECDH key exchange, including anonymous, ephemeral and fixed ECDH.

aRSA

cipher suites using RSA authentication, i.e. the certificates carry RSA keys.

aDSS, **DSS**

cipher suites using DSS authentication, i.e. the certificates carry DSS keys.

aDH

cipher suites effectively using DH authentication, i.e. the certificates carry DH keys. Not implemented.

aECDH

cipher suites effectively using ECDH authentication, i.e. the certificates carry ECDH keys.

aECDSA, **ECDSA**

cipher suites using ECDSA authentication, i.e. the certificates carry ECDSA keys.

kFZA, **aFZA**, **eFZA**, **FZA**

ciphers suites using FORTEZZA key exchange, authentication, encryption or all FORTEZZA algorithms. Not implemented.

TLSv1.2, **TLSv1**, **SSLv3**, **SSLv2**

TLS v1.2, TLS v1.0, SSL v3.0 or SSL v2.0 cipher suites respectively. Note: there are no ciphersuites specific to TLS v1.1.

AES128, **AES256**, **AES**

cipher suites using 128 bit AES, 256 bit AES or either 128 or 256 bit AES.

AESGCM

AES in Galois Counter Mode (GCM): these ciphersuites are only supported in TLS v1.2.

CAMELLIA128, **CAMELLIA256**, **CAMELLIA**

cipher suites using 128 bit CAMELLIA, 256 bit CAMELLIA or either 128 or 256 bit CAMELLIA.

3DES

cipher suites using triple DES.

DES

cipher suites using DES (not triple DES).

RC4

cipher suites using RC4.

RC2

cipher suites using RC2.

IDEA

cipher suites using IDEA.

SEED

cipher suites using SEED.

MD5

cipher suites using MD5.

SHA1, **SHA**

cipher suites using SHA1.

SHA256, **SHA384**

ciphersuites using SHA256 or SHA384.

aGOST

cipher suites using GOST R 34.10 (either 2001 or 94) for authenticaction (needs an engine supporting GOST algorithms).

aGOST01

cipher suites using GOST R 34.10-2001 authentication.

aGOST94

cipher suites using GOST R 34.10-94 authentication (note that R 34.10-94 standard has been expired so use GOST R 34.10-2001)

kGOST

cipher suites, using VKO 34.10 key exchange, specified in the RFC 4357.

GOST94

cipher suites, using HMAC based on GOST R 34.11-94.

GOST89MAC

cipher suites using GOST 28147-89 MAC **instead of** HMAC.

PSK

cipher suites using pre-shared keys (PSK).

CIPHER SUITE NAMES

The following lists give the SSL or TLS cipher suites names from the relevant specification and their OpenSSL equivalents. It should be noted, that several cipher suite names do not include the authentication used, e.g. DES-CBC3-SHA. In these cases, RSA authentication is used.

SSL v3.0 cipher suites.

```
SSL_RSA_WITH_NULL_MD5                   NULL-MD5
SSL_RSA_WITH_NULL_SHA                   NULL-SHA
SSL_RSA_EXPORT_WITH_RC4_40_MD5          EXP-RC4-MD5
SSL_RSA_WITH_RC4_128_MD5                RC4-MD5
SSL_RSA_WITH_RC4_128_SHA                RC4-SHA
SSL_RSA_EXPORT_WITH_RC2_CBC_40_MD5      EXP-RC2-CBC-MD5
SSL_RSA_WITH_IDEA_CBC_SHA               IDEA-CBC-SHA
SSL_RSA_EXPORT_WITH_DES40_CBC_SHA       EXP-DES-CBC-SHA
SSL_RSA_WITH_DES_CBC_SHA                DES-CBC-SHA
SSL_RSA_WITH_3DES_EDE_CBC_SHA           DES-CBC3-SHA

SSL_DH_DSS_EXPORT_WITH_DES40_CBC_SHA    Not implemented.
SSL_DH_DSS_WITH_DES_CBC_SHA             Not implemented.
SSL_DH_DSS_WITH_3DES_EDE_CBC_SHA        Not implemented.
SSL_DH_RSA_EXPORT_WITH_DES40_CBC_SHA    Not implemented.
SSL_DH_RSA_WITH_DES_CBC_SHA             Not implemented.
SSL_DH_RSA_WITH_3DES_EDE_CBC_SHA        Not implemented.
SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA   EXP-EDH-DSS-DES-CBC-SHA
SSL_DHE_DSS_WITH_DES_CBC_SHA            EDH-DSS-CBC-SHA
SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA       EDH-DSS-DES-CBC3-SHA
SSL_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA   EXP-EDH-RSA-DES-CBC-SHA
SSL_DHE_RSA_WITH_DES_CBC_SHA            EDH-RSA-DES-CBC-SHA
SSL_DHE_RSA_WITH_3DES_EDE_CBC_SHA       EDH-RSA-DES-CBC3-SHA

SSL_DH_anon_EXPORT_WITH_RC4_40_MD5      EXP-ADH-RC4-MD5
SSL_DH_anon_WITH_RC4_128_MD5            ADH-RC4-MD5
SSL_DH_anon_EXPORT_WITH_DES40_CBC_SHA   EXP-ADH-DES-CBC-SHA
SSL_DH_anon_WITH_DES_CBC_SHA            ADH-DES-CBC-SHA
SSL_DH_anon_WITH_3DES_EDE_CBC_SHA       ADH-DES-CBC3-SHA

SSL_FORTEZZA_KEA_WITH_NULL_SHA          Not implemented.
SSL_FORTEZZA_KEA_WITH_FORTEZZA_CBC_SHA  Not implemented.
SSL_FORTEZZA_KEA_WITH_RC4_128_SHA       Not implemented.
```

TLS v1.0 cipher suites.

```
TLS_RSA_WITH_NULL_MD5                   NULL-MD5
TLS_RSA_WITH_NULL_SHA                   NULL-SHA
TLS_RSA_EXPORT_WITH_RC4_40_MD5          EXP-RC4-MD5
TLS_RSA_WITH_RC4_128_MD5                RC4-MD5
TLS_RSA_WITH_RC4_128_SHA                RC4-SHA
TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5      EXP-RC2-CBC-MD5
TLS_RSA_WITH_IDEA_CBC_SHA               IDEA-CBC-SHA
TLS_RSA_EXPORT_WITH_DES40_CBC_SHA       EXP-DES-CBC-SHA
TLS_RSA_WITH_DES_CBC_SHA                DES-CBC-SHA
TLS_RSA_WITH_3DES_EDE_CBC_SHA           DES-CBC3-SHA

TLS_DH_DSS_EXPORT_WITH_DES40_CBC_SHA    Not implemented.
TLS_DH_DSS_WITH_DES_CBC_SHA             Not implemented.
TLS_DH_DSS_WITH_3DES_EDE_CBC_SHA        Not implemented.
TLS_DH_RSA_EXPORT_WITH_DES40_CBC_SHA    Not implemented.
TLS_DH_RSA_WITH_DES_CBC_SHA             Not implemented.
TLS_DH_RSA_WITH_3DES_EDE_CBC_SHA        Not implemented.
TLS_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA   EXP-EDH-DSS-DES-CBC-SHA
TLS_DHE_DSS_WITH_DES_CBC_SHA            EDH-DSS-CBC-SHA
TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA       EDH-DSS-DES-CBC3-SHA
TLS_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA   EXP-EDH-RSA-DES-CBC-SHA
```

```
TLS_DHE_RSA_WITH_DES_CBC_SHA            EDH-RSA-DES-CBC-SHA
TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA       EDH-RSA-DES-CBC3-SHA

TLS_DH_anon_EXPORT_WITH_RC4_40_MD5      EXP-ADH-RC4-MD5
TLS_DH_anon_WITH_RC4_128_MD5            ADH-RC4-MD5
TLS_DH_anon_EXPORT_WITH_DES40_CBC_SHA   EXP-ADH-DES-CBC-SHA
TLS_DH_anon_WITH_DES_CBC_SHA            ADH-DES-CBC-SHA
TLS_DH_anon_WITH_3DES_EDE_CBC_SHA       ADH-DES-CBC3-SHA
```

AES ciphersuites from RFC3268, extending TLS v1.0

```
TLS_RSA_WITH_AES_128_CBC_SHA            AES128-SHA
TLS_RSA_WITH_AES_256_CBC_SHA            AES256-SHA

TLS_DH_DSS_WITH_AES_128_CBC_SHA         Not implemented.
TLS_DH_DSS_WITH_AES_256_CBC_SHA         Not implemented.
TLS_DH_RSA_WITH_AES_128_CBC_SHA         Not implemented.
TLS_DH_RSA_WITH_AES_256_CBC_SHA         Not implemented.

TLS_DHE_DSS_WITH_AES_128_CBC_SHA        DHE-DSS-AES128-SHA
TLS_DHE_DSS_WITH_AES_256_CBC_SHA        DHE-DSS-AES256-SHA
TLS_DHE_RSA_WITH_AES_128_CBC_SHA        DHE-RSA-AES128-SHA
TLS_DHE_RSA_WITH_AES_256_CBC_SHA        DHE-RSA-AES256-SHA

TLS_DH_anon_WITH_AES_128_CBC_SHA        ADH-AES128-SHA
TLS_DH_anon_WITH_AES_256_CBC_SHA        ADH-AES256-SHA
```

Camellia ciphersuites from RFC4132, extending TLS v1.0

```
TLS_RSA_WITH_CAMELLIA_128_CBC_SHA      CAMELLIA128-SHA
TLS_RSA_WITH_CAMELLIA_256_CBC_SHA      CAMELLIA256-SHA

TLS_DH_DSS_WITH_CAMELLIA_128_CBC_SHA   Not implemented.
TLS_DH_DSS_WITH_CAMELLIA_256_CBC_SHA   Not implemented.
TLS_DH_RSA_WITH_CAMELLIA_128_CBC_SHA   Not implemented.
TLS_DH_RSA_WITH_CAMELLIA_256_CBC_SHA   Not implemented.

TLS_DHE_DSS_WITH_CAMELLIA_128_CBC_SHA  DHE-DSS-CAMELLIA128-SHA
TLS_DHE_DSS_WITH_CAMELLIA_256_CBC_SHA  DHE-DSS-CAMELLIA256-SHA
TLS_DHE_RSA_WITH_CAMELLIA_128_CBC_SHA  DHE-RSA-CAMELLIA128-SHA
TLS_DHE_RSA_WITH_CAMELLIA_256_CBC_SHA  DHE-RSA-CAMELLIA256-SHA

TLS_DH_anon_WITH_CAMELLIA_128_CBC_SHA  ADH-CAMELLIA128-SHA
TLS_DH_anon_WITH_CAMELLIA_256_CBC_SHA  ADH-CAMELLIA256-SHA
```

SEED ciphersuites from RFC4162, extending TLS v1.0

```
TLS_RSA_WITH_SEED_CBC_SHA              SEED-SHA

TLS_DH_DSS_WITH_SEED_CBC_SHA           Not implemented.
TLS_DH_RSA_WITH_SEED_CBC_SHA           Not implemented.

TLS_DHE_DSS_WITH_SEED_CBC_SHA          DHE-DSS-SEED-SHA
TLS_DHE_RSA_WITH_SEED_CBC_SHA          DHE-RSA-SEED-SHA

TLS_DH_anon_WITH_SEED_CBC_SHA          ADH-SEED-SHA
```

GOST ciphersuites from draft-chudov-cryptopro-cptls, extending TLS v1.0

Note: these ciphers require an engine which including GOST cryptographic algorithms, such as the **ccgost** engine, included in the OpenSSL distribution.

```
TLS_GOSTR341094_WITH_28147_CNT_IMIT GOST94-GOST89-GOST89
TLS_GOSTR341001_WITH_28147_CNT_IMIT GOST2001-GOST89-GOST89
TLS_GOSTR341094_WITH_NULL_GOSTR3411 GOST94-NULL-GOST94
```

```
TLS_GOSTR341001_WITH_NULL_GOSTR3411 GOST2001-NULL-GOST94
```

Additional Export 1024 and other cipher suites

Note: these ciphers can also be used in SSL v3.

```
TLS_RSA_EXPORT1024_WITH_DES_CBC_SHA     EXP1024-DES-CBC-SHA
TLS_RSA_EXPORT1024_WITH_RC4_56_SHA      EXP1024-RC4-SHA
TLS_DHE_DSS_EXPORT1024_WITH_DES_CBC_SHA EXP1024-DHE-DSS-DES-CBC-SHA
TLS_DHE_DSS_EXPORT1024_WITH_RC4_56_SHA  EXP1024-DHE-DSS-RC4-SHA
TLS_DHE_DSS_WITH_RC4_128_SHA            DHE-DSS-RC4-SHA
```

Elliptic curve cipher suites.

```
TLS_ECDH_RSA_WITH_NULL_SHA              ECDH-RSA-NULL-SHA
TLS_ECDH_RSA_WITH_RC4_128_SHA           ECDH-RSA-RC4-SHA
TLS_ECDH_RSA_WITH_3DES_EDE_CBC_SHA      ECDH-RSA-DES-CBC3-SHA
TLS_ECDH_RSA_WITH_AES_128_CBC_SHA       ECDH-RSA-AES128-SHA
TLS_ECDH_RSA_WITH_AES_256_CBC_SHA       ECDH-RSA-AES256-SHA

TLS_ECDH_ECDSA_WITH_NULL_SHA            ECDH-ECDSA-NULL-SHA
TLS_ECDH_ECDSA_WITH_RC4_128_SHA         ECDH-ECDSA-RC4-SHA
TLS_ECDH_ECDSA_WITH_3DES_EDE_CBC_SHA    ECDH-ECDSA-DES-CBC3-SHA
TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA     ECDH-ECDSA-AES128-SHA
TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA     ECDH-ECDSA-AES256-SHA

TLS_ECDHE_RSA_WITH_NULL_SHA             ECDHE-RSA-NULL-SHA
TLS_ECDHE_RSA_WITH_RC4_128_SHA          ECDHE-RSA-RC4-SHA
TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA     ECDHE-RSA-DES-CBC3-SHA
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA      ECDHE-RSA-AES128-SHA
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA      ECDHE-RSA-AES256-SHA

TLS_ECDHE_ECDSA_WITH_NULL_SHA           ECDHE-ECDSA-NULL-SHA
TLS_ECDHE_ECDSA_WITH_RC4_128_SHA        ECDHE-ECDSA-RC4-SHA
TLS_ECDHE_ECDSA_WITH_3DES_EDE_CBC_SHA   ECDHE-ECDSA-DES-CBC3-SHA
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA    ECDHE-ECDSA-AES128-SHA
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA    ECDHE-ECDSA-AES256-SHA

TLS_ECDH_anon_WITH_NULL_SHA             AECDH-NULL-SHA
TLS_ECDH_anon_WITH_RC4_128_SHA          AECDH-RC4-SHA
TLS_ECDH_anon_WITH_3DES_EDE_CBC_SHA     AECDH-DES-CBC3-SHA
TLS_ECDH_anon_WITH_AES_128_CBC_SHA      AECDH-AES128-SHA
TLS_ECDH_anon_WITH_AES_256_CBC_SHA      AECDH-AES256-SHA
```

TLS v1.2 cipher suites

```
TLS_RSA_WITH_NULL_SHA256                  NULL-SHA256

TLS_RSA_WITH_AES_128_CBC_SHA256           AES128-SHA256
TLS_RSA_WITH_AES_256_CBC_SHA256           AES256-SHA256
TLS_RSA_WITH_AES_128_GCM_SHA256           AES128-GCM-SHA256
TLS_RSA_WITH_AES_256_GCM_SHA384           AES256-GCM-SHA384

TLS_DH_RSA_WITH_AES_128_CBC_SHA256        Not implemented.
TLS_DH_RSA_WITH_AES_256_CBC_SHA256        Not implemented.
TLS_DH_RSA_WITH_AES_128_GCM_SHA256        Not implemented.
TLS_DH_RSA_WITH_AES_256_GCM_SHA384        Not implemented.

TLS_DH_DSS_WITH_AES_128_CBC_SHA256        Not implemented.
TLS_DH_DSS_WITH_AES_256_CBC_SHA256        Not implemented.
TLS_DH_DSS_WITH_AES_128_GCM_SHA256        Not implemented.
TLS_DH_DSS_WITH_AES_256_GCM_SHA384        Not implemented.

TLS_DHE_RSA_WITH_AES_128_CBC_SHA256       DHE-RSA-AES128-SHA256
TLS_DHE_RSA_WITH_AES_256_CBC_SHA256       DHE-RSA-AES256-SHA256
TLS_DHE_RSA_WITH_AES_128_GCM_SHA256       DHE-RSA-AES128-GCM-SHA256
```

```
TLS_DHE_RSA_WITH_AES_256_GCM_SHA384       DHE-RSA-AES256-GCM-SHA384

TLS_DHE_DSS_WITH_AES_128_CBC_SHA256       DHE-DSS-AES128-SHA256
TLS_DHE_DSS_WITH_AES_256_CBC_SHA256       DHE-DSS-AES256-SHA256
TLS_DHE_DSS_WITH_AES_128_GCM_SHA256       DHE-DSS-AES128-GCM-SHA256
TLS_DHE_DSS_WITH_AES_256_GCM_SHA384       DHE-DSS-AES256-GCM-SHA384

TLS_ECDH_RSA_WITH_AES_128_CBC_SHA256      ECDH-RSA-AES128-SHA256
TLS_ECDH_RSA_WITH_AES_256_CBC_SHA384      ECDH-RSA-AES256-SHA384
TLS_ECDH_RSA_WITH_AES_128_GCM_SHA256      ECDH-RSA-AES128-GCM-SHA256
TLS_ECDH_RSA_WITH_AES_256_GCM_SHA384      ECDH-RSA-AES256-GCM-SHA384

TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA256    ECDH-ECDSA-AES128-SHA256
TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA384    ECDH-ECDSA-AES256-SHA384
TLS_ECDH_ECDSA_WITH_AES_128_GCM_SHA256    ECDH-ECDSA-AES128-GCM-SHA256
TLS_ECDH_ECDSA_WITH_AES_256_GCM_SHA384    ECDH-ECDSA-AES256-GCM-SHA384

TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256     ECDHE-RSA-AES128-SHA256
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384     ECDHE-RSA-AES256-SHA384
TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256     ECDHE-RSA-AES128-GCM-SHA256
TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384     ECDHE-RSA-AES256-GCM-SHA384

TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256   ECDHE-ECDSA-AES128-SHA256
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384   ECDHE-ECDSA-AES256-SHA384
TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256   ECDHE-ECDSA-AES128-GCM-SHA256
TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384   ECDHE-ECDSA-AES256-GCM-SHA384

TLS_DH_anon_WITH_AES_128_CBC_SHA256       ADH-AES128-SHA256
TLS_DH_anon_WITH_AES_256_CBC_SHA256       ADH-AES256-SHA256
TLS_DH_anon_WITH_AES_128_GCM_SHA256       ADH-AES128-GCM-SHA256
TLS_DH_anon_WITH_AES_256_GCM_SHA384       ADH-AES256-GCM-SHA384
```

Pre shared keying (PSK) cipheruites

```
TLS_PSK_WITH_RC4_128_SHA                  PSK-RC4-SHA
TLS_PSK_WITH_3DES_EDE_CBC_SHA             PSK-3DES-EDE-CBC-SHA
TLS_PSK_WITH_AES_128_CBC_SHA              PSK-AES128-CBC-SHA
TLS_PSK_WITH_AES_256_CBC_SHA              PSK-AES256-CBC-SHA
```

Deprecated SSL v2.0 cipher suites.

```
SSL_CK_RC4_128_WITH_MD5                 RC4-MD5
SSL_CK_RC4_128_EXPORT40_WITH_MD5        Not implemented.
SSL_CK_RC2_128_CBC_WITH_MD5             RC2-CBC-MD5
SSL_CK_RC2_128_CBC_EXPORT40_WITH_MD5    Not implemented.
SSL_CK_IDEA_128_CBC_WITH_MD5            IDEA-CBC-MD5
SSL_CK_DES_64_CBC_WITH_MD5              Not implemented.
SSL_CK_DES_192_EDE3_CBC_WITH_MD5        DES-CBC3-MD5
```

NOTES

The non-ephemeral DH modes are currently unimplemented in OpenSSL because there is no support for DH certificates.

Some compiled versions of OpenSSL may not include all the ciphers listed here because some ciphers were excluded at compile time.

EXAMPLES

Verbose listing of all OpenSSL ciphers including NULL ciphers:

```
openssl ciphers -v 'ALL:eNULL'
```

Include all ciphers except NULL and anonymous DH then sort by strength:

```
openssl ciphers -v 'ALL:!ADH:@STRENGTH'
```

Include all ciphers except ones with no encryption (eNULL) or no authentication (aNULL):

```
openssl ciphers -v 'ALL:!aNULL'
```

Include only 3DES ciphers and then place RSA ciphers last:

```
openssl ciphers -v '3DES:+RSA'
```

Include all RC4 ciphers but leave out those without authentication:

```
openssl ciphers -v 'RC4:!COMPLEMENTOFDEFAULT'
```

Include all chiphers with RSA authentication but leave out ciphers without encryption.

```
openssl ciphers -v 'RSA:!COMPLEMENTOFALL'
```

SEE ALSO

s_client(1), s_server(1), ssl(3)

HISTORY

The **COMPLENTOFALL** and **COMPLEMENTOFDEFAULT** selection options for cipherlist strings were added in OpenSSL 0.9.7. The **-V** option for the **ciphers** command was added in OpenSSL 1.0.0.

Name

cms — CMS utility

Synopsis

```
opensslcms
[-encrypt]
[-decrypt]
[-sign]
[-verify]
[-cmsout]
[-resign]
[-data_create]
[-data_out]
[-digest_create]
[-digest_verify]
[-compress]
[-uncompress]
[-EncryptedData_encrypt]
[-sign_receipt]
[-verify_receipt receipt]
[-in filename]
[-inform SMIME|PEM|DER]
[-rctform SMIME|PEM|DER]
[-out filename]
[-outform SMIME|PEM|DER]
[-stream -indef -noindef]
[-noindef]
[-content filename]
[-text]
[-noout]
[-print]
[-CAfile file]
[-CApath dir]
[-no_alt_chains]
[-md digest]
[-[cipher]]
[-nointern]
[-no_signer_cert_verify]
[-nocerts]
[-noattr]
[-nosmimecap]
[-binary]
[-nodetach]
[-certfile file]
[-certsout file]
[-signer file]
[-recip file]
[-keyid]
[-receipt_request_all -receipt_request_first]
[-receipt_request_from emailaddress]
[-receipt_request_to emailaddress]
[-receipt_request_print]
[-secretkey key]
[-secretkeyid id]
[-econtent_type type]
[-inkey file]
[-passin arg]
[-rand file(s)]
[cert.pem…]
[-to addr]
[-from addr]
[-subject subj]
[cert.pem]…
```

DESCRIPTION

The **cms** command handles S/MIME v3.1 mail. It can encrypt, decrypt, sign and verify, compress and uncompress S/MIME messages.

COMMAND OPTIONS

There are fourteen operation options that set the type of operation to be performed. The meaning of the other options varies according to the operation type.

-encrypt

encrypt mail for the given recipient certificates. Input file is the message to be encrypted. The output file is the encrypted mail in MIME format. The actual CMS type is <B>EnvelopedData<B>.

Note that no revocation check is done for the recipient cert, so if that key has been compromised, others may be able to decrypt the text.

-decrypt

decrypt mail using the supplied certificate and private key. Expects an encrypted mail message in MIME format for the input file. The decrypted mail is written to the output file.

-debug_decrypt

this option sets the **CMS_DEBUG_DECRYPT** flag. This option should be used with caution: see the notes section below.

-sign

sign mail using the supplied certificate and private key. Input file is the message to be signed. The signed message in MIME format is written to the output file.

-verify

verify signed mail. Expects a signed mail message on input and outputs the signed data. Both clear text and opaque signing is supported.

-cmsout

takes an input message and writes out a PEM encoded CMS structure.

-resign

resign a message: take an existing message and one or more new signers.

-data_create

Create a CMS **Data** type.

-data_out

Data type and output the content.

-digest_create

Create a CMS **DigestedData** type.

-digest_verify

Verify a CMS **DigestedData** type and output the content.

-compress

Create a CMS **CompressedData** type. OpenSSL must be compiled with **zlib** support for this option to work, otherwise it will output an error.

-uncompress

Uncompress a CMS **CompressedData** type and output the content. OpenSSL must be compiled with **zlib** support for this option to work, otherwise it will output an error.

-EncryptedData_encrypt

Encrypt content using supplied symmetric key and algorithm using a CMS **EncrytedData** type and output the content.

-sign_receipt

Generate and output a signed receipt for the supplied message. The input message **must** contain a signed receipt request. Functionality is otherwise similar to the **-sign** operation.

-verify_receipt receipt

Verify a signed receipt in filename **receipt**. The input message **must** contain the original receipt request. Functionality is otherwise similar to the **-verify** operation.

-in filename

the input message to be encrypted or signed or the message to be decrypted or verified.

-inform SMIME|PEM|DER

this specifies the input format for the CMS structure. The default is **SMIME** which reads an S/MIME format message. **PEM** and **DER** format change this to expect PEM and DER format CMS structures instead. This currently only affects the input format of the CMS structure, if no CMS structure is being input (for example with **-encrypt** or **-sign**) this option has no effect.

-rctform SMIME|PEM|DER

specify the format for a signed receipt for use with the **-receipt_verify** operation.

-out filename

the message text that has been decrypted or verified or the output MIME format message that has been signed or verified.

-outform SMIME|PEM|DER

this specifies the output format for the CMS structure. The default is **SMIME** which writes an S/MIME format message. **PEM** and **DER** format change this to write PEM and DER format CMS structures instead. This currently only affects the output format of the CMS structure, if no CMS structure is being output (for example with **-verify** or **-decrypt**) this option has no effect.

-stream -indef -noindef

the **-stream** and **-indef** options are equivalent and enable streaming I/O for encoding operations. This permits single pass processing of data without the need to hold the entire contents in memory, potentially supporting very large files. Streaming is automatically set for S/MIME signing with detached data if the output format is **SMIME** it is currently off by default for all other operations.

-noindef

disable streaming I/O where it would produce and indefinite length constructed encoding. This option currently has no effect. In future streaming will be enabled by default on all relevant operations and this option will disable it.

-content filename

This specifies a file containing the detached content, this is only useful with the **-verify** command. This is only usable if the CMS structure is using the detached signature form where the content is not included. This option will override any content if the input format is S/MIME and it uses the multipart/signed MIME content type.

-text

this option adds plain text (text/plain) MIME headers to the supplied message if encrypting or signing. If decrypting or verifying it strips off text headers: if the decrypted or verified message is not of MIME type text/plain then an error occurs.

-noout

for the **-cmsout** operation do not output the parsed CMS structure. This is useful when combined with the **-print** option or if the syntax of the CMS structure is being checked.

-print

for the **-cmsout** operation print out all fields of the CMS structure. This is mainly useful for testing purposes.

-CAfile file

a file containing trusted CA certificates, only used with **-verify**.

-CApath dir

a directory containing trusted CA certificates, only used with **-verify**. This directory must be a standard certificate directory: that is a hash of each subject name (using **x509 -hash**) should be linked to each certificate.

-md digest

digest algorithm to use when signing or resigning. If not present then the default digest algorithm for the signing key will be used (usually SHA1).

-[cipher]

the encryption algorithm to use. For example triple DES (168 bits) - **-des3** or 256 bit AES - **-aes256**. Any standard algorithm name (as used by the EVP_get_cipherbyname() function) can also be used preceded by a dash, for example **-aes_128_cbc**. See **enc** for a list of ciphers supported by your version of OpenSSL.

If not specified triple DES is used. Only used with **-encrypt** and **-EncryptedData_create** commands.

-nointern

when verifying a message normally certificates (if any) included in the message are searched for the signing certificate. With this option only the certificates specified in the **-certfile** option are used. The supplied certificates can still be used as untrusted CAs however.

-no_signer_cert_verify

do not verify the signers certificate of a signed message.

-nocerts

when signing a message the signer's certificate is normally included with this option it is excluded. This will reduce the size of the signed message but the verifier must have a copy of the signers certificate available locally (passed using the **-certfile** option for example).

-noattr

normally when a message is signed a set of attributes are included which include the signing time and supported symmetric algorithms. With this option they are not included.

-nosmimecap

exclude the list of supported algorithms from signed attributes, other options such as signing time and content type are still included.

-binary

normally the input message is converted to "canonical" format which is effectively using CR and LF as end of line: as required by the S/MIME specification. When this option is present no translation occurs. This is useful when handling binary data which may not be in MIME format.

-nodetach

when signing a message use opaque signing: this form is more resistant to translation by mail relays but it cannot be read by mail agents that do not support S/MIME. Without this option cleartext signing with the MIME type multipart/signed is used.

-certfile file

allows additional certificates to be specified. When signing these will be included with the message. When verifying these will be searched for the signers certificates. The certificates should be in PEM format.

-certsout file

any certificates contained in the message are written to **file**.

-signer file

a signing certificate when signing or resigning a message, this option can be used multiple times if more than one signer is required. If a message is being verified then the signers certificates will be written to this file if the verification was successful.

-recip file

the recipients certificate when decrypting a message. This certificate must match one of the recipients of the message or an error occurs.

-keyid

use subject key identifier to identify certificates instead of issuer name and serial number. The supplied certificate **must** include a subject key identifier extension. Supported by **-sign** and **-encrypt** options.

-receipt_request_all -receipt_request_first

for **-sign** option include a signed receipt request. Indicate requests should be provided by all receipient or first tier recipients (those mailed directly and not from a mailing list). Ignored it **-receipt_request_from** is included.

-receipt_request_from emailaddress

for **-sign** option include a signed receipt request. Add an explicit email address where receipts should be supplied.

-receipt_request_to emailaddress

Add an explicit email address where signed receipts should be sent to. This option **must** but supplied if a signed receipt it requested.

-receipt_request_print

For the **-verify** operation print out the contents of any signed receipt requests.

-secretkey key

specify symmetric key to use. The key must be supplied in hex format and be consistent with the algorithm used. Supported by the **-EncryptedData_encrypt -EncrryptedData_decrypt**, **-encrypt** and **-decrypt** options. When used with **-encrypt** or **-decrypt** the supplied key is used to wrap or unwrap the content encryption key using an AES key in the **KEKRecipientInfo** type.

-secretkeyid id

the key identifier for the supplied symmetric key for **KEKRecipientInfo** type. This option **must** be present if the **-secretkey** option is used with **-encrypt**. With **-decrypt** operations the **id** is used to locate the relevant key if it is not supplied then an attempt is used to decrypt any **KEKRecipientInfo** structures.

-econtent_type type

set the encapsulated content type to **type** if not supplied the **Data** type is used. The **type** argument can be any valid OID name in either text or numerical format.

-inkey file

the private key to use when signing or decrypting. This must match the corresponding certificate. If this option is not specified then the private key must be included in the certificate file specified with the **-recip** or **-signer** file. When signing this option can be used multiple times to specify successive keys.

-passin arg

the private key password source. For more information about the format of **arg** see the **PASS PHRASE ARGUMENTS** section in openssl(1).

-rand file(s)

a file or files containing random data used to seed the random number generator, or an EGD socket (see RAND_egd(3)). Multiple files can be specified separated by a OS-dependent character. The separator is **;** for MS-Windows, **,** for OpenVMS, and **:** for all others.

cert.pem…

one or more certificates of message recipients: used when encrypting a message.

-to, -from, -subject

the relevant mail headers. These are included outside the signed portion of a message so they may be included manually. If signing then many S/MIME mail clients check the signers certificate's email address matches that specified in the From: address.

-purpose, -ignore_critical, -issuer_checks, -crl_check, -crl_check_all, -policy_check, -extended_crl, -x509_strict, -policy -check_ss_sig -no_alt_chains

Set various certificate chain valiadition option. See the **verify** manual page for details.

NOTES

The MIME message must be sent without any blank lines between the headers and the output. Some mail programs will automatically add a blank line. Piping the mail directly to sendmail is one way to achieve the correct format.

The supplied message to be signed or encrypted must include the necessary MIME headers or many S/MIME clients wont display it properly (if at all). You can use the **-text** option to automatically add plain text headers.

A "signed and encrypted" message is one where a signed message is then encrypted. This can be produced by encrypting an already signed message: see the examples section.

This version of the program only allows one signer per message but it will verify multiple signers on received messages. Some S/MIME clients choke if a message contains multiple signers. It is possible to sign messages "in parallel" by signing an already signed message.

The options **-encrypt** and **-decrypt** reflect common usage in S/MIME clients. Strictly speaking these process CMS enveloped data: CMS encrypted data is used for other purposes.

The **-resign** option uses an existing message digest when adding a new signer. This means that attributes must be present in at least one existing signer using the same message digest or this operation will fail.

The **-stream** and **-indef** options enable experimental streaming I/O support. As a result the encoding is BER using indefinite length constructed encoding and no longer DER. Streaming is supported for the **-encrypt** operation and the **-sign** operation if the content is not detached.

Streaming is always used for the **-sign** operation with detached data but since the content is no longer part of the CMS structure the encoding remains DER.

If the **-decrypt** option is used without a recipient certificate then an attempt is made to locate the recipient by trying each potential recipient in turn using the supplied private key. To thwart the MMA attack (Bleichenbacher's attack on PKCS #1 v1.5 RSA padding) all recipients are tried whether they succeed or not and if no recipients match the message is "decrypted" using a random key which will typically output garbage. The **-debug_decrypt** option can be used to disable the MMA attack protection and return an error if no recipient can be found: this option should be used with caution. For a fuller description see CMS_decrypt(3)).

EXIT CODES

0 the operation was completely successfully.

1 an error occurred parsing the command options.

2 one of the input files could not be read.

3 an error occurred creating the CMS file or when reading the MIME message.

4 an error occurred decrypting or verifying the message.

5 the message was verified correctly but an error occurred writing out the signers certificates.

COMPATIBILITY WITH PKCS#7 format.

The **smime** utility can only process the older **PKCS#7** format. The **cms** utility supports Cryptographic Message Syntax format. Use of some features will result in messages which cannot be processed by applications which only support the older format. These are detailed below.

The use of the **-keyid** option with **-sign** or **-encrypt**.

The **-outform PEM** option uses different headers.

The **-compress** option.

The **-secretkey** option when used with **-encrypt**.

Additionally the **-EncryptedData_create** and **-data_create** type cannot be processed by the older **smime** command.

EXAMPLES

Create a cleartext signed message:

```
openssl cms -sign -in message.txt -text -out mail.msg \
       -signer mycert.pem
```

Create an opaque signed message

```
openssl cms -sign -in message.txt -text -out mail.msg -nodetach \
       -signer mycert.pem
```

Create a signed message, include some additional certificates and read the private key from another file:

```
openssl cms -sign -in in.txt -text -out mail.msg \
       -signer mycert.pem -inkey mykey.pem -certfile mycerts.pem
```

Create a signed message with two signers, use key identifier:

```
openssl cms -sign -in message.txt -text -out mail.msg \
       -signer mycert.pem -signer othercert.pem -keyid
```

Send a signed message under Unix directly to sendmail, including headers:

```
openssl cms -sign -in in.txt -text -signer mycert.pem \
       -from steve@openssl.org -to someone@somewhere \
       -subject "Signed message" | sendmail someone@somewhere
```

Verify a message and extract the signer's certificate if successful:

```
openssl cms -verify -in mail.msg -signer user.pem -out signedtext.txt
```

Send encrypted mail using triple DES:

```
openssl cms -encrypt -in in.txt -from steve@openssl.org \
       -to someone@somewhere -subject "Encrypted message" \
       -des3 user.pem -out mail.msg
```

Sign and encrypt mail:

```
openssl cms -sign -in ml.txt -signer my.pem -text \
       | openssl cms -encrypt -out mail.msg \
       -from steve@openssl.org -to someone@somewhere \
       -subject "Signed and Encrypted message" -des3 user.pem
```

Note: the encryption command does not include the **-text** option because the message being encrypted already has MIME headers.

Decrypt mail:

```
openssl cms -decrypt -in mail.msg -recip mycert.pem -inkey key.pem
```

The output from Netscape form signing is a PKCS#7 structure with the detached signature format. You can use this program to verify the signature by line wrapping the base64 encoded structure and surrounding it with:

```
-----BEGIN PKCS7-----
-----END PKCS7-----
```

and using the command,

```
openssl cms -verify -inform PEM -in signature.pem -content content.txt
```

alternatively you can base64 decode the signature and use

```
openssl cms -verify -inform DER -in signature.der -content content.txt
```

Create an encrypted message using 128 bit Camellia:

```
openssl cms -encrypt -in plain.txt -camellia128 -out mail.msg cert.pem
```

Add a signer to an existing message:

```
openssl cms -resign -in mail.msg -signer newsign.pem -out mail2.msg
```

BUGS

The MIME parser isn't very clever: it seems to handle most messages that I've thrown at it but it may choke on others.

The code currently will only write out the signer's certificate to a file: if the signer has a separate encryption certificate this must be manually extracted. There should be some heuristic that determines the correct encryption certificate.

Ideally a database should be maintained of a certificates for each email address.

The code doesn't currently take note of the permitted symmetric encryption algorithms as supplied in the SMIMECapabilities signed attribute. this means the user has to manually include the correct encryption algorithm. It should store the list of permitted ciphers in a database and only use those.

No revocation checking is done on the signer's certificate.

HISTORY

The use of multiple **-signer** options and the **-resign** command were first added in OpenSSL 1.0.0

The -no_alt_chains options was first added to OpenSSL 1.0.1n and 1.0.2b.

Name

crl — CRL utility

Synopsis

```
opensslcrl
[-inform PEM|DER]
[-outform PEM|DER]
[-text]
[-in filename]
[-out filename]
[-nameopt option]
[-noout]
[-hash]
[-issuer]
[-lastupdate]
[-nextupdate]
[-CAfile file]
[-CApath dir]
```

DESCRIPTION

The **crl** command processes CRL files in DER or PEM format.

COMMAND OPTIONS

-inform DER|PEM

This specifies the input format. **DER** format is DER encoded CRL structure. **PEM** (the default) is a base64 encoded version of the DER form with header and footer lines.

-outform DER|PEM

This specifies the output format, the options have the same meaning as the **-inform** option.

-in filename

This specifies the input filename to read from or standard input if this option is not specified.

-out filename

specifies the output filename to write to or standard output by default.

-text

print out the CRL in text form.

-nameopt option

option which determines how the subject or issuer names are displayed. See the description of **-nameopt** in x509(1).

-noout

don't output the encoded version of the CRL.

-hash

output a hash of the issuer name. This can be use to lookup CRLs in a directory by issuer name.

-hash_old

outputs the "hash" of the CRL issuer name using the older algorithm as used by OpenSSL versions before 1.0.0.

-issuer

output the issuer name.

-lastupdate

output the lastUpdate field.

-nextupdate

output the nextUpdate field.

-CAfile file

verify the signature on a CRL by looking up the issuing certificate in **file**

-CApath dir

verify the signature on a CRL by looking up the issuing certificate in **dir**. This directory must be a standard certificate directory: that is a hash of each subject name (using **x509 -hash**) should be linked to each certificate.

NOTES

The PEM CRL format uses the header and footer lines:

```
-----BEGIN X509 CRL-----
-----END X509 CRL-----
```

EXAMPLES

Convert a CRL file from PEM to DER:

```
openssl crl -in crl.pem -outform DER -out crl.der
```

Output the text form of a DER encoded certificate:

```
openssl crl -in crl.der -text -noout
```

BUGS

Ideally it should be possible to create a CRL using appropriate options and files too.

SEE ALSO

crl2pkcs7(1), ca(1), x509(1)

Name

crl2pkcs7 — Create a PKCS#7 structure from a CRL and certificates.

Synopsis

```
opensslcrl2pkcs7
[-inform PEM|DER]
[-outform PEM|DER]
[-in filename]
[-out filename]
[-certfile filename]
[-nocrl]
```

DESCRIPTION

The **crl2pkcs7** command takes an optional CRL and one or more certificates and converts them into a PKCS#7 degenerate "certificates only" structure.

COMMAND OPTIONS

-inform DER|PEM

This specifies the CRL input format. **DER** format is DER encoded CRL structure.**PEM** (the default) is a base64 encoded version of the DER form with header and footer lines.

-outform DER|PEM

This specifies the PKCS#7 structure output format. **DER** format is DER encoded PKCS#7 structure.**PEM** (the default) is a base64 encoded version of the DER form with header and footer lines.

-in filename

This specifies the input filename to read a CRL from or standard input if this option is not specified.

-out filename

specifies the output filename to write the PKCS#7 structure to or standard output by default.

-certfile filename

specifies a filename containing one or more certificates in **PEM** format. All certificates in the file will be added to the PKCS#7 structure. This option can be used more than once to read certificates form multiple files.

-nocrl

normally a CRL is included in the output file. With this option no CRL is included in the output file and a CRL is not read from the input file.

EXAMPLES

Create a PKCS#7 structure from a certificate and CRL:

```
openssl crl2pkcs7 -in crl.pem -certfile cert.pem -out p7.pem
```

Creates a PKCS#7 structure in DER format with no CRL from several different certificates:

```
openssl crl2pkcs7 -nocrl -certfile newcert.pem
```

```
       -certfile demoCA/cacert.pem -outform DER -out p7.der
```

NOTES

The output file is a PKCS#7 signed data structure containing no signers and just certificates and an optional CRL.

This utility can be used to send certificates and CAs to Netscape as part of the certificate enrollment process. This involves sending the DER encoded output as MIME type application/x-x509-user-cert.

The **PEM** encoded form with the header and footer lines removed can be used to install user certificates and CAs in MSIE using the Xenroll control.

SEE ALSO

pkcs7(1)

Name

dgst, sha, sha1, mdc2, ripemd160, sha224, sha256, sha384, sha512, md2, md4, md5 and dss1 — message digests

Synopsis

```
openssldgst
[-sha|-sha1|-mdc2|-ripemd160|-sha224|-sha256|-sha384|-sha512|-md2|-md4|-md5|-dss1]
[-c]
[-d]
[-hex]
[-binary]
[-r]
[-non-fips-allow]
[-out filename]
[-sign filename]
[-keyform arg]
[-passin arg]
[-verify filename]
[-prverify filename]
[-signature filename]
[-hmac key]
[-non-fips-allow]
[-fips-fingerprint]
[file…]
```

```
openssl
[digest]
[…]
```

DESCRIPTION

The digest functions output the message digest of a supplied file or files in hexadecimal. The digest functions also generate and verify digital signatures using message digests.

OPTIONS

-c

print out the digest in two digit groups separated by colons, only relevant if **hex** format output is used.

-d

print out BIO debugging information.

-hex

digest is to be output as a hex dump. This is the default case for a "normal" digest as opposed to a digital signature. See NOTES below for digital signatures using **-hex**.

-binary

output the digest or signature in binary form.

-r

output the digest in the "coreutils" format used by programs like **sha1sum**.

-non-fips-allow

Allow use of non FIPS digest when in FIPS mode. This has no effect when not in FIPS mode.

-out filename

filename to output to, or standard output by default.

-sign filename

digitally sign the digest using the private key in "filename".

-keyform arg

Specifies the key format to sign digest with. The DER, PEM, P12, and ENGINE formats are supported.

-engine id

Use engine **id** for operations (including private key storage). This engine is not used as source for digest algorithms, unless it is also specified in the configuration file.

-sigopt nm:v

Pass options to the signature algorithm during sign or verify operations. Names and values of these options are algorithm-specific.

-passin arg

the private key password source. For more information about the format of **arg** see the **PASS PHRASE ARGUMENTS** section in openssl(1).

-verify filename

verify the signature using the the public key in "filename". The output is either "Verification OK" or "Verification Failure".

-prverify filename

verify the signature using the the private key in "filename".

-signature filename

the actual signature to verify.

-hmac key

create a hashed MAC using "key".

-mac alg

create MAC (keyed Message Authentication Code). The most popular MAC algorithm is HMAC (hash-based MAC), but there are other MAC algorithms which are not based on hash, for instance **gost-mac** algorithm, supported by **ccgost** engine. MAC keys and other options should be set via **-macopt** parameter.

-macopt nm:v

Passes options to MAC algorithm, specified by **-mac** key. Following options are supported by both by **HMAC** and **gost-mac**:

key:string

Specifies MAC key as alphnumeric string (use if key contain printable characters only). String length must conform to any restrictions of the MAC algorithm for example exactly 32 chars for gost-mac.

hexkey:string

Specifies MAC key in hexadecimal form (two hex digits per byte). Key length must conform to any restrictions of the MAC algorithm for example exactly 32 chars for gost-mac.

-rand file(s)

a file or files containing random data used to seed the random number generator, or an EGD socket (see RAND_egd(3)). Multiple files can be specified separated by a OS-dependent character. The separator is **;** for MS-Windows, **,** for OpenVMS, and **:** for all others.

-non-fips-allow

enable use of non-FIPS algorithms such as MD5 even in FIPS mode.

-fips-fingerprint

compute HMAC using a specific key for certain OpenSSL-FIPS operations.

file…

file or files to digest. If no files are specified then standard input is used.

EXAMPLES

To create a hex-encoded message digest of a file: openssl dgst -md5 -hex file.txt

To sign a file using SHA-256 with binary file output: openssl dgst -sha256 -sign privatekey.pem -out signature.sign file.txt

To verify a signature: openssl dgst -sha256 -verify publickey.pem \ -signature signature.sign \ file.txt

NOTES

The digest of choice for all new applications is SHA1. Other digests are however still widely used.

When signing a file, **dgst** will automatically determine the algorithm (RSA, ECC, etc) to use for signing based on the private key's ASN.1 info. When verifying signatures, it only handles the RSA, DSA, or ECDSA signature itself, not the related data to identify the signer and algorithm used in formats such as x.509, CMS, and S/MIME.

A source of random numbers is required for certain signing algorithms, in particular ECDSA and DSA.

The signing and verify options should only be used if a single file is being signed or verified.

Hex signatures cannot be verified using **openssl**. Instead, use "xxd -r" or similar program to transform the hex signature into a binary signature prior to verification.

Name

dhparam — DH parameter manipulation and generation

Synopsis

```
openssl dhparam
[-inform DER|PEM]
[-outform DER|PEM]
[-in filename]
[-out filename]
[-dsaparam]
[-check]
[-noout]
[-text]
[-C]
[-2]
[-5]
[-rand file(s)]
[-engine id]
[numbits]
```

DESCRIPTION

This command is used to manipulate DH parameter files.

OPTIONS

-inform DER|PEM

This specifies the input format. The **DER** option uses an ASN1 DER encoded form compatible with the PKCS#3 DHparameter structure. The PEM form is the default format: it consists of the **DER** format base64 encoded with additional header and footer lines.

-outform DER|PEM

This specifies the output format, the options have the same meaning as the **-inform** option.

-in *filename*

This specifies the input filename to read parameters from or standard input if this option is not specified.

-out *filename*

This specifies the output filename parameters to. Standard output is used if this option is not present. The output filename should **not** be the same as the input filename.

-dsaparam

If this option is used, DSA rather than DH parameters are read or created; they are converted to DH format. Otherwise, "strong" primes (such that (p-1)/2 is also prime) will be used for DH parameter generation.

DH parameter generation with the **-dsaparam** option is much faster, and the recommended exponent length is shorter, which makes DH key exchange more efficient. Beware that with such DSA-style DH parameters, a fresh DH key should be created for each use to avoid small-subgroup attacks that may be possible otherwise.

-check

check if the parameters are valid primes and generator.

-2, **-5**

The generator to use, either 2 or 5. If present then the input file is ignored and parameters are generated instead. If not present but **numbits** is present, parameters are generated with the default generator 2.

-rand *file(s)*

a file or files containing random data used to seed the random number generator, or an EGD socket (see RAND_egd(3)). Multiple files can be specified separated by a OS-dependent character. The separator is **;** for MS-Windows, **,** for OpenVMS, and **:** for all others.

numbits

this option specifies that a parameter set should be generated of size *numbits*. It must be the last option. If this option is present then the input file is ignored and parameters are generated instead. If this option is not present but a generator (**-2** or **-5**) is present, parameters are generated with a default length of 2048 bits.

-noout

this option inhibits the output of the encoded version of the parameters.

-text

this option prints out the DH parameters in human readable form.

-C

this option converts the parameters into C code. The parameters can then be loaded by calling the **get_dh***numbits***()** function.

-engine id

specifying an engine (by its unique **id** string) will cause **dhparam** to attempt to obtain a functional reference to the specified engine, thus initialising it if needed. The engine will then be set as the default for all available algorithms.

WARNINGS

The program **dhparam** combines the functionality of the programs **dh** and **gendh** in previous versions of OpenSSL and SSLeay. The **dh** and **gendh** programs are retained for now but may have different purposes in future versions of OpenSSL.

NOTES

PEM format DH parameters use the header and footer lines:

```
-----BEGIN DH PARAMETERS-----
-----END DH PARAMETERS-----
```

OpenSSL currently only supports the older PKCS#3 DH, not the newer X9.42 DH.

This program manipulates DH parameters not keys.

BUGS

There should be a way to generate and manipulate DH keys.

SEE ALSO

dsaparam(1)

HISTORY

The **dhparam** command was added in OpenSSL 0.9.5. The **-dsaparam** option was added in OpenSSL 0.9.6.

Name

dsa — DSA key processing

Synopsis

```
openssldsa
[-inform PEM|DER]
[-outform PEM|DER]
[-in filename]
[-passin arg]
[-out filename]
[-passout arg]
[-aes128]
[-aes192]
[-aes256]
[-camellia128]
[-camellia192]
[-camellia256]
[-des]
[-des3]
[-idea]
[-text]
[-noout]
[-modulus]
[-pubin]
[-pubout]
[-engine id]
```

DESCRIPTION

The **dsa** command processes DSA keys. They can be converted between various forms and their components printed out. **Note** This command uses the traditional SSLeay compatible format for private key encryption: newer applications should use the more secure PKCS#8 format using the **pkcs8**

COMMAND OPTIONS

-inform DER|PEM

This specifies the input format. The **DER** option with a private key uses an ASN1 DER encoded form of an ASN.1 SEQUENCE consisting of the values of version (currently zero), p, q, g, the public and private key components respectively as ASN.1 INTEGERs. When used with a public key it uses a SubjectPublicKeyInfo structure: it is an error if the key is not DSA.

The **PEM** form is the default format: it consists of the **DER** format base64 encoded with additional header and footer lines. In the case of a private key PKCS#8 format is also accepted.

-outform DER|PEM

This specifies the output format, the options have the same meaning as the **-inform** option.

-in filename

This specifies the input filename to read a key from or standard input if this option is not specified. If the key is encrypted a pass phrase will be prompted for.

-passin arg

the input file password source. For more information about the format of **arg** see the **PASS PHRASE ARGUMENTS** section in openssl(1).

-out filename

This specifies the output filename to write a key to or standard output by is not specified. If any encryption options are set then a pass phrase will be prompted for. The output filename should **not** be the same as the input filename.

-passout arg

the output file password source. For more information about the format of **arg** see the **PASS PHRASE ARGUMENTS** section in openssl(1).

-aes128|-aes192|-aes256|-camellia128|-camellia192|-camellia256|-des|-des3|-idea

These options encrypt the private key with the specified cipher before outputting it. A pass phrase is prompted for. If none of these options is specified the key is written in plain text. This means that using the **dsa** utility to read in an encrypted key with no encryption option can be used to remove the pass phrase from a key, or by setting the encryption options it can be use to add or change the pass phrase. These options can only be used with PEM format output files.

-text

prints out the public, private key components and parameters.

-noout

this option prevents output of the encoded version of the key.

-modulus

this option prints out the value of the public key component of the key.

-pubin

by default a private key is read from the input file: with this option a public key is read instead.

-pubout

by default a private key is output. With this option a public key will be output instead. This option is automatically set if the input is a public key.

-engine id

specifying an engine (by its unique **id** string) will cause **dsa** to attempt to obtain a functional reference to the specified engine, thus initialising it if needed. The engine will then be set as the default for all available algorithms.

NOTES

The PEM private key format uses the header and footer lines:

```
-----BEGIN DSA PRIVATE KEY-----
-----END DSA PRIVATE KEY-----
```

The PEM public key format uses the header and footer lines:

```
-----BEGIN PUBLIC KEY-----
-----END PUBLIC KEY-----
```

EXAMPLES

To remove the pass phrase on a DSA private key:

```
openssl dsa -in key.pem -out keyout.pem
```

To encrypt a private key using triple DES:

```
openssl dsa -in key.pem -des3 -out keyout.pem
```

To convert a private key from PEM to DER format:

```
openssl dsa -in key.pem -outform DER -out keyout.der
```

To print out the components of a private key to standard output:

```
openssl dsa -in key.pem -text -noout
```

To just output the public part of a private key:

```
openssl dsa -in key.pem -pubout -out pubkey.pem
```

SEE ALSO

dsaparam(1), gendsa(1), rsa(1), genrsa(1)

Name

dsaparam — DSA parameter manipulation and generation

Synopsis

```
openssl dsaparam
[-inform DER|PEM]
[-outform DER|PEM]
[-in filename]
[-out filename]
[-noout]
[-text]
[-C]
[-rand file(s)]
[-genkey]
[-engine id]
[numbits]
```

DESCRIPTION

This command is used to manipulate or generate DSA parameter files.

OPTIONS

-inform DER|PEM

This specifies the input format. The **DER** option uses an ASN1 DER encoded form compatible with RFC2459 (PKIX) DSS-Parms that is a SEQUENCE consisting of p, q and g respectively. The PEM form is the default format: it consists of the **DER** format base64 encoded with additional header and footer lines.

-outform DER|PEM

This specifies the output format, the options have the same meaning as the **-inform** option.

-in filename

This specifies the input filename to read parameters from or standard input if this option is not specified. If the **numbits** parameter is included then this option will be ignored.

-out filename

This specifies the output filename parameters to. Standard output is used if this option is not present. The output filename should **not** be the same as the input filename.

-noout

this option inhibits the output of the encoded version of the parameters.

-text

this option prints out the DSA parameters in human readable form.

-C

this option converts the parameters into C code. The parameters can then be loaded by calling the **get_dsaXXX()** function.

-genkey

this option will generate a DSA either using the specified or generated parameters.

-rand file(s)

a file or files containing random data used to seed the random number generator, or an EGD socket (see RAND_egd(3)). Multiple files can be specified separated by a OS-dependent character. The separator is **;** for MS-Windows, **,** for OpenVMS, and **:** for all others.

numbits

this option specifies that a parameter set should be generated of size **numbits**. It must be the last option. If this option is included then the input file (if any) is ignored.

-engine id

specifying an engine (by its unique **id** string) will cause **dsaparam** to attempt to obtain a functional reference to the specified engine, thus initialising it if needed. The engine will then be set as the default for all available algorithms.

NOTES

PEM format DSA parameters use the header and footer lines:

```
-----BEGIN DSA PARAMETERS-----
-----END DSA PARAMETERS-----
```

DSA parameter generation is a slow process and as a result the same set of DSA parameters is often used to generate several distinct keys.

SEE ALSO

gendsa(1), dsa(1), genrsa(1), rsa(1)

Name

ec — EC key processing

Synopsis

```
opensslec
[-inform PEM|DER]
[-outform PEM|DER]
[-in filename]
[-passin arg]
[-out filename]
[-passout arg]
[-des]
[-des3]
[-idea]
[-text]
[-noout]
[-param_out]
[-pubin]
[-pubout]
[-conv_form arg]
[-param_enc arg]
[-engine id]
```

DESCRIPTION

The **ec** command processes EC keys. They can be converted between various forms and their components printed out. **Note** OpenSSL uses the private key format specified in 'SEC 1: Elliptic Curve Cryptography' (http://www.secg.org/). To convert a OpenSSL EC private key into the PKCS#8 private key format use the **pkcs8** command.

COMMAND OPTIONS

-inform DER|PEM

This specifies the input format. The **DER** option with a private key uses an ASN.1 DER encoded SEC1 private key. When used with a public key it uses the SubjectPublicKeyInfo structure as specified in RFC 3280. The **PEM** form is the default format: it consists of the **DER** format base64 encoded with additional header and footer lines. In the case of a private key PKCS#8 format is also accepted.

-outform DER|PEM

This specifies the output format, the options have the same meaning as the **-inform** option.

-in filename

This specifies the input filename to read a key from or standard input if this option is not specified. If the key is encrypted a pass phrase will be prompted for.

-passin arg

the input file password source. For more information about the format of **arg** see the **PASS PHRASE ARGUMENTS** section in openssl(1).

-out filename

This specifies the output filename to write a key to or standard output by is not specified. If any encryption options are set then a pass phrase will be prompted for. The output filename should **not** be the same as the input filename.

-passout arg

the output file password source. For more information about the format of **arg** see the **PASS PHRASE ARGUMENTS** section in openssl(1).

-des|-des3|-idea

These options encrypt the private key with the DES, triple DES, IDEA or any other cipher supported by OpenSSL before outputting it. A pass phrase is prompted for. If none of these options is specified the key is written in plain text. This means that using the **ec** utility to read in an encrypted key with no encryption option can be used to remove the pass phrase from a key, or by setting the encryption options it can be use to add or change the pass phrase. These options can only be used with PEM format output files.

-text

prints out the public, private key components and parameters.

-noout

this option prevents output of the encoded version of the key.

-modulus

this option prints out the value of the public key component of the key.

-pubin

by default a private key is read from the input file: with this option a public key is read instead.

-pubout

by default a private key is output. With this option a public key will be output instead. This option is automatically set if the input is a public key.

-conv_form

This specifies how the points on the elliptic curve are converted into octet strings. Possible values are: **compressed** (the default value), **uncompressed** and **hybrid**. For more information regarding the point conversion forms please read the X9.62 standard. **Note** Due to patent issues the **compressed** option is disabled by default for binary curves and can be enabled by defining the preprocessor macro **OPENSSL_EC_BIN_PT_COMP** at compile time.

-param_enc arg

This specifies how the elliptic curve parameters are encoded. Possible value are: **named_curve**, i.e. the ec parameters are specified by a OID, or **explicit** where the ec parameters are explicitly given (see RFC 3279 for the definition of the EC parameters structures). The default value is **named_curve**. **Note** the **implicitlyCA** alternative ,as specified in RFC 3279, is currently not implemented in OpenSSL.

-engine id

specifying an engine (by its unique **id** string) will cause **ec** to attempt to obtain a functional reference to the specified engine, thus initialising it if needed. The engine will then be set as the default for all available algorithms.

NOTES

The PEM private key format uses the header and footer lines:

```
-----BEGIN EC PRIVATE KEY-----
```

```
-----END EC PRIVATE KEY-----
```

The PEM public key format uses the header and footer lines:

```
-----BEGIN PUBLIC KEY-----
-----END PUBLIC KEY-----
```

EXAMPLES

To encrypt a private key using triple DES:

```
openssl ec -in key.pem -des3 -out keyout.pem
```

To convert a private key from PEM to DER format:

```
openssl ec -in key.pem -outform DER -out keyout.der
```

To print out the components of a private key to standard output:

```
openssl ec -in key.pem -text -noout
```

To just output the public part of a private key:

```
openssl ec -in key.pem -pubout -out pubkey.pem
```

To change the parameters encoding to **explicit**:

```
openssl ec -in key.pem -param_enc explicit -out keyout.pem
```

To change the point conversion form to **compressed**:

```
openssl ec -in key.pem -conv_form compressed -out keyout.pem
```

SEE ALSO

ecparam(1), dsa(1), rsa(1)

HISTORY

The ec command was first introduced in OpenSSL 0.9.8.

AUTHOR

Nils Larsch for the OpenSSL project (http://www.openssl.org).

Name

ecparam — EC parameter manipulation and generation

Synopsis

```
openssl ecparam
[-inform DER|PEM]
[-outform DER|PEM]
[-in filename]
[-out filename]
[-noout]
[-text]
[-C]
[-check]
[-name arg]
[-list_curves]
[-conv_form arg]
[-param_enc arg]
[-no_seed]
[-rand file(s)]
[-genkey]
[-engine id]
```

DESCRIPTION

This command is used to manipulate or generate EC parameter files.

OPTIONS

-inform DER|PEM

This specifies the input format. The **DER** option uses an ASN.1 DER encoded form compatible with RFC 3279 EcpkParameters. The PEM form is the default format: it consists of the **DER** format base64 encoded with additional header and footer lines.

-outform DER|PEM

This specifies the output format, the options have the same meaning as the **-inform** option.

-in filename

This specifies the input filename to read parameters from or standard input if this option is not specified.

-out filename

This specifies the output filename parameters to. Standard output is used if this option is not present. The output filename should **not** be the same as the input filename.

-noout

This option inhibits the output of the encoded version of the parameters.

-text

This option prints out the EC parameters in human readable form.

-C

This option converts the EC parameters into C code. The parameters can then be loaded by calling the **get_ec_group_XXX()** function.

-check

Validate the elliptic curve parameters.

-name arg

Use the EC parameters with the specified 'short' name. Use **-list_curves** to get a list of all currently implemented EC parameters.

-list_curves

If this options is specified **ecparam** will print out a list of all currently implemented EC parameters names and exit.

-conv_form

This specifies how the points on the elliptic curve are converted into octet strings. Possible values are: **compressed** (the default value), **uncompressed** and **hybrid**. For more information regarding the point conversion forms please read the X9.62 standard. **Note** Due to patent issues the **compressed** option is disabled by default for binary curves and can be enabled by defining the preprocessor macro **OPENSSL_EC_BIN_PT_COMP** at compile time.

-param_enc arg

This specifies how the elliptic curve parameters are encoded. Possible value are: **named_curve**, i.e. the ec parameters are specified by a OID, or **explicit** where the ec parameters are explicitly given (see RFC 3279 for the definition of the EC parameters structures). The default value is **named_curve**. **Note** the **implicitlyCA** alternative ,as specified in RFC 3279, is currently not implemented in OpenSSL.

-no_seed

This option inhibits that the 'seed' for the parameter generation is included in the ECParameters structure (see RFC 3279).

-genkey

This option will generate a EC private key using the specified parameters.

-rand file(s)

a file or files containing random data used to seed the random number generator, or an EGD socket (see RAND_egd(3)). Multiple files can be specified separated by a OS-dependent character. The separator is **;** for MS-Windows, **,** for OpenVMS, and **:** for all others.

-engine id

specifying an engine (by its unique **id** string) will cause **ecparam** to attempt to obtain a functional reference to the specified engine, thus initialising it if needed. The engine will then be set as the default for all available algorithms.

NOTES

PEM format EC parameters use the header and footer lines:

```
-----BEGIN EC PARAMETERS-----
-----END EC PARAMETERS-----
```

OpenSSL is currently not able to generate new groups and therefore **ecparam** can only create EC parameters from known (named) curves.

EXAMPLES

To create EC parameters with the group 'prime192v1':

```
openssl ecparam -out ec_param.pem -name prime192v1
```

To create EC parameters with explicit parameters:

```
openssl ecparam -out ec_param.pem -name prime192v1 -param_enc explicit
```

To validate given EC parameters:

```
openssl ecparam -in ec_param.pem -check
```

To create EC parameters and a private key:

```
openssl ecparam -out ec_key.pem -name prime192v1 -genkey
```

To change the point encoding to 'compressed':

```
openssl ecparam -in ec_in.pem -out ec_out.pem -conv_form compressed
```

To print out the EC parameters to standard output:

```
openssl ecparam -in ec_param.pem -noout -text
```

SEE ALSO

ec(1), dsaparam(1)

HISTORY

The ecparam command was first introduced in OpenSSL 0.9.8.

AUTHOR

Nils Larsch for the OpenSSL project (http://www.openssl.org)

Name

enc — symmetric cipher routines

Synopsis

```
openssl enc -ciphername
[-in filename]
[-out filename]
[-pass arg]
[-e]
[-d]
[-a/-base64]
[-A]
[-k password]
[-kfile filename]
[-K key]
[-iv IV]
[-S salt]
[-salt]
[-nosalt]
[-z]
[-md]
[-p]
[-P]
[-bufsize number]
[-nopad]
[-debug]
[-none]
[-engine id]
```

DESCRIPTION

The symmetric cipher commands allow data to be encrypted or decrypted using various block and stream ciphers using keys based on passwords or explicitly provided. Base64 encoding or decoding can also be performed either by itself or in addition to the encryption or decryption.

OPTIONS

-in filename

the input filename, standard input by default.

-out filename

the output filename, standard output by default.

-pass arg

the password source. For more information about the format of **arg** see the **PASS PHRASE ARGUMENTS** section in openssl(1).

-salt

use a salt in the key derivation routines. This is the default.

-nosalt

don't use a salt in the key derivation routines. This option **SHOULD NOT** be used except for test purposes or compatibility with ancient versions of OpenSSL and SSLeay.

-e

encrypt the input data: this is the default.

-d

decrypt the input data.

-a

base64 process the data. This means that if encryption is taking place the data is base64 encoded after encryption. If decryption is set then the input data is base64 decoded before being decrypted.

-base64

same as **-a**

-A

if the **-a** option is set then base64 process the data on one line.

-k password

the password to derive the key from. This is for compatibility with previous versions of OpenSSL. Superseded by the **-pass** argument.

-kfile filename

read the password to derive the key from the first line of **filename**. This is for compatibility with previous versions of OpenSSL. Superseded by the **-pass** argument.

-nosalt

do not use a salt

-salt

use salt (randomly generated or provide with **-S** option) when encrypting (this is the default).

-S salt

the actual salt to use: this must be represented as a string of hex digits.

-K key

the actual key to use: this must be represented as a string comprised only of hex digits. If only the key is specified, the IV must additionally specified using the **-iv** option. When both a key and a password are specified, the key given with the **-K** option will be used and the IV generated from the password will be taken. It probably does not make much sense to specify both key and password.

-iv IV

the actual IV to use: this must be represented as a string comprised only of hex digits. When only the key is specified using the **-K** option, the IV must explicitly be defined. When a password is being specified using one of the other options, the IV is generated from this password.

-p

print out the key and IV used.

-P

print out the key and IV used then immediately exit: don't do any encryption or decryption.

-bufsize number

set the buffer size for I/O

-nopad

disable standard block padding

-debug

debug the BIOs used for I/O.

-z

Compress or decompress clear text using zlib before encryption or after decryption. This option exists only if OpenSSL with compiled with zlib or zlib-dynamic option.

-none

Use NULL cipher (no encryption or decryption of input).

NOTES

The program can be called either as **openssl ciphername** or **openssl enc -ciphername**. But the first form doesn't work with engine-provided ciphers, because this form is processed before the configuration file is read and any ENGINEs loaded.

Engines which provide entirely new encryption algorithms (such as ccgost engine which provides gost89 algorithm) should be configured in the configuration file. Engines, specified in the command line using -engine options can only be used for hadrware-assisted implementations of ciphers, which are supported by OpenSSL core or other engine, specified in the configuration file.

When enc command lists supported ciphers, ciphers provided by engines, specified in the configuration files are listed too.

A password will be prompted for to derive the key and IV if necessary.

The **-salt** option should **ALWAYS** be used if the key is being derived from a password unless you want compatibility with previous versions of OpenSSL and SSLeay.

Without the **-salt** option it is possible to perform efficient dictionary attacks on the password and to attack stream cipher encrypted data. The reason for this is that without the salt the same password always generates the same encryption key. When the salt is being used the first eight bytes of the encrypted data are reserved for the salt: it is generated at random when encrypting a file and read from the encrypted file when it is decrypted.

Some of the ciphers do not have large keys and others have security implications if not used correctly. A beginner is advised to just use a strong block cipher in CBC mode such as bf or des3.

All the block ciphers normally use PKCS#5 padding also known as standard block padding: this allows a rudimentary integrity or password check to be performed. However since the chance of random data passing the test is better than 1 in 256 it isn't a very good test.

If padding is disabled then the input data must be a multiple of the cipher block length.

All RC2 ciphers have the same key and effective key length.

Blowfish and RC5 algorithms use a 128 bit key.

SUPPORTED CIPHERS

Note that some of these ciphers can be disabled at compile time and some are available only if an appropriate engine is configured in the configuration file. The output of the **enc** command run with unsupported options (for example **openssl enc -help**) includes a list of ciphers, supported by your versesion of OpenSSL, including ones provided by configured engines.

The **enc** program does not support authenticated encryption modes like CCM and GCM. The utility does not store or retrieve the authentication tag.

```
base64             Base 64

bf-cbc             Blowfish in CBC mode
bf                 Alias for bf-cbc
bf-cfb             Blowfish in CFB mode
bf-ecb             Blowfish in ECB mode
bf-ofb             Blowfish in OFB mode

cast-cbc           CAST in CBC mode
cast               Alias for cast-cbc
cast5-cbc          CAST5 in CBC mode
cast5-cfb          CAST5 in CFB mode
cast5-ecb          CAST5 in ECB mode
cast5-ofb          CAST5 in OFB mode

des-cbc            DES in CBC mode
des                Alias for des-cbc
des-cfb            DES in CBC mode
des-ofb            DES in OFB mode
des-ecb            DES in ECB mode

des-ede-cbc        Two key triple DES EDE in CBC mode
des-ede            Two key triple DES EDE in ECB mode
des-ede-cfb        Two key triple DES EDE in CFB mode
des-ede-ofb        Two key triple DES EDE in OFB mode

des-ede3-cbc       Three key triple DES EDE in CBC mode
des-ede3           Three key triple DES EDE in ECB mode
des3               Alias for des-ede3-cbc
des-ede3-cfb       Three key triple DES EDE CFB mode
des-ede3-ofb       Three key triple DES EDE in OFB mode

desx               DESX algorithm.

gost89             GOST 28147-89 in CFB mode (provided by ccgost engine)
gost89-cnt        `GOST 28147-89 in CNT mode (provided by ccgost engine)

idea-cbc           IDEA algorithm in CBC mode
idea               same as idea-cbc
idea-cfb           IDEA in CFB mode
idea-ecb           IDEA in ECB mode
idea-ofb           IDEA in OFB mode

rc2-cbc            128 bit RC2 in CBC mode
rc2                Alias for rc2-cbc
rc2-cfb            128 bit RC2 in CFB mode
rc2-ecb            128 bit RC2 in ECB mode
rc2-ofb            128 bit RC2 in OFB mode
rc2-64-cbc         64 bit RC2 in CBC mode
rc2-40-cbc         40 bit RC2 in CBC mode

rc4                128 bit RC4
rc4-64             64 bit RC4
rc4-40             40 bit RC4
```

```
rc5-cbc            RC5 cipher in CBC mode
rc5                Alias for rc5-cbc
rc5-cfb            RC5 cipher in CFB mode
rc5-ecb            RC5 cipher in ECB mode
rc5-ofb            RC5 cipher in OFB mode

aes-[128|192|256]-cbc  128/192/256 bit AES in CBC mode
aes-[128|192|256]      Alias for aes-[128|192|256]-cbc
aes-[128|192|256]-cfb  128/192/256 bit AES in 128 bit CFB mode
aes-[128|192|256]-cfb1 128/192/256 bit AES in 1 bit CFB mode
aes-[128|192|256]-cfb8 128/192/256 bit AES in 8 bit CFB mode
aes-[128|192|256]-ecb  128/192/256 bit AES in ECB mode
aes-[128|192|256]-ofb  128/192/256 bit AES in OFB mode
```

EXAMPLES

Just base64 encode a binary file:

```
openssl base64 -in file.bin -out file.b64
```

Decode the same file

```
openssl base64 -d -in file.b64 -out file.bin
```

Encrypt a file using triple DES in CBC mode using a prompted password:

```
openssl des3 -salt -in file.txt -out file.des3
```

Decrypt a file using a supplied password:

```
openssl des3 -d -salt -in file.des3 -out file.txt -k mypassword
```

Encrypt a file then base64 encode it (so it can be sent via mail for example) using Blowfish in CBC mode:

```
openssl bf -a -salt -in file.txt -out file.bf
```

Base64 decode a file then decrypt it:

```
openssl bf -d -salt -a -in file.bf -out file.txt
```

Decrypt some data using a supplied 40 bit RC4 key:

```
openssl rc4-40 -in file.rc4 -out file.txt -K 0102030405
```

BUGS

The **-A** option when used with large files doesn't work properly.

There should be an option to allow an iteration count to be included.

The **enc** program only supports a fixed number of algorithms with certain parameters. So if, for example, you want to use RC2 with a 76 bit key or RC4 with an 84 bit key you can't use this program.

Name

errstr — lookup error codes

Synopsis

```
openssl errstr error_code
```

DESCRIPTION

Sometimes an application will not load error message and only numerical forms will be available. The **errstr** utility can be used to display the meaning of the hex code. The hex code is the hex digits after the second colon.

EXAMPLE

The error code:

```
27594:error:2006D080:lib(32):func(109):reason(128):bss_file.c:107:
```

can be displayed with:

```
openssl errstr 2006D080
```

to produce the error message:

```
error:2006D080:BIO routines:BIO_new_file:no such file
```

SEE ALSO

err(3), ERR_load_crypto_strings(3), SSL_load_error_strings(3)

Name

gendsa — generate a DSA private key from a set of parameters

Synopsis

```
opensslgendsa
[-out filename]
[-aes128]
[-aes192]
[-aes256]
[-camellia128]
[-camellia192]
[-camellia256]
[-des]
[-des3]
[-idea]
[-rand file(s)]
[-engine id]
[paramfile]
```

DESCRIPTION

The **gendsa** command generates a DSA private key from a DSA parameter file (which will be typically generated by the **openssl dsaparam** command).

OPTIONS

-aes128|-aes192|-aes256|-camellia128|-camellia192|-camellia256|-des|-des3|-idea

These options encrypt the private key with specified cipher before outputting it. A pass phrase is prompted for. If none of these options is specified no encryption is used.

-rand file(s)

a file or files containing random data used to seed the random number generator, or an EGD socket (see RAND_egd(3)). Multiple files can be specified separated by a OS-dependent character. The separator is **;** for MS-Windows, **,** for OpenVMS, and **:** for all others.

-engine id

specifying an engine (by its unique **id** string) will cause **gendsa** to attempt to obtain a functional reference to the specified engine, thus initialising it if needed. The engine will then be set as the default for all available algorithms.

paramfile

This option specifies the DSA parameter file to use. The parameters in this file determine the size of the private key. DSA parameters can be generated and examined using the **openssl dsaparam** command.

NOTES

DSA key generation is little more than random number generation so it is much quicker that RSA key generation for example.

SEE ALSO

dsaparam(1), dsa(1), genrsa(1), rsa(1)

Name

genpkey — generate a private key

Synopsis

```
opensslgenpkey
[-out filename]
[-outform PEM|DER]
[-pass arg]
[-cipher]
[-engine id]
[-paramfile file]
[-algorithm alg]
[-pkeyopt opt:value]
[-genparam]
[-text]
```

DESCRIPTION

The **genpkey** command generates a private key.

OPTIONS

-out filename

the output filename. If this argument is not specified then standard output is used.

-outform DER|PEM

This specifies the output format DER or PEM.

-pass arg

the output file password source. For more information about the format of **arg** see the **PASS PHRASE ARGUMENTS** section in openssl(1).

-cipher

This option encrypts the private key with the supplied cipher. Any algorithm name accepted by EVP_get_cipherbyname() is acceptable such as **des3**.

-engine id

specifying an engine (by its unique **id** string) will cause **genpkey** to attempt to obtain a functional reference to the specified engine, thus initialising it if needed. The engine will then be set as the default for all available algorithms. If used this option should precede all other options.

-algorithm alg

public key algorithm to use such as RSA, DSA or DH. If used this option must precede any **-pkeyopt** options. The options **-paramfile** and **-algorithm** are mutually exclusive.

-pkeyopt opt:value

set the public key algorithm option **opt** to **value**. The precise set of options supported depends on the public key algorithm used and its implementation. See **KEY GENERATION OPTIONS** below for more details.

-genparam

generate a set of parameters instead of a private key. If used this option must precede and **-algorithm**, **-paramfile** or **-pkeyopt** options.

-paramfile filename

Some public key algorithms generate a private key based on a set of parameters. They can be supplied using this option. If this option is used the public key algorithm used is determined by the parameters. If used this option must precede and **-pkeyopt** options. The options **-paramfile** and **-algorithm** are mutually exclusive.

-text

Print an (unencrypted) text representation of private and public keys and parameters along with the PEM or DER structure.

KEY GENERATION OPTIONS

The options supported by each algorith and indeed each implementation of an algorithm can vary. The options for the OpenSSL implementations are detailed below.

RSA KEY GENERATION OPTIONS

rsa_keygen_bits:numbits

The number of bits in the generated key. If not specified 1024 is used.

rsa_keygen_pubexp:value

The RSA public exponent value. This can be a large decimal or hexadecimal value if preceded by **0x**. Default value is 65537.

DSA PARAMETER GENERATION OPTIONS

dsa_paramgen_bits:numbits

The number of bits in the generated parameters. If not specified 1024 is used.

DH PARAMETER GENERATION OPTIONS

dh_paramgen_prime_len:numbits

The number of bits in the prime parameter **p**.

dh_paramgen_generator:value

The value to use for the generator **g**.

EC PARAMETER GENERATION OPTIONS

ec_paramgen_curve:curve

the EC curve to use.

GOST2001 KEY GENERATION AND PARAMETER OPTIONS

Gost 2001 support is not enabled by default. To enable this algorithm, one should load the ccgost engine in the OpenSSL configuration file. See README.gost file in the engines/ccgost directiry of the source distribution for more details.

Use of a parameter file for the GOST R 34.10 algorithm is optional. Parameters can be specified during key generation directly as well as during generation of parameter file.

paramset:name

Specifies GOST R 34.10-2001 parameter set according to RFC 4357. Parameter set can be specified using abbreviated name, object short name or numeric OID. Following parameter sets are supported:

paramset	OID	Usage
A	1.2.643.2.2.35.1	Signature
B	1.2.643.2.2.35.2	Signature
C	1.2.643.2.2.35.3	Signature
XA	1.2.643.2.2.36.0	Key exchange
XB	1.2.643.2.2.36.1	Key exchange
test	1.2.643.2.2.35.0	Test purposes

NOTES

The use of the genpkey program is encouraged over the algorithm specific utilities because additional algorithm options and ENGINE provided algorithms can be used.

EXAMPLES

Generate an RSA private key using default parameters:

```
openssl genpkey -algorithm RSA -out key.pem
```

Encrypt output private key using 128 bit AES and the passphrase "hello":

```
openssl genpkey -algorithm RSA -out key.pem -aes-128-cbc -pass pass:hello
```

Generate a 2048 bit RSA key using 3 as the public exponent:

```
openssl genpkey -algorithm RSA -out key.pem -pkeyopt rsa_keygen_bits:2048 \
                                               -pkeyopt rsa_keygen_pubexp:3
```

Generate 1024 bit DSA parameters:

```
openssl genpkey -genparam -algorithm DSA -out dsap.pem \
                                               -pkeyopt dsa_paramgen_bits:1024
```

Generate DSA key from parameters:

```
openssl genpkey -paramfile dsap.pem -out dsakey.pem
```

Generate 1024 bit DH parameters:

```
openssl genpkey -genparam -algorithm DH -out dhp.pem \
                                       -pkeyopt dh_paramgen_prime_len:1024
```

Generate DH key from parameters:

```
openssl genpkey -paramfile dhp.pem -out dhkey.pem
```

Name

genrsa — generate an RSA private key

Synopsis

```
opensslgenrsa
[-out filename]
[-passout arg]
[-aes128]
[-aes192]
[-aes256]
[-camellia128]
[-camellia192]
[-camellia256]
[-des]
[-des3]
[-idea]
[-f4]
[-3]
[-rand file(s)]
[-engine id]
[numbits]
```

DESCRIPTION

The **genrsa** command generates an RSA private key.

OPTIONS

-out filename

the output filename. If this argument is not specified then standard output is used.

-passout arg

the output file password source. For more information about the format of **arg** see the **PASS PHRASE ARGUMENTS** section in openssl(1).

-aes128|-aes192|-aes256|-camellia128|-camellia192|-camellia256|-des|-des3|-idea

These options encrypt the private key with specified cipher before outputting it. If none of these options is specified no encryption is used. If encryption is used a pass phrase is prompted for if it is not supplied via the **-passout** argument.

-F4|-3

the public exponent to use, either 65537 or 3. The default is 65537.

-rand file(s)

a file or files containing random data used to seed the random number generator, or an EGD socket (see RAND_egd(3)). Multiple files can be specified separated by a OS-dependent character. The separator is **;** for MS-Windows, **,** for OpenVMS, and **:** for all others.

-engine id

specifying an engine (by its unique **id** string) will cause **genrsa** to attempt to obtain a functional reference to the specified engine, thus initialising it if needed. The engine will then be set as the default for all available algorithms.

numbits

the size of the private key to generate in bits. This must be the last option specified. The default is 512.

NOTES

RSA private key generation essentially involves the generation of two prime numbers. When generating a private key various symbols will be output to indicate the progress of the generation. A **.** represents each number which has passed an initial sieve test, **+** means a number has passed a single round of the Miller-Rabin primality test. A newline means that the number has passed all the prime tests (the actual number depends on the key size).

Because key generation is a random process the time taken to generate a key may vary somewhat.

BUGS

A quirk of the prime generation algorithm is that it cannot generate small primes. Therefore the number of bits should not be less that 64. For typical private keys this will not matter because for security reasons they will be much larger (typically 1024 bits).

SEE ALSO

gendsa(1)

Name

nseq — create or examine a netscape certificate sequence

Synopsis

```
opensslnseq
[-in filename]
[-out filename]
[-toseq]
```

DESCRIPTION

The **nseq** command takes a file containing a Netscape certificate sequence and prints out the certificates contained in it or takes a file of certificates and converts it into a Netscape certificate sequence.

COMMAND OPTIONS

-in filename

This specifies the input filename to read or standard input if this option is not specified.

-out filename

specifies the output filename or standard output by default.

-toseq

normally a Netscape certificate sequence will be input and the output is the certificates contained in it. With the **-toseq** option the situation is reversed: a Netscape certificate sequence is created from a file of certificates.

EXAMPLES

Output the certificates in a Netscape certificate sequence

```
openssl nseq -in nseq.pem -out certs.pem
```

Create a Netscape certificate sequence

```
openssl nseq -in certs.pem -toseq -out nseq.pem
```

NOTES

The **PEM** encoded form uses the same headers and footers as a certificate:

```
-----BEGIN CERTIFICATE-----
-----END CERTIFICATE-----
```

A Netscape certificate sequence is a Netscape specific form that can be sent to browsers as an alternative to the standard PKCS#7 format when several certificates are sent to the browser: for example during certificate enrollment. It is used by Netscape certificate server for example.

BUGS

This program needs a few more options: like allowing DER or PEM input and output files and allowing multiple certificate files to be used.

Name

ocsp — Online Certificate Status Protocol utility

Synopsis

```
opensslocsp
[-out file]
[-issuer file]
[-cert file]
[-serial n]
[-signer file]
[-signkey file]
[-sign_other file]
[-no_certs]
[-req_text]
[-resp_text]
[-text]
[-reqout file]
[-respout file]
[-reqin file]
[-respin file]
[-nonce]
[-no_nonce]
[-url URL]
[-host host:n]
[-path]
[-CApath dir]
[-CAfile file]
[-no_alt_chains]]
[-VAfile file]
[-validity_period n]
[-status_age n]
[-noverify]
[-verify_other file]
[-trust_other]
[-no_intern]
[-no_signature_verify]
[-no_cert_verify]
[-no_chain]
[-no_cert_checks]
[-no_explicit]
[-port num]
[-index file]
[-CA file]
[-rsigner file]
[-rkey file]
[-rother file]
[-resp_no_certs]
[-nmin n]
[-ndays n]
[-resp_key_id]
[-nrequest n]
[-md5|-sha1|…]
```

DESCRIPTION

The Online Certificate Status Protocol (OCSP) enables applications to determine the (revocation) state of an identified certificate (RFC 2560).

The **ocsp** command performs many common OCSP tasks. It can be used to print out requests and responses, create requests and send queries to an OCSP responder and behave like a mini OCSP server itself.

OCSP CLIENT OPTIONS

-out filename

specify output filename, default is standard output.

-issuer filename

This specifies the current issuer certificate. This option can be used multiple times. The certificate specified in **filename** must be in PEM format. This option **MUST** come before any **-cert** options.

-cert filename

Add the certificate **filename** to the request. The issuer certificate is taken from the previous **issuer** option, or an error occurs if no issuer certificate is specified.

-serial num

Same as the **cert** option except the certificate with serial number **num** is added to the request. The serial number is interpreted as a decimal integer unless preceded by **0x**. Negative integers can also be specified by preceding the value by a **-** sign.

-signer filename, **-signkey filename**

Sign the OCSP request using the certificate specified in the **signer** option and the private key specified by the **signkey** option. If the **signkey** option is not present then the private key is read from the same file as the certificate. If neither option is specified then the OCSP request is not signed.

-sign_other filename

Additional certificates to include in the signed request.

-nonce, **-no_nonce**

Add an OCSP nonce extension to a request or disable OCSP nonce addition. Normally if an OCSP request is input using the **respin** option no nonce is added: using the **nonce** option will force addition of a nonce. If an OCSP request is being created (using **cert** and **serial** options) a nonce is automatically added specifying **no_nonce** overrides this.

-req_text, **-resp_text**, **-text**

print out the text form of the OCSP request, response or both respectively.

-reqout file, **-respout file**

write out the DER encoded certificate request or response to **file**.

-reqin file, **-respin file**

read OCSP request or response file from **file**. These option are ignored if OCSP request or response creation is implied by other options (for example with **serial**, **cert** and **host** options).

-url responder_url

specify the responder URL. Both HTTP and HTTPS (SSL/TLS) URLs can be specified.

-host hostname:port, **-path pathname**

if the **host** option is present then the OCSP request is sent to the host **hostname** on port **port**. **path** specifies the HTTP path name to use or "/" by default.

-timeout seconds

connection timeout to the OCSP responder in seconds

-CAfile file, **-CApath pathname**

file or pathname containing trusted CA certificates. These are used to verify the signature on the OCSP response.

-no_alt_chains

See **verify** manual page for details.

-verify_other file

file containing additional certificates to search when attempting to locate the OCSP response signing certificate. Some responders omit the actual signer's certificate from the response: this option can be used to supply the necessary certificate in such cases.

-trust_other

the certificates specified by the **-verify_other** option should be explicitly trusted and no additional checks will be performed on them. This is useful when the complete responder certificate chain is not available or trusting a root CA is not appropriate.

-VAfile file

file containing explicitly trusted responder certificates. Equivalent to the **-verify_other** and **-trust_other** options.

-noverify

don't attempt to verify the OCSP response signature or the nonce values. This option will normally only be used for debugging since it disables all verification of the responders certificate.

-no_intern

ignore certificates contained in the OCSP response when searching for the signers certificate. With this option the signers certificate must be specified with either the **-verify_other** or **-VAfile** options.

-no_signature_verify

don't check the signature on the OCSP response. Since this option tolerates invalid signatures on OCSP responses it will normally only be used for testing purposes.

-no_cert_verify

don't verify the OCSP response signers certificate at all. Since this option allows the OCSP response to be signed by any certificate it should only be used for testing purposes.

-no_chain

do not use certificates in the response as additional untrusted CA certificates.

-no_explicit

do not explicitly trust the root CA if it is set to be trusted for OCSP signing.

-no_cert_checks

don't perform any additional checks on the OCSP response signers certificate. That is do not make any checks to see if the signers certificate is authorised to provide the necessary status information: as a result this option should only be used for testing purposes.

-validity_period nsec, **-status_age age**

these options specify the range of times, in seconds, which will be tolerated in an OCSP response. Each certificate status response includes a **notBefore** time and an optional **notAfter** time. The current time should fall between these two values, but the interval between the two times may be only a few seconds. In practice the OCSP responder and clients clocks may not be precisely synchronised and so such a check may fail. To avoid this the **-validity_period** option can be used to specify an acceptable error range in seconds, the default value is 5 minutes.

If the **notAfter** time is omitted from a response then this means that new status information is immediately available. In this case the age of the **notBefore** field is checked to see it is not older than **age** seconds old. By default this additional check is not performed.

-md5|-sha1|-sha256|-ripemod160|…

this option sets digest algorithm to use for certificate identification in the OCSP request. By default SHA-1 is used.

OCSP SERVER OPTIONS

-index indexfile

indexfile is a text index file in **ca** format containing certificate revocation information.

If the **index** option is specified the **ocsp** utility is in responder mode, otherwise it is in client mode. The request(s) the responder processes can be either specified on the command line (using **issuer** and **serial** options), supplied in a file (using the **respin** option) or via external OCSP clients (if **port** or **url** is specified).

If the **index** option is present then the **CA** and **rsigner** options must also be present.

-CA file

CA certificate corresponding to the revocation information in **indexfile**.

-rsigner file

The certificate to sign OCSP responses with.

-rother file

Additional certificates to include in the OCSP response.

-resp_no_certs

Don't include any certificates in the OCSP response.

-resp_key_id

Identify the signer certificate using the key ID, default is to use the subject name.

-rkey file

The private key to sign OCSP responses with: if not present the file specified in the **rsigner** option is used.

-port portnum

Port to listen for OCSP requests on. The port may also be specified using the **url** option.

-nrequest number

The OCSP server will exit after receiving **number** requests, default unlimited.

-nmin minutes, **-ndays days**

Number of minutes or days when fresh revocation information is available: used in the **nextUpdate** field. If neither option is present then the **nextUpdate** field is omitted meaning fresh revocation information is immediately available.

OCSP Response verification.

OCSP Response follows the rules specified in RFC2560.

Initially the OCSP responder certificate is located and the signature on the OCSP request checked using the responder certificate's public key.

Then a normal certificate verify is performed on the OCSP responder certificate building up a certificate chain in the process. The locations of the trusted certificates used to build the chain can be specified by the **CAfile** and **CApath** options or they will be looked for in the standard OpenSSL certificates directory.

If the initial verify fails then the OCSP verify process halts with an error.

Otherwise the issuing CA certificate in the request is compared to the OCSP responder certificate: if there is a match then the OCSP verify succeeds.

Otherwise the OCSP responder certificate's CA is checked against the issuing CA certificate in the request. If there is a match and the OCSPSigning extended key usage is present in the OCSP responder certificate then the OCSP verify succeeds.

Otherwise, if **-no_explicit** is **not** set the root CA of the OCSP responders CA is checked to see if it is trusted for OCSP signing. If it is the OCSP verify succeeds.

If none of these checks is successful then the OCSP verify fails.

What this effectively means if that if the OCSP responder certificate is authorised directly by the CA it is issuing revocation information about (and it is correctly configured) then verification will succeed.

If the OCSP responder is a "global responder" which can give details about multiple CAs and has its own separate certificate chain then its root CA can be trusted for OCSP signing. For example:

```
openssl x509 -in ocspCA.pem -addtrust OCSPSigning -out trustedCA.pem
```

Alternatively the responder certificate itself can be explicitly trusted with the **-VAfile** option.

NOTES

As noted, most of the verify options are for testing or debugging purposes. Normally only the **-CApath**, **-CAfile** and (if the responder is a 'global VA') **-VAfile** options need to be used.

The OCSP server is only useful for test and demonstration purposes: it is not really usable as a full OCSP responder. It contains only a very simple HTTP request handling and can only handle the POST form of OCSP queries. It also handles requests serially meaning it cannot respond to new requests until it has processed the current one. The text index file format of revocation is also inefficient for large quantities of revocation data.

It is possible to run the **ocsp** application in responder mode via a CGI script using the **respin** and **respout** options.

EXAMPLES

Create an OCSP request and write it to a file:

```
openssl ocsp -issuer issuer.pem -cert c1.pem -cert c2.pem -reqout req.der
```

Send a query to an OCSP responder with URL http://ocsp.myhost.com/ save the response to a file and print it out in text form

```
openssl ocsp -issuer issuer.pem -cert c1.pem -cert c2.pem \
    -url http://ocsp.myhost.com/ -resp_text -respout resp.der
```

Read in an OCSP response and print out text form:

```
openssl ocsp -respin resp.der -text
```

OCSP server on port 8888 using a standard **ca** configuration, and a separate responder certificate. All requests and responses are printed to a file.

```
openssl ocsp -index demoCA/index.txt -port 8888 -rsigner rcert.pem -CA demoCA/cacert.pem \
       -text -out log.txt
```

As above but exit after processing one request:

```
openssl ocsp -index demoCA/index.txt -port 8888 -rsigner rcert.pem -CA demoCA/cacert.pem \
    -nrequest 1
```

Query status information using internally generated request:

```
openssl ocsp -index demoCA/index.txt -rsigner rcert.pem -CA demoCA/cacert.pem \
    -issuer demoCA/cacert.pem -serial 1
```

Query status information using request read from a file, write response to a second file.

```
openssl ocsp -index demoCA/index.txt -rsigner rcert.pem -CA demoCA/cacert.pem \
    -reqin req.der -respout resp.der
```

HISTORY

The -no_alt_chains options was first added to OpenSSL 1.0.1n and 1.0.2b.

Name

passwd — compute password hashes

Synopsis

```
openssl passwd
[-crypt]
[-1]
[-apr1]
[-salt string]
[-in file]
[-stdin]
[-noverify]
[-quiet]
[-table]
{password}
```

DESCRIPTION

The **passwd** command computes the hash of a password typed at run-time or the hash of each password in a list. The password list is taken from the named file for option **-in file**, from stdin for option **-stdin**, or from the command line, or from the terminal otherwise. The Unix standard algorithm **crypt** and the MD5-based BSD password algorithm **1** and its Apache variant **apr1** are available.

OPTIONS

-crypt

Use the **crypt** algorithm (default).

-1

Use the MD5 based BSD password algorithm **1**.

-apr1

Use the **apr1** algorithm (Apache variant of the BSD algorithm).

-salt *string*

Use the specified salt. When reading a password from the terminal, this implies **-noverify**.

-in *file*

Read passwords from *file*.

-stdin

Read passwords from **stdin**.

-noverify

Don't verify when reading a password from the terminal.

-quiet

Don't output warnings when passwords given at the command line are truncated.

-table

In the output list, prepend the cleartext password and a TAB character to each password hash.

EXAMPLES

openssl passwd -crypt -salt xx password prints **xxj31ZMTZzkVA**.

openssl passwd -1 -salt xxxxxxxx password prints **$1$xxxxxxxx$UYCIxa628.9qXjpQCjM4a.**.

openssl passwd -apr1 -salt xxxxxxxx password prints **$apr1$xxxxxxxx$dxHfLAsjHkDRmG83UXe8K0**.

Name

pkcs12 — PKCS#12 file utility

Synopsis

```
opensslpkcs12
[-export]
[-chain]
[-inkey filename]
[-certfile filename]
[-name name]
[-caname name]
[-in filename]
[-out filename]
[-noout]
[-nomacver]
[-nocerts]
[-clcerts]
[-cacerts]
[-nokeys]
[-info]
[-des|-des3|-idea|-aes128|-aes192|-aes256|-camellia128|-camellia192|-camellia256|-nodes]
[-noiter]
[-maciter | -nomaciter | -nomac]
[-twopass]
[-descert]
[-certpbe cipher]
[-keypbe cipher]
[-macalg digest]
[-keyex]
[-keysig]
[-password arg]
[-passin arg]
[-passout arg]
[-rand file(s)]
[-CAfile file]
[-CApath dir]
[-CSP name]
```

DESCRIPTION

The **pkcs12** command allows PKCS#12 files (sometimes referred to as PFX files) to be created and parsed. PKCS#12 files are used by several programs including Netscape, MSIE and MS Outlook.

COMMAND OPTIONS

There are a lot of options the meaning of some depends of whether a PKCS#12 file is being created or parsed. By default a PKCS#12 file is parsed. A PKCS#12 file can be created by using the **-export** option (see below).

PARSING OPTIONS

-in filename

This specifies filename of the PKCS#12 file to be parsed. Standard input is used by default.

-out filename

The filename to write certificates and private keys to, standard output by default. They are all written in PEM format.

-passin arg

the PKCS#12 file (i.e. input file) password source. For more information about the format of **arg** see the **PASS PHRASE ARGUMENTS** section in openssl(1).

-passout arg

pass phrase source to encrypt any outputted private keys with. For more information about the format of **arg** see the **PASS PHRASE ARGUMENTS** section in openssl(1).

-password arg

With -export, -password is equivalent to -passout. Otherwise, -password is equivalent to -passin.

-noout

this option inhibits output of the keys and certificates to the output file version of the PKCS#12 file.

-clcerts

only output client certificates (not CA certificates).

-cacerts

only output CA certificates (not client certificates).

-nocerts

no certificates at all will be output.

-nokeys

no private keys will be output.

-info

output additional information about the PKCS#12 file structure, algorithms used and iteration counts.

-des

use DES to encrypt private keys before outputting.

-des3

use triple DES to encrypt private keys before outputting, this is the default.

-idea

use IDEA to encrypt private keys before outputting.

-aes128, **-aes192**, **-aes256**

use AES to encrypt private keys before outputting.

-camellia128, **-camellia192**, **-camellia256**

use Camellia to encrypt private keys before outputting.

-nodes

don't encrypt the private keys at all.

-nomacver

don't attempt to verify the integrity MAC before reading the file.

-twopass

prompt for separate integrity and encryption passwords: most software always assumes these are the same so this option will render such PKCS#12 files unreadable.

FILE CREATION OPTIONS

-export

This option specifies that a PKCS#12 file will be created rather than parsed.

-out filename

This specifies filename to write the PKCS#12 file to. Standard output is used by default.

-in filename

The filename to read certificates and private keys from, standard input by default. They must all be in PEM format. The order doesn't matter but one private key and its corresponding certificate should be present. If additional certificates are present they will also be included in the PKCS#12 file.

-inkey filename

file to read private key from. If not present then a private key must be present in the input file.

-name friendlyname

This specifies the "friendly name" for the certificate and private key. This name is typically displayed in list boxes by software importing the file.

-certfile filename

A filename to read additional certificates from.

-caname friendlyname

This specifies the "friendly name" for other certificates. This option may be used multiple times to specify names for all certificates in the order they appear. Netscape ignores friendly names on other certificates whereas MSIE displays them.

-pass arg, **-passout arg**

the PKCS#12 file (i.e. output file) password source. For more information about the format of **arg** see the **PASS PHRASE ARGUMENTS** section in openssl(1).

-passin password

pass phrase source to decrypt any input private keys with. For more information about the format of **arg** see the **PASS PHRASE ARGUMENTS** section in openssl(1).

-chain

if this option is present then an attempt is made to include the entire certificate chain of the user certificate. The standard CA store is used for this search. If the search fails it is considered a fatal error.

-descert

encrypt the certificate using triple DES, this may render the PKCS#12 file unreadable by some "export grade" software. By default the private key is encrypted using triple DES and the certificate using 40 bit RC2.

-keypbe alg, **-certpbe alg**

these options allow the algorithm used to encrypt the private key and certificates to be selected. Any PKCS#5 v1.5 or PKCS#12 PBE algorithm name can be used (see **NOTES** section for more information). If a a cipher name (as output by the **list-cipher-algorithms** command is specified then it is used with PKCS#5 v2.0. For interoperability reasons it is advisable to only use PKCS#12 algorithms.

-keyex|-keysig

specifies that the private key is to be used for key exchange or just signing. This option is only interpreted by MSIE and similar MS software. Normally "export grade" software will only allow 512 bit RSA keys to be used for encryption purposes but arbitrary length keys for signing. The **-keysig** option marks the key for signing only. Signing only keys can be used for S/MIME signing, authenticode (ActiveX control signing) and SSL client authentication, however due to a bug only MSIE 5.0 and later support the use of signing only keys for SSL client authentication.

-macalg digest

specify the MAC digest algorithm. If not included them SHA1 will be used.

-nomaciter, **-noiter**

these options affect the iteration counts on the MAC and key algorithms. Unless you wish to produce files compatible with MSIE 4.0 you should leave these options alone.

To discourage attacks by using large dictionaries of common passwords the algorithm that derives keys from passwords can have an iteration count applied to it: this causes a certain part of the algorithm to be repeated and slows it down. The MAC is used to check the file integrity but since it will normally have the same password as the keys and certificates it could also be attacked. By default both MAC and encryption iteration counts are set to 2048, using these options the MAC and encryption iteration counts can be set to 1, since this reduces the file security you should not use these options unless you really have to. Most software supports both MAC and key iteration counts. MSIE 4.0 doesn't support MAC iteration counts so it needs the **-nomaciter** option.

-maciter

This option is included for compatibility with previous versions, it used to be needed to use MAC iterations counts but they are now used by default.

-nomac

don't attempt to provide the MAC integrity.

-rand file(s)

a file or files containing random data used to seed the random number generator, or an EGD socket (see RAND_egd(3)). Multiple files can be specified separated by a OS-dependent character. The separator is **;** for MS-Windows, **,** for OpenVMS, and **:** for all others.

-CAfile file

CA storage as a file.

-CApath dir

CA storage as a directory. This directory must be a standard certificate directory: that is a hash of each subject name (using **x509 -hash**) should be linked to each certificate.

-CSP name

write **name** as a Microsoft CSP name.

NOTES

Although there are a large number of options most of them are very rarely used. For PKCS#12 file parsing only **-in** and **-out** need to be used for PKCS#12 file creation **-export** and **-name** are also used.

If none of the **-clcerts**, **-cacerts** or **-nocerts** options are present then all certificates will be output in the order they appear in the input PKCS#12 files. There is no guarantee that the first certificate present is the one corresponding to the private key. Certain software which requires a private key and certificate and assumes the first certificate in the file is the one corresponding to the private key: this may not always be the case. Using the **-clcerts** option will solve this problem by only outputting the certificate corresponding to the private key. If the CA certificates are required then they can be output to a separate file using the **-nokeys -cacerts** options to just output CA certificates.

The **-keypbe** and **-certpbe** algorithms allow the precise encryption algorithms for private keys and certificates to be specified. Normally the defaults are fine but occasionally software can't handle triple DES encrypted private keys, then the option **-keypbe PBE-SHA1-RC2-40** can be used to reduce the private key encryption to 40 bit RC2. A complete description of all algorithms is contained in the **pkcs8** manual page.

EXAMPLES

Parse a PKCS#12 file and output it to a file:

```
openssl pkcs12 -in file.p12 -out file.pem
```

Output only client certificates to a file:

```
openssl pkcs12 -in file.p12 -clcerts -out file.pem
```

Don't encrypt the private key:

```
openssl pkcs12 -in file.p12 -out file.pem -nodes
```

Print some info about a PKCS#12 file:

```
openssl pkcs12 -in file.p12 -info -noout
```

Create a PKCS#12 file:

```
openssl pkcs12 -export -in file.pem -out file.p12 -name "My Certificate"
```

Include some extra certificates:

```
openssl pkcs12 -export -in file.pem -out file.p12 -name "My Certificate" \
 -certfile othercerts.pem
```

BUGS

Some would argue that the PKCS#12 standard is one big bug :-)

Versions of OpenSSL before 0.9.6a had a bug in the PKCS#12 key generation routines. Under rare circumstances this could produce a PKCS#12 file encrypted with an invalid key. As a result some PKCS#12 files which triggered this bug from other implementations (MSIE or Netscape) could not be decrypted by OpenSSL and similarly OpenSSL could produce PKCS#12 files which could not be decrypted by other implementations. The chances of producing such a file are relatively small: less than 1 in 256.

A side effect of fixing this bug is that any old invalidly encrypted PKCS#12 files cannot no longer be parsed by the fixed version. Under such circumstances the **pkcs12** utility will report that the MAC is OK but fail with a decryption error when extracting private keys.

This problem can be resolved by extracting the private keys and certificates from the PKCS#12 file using an older version of OpenSSL and recreating the PKCS#12 file from the keys and certificates using a newer version of OpenSSL. For example:

```
old-openssl -in bad.p12 -out keycerts.pem
openssl -in keycerts.pem -export -name "My PKCS#12 file" -out fixed.p12
```

SEE ALSO

pkcs8(1)

Name

pkcs7 — PKCS#7 utility

Synopsis

```
opensslpkcs7
[-inform PEM|DER]
[-outform PEM|DER]
[-in filename]
[-out filename]
[-print_certs]
[-text]
[-noout]
[-engine id]
```

DESCRIPTION

The **pkcs7** command processes PKCS#7 files in DER or PEM format.

COMMAND OPTIONS

-inform DER|PEM

This specifies the input format. **DER** format is DER encoded PKCS#7 v1.5 structure.**PEM** (the default) is a base64 encoded version of the DER form with header and footer lines.

-outform DER|PEM

This specifies the output format, the options have the same meaning as the **-inform** option.

-in filename

This specifies the input filename to read from or standard input if this option is not specified.

-out filename

specifies the output filename to write to or standard output by default.

-print_certs

prints out any certificates or CRLs contained in the file. They are preceded by their subject and issuer names in one line format.

-text

prints out certificates details in full rather than just subject and issuer names.

-noout

don't output the encoded version of the PKCS#7 structure (or certificates is **-print_certs** is set).

-engine id

specifying an engine (by its unique **id** string) will cause **pkcs7** to attempt to obtain a functional reference to the specified engine, thus initialising it if needed. The engine will then be set as the default for all available algorithms.

EXAMPLES

Convert a PKCS#7 file from PEM to DER:

```
openssl pkcs7 -in file.pem -outform DER -out file.der
```

Output all certificates in a file:

```
openssl pkcs7 -in file.pem -print_certs -out certs.pem
```

NOTES

The PEM PKCS#7 format uses the header and footer lines:

```
-----BEGIN PKCS7-----
-----END PKCS7-----
```

For compatibility with some CAs it will also accept:

```
-----BEGIN CERTIFICATE-----
-----END CERTIFICATE-----
```

RESTRICTIONS

There is no option to print out all the fields of a PKCS#7 file.

This PKCS#7 routines only understand PKCS#7 v 1.5 as specified in RFC2315 they cannot currently parse, for example, the new CMS as described in RFC2630.

SEE ALSO

crl2pkcs7(1)

Name

pkcs8 — PKCS#8 format private key conversion tool

Synopsis

```
opensslpkcs8
[-topk8]
[-inform PEM|DER]
[-outform PEM|DER]
[-in filename]
[-passin arg]
[-out filename]
[-passout arg]
[-noiter]
[-nocrypt]
[-nooct]
[-embed]
[-nsdb]
[-v2 alg]
[-v1 alg]
[-engine id]
```

DESCRIPTION

The **pkcs8** command processes private keys in PKCS#8 format. It can handle both unencrypted PKCS#8 PrivateKeyInfo format and EncryptedPrivateKeyInfo format with a variety of PKCS#5 (v1.5 and v2.0) and PKCS#12 algorithms.

COMMAND OPTIONS

-topk8

Normally a PKCS#8 private key is expected on input and a traditional format private key will be written. With the **-topk8** option the situation is reversed: it reads a traditional format private key and writes a PKCS#8 format key.

-inform DER|PEM

This specifies the input format. If a PKCS#8 format key is expected on input then either a **DER** or **PEM** encoded version of a PKCS#8 key will be expected. Otherwise the **DER** or **PEM** format of the traditional format private key is used.

-outform DER|PEM

This specifies the output format, the options have the same meaning as the **-inform** option.

-in filename

This specifies the input filename to read a key from or standard input if this option is not specified. If the key is encrypted a pass phrase will be prompted for.

-passin arg

the input file password source. For more information about the format of **arg** see the **PASS PHRASE ARGUMENTS** section in openssl(1).

-out filename

This specifies the output filename to write a key to or standard output by default. If any encryption options are set then a pass phrase will be prompted for. The output filename should **not** be the same as the input filename.

-passout arg

the output file password source. For more information about the format of **arg** see the **PASS PHRASE ARGUMENTS** section in openssl(1).

-nocrypt

PKCS#8 keys generated or input are normally PKCS#8 EncryptedPrivateKeyInfo structures using an appropriate password based encryption algorithm. With this option an unencrypted PrivateKeyInfo structure is expected or output. This option does not encrypt private keys at all and should only be used when absolutely necessary. Certain software such as some versions of Java code signing software used unencrypted private keys.

-nooct

This option generates RSA private keys in a broken format that some software uses. Specifically the private key should be enclosed in a OCTET STRING but some software just includes the structure itself without the surrounding OCTET STRING.

-embed

This option generates DSA keys in a broken format. The DSA parameters are embedded inside the PrivateKey structure. In this form the OCTET STRING contains an ASN1 SEQUENCE consisting of two structures: a SEQUENCE containing the parameters and an ASN1 INTEGER containing the private key.

-nsdb

This option generates DSA keys in a broken format compatible with Netscape private key databases. The PrivateKey contains a SEQUENCE consisting of the public and private keys respectively.

-v2 alg

This option enables the use of PKCS#5 v2.0 algorithms. Normally PKCS#8 private keys are encrypted with the password based encryption algorithm called **pbeWithMD5AndDES-CBC** this uses 56 bit DES encryption but it was the strongest encryption algorithm supported in PKCS#5 v1.5. Using the **-v2** option PKCS#5 v2.0 algorithms are used which can use any encryption algorithm such as 168 bit triple DES or 128 bit RC2 however not many implementations support PKCS#5 v2.0 yet. If you are just using private keys with OpenSSL then this doesn't matter.

The **alg** argument is the encryption algorithm to use, valid values include **des**, **des3** and **rc2**. It is recommended that **des3** is used.

-v1 alg

This option specifies a PKCS#5 v1.5 or PKCS#12 algorithm to use. A complete list of possible algorithms is included below.

-engine id

specifying an engine (by its unique **id** string) will cause **pkcs8** to attempt to obtain a functional reference to the specified engine, thus initialising it if needed. The engine will then be set as the default for all available algorithms.

NOTES

The encrypted form of a PEM encode PKCS#8 files uses the following headers and footers:

```
-----BEGIN ENCRYPTED PRIVATE KEY-----
-----END ENCRYPTED PRIVATE KEY-----
```

The unencrypted form uses:

```
-----BEGIN PRIVATE KEY-----
-----END PRIVATE KEY-----
```

Private keys encrypted using PKCS#5 v2.0 algorithms and high iteration counts are more secure that those encrypted using the traditional SSLeay compatible formats. So if additional security is considered important the keys should be converted.

The default encryption is only 56 bits because this is the encryption that most current implementations of PKCS#8 will support.

Some software may use PKCS#12 password based encryption algorithms with PKCS#8 format private keys: these are handled automatically but there is no option to produce them.

It is possible to write out DER encoded encrypted private keys in PKCS#8 format because the encryption details are included at an ASN1 level whereas the traditional format includes them at a PEM level.

PKCS#5 v1.5 and PKCS#12 algorithms.

Various algorithms can be used with the **-v1** command line option, including PKCS#5 v1.5 and PKCS#12. These are described in more detail below.

PBE-MD2-DES PBE-MD5-DES

> These algorithms were included in the original PKCS#5 v1.5 specification. They only offer 56 bits of protection since they both use DES.

PBE-SHA1-RC2-64 PBE-MD2-RC2-64 PBE-MD5-RC2-64 PBE-SHA1-DES

> These algorithms are not mentioned in the original PKCS#5 v1.5 specification but they use the same key derivation algorithm and are supported by some software. They are mentioned in PKCS#5 v2.0. They use either 64 bit RC2 or 56 bit DES.

PBE-SHA1-RC4-128 PBE-SHA1-RC4-40 PBE-SHA1-3DES PBE-SHA1-2DES PBE-SHA1-RC2-128 PBE-SHA1-RC2-40

> These algorithms use the PKCS#12 password based encryption algorithm and allow strong encryption algorithms like triple DES or 128 bit RC2 to be used.

EXAMPLES

Convert a private from traditional to PKCS#5 v2.0 format using triple DES:

```
openssl pkcs8 -in key.pem -topk8 -v2 des3 -out enckey.pem
```

Convert a private key to PKCS#8 using a PKCS#5 1.5 compatible algorithm (DES):

```
openssl pkcs8 -in key.pem -topk8 -out enckey.pem
```

Convert a private key to PKCS#8 using a PKCS#12 compatible algorithm (3DES):

```
openssl pkcs8 -in key.pem -topk8 -out enckey.pem -v1 PBE-SHA1-3DES
```

Read a DER unencrypted PKCS#8 format private key:

```
openssl pkcs8 -inform DER -nocrypt -in key.der -out key.pem
```

Convert a private key from any PKCS#8 format to traditional format:

```
openssl pkcs8 -in pk8.pem -out key.pem
```

STANDARDS

Test vectors from this PKCS#5 v2.0 implementation were posted to the pkcs-tng mailing list using triple DES, DES and RC2 with high iteration counts, several people confirmed that they could decrypt the private keys produced and Therefore it can be assumed that the PKCS#5 v2.0 implementation is reasonably accurate at least as far as these algorithms are concerned.

The format of PKCS#8 DSA (and other) private keys is not well documented: it is hidden away in PKCS#11 v2.01, section 11.9. OpenSSL's default DSA PKCS#8 private key format complies with this standard.

BUGS

There should be an option that prints out the encryption algorithm in use and other details such as the iteration count.

PKCS#8 using triple DES and PKCS#5 v2.0 should be the default private key format for OpenSSL: for compatibility several of the utilities use the old format at present.

SEE ALSO

dsa(1), rsa(1), genrsa(1), gendsa(1)

Name

pkey — public or private key processing tool

Synopsis

```
opensslpkey
[-inform PEM|DER]
[-outform PEM|DER]
[-in filename]
[-passin arg]
[-out filename]
[-passout arg]
[-cipher]
[-text]
[-text_pub]
[-noout]
[-pubin]
[-pubout]
[-engine id]
```

DESCRIPTION

The **pkey** command processes public or private keys. They can be converted between various forms and their components printed out.

COMMAND OPTIONS

-inform DER|PEM

This specifies the input format DER or PEM.

-outform DER|PEM

This specifies the output format, the options have the same meaning as the **-inform** option.

-in filename

This specifies the input filename to read a key from or standard input if this option is not specified. If the key is encrypted a pass phrase will be prompted for.

-passin arg

the input file password source. For more information about the format of **arg** see the **PASS PHRASE ARGUMENTS** section in openssl(1).

-out filename

This specifies the output filename to write a key to or standard output if this option is not specified. If any encryption options are set then a pass phrase will be prompted for. The output filename should **not** be the same as the input filename.

-passout password

the output file password source. For more information about the format of **arg** see the **PASS PHRASE ARGUMENTS** section in openssl(1).

-cipher

These options encrypt the private key with the supplied cipher. Any algorithm name accepted by EVP_get_cipherbyname() is acceptable such as **des3**.

-text

prints out the various public or private key components in plain text in addition to the encoded version.

-text_pub

print out only public key components even if a private key is being processed.

-noout

do not output the encoded version of the key.

-pubin

by default a private key is read from the input file: with this option a public key is read instead.

-pubout

by default a private key is output: with this option a public key will be output instead. This option is automatically set if the input is a public key.

-engine id

specifying an engine (by its unique **id** string) will cause **pkey** to attempt to obtain a functional reference to the specified engine, thus initialising it if needed. The engine will then be set as the default for all available algorithms.

EXAMPLES

To remove the pass phrase on an RSA private key:

```
openssl pkey -in key.pem -out keyout.pem
```

To encrypt a private key using triple DES:

```
openssl pkey -in key.pem -des3 -out keyout.pem
```

To convert a private key from PEM to DER format:

```
openssl pkey -in key.pem -outform DER -out keyout.der
```

To print out the components of a private key to standard output:

```
openssl pkey -in key.pem -text -noout
```

To print out the public components of a private key to standard output:

```
openssl pkey -in key.pem -text_pub -noout
```

To just output the public part of a private key:

```
openssl pkey -in key.pem -pubout -out pubkey.pem
```

SEE ALSO

genpkey(1), rsa(1), pkcs8(1), dsa(1), genrsa(1), gendsa(1)

Name

pkeyparam — public key algorithm parameter processing tool

Synopsis

```
opensslpkeyparam
[-in filename]
[-out filename]
[-text]
[-noout]
[-engine id]
```

DESCRIPTION

The **pkey** command processes public or private keys. They can be converted between various forms and their components printed out.

COMMAND OPTIONS

-in filename

This specifies the input filename to read parameters from or standard input if this option is not specified.

-out filename

This specifies the output filename to write parameters to or standard output if this option is not specified.

-text

prints out the parameters in plain text in addition to the encoded version.

-noout

do not output the encoded version of the parameters.

-engine id

specifying an engine (by its unique **id** string) will cause **pkeyparam** to attempt to obtain a functional reference to the specified engine, thus initialising it if needed. The engine will then be set as the default for all available algorithms.

EXAMPLE

Print out text version of parameters:

```
openssl pkeyparam -in param.pem -text
```

NOTES

There are no **-inform** or **-outform** options for this command because only PEM format is supported because the key type is determined by the PEM headers.

SEE ALSO

genpkey(1), rsa(1), pkcs8(1), dsa(1), genrsa(1), gendsa(1)

Name

pkeyutl — public key algorithm utility

Synopsis

```
opensslpkeyutl
[-in file]
[-out file]
[-sigfile file]
[-inkey file]
[-keyform PEM|DER]
[-passin arg]
[-peerkey file]
[-peerform PEM|DER]
[-pubin]
[-certin]
[-rev]
[-sign]
[-verify]
[-verifyrecover]
[-encrypt]
[-decrypt]
[-derive]
[-pkeyopt opt:value]
[-hexdump]
[-asn1parse]
[-engine id]
```

DESCRIPTION

The **pkeyutl** command can be used to perform public key operations using any supported algorithm.

COMMAND OPTIONS

-in filename

This specifies the input filename to read data from or standard input if this option is not specified.

-out filename

specifies the output filename to write to or standard output by default.

-inkey file

the input key file, by default it should be a private key.

-keyform PEM|DER

the key format PEM, DER or ENGINE.

-passin arg

the input key password source. For more information about the format of **arg** see the **PASS PHRASE ARGUMENTS** section in openssl(1).

-peerkey file

the peer key file, used by key derivation (agreement) operations.

-peerform PEM|DER

the peer key format PEM, DER or ENGINE.

-engine id

specifying an engine (by its unique **id** string) will cause **pkeyutl** to attempt to obtain a functional reference to the specified engine, thus initialising it if needed. The engine will then be set as the default for all available algorithms.

-pubin

the input file is a public key.

-certin

the input is a certificate containing a public key.

-rev

reverse the order of the input buffer. This is useful for some libraries (such as CryptoAPI) which represent the buffer in little endian format.

-sign

sign the input data and output the signed result. This requires a private key.

-verify

verify the input data against the signature file and indicate if the verification succeeded or failed.

-verifyrecover

verify the input data and output the recovered data.

-encrypt

encrypt the input data using a public key.

-decrypt

decrypt the input data using a private key.

-derive

derive a shared secret using the peer key.

-hexdump

hex dump the output data.

-asn1parse

asn1parse the output data, this is useful when combined with the **-verifyrecover** option when an ASN1 structure is signed.

NOTES

The operations and options supported vary according to the key algorithm and its implementation. The OpenSSL operations and options are indicated below.

Unless otherwise mentioned all algorithms support the **digest:alg** option which specifies the digest in use for sign, verify and verifyrecover operations. The value **alg** should represent a digest name as used in the EVP_get_digestbyname() function for example **sha1**.

RSA ALGORITHM

The RSA algorithm supports encrypt, decrypt, sign, verify and verifyrecover operations in general. Some padding modes only support some of these operations however.

-rsa_padding_mode:mode

This sets the RSA padding mode. Acceptable values for **mode** are **pkcs1** for PKCS#1 padding, **sslv23** for SSLv23 padding, **none** for no padding, **oaep** for **OAEP** mode, **x931** for X9.31 mode and **pss** for PSS.

In PKCS#1 padding if the message digest is not set then the supplied data is signed or verified directly instead of using a **DigestInfo** structure. If a digest is set then the a **DigestInfo** structure is used and its the length must correspond to the digest type.

For **oeap** mode only encryption and decryption is supported.

For **x931** if the digest type is set it is used to format the block data otherwise the first byte is used to specify the X9.31 digest ID. Sign, verify and verifyrecover are can be performed in this mode.

For **pss** mode only sign and verify are supported and the digest type must be specified.

rsa_pss_saltlen:len

For **pss** mode only this option specifies the salt length. Two special values are supported: -1 sets the salt length to the digest length. When signing -2 sets the salt length to the maximum permissible value. When verifying -2 causes the salt length to be automatically determined based on the **PSS** block structure.

DSA ALGORITHM

The DSA algorithm supports signing and verification operations only. Currently there are no additional options other than **digest**. Only the SHA1 digest can be used and this digest is assumed by default.

DH ALGORITHM

The DH algorithm only supports the derivation operation and no additional options.

EC ALGORITHM

The EC algorithm supports sign, verify and derive operations. The sign and verify operations use ECDSA and derive uses ECDH. Currently there are no additional options other than **digest**. Only the SHA1 digest can be used and this digest is assumed by default.

EXAMPLES

Sign some data using a private key:

```
openssl pkeyutl -sign -in file -inkey key.pem -out sig
```

Recover the signed data (e.g. if an RSA key is used):

```
openssl pkeyutl -verifyrecover -in sig -inkey key.pem
```

Verify the signature (e.g. a DSA key):

```
openssl pkeyutl -verify -in file -sigfile sig -inkey key.pem
```

Sign data using a message digest value (this is currently only valid for RSA):

```
openssl pkeyutl -sign -in file -inkey key.pem -out sig -pkeyopt digest:sha256
```

Derive a shared secret value:

```
openssl pkeyutl -derive -inkey key.pem -peerkey pubkey.pem -out secret
```

SEE ALSO

genpkey(1), pkey(1), rsautl(1)dgst(1), rsa(1), genrsa(1)

Name

rand — generate pseudo-random bytes

Synopsis

```
openssl rand
[-out file]
[-rand file(s)]
[-base64]
[-hex]
num
```

DESCRIPTION

The **rand** command outputs *num* pseudo-random bytes after seeding the random number generator once. As in other **openssl** command line tools, PRNG seeding uses the file *$HOME/***.rnd** or **.rnd** in addition to the files given in the **-rand** option. A new *$HOME*/**.rnd** or **.rnd** file will be written back if enough seeding was obtained from these sources.

OPTIONS

-out *file*

Write to *file* instead of standard output.

-rand *file(s)*

Use specified file or files or EGD socket (see RAND_egd(3)) for seeding the random number generator. Multiple files can be specified separated by a OS-dependent character. The separator is **;** for MS-Windows, **,** for OpenVMS, and **:** for all others.

-base64

Perform base64 encoding on the output.

-hex

Show the output as a hex string.

SEE ALSO

RAND_bytes(3)

Name

req — PKCS#10 certificate request and certificate generating utility.

Synopsis

```
opensslreq
[-inform PEM|DER]
[-outform PEM|DER]
[-in filename]
[-passin arg]
[-out filename]
[-passout arg]
[-text]
[-pubkey]
[-noout]
[-verify]
[-modulus]
[-new]
[-rand file(s)]
[-newkey rsa:bits]
[-newkey alg:file]
[-nodes]
[-key filename]
[-keyform PEM|DER]
[-keyout filename]
[-keygen_engine id]
[-[digest]]
[-config filename]
[-subj arg]
[-multivalue-rdn]
[-x509]
[-days n]
[-set_serial n]
[-asn1-kludge]
[-no-asn1-kludge]
[-newhdr]
[-extensions section]
[-reqexts section]
[-utf8]
[-nameopt]
[-reqopt]
[-subject]
[-subj arg]
[-batch]
[-verbose]
[-engine id]
```

DESCRIPTION

The **req** command primarily creates and processes certificate requests in PKCS#10 format. It can additionally create self signed certificates for use as root CAs for example.

COMMAND OPTIONS

-inform DER|PEM

> This specifies the input format. The **DER** option uses an ASN1 DER encoded form compatible with the PKCS#10. The **PEM** form is the default format: it consists of the **DER** format base64 encoded with additional header and footer lines.

-outform DER|PEM

> This specifies the output format, the options have the same meaning as the **-inform** option.

-in filename

This specifies the input filename to read a request from or standard input if this option is not specified. A request is only read if the creation options (**-new** and **-newkey**) are not specified.

-passin arg

the input file password source. For more information about the format of **arg** see the **PASS PHRASE ARGUMENTS** section in openssl(1).

-out filename

This specifies the output filename to write to or standard output by default.

-passout arg

the output file password source. For more information about the format of **arg** see the **PASS PHRASE ARGUMENTS** section in openssl(1).

-text

prints out the certificate request in text form.

-subject

prints out the request subject (or certificate subject if **-x509** is specified)

-pubkey

outputs the public key.

-noout

this option prevents output of the encoded version of the request.

-modulus

this option prints out the value of the modulus of the public key contained in the request.

-verify

verifies the signature on the request.

-new

this option generates a new certificate request. It will prompt the user for the relevant field values. The actual fields prompted for and their maximum and minimum sizes are specified in the configuration file and any requested extensions.

If the **-key** option is not used it will generate a new RSA private key using information specified in the configuration file.

-subj arg

Replaces subject field of input request with specified data and outputs modified request. The arg must be formatted as */type0=value0/type1=value1/type2=...*, characters may be escaped by \ (backslash), no spaces are skipped.

-rand file(s)

a file or files containing random data used to seed the random number generator, or an EGD socket (see RAND_egd(3)). Multiple files can be specified separated by a OS-dependent character. The separator is **;** for MS-Windows, **,** for OpenVMS, and **:** for all others.

-newkey arg

this option creates a new certificate request and a new private key. The argument takes one of several forms. **rsa:nbits**, where **nbits** is the number of bits, generates an RSA key **nbits** in size. If **nbits** is omitted, i.e. **-newkey rsa** specified, the default key size, specified in the configuration file is used.

All other algorithms support the **-newkey alg:file** form, where file may be an algorithm parameter file, created by the **genpkey -genparam** command or and X.509 certificate for a key with approriate algorithm.

param:file generates a key using the parameter file or certificate **file**, the algorithm is determined by the parameters. **algname:file** use algorithm **algname** and parameter file **file**: the two algorithms must match or an error occurs. **algname** just uses algorithm **algname**, and parameters, if neccessary should be specified via **-pkeyopt** parameter.

dsa:filename generates a DSA key using the parameters in the file **filename**. **ec:filename** generates EC key (usable both with ECDSA or ECDH algorithms), **gost2001:filename** generates GOST R 34.10-2001 key (requires **ccgost** engine configured in the configuration file). If just **gost2001** is specified a parameter set should be specified by **-pkeyopt paramset:X**

-pkeyopt opt:value

set the public key algorithm option **opt** to **value**. The precise set of options supported depends on the public key algorithm used and its implementation. See **KEY GENERATION OPTIONS** in the **genpkey** manual page for more details.

-key filename

This specifies the file to read the private key from. It also accepts PKCS#8 format private keys for PEM format files.

-keyform PEM|DER

the format of the private key file specified in the **-key** argument. PEM is the default.

-keyout filename

this gives the filename to write the newly created private key to. If this option is not specified then the filename present in the configuration file is used.

-nodes

if this option is specified then if a private key is created it will not be encrypted.

-[digest]

this specifies the message digest to sign the request with (such as **-md5**, **-sha1**). This overrides the digest algorithm specified in the configuration file.

Some public key algorithms may override this choice. For instance, DSA signatures always use SHA1, GOST R 34.10 signatures always use GOST R 34.11-94 (**-md_gost94**).

-config filename

this allows an alternative configuration file to be specified, this overrides the compile time filename or any specified in the **OPENSSL_CONF** environment variable.

-subj arg

sets subject name for new request or supersedes the subject name when processing a request. The arg must be formatted as */type0=value0/type1=value1/type2=…*, characters may be escaped by \ (backslash), no spaces are skipped.

-multivalue-rdn

this option causes the -subj argument to be interpreted with full support for multivalued RDNs. Example:

/DC=org/DC=OpenSSL/DC=users/UID=123456+CN=John Doe

If -multi-rdn is not used then the UID value is *123456+CN=John Doe*.

-x509

this option outputs a self signed certificate instead of a certificate request. This is typically used to generate a test certificate or a self signed root CA. The extensions added to the certificate (if any) are specified in the configuration file. Unless specified using the **set_serial** option **0** will be used for the serial number.

-days n

when the **-x509** option is being used this specifies the number of days to certify the certificate for. The default is 30 days.

-set_serial n

serial number to use when outputting a self signed certificate. This may be specified as a decimal value or a hex value if preceded by **0x**. It is possible to use negative serial numbers but this is not recommended.

-extensions section
-reqexts section

these options specify alternative sections to include certificate extensions (if the **-x509** option is present) or certificate request extensions. This allows several different sections to be used in the same configuration file to specify requests for a variety of purposes.

-utf8

this option causes field values to be interpreted as UTF8 strings, by default they are interpreted as ASCII. This means that the field values, whether prompted from a terminal or obtained from a configuration file, must be valid UTF8 strings.

-nameopt option

option which determines how the subject or issuer names are displayed. The **option** argument can be a single option or multiple options separated by commas. Alternatively the **-nameopt** switch may be used more than once to set multiple options. See the x509(1) manual page for details.

-reqopt

customise the output format used with **-text**. The **option** argument can be a single option or multiple options separated by commas.

See discission of the **-certopt** parameter in the **x509** command.

-asn1-kludge

by default the **req** command outputs certificate requests containing no attributes in the correct PKCS#10 format. However certain CAs will only accept requests containing no attributes in an invalid form: this option produces this invalid format.

More precisely the **Attributes** in a PKCS#10 certificate request are defined as a **SET OF Attribute**. They are **not OPTIONAL** so if no attributes are present then they should be encoded as an empty **SET OF**. The invalid form does not include the empty **SET OF** whereas the correct form does.

It should be noted that very few CAs still require the use of this option.

-no-asn1-kludge

Reverses effect of **-asn1-kludge**

-newhdr

Adds the word **NEW** to the PEM file header and footer lines on the outputted request. Some software (Netscape certificate server) and some CAs need this.

-batch

non-interactive mode.

-verbose

print extra details about the operations being performed.

-engine id

specifying an engine (by its unique **id** string) will cause **req** to attempt to obtain a functional reference to the specified engine, thus initialising it if needed. The engine will then be set as the default for all available algorithms.

-keygen_engine id

specifies an engine (by its unique **id** string) which would be used for key generation operations.

CONFIGURATION FILE FORMAT

The configuration options are specified in the **req** section of the configuration file. As with all configuration files if no value is specified in the specific section (i.e. **req**) then the initial unnamed or **default** section is searched too.

The options available are described in detail below.

input_password output_password

The passwords for the input private key file (if present) and the output private key file (if one will be created). The command line options **passin** and **passout** override the configuration file values.

default_bits

This specifies the default key size in bits. If not specified then 512 is used. It is used if the **-new** option is used. It can be overridden by using the **-newkey** option.

default_keyfile

This is the default filename to write a private key to. If not specified the key is written to standard output. This can be overridden by the **-keyout** option.

oid_file

This specifies a file containing additional **OBJECT IDENTIFIERS**. Each line of the file should consist of the numerical form of the object identifier followed by white space then the short name followed by white space and finally the long name.

oid_section

This specifies a section in the configuration file containing extra object identifiers. Each line should consist of the short name of the object identifier followed by **=** and the numerical form. The short and long names are the same when this option is used.

RANDFILE

This specifies a filename in which random number seed information is placed and read from, or an EGD socket (see RAND_egd(3)). It is used for private key generation.

encrypt_key

If this is set to **no** then if a private key is generated it is **not** encrypted. This is equivalent to the **-nodes** command line option. For compatibility **encrypt_rsa_key** is an equivalent option.

default_md

This option specifies the digest algorithm to use. Possible values include **md5 sha1 mdc2**. If not present then MD5 is used. This option can be overridden on the command line.

string_mask

This option masks out the use of certain string types in certain fields. Most users will not need to change this option.

It can be set to several values **default** which is also the default option uses PrintableStrings, T61Strings and BMPStrings if the **pkix** value is used then only PrintableStrings and BMPStrings will be used. This follows the PKIX recommendation in RFC2459. If the **utf8only** option is used then only UTF8Strings will be used: this is the PKIX recommendation in RFC2459 after 2003. Finally the **nombstr** option just uses PrintableStrings and T61Strings: certain software has problems with BMPStrings and UTF8Strings: in particular Netscape.

req_extensions

this specifies the configuration file section containing a list of extensions to add to the certificate request. It can be overridden by the **-reqexts** command line switch. See the x509v3_config(5) manual page for details of the extension section format.

x509_extensions

this specifies the configuration file section containing a list of extensions to add to certificate generated when the **-x509** switch is used. It can be overridden by the **-extensions** command line switch.

prompt

if set to the value **no** this disables prompting of certificate fields and just takes values from the config file directly. It also changes the expected format of the **distinguished_name** and **attributes** sections.

utf8

if set to the value **yes** then field values to be interpreted as UTF8 strings, by default they are interpreted as ASCII. This means that the field values, whether prompted from a terminal or obtained from a configuration file, must be valid UTF8 strings.

attributes

this specifies the section containing any request attributes: its format is the same as **distinguished_name**. Typically these may contain the challengePassword or unstructuredName types. They are currently ignored by OpenSSL's request signing utilities but some CAs might want them.

distinguished_name

This specifies the section containing the distinguished name fields to prompt for when generating a certificate or certificate request. The format is described in the next section.

DISTINGUISHED NAME AND ATTRIBUTE SECTION FORMAT

There are two separate formats for the distinguished name and attribute sections. If the **prompt** option is set to **no** then these sections just consist of field names and values: for example,

```
CN=My Name
OU=My Organization
```

```
emailAddress=someone@somewhere.org
```

This allows external programs (e.g. GUI based) to generate a template file with all the field names and values and just pass it to **req**. An example of this kind of configuration file is contained in the **EXAMPLES** section.

Alternatively if the **prompt** option is absent or not set to **no** then the file contains field prompting information. It consists of lines of the form:

```
fieldName="prompt"
fieldName_default="default field value"
fieldName_min= 2
fieldName_max= 4
```

"fieldName" is the field name being used, for example commonName (or CN). The "prompt" string is used to ask the user to enter the relevant details. If the user enters nothing then the default value is used if no default value is present then the field is omitted. A field can still be omitted if a default value is present if the user just enters the '.' character.

The number of characters entered must be between the fieldName_min and fieldName_max limits: there may be additional restrictions based on the field being used (for example countryName can only ever be two characters long and must fit in a PrintableString).

Some fields (such as organizationName) can be used more than once in a DN. This presents a problem because configuration files will not recognize the same name occurring twice. To avoid this problem if the fieldName contains some characters followed by a full stop they will be ignored. So for example a second organizationName can be input by calling it "1.organizationName".

The actual permitted field names are any object identifier short or long names. These are compiled into OpenSSL and include the usual values such as commonName, countryName, localityName, organizationName, organizationalUnitName, stateOrProvinceName. Additionally emailAddress is include as well as name, surname, givenName initials and dnQualifier.

Additional object identifiers can be defined with the **oid_file** or **oid_section** options in the configuration file. Any additional fields will be treated as though they were a DirectoryString.

EXAMPLES

Examine and verify certificate request:

```
openssl req -in req.pem -text -verify -noout
```

Create a private key and then generate a certificate request from it:

```
openssl genrsa -out key.pem 1024
openssl req -new -key key.pem -out req.pem
```

The same but just using req:

```
openssl req -newkey rsa:1024 -keyout key.pem -out req.pem
```

Generate a self signed root certificate:

```
openssl req -x509 -newkey rsa:1024 -keyout key.pem -out req.pem
```

Example of a file pointed to by the **oid_file** option:

```
1.2.3.4        shortName       A longer Name
1.2.3.6        otherName       Other longer Name
```

Example of a section pointed to by **oid_section** making use of variable expansion:

```
testoid1=1.2.3.5
testoid2=${testoid1}.6
```

Sample configuration file prompting for field values:

```
[ req ]
default_bits           = 1024
default_keyfile        = privkey.pem
distinguished_name     = req_distinguished_name
attributes             = req_attributes
x509_extensions        = v3_ca

dirstring_type = nobmp

[ req_distinguished_name ]
countryName                    = Country Name (2 letter code)
countryName_default            = AU
countryName_min                = 2
countryName_max                = 2

localityName                   = Locality Name (eg, city)

organizationalUnitName         = Organizational Unit Name (eg, section)

commonName                     = Common Name (eg, YOUR name)
commonName_max                 = 64

emailAddress                   = Email Address
emailAddress_max               = 40

[ req_attributes ]
challengePassword              = A challenge password
challengePassword_min          = 4
challengePassword_max          = 20

[ v3_ca ]

subjectKeyIdentifier=hash
authorityKeyIdentifier=keyid:always,issuer:always
basicConstraints = CA:true
```

Sample configuration containing all field values:

```
RANDFILE               = $ENV::HOME/.rnd

[ req ]
default_bits           = 1024
default_keyfile        = keyfile.pem
distinguished_name     = req_distinguished_name
attributes             = req_attributes
prompt                 = no
output_password        = mypass

[ req_distinguished_name ]
C                      = GB
ST                     = Test State or Province
L                      = Test Locality
O                      = Organization Name
OU                     = Organizational Unit Name
CN                     = Common Name
emailAddress           = test@email.address

[ req_attributes ]
challengePassword              = A challenge password
```

NOTES

The header and footer lines in the **PEM** format are normally:

```
-----BEGIN CERTIFICATE REQUEST-----
-----END CERTIFICATE REQUEST-----
```

some software (some versions of Netscape certificate server) instead needs:

```
-----BEGIN NEW CERTIFICATE REQUEST-----
-----END NEW CERTIFICATE REQUEST-----
```

which is produced with the **-newhdr** option but is otherwise compatible. Either form is accepted transparently on input.

The certificate requests generated by **Xenroll** with MSIE have extensions added. It includes the **keyUsage** extension which determines the type of key (signature only or general purpose) and any additional OIDs entered by the script in an extendedKeyUsage extension.

DIAGNOSTICS

The following messages are frequently asked about:

```
Using configuration from /some/path/openssl.cnf
Unable to load config info
```

This is followed some time later by…

```
unable to find 'distinguished_name' in config
problems making Certificate Request
```

The first error message is the clue: it can't find the configuration file! Certain operations (like examining a certificate request) don't need a configuration file so its use isn't enforced. Generation of certificates or requests however does need a configuration file. This could be regarded as a bug.

Another puzzling message is this:

```
Attributes:
    a0:00
```

this is displayed when no attributes are present and the request includes the correct empty **SET OF** structure (the DER encoding of which is 0xa0 0x00). If you just see:

```
Attributes:
```

then the **SET OF** is missing and the encoding is technically invalid (but it is tolerated). See the description of the command line option **-asn1-kludge** for more information.

ENVIRONMENT VARIABLES

The variable **OPENSSL_CONF** if defined allows an alternative configuration file location to be specified, it will be overridden by the **-config** command line switch if it is present. For compatibility reasons the **SSLEAY_CONF** environment variable serves the same purpose but its use is discouraged.

BUGS

OpenSSL's handling of T61Strings (aka TeletexStrings) is broken: it effectively treats them as ISO-8859-1 (Latin 1), Netscape and MSIE have similar behaviour. This can cause problems if you need characters that aren't available in PrintableStrings and you don't want to or can't use BMPStrings.

As a consequence of the T61String handling the only correct way to represent accented characters in OpenSSL is to use a BMPString: unfortunately Netscape currently chokes on these. If you have to use accented characters with Netscape and MSIE then you currently need to use the invalid T61String form.

The current prompting is not very friendly. It doesn't allow you to confirm what you've just entered. Other things like extensions in certificate requests are statically defined in the configuration file. Some of these: like an email address in subjectAltName should be input by the user.

SEE ALSO

x509(1), ca(1), genrsa(1), gendsa(1), config(5), x509v3_config(5)

Name

rsa — RSA key processing tool

Synopsis

```
opensslrsa
[-inform PEM|NET|DER]
[-outform PEM|NET|DER]
[-in filename]
[-passin arg]
[-out filename]
[-passout arg]
[-sgckey]
[-aes128]
[-aes192]
[-aes256]
[-camellia128]
[-camellia192]
[-camellia256]
[-des]
[-des3]
[-idea]
[-text]
[-noout]
[-modulus]
[-check]
[-pubin]
[-pubout]
[-RSAPublicKey_in]
[-RSAPublicKey_out]
[-engine id]
```

DESCRIPTION

The **rsa** command processes RSA keys. They can be converted between various forms and their components printed out. **Note** this command uses the traditional SSLeay compatible format for private key encryption: newer applications should use the more secure PKCS#8 format using the **pkcs8** utility.

COMMAND OPTIONS

-inform DER|NET|PEM

This specifies the input format. The **DER** option uses an ASN1 DER encoded form compatible with the PKCS#1 RSAPrivateKey or SubjectPublicKeyInfo format. The **PEM** form is the default format: it consists of the **DER** format base64 encoded with additional header and footer lines. On input PKCS#8 format private keys are also accepted. The **NET** form is a format is described in the **NOTES** section.

-outform DER|NET|PEM

This specifies the output format, the options have the same meaning as the **-inform** option.

-in filename

This specifies the input filename to read a key from or standard input if this option is not specified. If the key is encrypted a pass phrase will be prompted for.

-passin arg

the input file password source. For more information about the format of **arg** see the **PASS PHRASE ARGUMENTS** section in openssl(1).

-out filename

This specifies the output filename to write a key to or standard output if this option is not specified. If any encryption options are set then a pass phrase will be prompted for. The output filename should **not** be the same as the input filename.

-passout password

the output file password source. For more information about the format of **arg** see the **PASS PHRASE ARGUMENTS** section in openssl(1).

-sgckey

use the modified NET algorithm used with some versions of Microsoft IIS and SGC keys.

-aes128|-aes192|-aes256|-camellia128|-camellia192|-camellia256|-des|-des3|-idea

These options encrypt the private key with the specified cipher before outputting it. A pass phrase is prompted for. If none of these options is specified the key is written in plain text. This means that using the **rsa** utility to read in an encrypted key with no encryption option can be used to remove the pass phrase from a key, or by setting the encryption options it can be use to add or change the pass phrase. These options can only be used with PEM format output files.

-text

prints out the various public or private key components in plain text in addition to the encoded version.

-noout

this option prevents output of the encoded version of the key.

-modulus

this option prints out the value of the modulus of the key.

-check

this option checks the consistency of an RSA private key.

-pubin

by default a private key is read from the input file: with this option a public key is read instead.

-pubout

by default a private key is output: with this option a public key will be output instead. This option is automatically set if the input is a public key.

-RSAPublicKey_in, **-RSAPublicKey_out**

like **-pubin** and **-pubout** except **RSAPublicKey** format is used instead.

-engine id

specifying an engine (by its unique **id** string) will cause **rsa** to attempt to obtain a functional reference to the specified engine, thus initialising it if needed. The engine will then be set as the default for all available algorithms.

NOTES

The PEM private key format uses the header and footer lines:

```
-----BEGIN RSA PRIVATE KEY-----
-----END RSA PRIVATE KEY-----
```

The PEM public key format uses the header and footer lines:

```
-----BEGIN PUBLIC KEY-----
-----END PUBLIC KEY-----
```

The PEM **RSAPublicKey** format uses the header and footer lines:

```
-----BEGIN RSA PUBLIC KEY-----
-----END RSA PUBLIC KEY-----
```

The **NET** form is a format compatible with older Netscape servers and Microsoft IIS .key files, this uses unsalted RC4 for its encryption. It is not very secure and so should only be used when necessary.

Some newer version of IIS have additional data in the exported .key files. To use these with the utility, view the file with a binary editor and look for the string "private-key", then trace back to the byte sequence 0x30, 0x82 (this is an ASN1 SEQUENCE). Copy all the data from this point onwards to another file and use that as the input to the **rsa** utility with the **-inform NET** option. If you get an error after entering the password try the **-sgckey** option.

EXAMPLES

To remove the pass phrase on an RSA private key:

```
openssl rsa -in key.pem -out keyout.pem
```

To encrypt a private key using triple DES:

```
openssl rsa -in key.pem -des3 -out keyout.pem
```

To convert a private key from PEM to DER format:

```
openssl rsa -in key.pem -outform DER -out keyout.der
```

To print out the components of a private key to standard output:

```
openssl rsa -in key.pem -text -noout
```

To just output the public part of a private key:

```
openssl rsa -in key.pem -pubout -out pubkey.pem
```

Output the public part of a private key in **RSAPublicKey** format:

```
openssl rsa -in key.pem -RSAPublicKey_out -out pubkey.pem
```

BUGS

The command line password arguments don't currently work with **NET** format.

There should be an option that automatically handles .key files, without having to manually edit them.

SEE ALSO

pkcs8(1), dsa(1), genrsa(1), gendsa(1)

Name

rsautl — RSA utility

Synopsis

```
opensslrsautl
[-in file]
[-out file]
[-inkey file]
[-pubin]
[-certin]
[-sign]
[-verify]
[-encrypt]
[-decrypt]
[-pkcs]
[-ssl]
[-raw]
[-hexdump]
[-asn1parse]
```

DESCRIPTION

The **rsautl** command can be used to sign, verify, encrypt and decrypt data using the RSA algorithm.

COMMAND OPTIONS

-in filename

This specifies the input filename to read data from or standard input if this option is not specified.

-out filename

specifies the output filename to write to or standard output by default.

-inkey file

the input key file, by default it should be an RSA private key.

-pubin

the input file is an RSA public key.

-certin

the input is a certificate containing an RSA public key.

-sign

sign the input data and output the signed result. This requires and RSA private key.

-verify

verify the input data and output the recovered data.

-encrypt

encrypt the input data using an RSA public key.

-decrypt

decrypt the input data using an RSA private key.

-pkcs, -oaep, -ssl, -raw

the padding to use: PKCS#1 v1.5 (the default), PKCS#1 OAEP, special padding used in SSL v2 backwards compatible handshakes, or no padding, respectively. For signatures, only **-pkcs** and **-raw** can be used.

-hexdump

hex dump the output data.

-asn1parse

asn1parse the output data, this is useful when combined with the **-verify** option.

NOTES

rsautl because it uses the RSA algorithm directly can only be used to sign or verify small pieces of data.

EXAMPLES

Sign some data using a private key:

```
openssl rsautl -sign -in file -inkey key.pem -out sig
```

Recover the signed data

```
openssl rsautl -verify -in sig -inkey key.pem
```

Examine the raw signed data:

```
openssl rsautl -verify -in file -inkey key.pem -raw -hexdump

0000 - 00 01 ff ff ff ff ff ff-ff ff ff ff ff ff ff ff   ……..........
0010 - ff ff ff ff ff ff ff ff-ff ff ff ff ff ff ff ff   ……..........
0020 - ff ff ff ff ff ff ff ff-ff ff ff ff ff ff ff ff   ……..........
0030 - ff ff ff ff ff ff ff ff-ff ff ff ff ff ff ff ff   ……..........
0040 - ff ff ff ff ff ff ff ff-ff ff ff ff ff ff ff ff   ……..........
0050 - ff ff ff ff ff ff ff ff-ff ff ff ff ff ff ff ff   ……..........
0060 - ff ff ff ff ff ff ff ff-ff ff ff ff ff ff ff ff   ……..........
0070 - ff ff ff ff 00 68 65 6c-6c 6f 20 77 6f 72 6c 64   …..hello world
```

The PKCS#1 block formatting is evident from this. If this was done using encrypt and decrypt the block would have been of type 2 (the second byte) and random padding data visible instead of the 0xff bytes.

It is possible to analyse the signature of certificates using this utility in conjunction with **asn1parse**. Consider the self signed example in certs/pca-cert.pem . Running **asn1parse** as follows yields:

```
openssl asn1parse -in pca-cert.pem

  0:d=0  hl=4 l= 742 cons: SEQUENCE
  4:d=1  hl=4 l= 591 cons:  SEQUENCE
  8:d=2  hl=2 l=   3 cons:   cont [ 0 ]
 10:d=3  hl=2 l=   1 prim:    INTEGER           :02
 13:d=2  hl=2 l=   1 prim:   INTEGER           :00
 16:d=2  hl=2 l=  13 cons:   SEQUENCE
 18:d=3  hl=2 l=   9 prim:    OBJECT            :md5WithRSAEncryption
 29:d=3  hl=2 l=   0 prim:    NULL
 31:d=2  hl=2 l=  92 cons:   SEQUENCE
 33:d=3  hl=2 l=  11 cons:    SET
 35:d=4  hl=2 l=   9 cons:     SEQUENCE
 37:d=5  hl=2 l=   3 prim:      OBJECT            :countryName
 42:d=5  hl=2 l=   2 prim:      PRINTABLESTRING   :AU
….
599:d=1  hl=2 l=  13 cons:  SEQUENCE
601:d=2  hl=2 l=   9 prim:   OBJECT            :md5WithRSAEncryption
612:d=2  hl=2 l=   0 prim:   NULL
614:d=1  hl=3 l= 129 prim:  BIT STRING
```

The final BIT STRING contains the actual signature. It can be extracted with:

```
openssl asn1parse -in pca-cert.pem -out sig -noout -strparse 614
```

The certificate public key can be extracted with:

```
openssl x509 -in test/testx509.pem -pubkey -noout >pubkey.pem
```

The signature can be analysed with:

```
openssl rsautl -in sig -verify -asn1parse -inkey pubkey.pem -pubin

 0:d=0  hl=2 l=  32 cons: SEQUENCE
 2:d=1  hl=2 l=  12 cons:  SEQUENCE
 4:d=2  hl=2 l=   8 prim:   OBJECT            :md5
14:d=2  hl=2 l=   0 prim:   NULL
16:d=1  hl=2 l=  16 prim:  OCTET STRING
   0000 - f3 46 9e aa 1a 4a 73 c9-37 ea 93 00 48 25 08 b5   .F…Js.7…H%..
```

This is the parsed version of an ASN1 DigestInfo structure. It can be seen that the digest used was md5. The actual part of the certificate that was signed can be extracted with:

```
openssl asn1parse -in pca-cert.pem -out tbs -noout -strparse 4
```

and its digest computed with:

```
openssl md5 -c tbs
MD5(tbs)= f3:46:9e:aa:1a:4a:73:c9:37:ea:93:00:48:25:08:b5
```

which it can be seen agrees with the recovered value above.

SEE ALSO

dgst(1), rsa(1), genrsa(1)

Name

s_client — SSL/TLS client program

Synopsis

```
openssls_client
[-connect host:port]
[-servername name]
[-verify depth]
[-verify_return_error]
[-cert filename]
[-certform DER|PEM]
[-key filename]
[-keyform DER|PEM]
[-pass arg]
[-CApath directory]
[-CAfile filename]
[-no_alt_chains]
[-reconnect]
[-pause]
[-showcerts]
[-debug]
[-msg]
[-nbio_test]
[-state]
[-nbio]
[-crlf]
[-ign_eof]
[-no_ign_eof]
[-quiet]
[-ssl2]
[-ssl3]
[-tls1]
[-no_ssl2]
[-no_ssl3]
[-no_tls1]
[-bugs]
[-cipher cipherlist]
[-serverpref]
[-starttls protocol]
[-engine id]
[-tlsextdebug]
[-no_ticket]
[-sess_out filename]
[-sess_in filename]
[-rand file(s)]
[-status]
[-nextprotoneg protocols]
```

DESCRIPTION

The **s_client** command implements a generic SSL/TLS client which connects to a remote host using SSL/TLS. It is a *very* useful diagnostic tool for SSL servers.

OPTIONS

-connect host:port

> This specifies the host and optional port to connect to. If not specified then an attempt is made to connect to the local host on port 4433.

-servername name

Set the TLS SNI (Server Name Indication) extension in the ClientHello message.

-cert certname

The certificate to use, if one is requested by the server. The default is not to use a certificate.

-certform format

The certificate format to use: DER or PEM. PEM is the default.

-key keyfile

The private key to use. If not specified then the certificate file will be used.

-keyform format

The private format to use: DER or PEM. PEM is the default.

-pass arg

the private key password source. For more information about the format of **arg** see the **PASS PHRASE ARGUMENTS** section in openssl(1).

-verify depth

The verify depth to use. This specifies the maximum length of the server certificate chain and turns on server certificate verification. Currently the verify operation continues after errors so all the problems with a certificate chain can be seen. As a side effect the connection will never fail due to a server certificate verify failure.

-verify_return_error

Return verification errors instead of continuing. This will typically abort the handshake with a fatal error.

-CApath directory

The directory to use for server certificate verification. This directory must be in "hash format", see **verify** for more information. These are also used when building the client certificate chain.

-CAfile file

A file containing trusted certificates to use during server authentication and to use when attempting to build the client certificate chain.

-purpose, -ignore_critical, -issuer_checks, -crl_check, -crl_check_all, -policy_check, -extended_crl, -x509_strict, -policy, -check_ss_sig, -no_alt_chains

Set various certificate chain valiadition option. See the **verify** manual page for details.

-reconnect

reconnects to the same server 5 times using the same session ID, this can be used as a test that session caching is working.

-pause

pauses 1 second between each read and write call.

-showcerts

display the whole server certificate chain: normally only the server certificate itself is displayed.

-prexit

print session information when the program exits. This will always attempt to print out information even if the connection fails. Normally information will only be printed out once if the connection succeeds. This option is useful because the cipher in use may be renegotiated or the connection may fail because a client certificate is required or is requested only after an attempt is made to access a certain URL. Note: the output produced by this option is not always accurate because a connection might never have been established.

-state

prints out the SSL session states.

-debug

print extensive debugging information including a hex dump of all traffic.

-msg

show all protocol messages with hex dump.

-nbio_test

tests non-blocking I/O

-nbio

turns on non-blocking I/O

-crlf

this option translated a line feed from the terminal into CR+LF as required by some servers.

-ign_eof

inhibit shutting down the connection when end of file is reached in the input.

-quiet

inhibit printing of session and certificate information. This implicitly turns on **-ign_eof** as well.

-no_ign_eof

shut down the connection when end of file is reached in the input. Can be used to override the implicit **-ign_eof** after **-quiet**.

-psk_identity identity

Use the PSK identity **identity** when using a PSK cipher suite.

-psk key

Use the PSK key **key** when using a PSK cipher suite. The key is given as a hexadecimal number without leading 0x, for example -psk 1a2b3c4d.

-ssl2, **-ssl3**, **-tls1**, **-tls1_1**, **-tls1_2**, **-no_ssl2**, **-no_ssl3**, **-no_tls1**, **-no_tls1_1**, **-no_tls1_2**

These options require or disable the use of the specified SSL or TLS protocols. By default the initial handshake uses a *version-flexible* method which will negotiate the highest mutually supported protocol version.

-bugs

there are several known bug in SSL and TLS implementations. Adding this option enables various workarounds.

-cipher cipherlist

this allows the cipher list sent by the client to be modified. Although the server determines which cipher suite is used it should take the first supported cipher in the list sent by the client. See the **ciphers** command for more information.

-serverpref

use the server's cipher preferences; only used for SSLV2.

-starttls protocol

send the protocol-specific message(s) to switch to TLS for communication. **protocol** is a keyword for the intended protocol. Currently, the only supported keywords are "smtp", "pop3", "imap", and "ftp".

-tlsextdebug

print out a hex dump of any TLS extensions received from the server.

-no_ticket

disable RFC4507bis session ticket support.

-sess_out filename

output SSL session to **filename**

-sess_in sess.pem

load SSL session from **filename**. The client will attempt to resume a connection from this session.

-engine id

specifying an engine (by its unique **id** string) will cause **s_client** to attempt to obtain a functional reference to the specified engine, thus initialising it if needed. The engine will then be set as the default for all available algorithms.

-rand file(s)

a file or files containing random data used to seed the random number generator, or an EGD socket (see RAND_egd(3)). Multiple files can be specified separated by a OS-dependent character. The separator is **;** for MS-Windows, **,** for OpenVMS, and **:** for all others.

-status

sends a certificate status request to the server (OCSP stapling). The server response (if any) is printed out.

-nextprotoneg protocols

enable Next Protocol Negotiation TLS extension and provide a list of comma-separated protocol names that the client should advertise support for. The list should contain most wanted protocols first. Protocol names are printable ASCII strings, for example "http/1.1" or "spdy/3". Empty list of protocols is treated specially and will cause the client to advertise support for the TLS extension but disconnect just after reciving ServerHello with a list of server supported protocols.

CONNECTED COMMANDS

If a connection is established with an SSL server then any data received from the server is displayed and any key presses will be sent to the server. When used interactively (which means neither **-quiet** nor **-ign_eof** have been given), the session will be renegotiated if the line begins with an **R**, and if the line begins with a **Q** or if end of file is reached, the connection will be closed down.

NOTES

s_client can be used to debug SSL servers. To connect to an SSL HTTP server the command:

```
openssl s_client -connect servername:443
```

would typically be used (https uses port 443). If the connection succeeds then an HTTP command can be given such as "GET /" to retrieve a web page.

If the handshake fails then there are several possible causes, if it is nothing obvious like no client certificate then the **-bugs**, **-ssl2**, **-ssl3**, **-tls1**, **-no_ssl2**, **-no_ssl3**, **-no_tls1** options can be tried in case it is a buggy server. In particular you should play with these options **before** submitting a bug report to an OpenSSL mailing list.

A frequent problem when attempting to get client certificates working is that a web client complains it has no certificates or gives an empty list to choose from. This is normally because the server is not sending the clients certificate authority in its "acceptable CA list" when it requests a certificate. By using **s_client** the CA list can be viewed and checked. However some servers only request client authentication after a specific URL is requested. To obtain the list in this case it is necessary to use the **-prexit** option and send an HTTP request for an appropriate page.

If a certificate is specified on the command line using the **-cert** option it will not be used unless the server specifically requests a client certificate. Therefor merely including a client certificate on the command line is no guarantee that the certificate works.

If there are problems verifying a server certificate then the **-showcerts** option can be used to show the whole chain.

Since the SSLv23 client hello cannot include compression methods or extensions these will only be supported if its use is disabled, for example by using the **-no_sslv2** option.

The **s_client** utility is a test tool and is designed to continue the handshake after any certificate verification errors. As a result it will accept any certificate chain (trusted or not) sent by the peer. None test applications should **not** do this as it makes them vulnerable to a MITM attack. This behaviour can be changed by with the **-verify_return_error** option: any verify errors are then returned aborting the handshake.

BUGS

Because this program has a lot of options and also because some of the techniques used are rather old, the C source of s_client is rather hard to read and not a model of how things should be done. A typical SSL client program would be much simpler.

The **-prexit** option is a bit of a hack. We should really report information whenever a session is renegotiated.

SEE ALSO

sess_id(1), s_server(1), ciphers(1)

HISTORY

The -no_alt_chains options was first added to OpenSSL 1.0.1n and 1.0.2b.

Name

s_server — SSL/TLS server program

Synopsis

```
openssls_server
[-accept port]
[-context id]
[-verify depth]
[-Verify depth]
[-crl_check]
[-crl_check_all]
[-cert filename]
[-certform DER|PEM]
[-key keyfile]
[-keyform DER|PEM]
[-pass arg]
[-dcert filename]
[-dcertform DER|PEM]
[-dkey keyfile]
[-dkeyform DER|PEM]
[-dpass arg]
[-dhparam filename]
[-nbio]
[-nbio_test]
[-crlf]
[-debug]
[-msg]
[-state]
[-CApath directory]
[-CAfile filename]
[-no_alt_chains]
[-nocert]
[-cipher cipherlist]
[-serverpref]
[-quiet]
[-no_tmp_rsa]
[-ssl2]
[-ssl3]
[-tls1]
[-no_ssl2]
[-no_ssl3]
[-no_tls1]
[-no_dhe]
[-no_ecdhe]
[-bugs]
[-hack]
[-www]
[-WWW]
[-HTTP]
[-engine id]
[-tlsextdebug]
[-no_ticket]
[-id_prefix arg]
[-rand file(s)]
[-status]
[-status_verbose]
[-status_timeout nsec]
[-status_url url]
[-nextprotoneg protocols]
```

DESCRIPTION

The **s_server** command implements a generic SSL/TLS server which listens for connections on a given port using SSL/TLS.

OPTIONS

-accept port

the TCP port to listen on for connections. If not specified 4433 is used.

-context id

sets the SSL context id. It can be given any string value. If this option is not present a default value will be used.

-cert certname

The certificate to use, most servers cipher suites require the use of a certificate and some require a certificate with a certain public key type: for example the DSS cipher suites require a certificate containing a DSS (DSA) key. If not specified then the filename "server.pem" will be used.

-certform format

The certificate format to use: DER or PEM. PEM is the default.

-key keyfile

The private key to use. If not specified then the certificate file will be used.

-keyform format

The private format to use: DER or PEM. PEM is the default.

-pass arg

the private key password source. For more information about the format of **arg** see the **PASS PHRASE ARGUMENTS** section in openssl(1).

-dcert filename, **-dkey keyname**

specify an additional certificate and private key, these behave in the same manner as the **-cert** and **-key** options except there is no default if they are not specified (no additional certificate and key is used). As noted above some cipher suites require a certificate containing a key of a certain type. Some cipher suites need a certificate carrying an RSA key and some a DSS (DSA) key. By using RSA and DSS certificates and keys a server can support clients which only support RSA or DSS cipher suites by using an appropriate certificate.

-dcertform format, **-dkeyform format**, **-dpass arg**

additional certificate and private key format and passphrase respectively.

-nocert

if this option is set then no certificate is used. This restricts the cipher suites available to the anonymous ones (currently just anonymous DH).

-dhparam filename

the DH parameter file to use. The ephemeral DH cipher suites generate keys using a set of DH parameters. If not specified then an attempt is made to load the parameters from the server certificate file. If this fails then a static set of parameters hard coded into the s_server program will be used.

-no_dhe

if this option is set then no DH parameters will be loaded effectively disabling the ephemeral DH cipher suites.

-no_ecdhe

if this option is set then no ECDH parameters will be loaded effectively disabling the ephemeral ECDH cipher suites.

-no_tmp_rsa

certain export cipher suites sometimes use a temporary RSA key, this option disables temporary RSA key generation.

-verify depth, **-Verify depth**

The verify depth to use. This specifies the maximum length of the client certificate chain and makes the server request a certificate from the client. With the **-verify** option a certificate is requested but the client does not have to send one, with the **-Verify** option the client must supply a certificate or an error occurs.

If the ciphersuite cannot request a client certificate (for example an anonymous ciphersuite or PSK) this option has no effect.

-crl_check, **-crl_check_all**

Check the peer certificate has not been revoked by its CA. The CRL(s) are appended to the certificate file. With the **-crl_check_all** option all CRLs of all CAs in the chain are checked.

-CApath directory

The directory to use for client certificate verification. This directory must be in "hash format", see **verify** for more information. These are also used when building the server certificate chain.

-CAfile file

A file containing trusted certificates to use during client authentication and to use when attempting to build the server certificate chain. The list is also used in the list of acceptable client CAs passed to the client when a certificate is requested.

-no_alt_chains

See the **verify** manual page for details.

-state

prints out the SSL session states.

-debug

print extensive debugging information including a hex dump of all traffic.

-msg

show all protocol messages with hex dump.

-nbio_test

tests non blocking I/O

-nbio

turns on non blocking I/O

-crlf

this option translated a line feed from the terminal into CR+LF.

-quiet

inhibit printing of session and certificate information.

-psk_hint hint

Use the PSK identity hint **hint** when using a PSK cipher suite.

-psk key

Use the PSK key **key** when using a PSK cipher suite. The key is given as a hexadecimal number without leading 0x, for example -psk 1a2b3c4d.

-ssl2, **-ssl3**, **-tls1**, **-tls1_1**, **-tls1_2**, **-no_ssl2**, **-no_ssl3**, **-no_tls1**, **-no_tls1_1**, **-no_tls1_2**

These options require or disable the use of the specified SSL or TLS protocols. By default the initial handshake uses a *version-flexible* method which will negotiate the highest mutually supported protocol version.

-bugs

there are several known bug in SSL and TLS implementations. Adding this option enables various workarounds.

-hack

this option enables a further workaround for some some early Netscape SSL code (?).

-cipher cipherlist

this allows the cipher list used by the server to be modified. When the client sends a list of supported ciphers the first client cipher also included in the server list is used. Because the client specifies the preference order, the order of the server cipherlist irrelevant. See the **ciphers** command for more information.

-serverpref

use the server's cipher preferences, rather than the client's preferences.

-tlsextdebug

print out a hex dump of any TLS extensions received from the server.

-no_ticket

disable RFC4507bis session ticket support.

-www

sends a status message back to the client when it connects. This includes lots of information about the ciphers used and various session parameters. The output is in HTML format so this option will normally be used with a web browser.

-WWW

emulates a simple web server. Pages will be resolved relative to the current directory, for example if the URL https://myhost/page.html is requested the file ./page.html will be loaded.

-HTTP

emulates a simple web server. Pages will be resolved relative to the current directory, for example if the URL https://myhost/page.html is requested the file ./page.html will be loaded. The files loaded are assumed to contain a complete and correct HTTP response (lines that are part of the HTTP response line and headers must end with CRLF).

-engine id

specifying an engine (by its unique **id** string) will cause **s_server** to attempt to obtain a functional reference to the specified engine, thus initialising it if needed. The engine will then be set as the default for all available algorithms.

-id_prefix arg

generate SSL/TLS session IDs prefixed by **arg**. This is mostly useful for testing any SSL/TLS code (eg. proxies) that wish to deal with multiple servers, when each of which might be generating a unique range of session IDs (eg. with a certain prefix).

-rand file(s)

a file or files containing random data used to seed the random number generator, or an EGD socket (see RAND_egd(3)). Multiple files can be specified separated by a OS-dependent character. The separator is **;** for MS-Windows, **,** for OpenVMS, and **:** for all others.

-status

enables certificate status request support (aka OCSP stapling).

-status_verbose

enables certificate status request support (aka OCSP stapling) and gives a verbose printout of the OCSP response.

-status_timeout nsec

sets the timeout for OCSP response to **nsec** seconds.

-status_url url

sets a fallback responder URL to use if no responder URL is present in the server certificate. Without this option an error is returned if the server certificate does not contain a responder address.

-nextprotoneg protocols

enable Next Protocol Negotiation TLS extension and provide a comma-separated list of supported protocol names. The list should contain most wanted protocols first. Protocol names are printable ASCII strings, for example "http/1.1" or "spdy/3".

CONNECTED COMMANDS

If a connection request is established with an SSL client and neither the **-www** nor the **-WWW** option has been used then normally any data received from the client is displayed and any key presses will be sent to the client.

Certain single letter commands are also recognized which perform special operations: these are listed below.

q end the current SSL connection but still accept new connections.

Q end the current SSL connection and exit.

r renegotiate the SSL session.

R renegotiate the SSL session and request a client certificate.

P send some plain text down the underlying TCP connection: this should cause the client to disconnect due to a protocol violation.

S print out some session cache status information.

NOTES

s_server can be used to debug SSL clients. To accept connections from a web browser the command:

```
openssl s_server -accept 443 -www
```

can be used for example.

Most web browsers (in particular Netscape and MSIE) only support RSA cipher suites, so they cannot connect to servers which don't use a certificate carrying an RSA key or a version of OpenSSL with RSA disabled.

Although specifying an empty list of CAs when requesting a client certificate is strictly speaking a protocol violation, some SSL clients interpret this to mean any CA is acceptable. This is useful for debugging purposes.

The session parameters can printed out using the **sess_id** program.

BUGS

Because this program has a lot of options and also because some of the techniques used are rather old, the C source of s_server is rather hard to read and not a model of how things should be done. A typical SSL server program would be much simpler.

The output of common ciphers is wrong: it just gives the list of ciphers that OpenSSL recognizes and the client supports.

There should be a way for the **s_server** program to print out details of any unknown cipher suites a client says it supports.

SEE ALSO

sess_id(1), s_client(1), ciphers(1)

HISTORY

The -no_alt_chains options was first added to OpenSSL 1.0.1n and 1.0.2b.

Name

s_time — SSL/TLS performance timing program

Synopsis

```
openssls_time
[-connect host:port]
[-www page]
[-cert filename]
[-key filename]
[-CApath directory]
[-CAfile filename]
[-reuse]
[-new]
[-verify depth]
[-nbio]
[-time seconds]
[-ssl2]
[-ssl3]
[-bugs]
[-cipher cipherlist]
```

DESCRIPTION

The **s_time** command implements a generic SSL/TLS client which connects to a remote host using SSL/TLS. It can request a page from the server and includes the time to transfer the payload data in its timing measurements. It measures the number of connections within a given timeframe, the amount of data transferred (if any), and calculates the average time spent for one connection.

OPTIONS

-connect host:port

This specifies the host and optional port to connect to.

-www page

This specifies the page to GET from the server. A value of '/' gets the index.htm[l] page. If this parameter is not specified, then **s_time** will only perform the handshake to establish SSL connections but not transfer any payload data.

-cert certname

The certificate to use, if one is requested by the server. The default is not to use a certificate. The file is in PEM format.

-key keyfile

The private key to use. If not specified then the certificate file will be used. The file is in PEM format.

-verify depth

The verify depth to use. This specifies the maximum length of the server certificate chain and turns on server certificate verification. Currently the verify operation continues after errors so all the problems with a certificate chain can be seen. As a side effect the connection will never fail due to a server certificate verify failure.

-CApath directory

The directory to use for server certificate verification. This directory must be in "hash format", see **verify** for more information. These are also used when building the client certificate chain.

-CAfile file

A file containing trusted certificates to use during server authentication and to use when attempting to build the client certificate chain.

-new

performs the timing test using a new session ID for each connection. If neither **-new** nor **-reuse** are specified, they are both on by default and executed in sequence.

-reuse

performs the timing test using the same session ID; this can be used as a test that session caching is working. If neither **-new** nor **-reuse** are specified, they are both on by default and executed in sequence.

-nbio

turns on non-blocking I/O.

-ssl2, **-ssl3**

these options disable the use of certain SSL or TLS protocols. By default the initial handshake uses a method which should be compatible with all servers and permit them to use SSL v3, SSL v2 or TLS as appropriate. The timing program is not as rich in options to turn protocols on and off as the s_client(1) program and may not connect to all servers.

Unfortunately there are a lot of ancient and broken servers in use which cannot handle this technique and will fail to connect. Some servers only work if TLS is turned off with the **-ssl3** option; others will only support SSL v2 and may need the **-ssl2** option.

-bugs

there are several known bug in SSL and TLS implementations. Adding this option enables various workarounds.

-cipher cipherlist

this allows the cipher list sent by the client to be modified. Although the server determines which cipher suite is used it should take the first supported cipher in the list sent by the client. See the ciphers(1) command for more information.

-time length

specifies how long (in seconds) **s_time** should establish connections and optionally transfer payload data from a server. Server and client performance and the link speed determine how many connections **s_time** can establish.

NOTES

s_time can be used to measure the performance of an SSL connection. To connect to an SSL HTTP server and get the default page the command

```
openssl s_time -connect servername:443 -www / -CApath yourdir -CAfile yourfile.pem \
    -cipher commoncipher [-ssl3]
```

would typically be used (https uses port 443). 'commoncipher' is a cipher to which both client and server can agree, see the ciphers(1) command for details.

If the handshake fails then there are several possible causes, if it is nothing obvious like no client certificate then the **-bugs**, **-ssl2**, **-ssl3** options can be tried in case it is a buggy server. In particular you should play with these options **before** submitting a bug report to an OpenSSL mailing list.

A frequent problem when attempting to get client certificates working is that a web client complains it has no certificates or gives an empty list to choose from. This is normally because the server is not sending the clients certificate authority in its "acceptable

CA list" when it requests a certificate. By using s_client(1) the CA list can be viewed and checked. However some servers only request client authentication after a specific URL is requested. To obtain the list in this case it is necessary to use the **-prexit** option of s_client(1) and send an HTTP request for an appropriate page.

If a certificate is specified on the command line using the **-cert** option it will not be used unless the server specifically requests a client certificate. Therefor merely including a client certificate on the command line is no guarantee that the certificate works.

BUGS

Because this program does not have all the options of the s_client(1) program to turn protocols on and off, you may not be able to measure the performance of all protocols with all servers.

The **-verify** option should really exit if the server verification fails.

SEE ALSO

s_client(1), s_server(1), ciphers(1)

Name

sess_id — SSL/TLS session handling utility

Synopsis

```
opensslsess_id
[-inform PEM|DER]
[-outform PEM|DER]
[-in filename]
[-out filename]
[-text]
[-noout]
[-context ID]
```

DESCRIPTION

The **sess_id** process the encoded version of the SSL session structure and optionally prints out SSL session details (for example the SSL session master key) in human readable format. Since this is a diagnostic tool that needs some knowledge of the SSL protocol to use properly, most users will not need to use it.

-inform DER|PEM

This specifies the input format. The **DER** option uses an ASN1 DER encoded format containing session details. The precise format can vary from one version to the next. The **PEM** form is the default format: it consists of the **DER** format base64 encoded with additional header and footer lines.

-outform DER|PEM

This specifies the output format, the options have the same meaning as the **-inform** option.

-in filename

This specifies the input filename to read session information from or standard input by default.

-out filename

This specifies the output filename to write session information to or standard output if this option is not specified.

-text

prints out the various public or private key components in plain text in addition to the encoded version.

-cert

if a certificate is present in the session it will be output using this option, if the **-text** option is also present then it will be printed out in text form.

-noout

this option prevents output of the encoded version of the session.

-context ID

this option can set the session id so the output session information uses the supplied ID. The ID can be any string of characters. This option wont normally be used.

OUTPUT

Typical output:

```
SSL-Session:
    Protocol  : TLSv1
    Cipher    : 0016
    Session-ID: 871E62626C554CE95488823752CBD5F3673A3EF3DCE9C67BD916C809914B40ED
    Session-ID-ctx: 01000000
    Master-Key: A7CEFC571974BE02CAC305269DC59F76EA9F0B180CB6642697A68251F2D2BB57E51DBBB4C7885573192AE9AEE220FACD
    Key-Arg   : None
    Start Time: 948459261
    Timeout   : 300 (sec)
    Verify return code 0 (ok)
```

Theses are described below in more detail.

Protocol

this is the protocol in use TLSv1, SSLv3 or SSLv2.

Cipher

the cipher used this is the actual raw SSL or TLS cipher code, see the SSL or TLS specifications for more information.

Session-ID

the SSL session ID in hex format.

Session-ID-ctx

the session ID context in hex format.

Master-Key

this is the SSL session master key.

Key-Arg

the key argument, this is only used in SSL v2.

Start Time

this is the session start time represented as an integer in standard Unix format.

Timeout

the timeout in seconds.

Verify return code

this is the return code when an SSL client certificate is verified.

NOTES

The PEM encoded session format uses the header and footer lines:

```
-----BEGIN SSL SESSION PARAMETERS-----
-----END SSL SESSION PARAMETERS-----
```

Since the SSL session output contains the master key it is possible to read the contents of an encrypted session using this information. Therefore appropriate security precautions should be taken if the information is being output by a "real" application. This is however strongly discouraged and should only be used for debugging purposes.

BUGS

The cipher and start time should be printed out in human readable form.

SEE ALSO

ciphers(1), s_server(1)

Name

smime — S/MIME utility

Synopsis

```
opensslsmime
[-encrypt]
[-decrypt]
[-sign]
[-resign]
[-verify]
[-pk7out]
[-[cipher]]
[-in file]
[-no_alt_chains]
[-certfile file]
[-signer file]
[-recip  file]
[-inform SMIME|PEM|DER]
[-passin arg]
[-inkey file]
[-out file]
[-outform SMIME|PEM|DER]
[-content file]
[-to addr]
[-from ad]
[-subject s]
[-text]
[-indef]
[-noindef]
[-stream]
[-rand file(s)]
[-md digest]
[cert.pem]…
```

DESCRIPTION

The **smime** command handles S/MIME mail. It can encrypt, decrypt, sign and verify S/MIME messages.

COMMAND OPTIONS

There are six operation options that set the type of operation to be performed. The meaning of the other options varies according to the operation type.

-encrypt

encrypt mail for the given recipient certificates. Input file is the message to be encrypted. The output file is the encrypted mail in MIME format.

Note that no revocation check is done for the recipient cert, so if that key has been compromised, others may be able to decrypt the text.

-decrypt

decrypt mail using the supplied certificate and private key. Expects an encrypted mail message in MIME format for the input file. The decrypted mail is written to the output file.

-sign

sign mail using the supplied certificate and private key. Input file is the message to be signed. The signed message in MIME format is written to the output file.

-verify

verify signed mail. Expects a signed mail message on input and outputs the signed data. Both clear text and opaque signing is supported.

-pk7out

takes an input message and writes out a PEM encoded PKCS#7 structure.

-resign

resign a message: take an existing message and one or more new signers.

-in filename

the input message to be encrypted or signed or the MIME message to be decrypted or verified.

-inform SMIME|PEM|DER

this specifies the input format for the PKCS#7 structure. The default is **SMIME** which reads an S/MIME format message. **PEM** and **DER** format change this to expect PEM and DER format PKCS#7 structures instead. This currently only affects the input format of the PKCS#7 structure, if no PKCS#7 structure is being input (for example with **-encrypt** or **-sign**) this option has no effect.

-out filename

the message text that has been decrypted or verified or the output MIME format message that has been signed or verified.

-outform SMIME|PEM|DER

this specifies the output format for the PKCS#7 structure. The default is **SMIME** which write an S/MIME format message. **PEM** and **DER** format change this to write PEM and DER format PKCS#7 structures instead. This currently only affects the output format of the PKCS#7 structure, if no PKCS#7 structure is being output (for example with **-verify** or **-decrypt**) this option has no effect.

-stream -indef -noindef

the **-stream** and **-indef** options are equivalent and enable streaming I/O for encoding operations. This permits single pass processing of data without the need to hold the entire contents in memory, potentially supporting very large files. Streaming is automatically set for S/MIME signing with detached data if the output format is **SMIME** it is currently off by default for all other operations.

-noindef

disable streaming I/O where it would produce and indefinite length constructed encoding. This option currently has no effect. In future streaming will be enabled by default on all relevant operations and this option will disable it.

-content filename

This specifies a file containing the detached content, this is only useful with the **-verify** command. This is only usable if the PKCS#7 structure is using the detached signature form where the content is not included. This option will override any content if the input format is S/MIME and it uses the multipart/signed MIME content type.

-text

this option adds plain text (text/plain) MIME headers to the supplied message if encrypting or signing. If decrypting or verifying it strips off text headers: if the decrypted or verified message is not of MIME type text/plain then an error occurs.

-CAfile file

a file containing trusted CA certificates, only used with **-verify**.

-CApath dir

a directory containing trusted CA certificates, only used with **-verify**. This directory must be a standard certificate directory: that is a hash of each subject name (using **x509 -hash**) should be linked to each certificate.

-md digest

digest algorithm to use when signing or resigning. If not present then the default digest algorithm for the signing key will be used (usually SHA1).

-[cipher]

the encryption algorithm to use. For example DES (56 bits) - **-des**, triple DES (168 bits) - **-des3**, EVP_get_cipherbyname() function) can also be used preceded by a dash, for example **-aes_128_cbc**. See **enc** for list of ciphers supported by your version of OpenSSL.

If not specified triple DES is used. Only used with **-encrypt**.

-nointern

when verifying a message normally certificates (if any) included in the message are searched for the signing certificate. With this option only the certificates specified in the **-certfile** option are used. The supplied certificates can still be used as untrusted CAs however.

-noverify

do not verify the signers certificate of a signed message.

-nochain

do not do chain verification of signers certificates: that is don't use the certificates in the signed message as untrusted CAs.

-nosigs

don't try to verify the signatures on the message.

-nocerts

when signing a message the signer's certificate is normally included with this option it is excluded. This will reduce the size of the signed message but the verifier must have a copy of the signers certificate available locally (passed using the **-certfile** option for example).

-noattr

normally when a message is signed a set of attributes are included which include the signing time and supported symmetric algorithms. With this option they are not included.

-binary

normally the input message is converted to "canonical" format which is effectively using CR and LF as end of line: as required by the S/MIME specification. When this option is present no translation occurs. This is useful when handling binary data which may not be in MIME format.

-nodetach

when signing a message use opaque signing: this form is more resistant to translation by mail relays but it cannot be read by mail agents that do not support S/MIME. Without this option cleartext signing with the MIME type multipart/signed is used.

-certfile file

allows additional certificates to be specified. When signing these will be included with the message. When verifying these will be searched for the signers certificates. The certificates should be in PEM format.

-signer file

a signing certificate when signing or resigning a message, this option can be used multiple times if more than one signer is required. If a message is being verified then the signers certificates will be written to this file if the verification was successful.

-recip file

the recipients certificate when decrypting a message. This certificate must match one of the recipients of the message or an error occurs.

-inkey file

the private key to use when signing or decrypting. This must match the corresponding certificate. If this option is not specified then the private key must be included in the certificate file specified with the **-recip** or **-signer** file. When signing this option can be used multiple times to specify successive keys.

-passin arg

the private key password source. For more information about the format of **arg** see the **PASS PHRASE ARGUMENTS** section in openssl(1).

-rand file(s)

a file or files containing random data used to seed the random number generator, or an EGD socket (see RAND_egd(3)). Multiple files can be specified separated by a OS-dependent character. The separator is **;** for MS-Windows, **,** for OpenVMS, and **:** for all others.

cert.pem…

one or more certificates of message recipients: used when encrypting a message.

-to, -from, -subject

the relevant mail headers. These are included outside the signed portion of a message so they may be included manually. If signing then many S/MIME mail clients check the signers certificate's email address matches that specified in the From: address.

-purpose, -ignore_critical, -issuer_checks, -crl_check, -crl_check_all, -policy_check, -extended_crl, -x509_strict, -policy -check_ss_sig -no_alt_chains

Set various options of certificate chain verification. See **verify** manual page for details.

NOTES

The MIME message must be sent without any blank lines between the headers and the output. Some mail programs will automatically add a blank line. Piping the mail directly to sendmail is one way to achieve the correct format.

The supplied message to be signed or encrypted must include the necessary MIME headers or many S/MIME clients wont display it properly (if at all). You can use the **-text** option to automatically add plain text headers.

A "signed and encrypted" message is one where a signed message is then encrypted. This can be produced by encrypting an already signed message: see the examples section.

This version of the program only allows one signer per message but it will verify multiple signers on received messages. Some S/MIME clients choke if a message contains multiple signers. It is possible to sign messages "in parallel" by signing an already signed message.

The options **-encrypt** and **-decrypt** reflect common usage in S/MIME clients. Strictly speaking these process PKCS#7 enveloped data: PKCS#7 encrypted data is used for other purposes.

The **-resign** option uses an existing message digest when adding a new signer. This means that attributes must be present in at least one existing signer using the same message digest or this operation will fail.

The **-stream** and **-indef** options enable experimental streaming I/O support. As a result the encoding is BER using indefinite length constructed encoding and no longer DER. Streaming is supported for the **-encrypt** operation and the **-sign** operation if the content is not detached.

Streaming is always used for the **-sign** operation with detached data but since the content is no longer part of the PKCS#7 structure the encoding remains DER.

EXIT CODES

0 the operation was completely successfully.

1 an error occurred parsing the command options.

2 one of the input files could not be read.

3 an error occurred creating the PKCS#7 file or when reading the MIME message.

4 an error occurred decrypting or verifying the message.

5 the message was verified correctly but an error occurred writing out the signers certificates.

EXAMPLES

Create a cleartext signed message:

```
openssl smime -sign -in message.txt -text -out mail.msg \
       -signer mycert.pem
```

Create an opaque signed message:

```
openssl smime -sign -in message.txt -text -out mail.msg -nodetach \
       -signer mycert.pem
```

Create a signed message, include some additional certificates and read the private key from another file:

```
openssl smime -sign -in in.txt -text -out mail.msg \
       -signer mycert.pem -inkey mykey.pem -certfile mycerts.pem
```

Create a signed message with two signers:

```
openssl smime -sign -in message.txt -text -out mail.msg \
       -signer mycert.pem -signer othercert.pem
```

Send a signed message under Unix directly to sendmail, including headers:

```
openssl smime -sign -in in.txt -text -signer mycert.pem \
       -from steve@openssl.org -to someone@somewhere \
       -subject "Signed message" | sendmail someone@somewhere
```

Verify a message and extract the signer's certificate if successful:

```
openssl smime -verify -in mail.msg -signer user.pem -out signedtext.txt
```

Send encrypted mail using triple DES:

```
openssl smime -encrypt -in in.txt -from steve@openssl.org \
       -to someone@somewhere -subject "Encrypted message" \
       -des3 user.pem -out mail.msg
```

Sign and encrypt mail:

```
openssl smime -sign -in ml.txt -signer my.pem -text \
       | openssl smime -encrypt -out mail.msg \
       -from steve@openssl.org -to someone@somewhere \
       -subject "Signed and Encrypted message" -des3 user.pem
```

Note: the encryption command does not include the **-text** option because the message being encrypted already has MIME headers.

Decrypt mail:

```
openssl smime -decrypt -in mail.msg -recip mycert.pem -inkey key.pem
```

The output from Netscape form signing is a PKCS#7 structure with the detached signature format. You can use this program to verify the signature by line wrapping the base64 encoded structure and surrounding it with:

```
-----BEGIN PKCS7-----
-----END PKCS7-----
```

and using the command:

```
openssl smime -verify -inform PEM -in signature.pem -content content.txt
```

Alternatively you can base64 decode the signature and use:

```
openssl smime -verify -inform DER -in signature.der -content content.txt
```

Create an encrypted message using 128 bit Camellia:

```
openssl smime -encrypt -in plain.txt -camellia128 -out mail.msg cert.pem
```

Add a signer to an existing message:

```
openssl smime -resign -in mail.msg -signer newsign.pem -out mail2.msg
```

BUGS

The MIME parser isn't very clever: it seems to handle most messages that I've thrown at it but it may choke on others.

The code currently will only write out the signer's certificate to a file: if the signer has a separate encryption certificate this must be manually extracted. There should be some heuristic that determines the correct encryption certificate.

Ideally a database should be maintained of a certificates for each email address.

The code doesn't currently take note of the permitted symmetric encryption algorithms as supplied in the SMIMECapabilities signed attribute. This means the user has to manually include the correct encryption algorithm. It should store the list of permitted ciphers in a database and only use those.

No revocation checking is done on the signer's certificate.

The current code can only handle S/MIME v2 messages, the more complex S/MIME v3 structures may cause parsing errors.

HISTORY

The use of multiple **-signer** options and the **-resign** command were first added in OpenSSL 1.0.0

The -no_alt_chains options was first added to OpenSSL 1.0.1n and 1.0.2b.

Name

speed — test library performance

Synopsis

```
openssl speed
[-engine id]
[md2]
[mdc2]
[md5]
[hmac]
[sha1]
[rmd160]
[idea-cbc]
[rc2-cbc]
[rc5-cbc]
[bf-cbc]
[des-cbc]
[des-ede3]
[rc4]
[rsa512]
[rsa1024]
[rsa2048]
[rsa4096]
[dsa512]
[dsa1024]
[dsa2048]
[idea]
[rc2]
[des]
[rsa]
[blowfish]
```

DESCRIPTION

This command is used to test the performance of cryptographic algorithms.

OPTIONS

-engine id

specifying an engine (by its unique **id** string) will cause **speed** to attempt to obtain a functional reference to the specified engine, thus initialising it if needed. The engine will then be set as the default for all available algorithms.

[zero or more test algorithms]

If any options are given, **speed** tests those algorithms, otherwise all of the above are tested.

Name

spkac — SPKAC printing and generating utility

Synopsis

```
opensslspkac
[-in filename]
[-out filename]
[-key keyfile]
[-passin arg]
[-challenge string]
[-pubkey]
[-spkac spkacname]
[-spksect section]
[-noout]
[-verify]
[-engine id]
```

DESCRIPTION

The **spkac** command processes Netscape signed public key and challenge (SPKAC) files. It can print out their contents, verify the signature and produce its own SPKACs from a supplied private key.

COMMAND OPTIONS

-in filename

This specifies the input filename to read from or standard input if this option is not specified. Ignored if the **-key** option is used.

-out filename

specifies the output filename to write to or standard output by default.

-key keyfile

create an SPKAC file using the private key in **keyfile**. The **-in**, **-noout**, **-spksect** and **-verify** options are ignored if present.

-passin password

the input file password source. For more information about the format of **arg** see the **PASS PHRASE ARGUMENTS** section in openssl(1).

-challenge string

specifies the challenge string if an SPKAC is being created.

-spkac spkacname

allows an alternative name form the variable containing the SPKAC. The default is "SPKAC". This option affects both generated and input SPKAC files.

-spksect section

allows an alternative name form the section containing the SPKAC. The default is the default section.

-noout

don't output the text version of the SPKAC (not used if an SPKAC is being created).

-pubkey

output the public key of an SPKAC (not used if an SPKAC is being created).

-verify

verifies the digital signature on the supplied SPKAC.

-engine id

specifying an engine (by its unique **id** string) will cause **spkac** to attempt to obtain a functional reference to the specified engine, thus initialising it if needed. The engine will then be set as the default for all available algorithms.

EXAMPLES

Print out the contents of an SPKAC:

```
openssl spkac -in spkac.cnf
```

Verify the signature of an SPKAC:

```
openssl spkac -in spkac.cnf -noout -verify
```

Create an SPKAC using the challenge string "hello":

```
openssl spkac -key key.pem -challenge hello -out spkac.cnf
```

Example of an SPKAC, (long lines split up for clarity):

```
SPKAC=MIG5MGUwXDANBgkqhkiG9w0BAQEFAANLADBIAkEA1cCoq2Wa3Ixs47uI7F\
PVwHVIPDx5yso105Y6zpozam135a8R0CpoRvkkigIyXfcCjiVi5oWk+6FfPaD03u\
PFoQIDAQABFgVoZWxsbzANBgkqhkiG9w0BAQQFAANBAFpQtY/FojdwkJh1bEIYuc\
2EeM2KHTWPEepWYeawvHD0gQ3DngSC75YCWnnDdq+NQ3F+X4deMx9AaEglZtULwV\
4=
```

NOTES

A created SPKAC with suitable DN components appended can be fed into the **ca** utility.

SPKACs are typically generated by Netscape when a form is submitted containing the **KEYGEN** tag as part of the certificate enrollment process.

The challenge string permits a primitive form of proof of possession of private key. By checking the SPKAC signature and a random challenge string some guarantee is given that the user knows the private key corresponding to the public key being certified. This is important in some applications. Without this it is possible for a previous SPKAC to be used in a "replay attack".

SEE ALSO

ca(1)

Name

ts — Time Stamping Authority tool (client/server)

Synopsis

```
opensslts-query
[-rand file:file…]
[-config configfile]
[-data file_to_hash]
[-digest digest_bytes]
[-md2|-md4|-md5|-sha|-sha1|-mdc2|-ripemd160|…]
[-policy object_id]
[-no_nonce]
[-cert]
[-in request.tsq]
[-out request.tsq]
[-text]
```

```
opensslts-reply
[-config configfile]
[-section tsa_section]
[-queryfile request.tsq]
[-passin password_src]
[-signer tsa_cert.pem]
[-inkey private.pem]
[-chain certs_file.pem]
[-policy object_id]
[-in response.tsr]
[-token_in]
[-out response.tsr]
[-token_out]
[-text]
[-engine id]
```

```
opensslts-verify
[-data file_to_hash]
[-digest digest_bytes]
[-queryfile request.tsq]
[-in response.tsr]
[-token_in]
[-CApath trusted_cert_path]
[-CAfile trusted_certs.pem]
[-untrusted cert_file.pem]
```

DESCRIPTION

The **ts** command is a basic Time Stamping Authority (TSA) client and server application as specified in RFC 3161 (Time-Stamp Protocol, TSP). A TSA can be part of a PKI deployment and its role is to provide long term proof of the existence of a certain datum before a particular time. Here is a brief description of the protocol:

1. The TSA client computes a one-way hash value for a data file and sends the hash to the TSA.

2. The TSA attaches the current date and time to the received hash value, signs them and sends the time stamp token back to the client. By creating this token the TSA certifies the existence of the original data file at the time of response generation.

3. The TSA client receives the time stamp token and verifies the signature on it. It also checks if the token contains the same hash value that it had sent to the TSA.

There is one DER encoded protocol data unit defined for transporting a time stamp request to the TSA and one for sending the time stamp response back to the client. The **ts** command has three main functions: creating a time stamp request based on a data file, creating a time stamp response based on a request, verifying if a response corresponds to a particular request or a data file.

There is no support for sending the requests/responses automatically over HTTP or TCP yet as suggested in RFC 3161. The users must send the requests either by ftp or e-mail.

OPTIONS

Time Stamp Request generation

The **-query** switch can be used for creating and printing a time stamp request with the following options:

-rand file:file…

The files containing random data for seeding the random number generator. Multiple files can be specified, the separator is **;** for MS-Windows, **,** for VMS and **:** for all other platforms. (Optional)

-config configfile

The configuration file to use, this option overrides the **OPENSSL_CONF** environment variable. Only the OID section of the config file is used with the **-query** command. (Optional)

-data file_to_hash

The data file for which the time stamp request needs to be created. stdin is the default if neither the **-data** nor the **-digest** parameter is specified. (Optional)

-digest digest_bytes

It is possible to specify the message imprint explicitly without the data file. The imprint must be specified in a hexadecimal format, two characters per byte, the bytes optionally separated by colons (e.g. 1A:F6:01:… or 1AF601…). The number of bytes must match the message digest algorithm in use. (Optional)

-md2|-md4|-md5|-sha|-sha1|-mdc2|-ripemd160|…

The message digest to apply to the data file, it supports all the message digest algorithms that are supported by the openssl **dgst** command. The default is SHA-1. (Optional)

-policy object_id

The policy that the client expects the TSA to use for creating the time stamp token. Either the dotted OID notation or OID names defined in the config file can be used. If no policy is requested the TSA will use its own default policy. (Optional)

-no_nonce

No nonce is specified in the request if this option is given. Otherwise a 64 bit long pseudo-random none is included in the request. It is recommended to use nonce to protect against replay-attacks. (Optional)

-cert

The TSA is expected to include its signing certificate in the response. (Optional)

-in request.tsq

This option specifies a previously created time stamp request in DER format that will be printed into the output file. Useful when you need to examine the content of a request in human-readable

format. (Optional)

-out request.tsq

Name of the output file to which the request will be written. Default is stdout. (Optional)

-text

If this option is specified the output is human-readable text format instead of DER. (Optional)

Time Stamp Response generation

A time stamp response (TimeStampResp) consists of a response status and the time stamp token itself (ContentInfo), if the token generation was successful. The **-reply** command is for creating a time stamp response or time stamp token based on a request and printing the response/token in human-readable format. If **-token_out** is not specified the output is always a time stamp response (TimeStampResp), otherwise it is a time stamp token (ContentInfo).

-config configfile

The configuration file to use, this option overrides the **OPENSSL_CONF** environment variable. See **CONFIGURATION FILE OPTIONS** for configurable variables. (Optional)

-section tsa_section

The name of the config file section conatining the settings for the response generation. If not specified the default TSA section is used, see **CONFIGURATION FILE OPTIONS** for details. (Optional)

-queryfile request.tsq

The name of the file containing a DER encoded time stamp request. (Optional)

-passin password_src

Specifies the password source for the private key of the TSA. See **PASS PHRASE ARGUMENTS** in openssl(1). (Optional)

-signer tsa_cert.pem

The signer certificate of the TSA in PEM format. The TSA signing certificate must have exactly one extended key usage assigned to it: timeStamping. The extended key usage must also be critical, otherwise the certificate is going to be refused. Overrides the **signer_cert** variable of the config file. (Optional)

-inkey private.pem

The signer private key of the TSA in PEM format. Overrides the **signer_key** config file option. (Optional)

-chain certs_file.pem

The collection of certificates in PEM format that will all be included in the response in addition to the signer certificate if the **-cert** option was used for the request. This file is supposed to contain the certificate chain for the signer certificate from its issuer upwards. The **-reply** command does not build a certificate chain automatically. (Optional)

-policy object_id

The default policy to use for the response unless the client explicitly requires a particular TSA policy. The OID can be specified either in dotted notation or with its name. Overrides the **default_policy** config file option. (Optional)

-in response.tsr

Specifies a previously created time stamp response or time stamp token (if **-token_in** is also specified) in DER format that will be written to the output file. This option does not require a request, it is useful e.g. when you need to examine the content of a response or token or you want to extract the time stamp token from a response. If the input is a token and the output is a time stamp response a default 'granted' status info is added to the token. (Optional)

-token_in

This flag can be used together with the **-in** option and indicates that the input is a DER encoded time stamp token (ContentInfo) instead of a time stamp response (TimeStampResp). (Optional)

-out response.tsr

The response is written to this file. The format and content of the file depends on other options (see **-text**, **-token_out**). The default is stdout. (Optional)

-token_out

The output is a time stamp token (ContentInfo) instead of time stamp response (TimeStampResp). (Optional)

-text

If this option is specified the output is human-readable text format instead of DER. (Optional)

-engine id

Specifying an engine (by its unique **id** string) will cause **ts** to attempt to obtain a functional reference to the specified engine, thus initialising it if needed. The engine will then be set as the default for all available algorithms. Default is builtin. (Optional)

Time Stamp Response verification

The **-verify** command is for verifying if a time stamp response or time stamp token is valid and matches a particular time stamp request or data file. The **-verify** command does not use the configuration file.

-data file_to_hash

The response or token must be verified against file_to_hash. The file is hashed with the message digest algorithm specified in the token. The **-digest** and **-queryfile** options must not be specified with this one. (Optional)

-digest digest_bytes

The response or token must be verified against the message digest specified with this option. The number of bytes must match the message digest algorithm specified in the token. The **-data** and **-queryfile** options must not be specified with this one. (Optional)

-queryfile request.tsq

The original time stamp request in DER format. The **-data** and **-digest** options must not be specified with this one. (Optional)

-in response.tsr

The time stamp response that needs to be verified in DER format. (Mandatory)

-token_in

This flag can be used together with the **-in** option and indicates that the input is a DER encoded time stamp token (ContentInfo) instead of a time stamp response (TimeStampResp). (Optional)

-CApath trusted_cert_path

The name of the directory containing the trused CA certificates of the client. See the similar option of verify(1) for additional details. Either this option or **-CAfile** must be specified. (Optional)

-CAfile trusted_certs.pem

The name of the file containing a set of trusted self-signed CA certificates in PEM format. See the similar option of verify(1) for additional details. Either this option or **-CApath** must be specified. (Optional)

-untrusted cert_file.pem

Set of additional untrusted certificates in PEM format which may be needed when building the certificate chain for the TSA's signing certificate. This file must contain the TSA signing certificate and all intermediate CA certificates unless the response includes them. (Optional)

CONFIGURATION FILE OPTIONS

The **-query** and **-reply** commands make use of a configuration file defined by the **OPENSSL_CONF** environment variable. See config(5) for a general description of the syntax of the config file. The **-query** command uses only the symbolic OID names section and it can work without it. However, the **-reply** command needs the config file for its operation.

When there is a command line switch equivalent of a variable the switch always overrides the settings in the config file.

tsa section, **default_tsa**

This is the main section and it specifies the name of another section that contains all the options for the **-reply** command. This default section can be overridden with the **-section** command line switch. (Optional)

oid_file

See ca(1) for description. (Optional)

oid_section

See ca(1) for description. (Optional)

RANDFILE

See ca(1) for description. (Optional)

serial

The name of the file containing the hexadecimal serial number of the last time stamp response created. This number is incremented by 1 for each response. If the file does not exist at the time of response generation a new file is created with serial number 1. (Mandatory)

crypto_device

Specifies the OpenSSL engine that will be set as the default for all available algorithms. The default value is builtin, you can specify any other engines supported by OpenSSL (e.g. use chil for the NCipher HSM). (Optional)

signer_cert

TSA signing certificate in PEM format. The same as the **-signer** command line option. (Optional)

certs

A file containing a set of PEM encoded certificates that need to be included in the response. The same as the **-chain** command line option. (Optional)

signer_key

The private key of the TSA in PEM format. The same as the **-inkey** command line option. (Optional)

default_policy

The default policy to use when the request does not mandate any policy. The same as the **-policy** command line option. (Optional)

other_policies

Comma separated list of policies that are also acceptable by the TSA and used only if the request explicitly specifies one of them. (Optional)

digests

The list of message digest algorithms that the TSA accepts. At least one algorithm must be specified. (Mandatory)

accuracy

The accuracy of the time source of the TSA in seconds, milliseconds and microseconds. E.g. secs:1, millisecs:500, microsecs:100. If any of the components is missing zero is assumed for that field. (Optional)

clock_precision_digits

Specifies the maximum number of digits, which represent the fraction of seconds, that need to be included in the time field. The trailing zeroes must be removed from the time, so there might actually be fewer digits, or no fraction of seconds at all. Supported only on UNIX platforms. The maximum value is 6, default is 0. (Optional)

ordering

If this option is yes the responses generated by this TSA can always be ordered, even if the time difference between two responses is less than the sum of their accuracies. Default is no. (Optional)

tsa_name

Set this option to yes if the subject name of the TSA must be included in the TSA name field of the response. Default is no. (Optional)

ess_cert_id_chain

The SignedData objects created by the TSA always contain the certificate identifier of the signing certificate in a signed attribute (see RFC 2634, Enhanced Security Services). If this option is set to yes and either the **certs** variable or the **-chain** option is specified then the certificate identifiers of the chain will also be included in the SigningCertificate signed attribute. If this variable is set to no, only the signing certificate identifier is included. Default is no. (Optional)

ENVIRONMENT VARIABLES

OPENSSL_CONF contains the path of the configuration file and can be overridden by the **-config** command line option.

EXAMPLES

All the examples below presume that **OPENSSL_CONF** is set to a proper configuration file, e.g. the example configuration file openssl/apps/openssl.cnf will do.

Time Stamp Request

To create a time stamp request for design1.txt with SHA-1 without nonce and policy and no certificate is required in the response:

```
openssl ts -query -data design1.txt -no_nonce \
      -out design1.tsq
```

To create a similar time stamp request with specifying the message imprint explicitly:

```
openssl ts -query -digest b7e5d3f93198b38379852f2c04e78d73abdd0f4b \
       -no_nonce -out design1.tsq
```

To print the content of the previous request in human readable format:

```
openssl ts -query -in design1.tsq -text
```

To create a time stamp request which includes the MD-5 digest of design2.txt, requests the signer certificate and nonce, specifies a policy id (assuming the tsa_policy1 name is defined in the OID section of the config file):

```
openssl ts -query -data design2.txt -md5 \
      -policy tsa_policy1 -cert -out design2.tsq
```

Time Stamp Response

Before generating a response a signing certificate must be created for the TSA that contains the **timeStamping** critical extended key usage extension without any other key usage extensions. You can add the 'extendedKeyUsage = critical,timeStamping' line to the user certificate section of the config file to generate a proper certificate. See req(1), ca(1), x509(1) for instructions. The examples below assume that cacert.pem contains the certificate of the CA, tsacert.pem is the signing certificate issued by cacert.pem and tsakey.pem is the private key of the TSA.

To create a time stamp response for a request:

```
openssl ts -reply -queryfile design1.tsq -inkey tsakey.pem \
      -signer tsacert.pem -out design1.tsr
```

If you want to use the settings in the config file you could just write:

```
openssl ts -reply -queryfile design1.tsq -out design1.tsr
```

To print a time stamp reply to stdout in human readable format:

```
openssl ts -reply -in design1.tsr -text
```

To create a time stamp token instead of time stamp response:

```
openssl ts -reply -queryfile design1.tsq -out design1_token.der -token_out
```

To print a time stamp token to stdout in human readable format:

```
openssl ts -reply -in design1_token.der -token_in -text -token_out
```

To extract the time stamp token from a response:

```
openssl ts -reply -in design1.tsr -out design1_token.der -token_out
```

To add 'granted' status info to a time stamp token thereby creating a valid response:

```
openssl ts -reply -in design1_token.der -token_in -out design1.tsr
```

Time Stamp Verification

To verify a time stamp reply against a request:

```
openssl ts -verify -queryfile design1.tsq -in design1.tsr \
      -CAfile cacert.pem -untrusted tsacert.pem
```

To verify a time stamp reply that includes the certificate chain:

```
openssl ts -verify -queryfile design2.tsq -in design2.tsr \
      -CAfile cacert.pem
```

To verify a time stamp token against the original data file: openssl ts -verify -data design2.txt -in design2.tsr \ -CAfile cacert.pem

To verify a time stamp token against a message imprint: openssl ts -verify -digest b7e5d3f93198b38379852f2c04e78d73abdd0f4b \ -in design2.tsr -CAfile cacert.pem

You could also look at the 'test' directory for more examples.

BUGS

If you find any bugs or you have suggestions please write to Zoltan Glozik <zglozik@opentsa.org>. Known issues:

- No support for time stamps over SMTP, though it is quite easy to implement an automatic e-mail based TSA with procmail and perl. HTTP server support is provided in the form of a separate apache module. HTTP client support is provided by tsget(1). Pure TCP/IP protocol is not supported.

- The file containing the last serial number of the TSA is not locked when being read or written. This is a problem if more than one instance of openssl(1) is trying to create a time stamp response at the same time. This is not an issue when using the apache server module, it does proper locking.

- Look for the FIXME word in the source files.

- The source code should really be reviewed by somebody else, too.

- More testing is needed, I have done only some basic tests (see test/testtsa).

AUTHOR

Zoltan Glozik <zglozik@opentsa.org>, OpenTSA project (http://www.opentsa.org)

SEE ALSO

tsget(1), openssl(1), req(1), x509(1), ca(1), genrsa(1), config(5)

Name

verify — Utility to verify certificates.

Synopsis

```
opensslverify
[-CApath directory]
[-CAfile file]
[-purpose purpose]
[-policy arg]
[-ignore_critical]
[-crl_check]
[-crl_check_all]
[-policy_check]
[-explicit_policy]
[-inhibit_any]
[-inhibit_map]
[-x509_strict]
[-extended_crl]
[-use_deltas]
[-policy_print]
[-no_alt_chains]
[-allow_proxy_certs]
[-untrusted file]
[-help]
[-issuer_checks]
[-attime timestamp]
[-verbose]
[-]
[certificates]
```

DESCRIPTION

The **verify** command verifies certificate chains.

COMMAND OPTIONS

-CApath directory

A directory of trusted certificates. The certificates should have names of the form: hash.0 or have symbolic links to them of this form ("hash" is the hashed certificate subject name: see the **-hash** option of the **x509** utility). Under Unix the **c_rehash** script will automatically create symbolic links to a directory of certificates.

-CAfile file A file of trusted certificates. The file should contain multiple certificates in PEM format concatenated together.

-untrusted file

A file of untrusted certificates. The file should contain multiple certificates in PEM format concatenated together.

-purpose purpose

The intended use for the certificate. If this option is not specified, **verify** will not consider certificate purpose during chain verification. Currently accepted uses are **sslclient**, **sslserver**, **nssslserver**, **smimesign**, **smimeencrypt**. See the **VERIFY OPERATION** section for more information.

-help

Print out a usage message.

-verbose

Print extra information about the operations being performed.

-issuer_checks

Print out diagnostics relating to searches for the issuer certificate of the current certificate. This shows why each candidate issuer certificate was rejected. The presence of rejection messages does not itself imply that anything is wrong; during the normal verification process, several rejections may take place.

-attime timestamp

Perform validation checks using time specified by **timestamp** and not current system time. **timestamp** is the number of seconds since 01.01.1970 (UNIX time).

-policy arg

Enable policy processing and add **arg** to the user-initial-policy-set (see RFC5280). The policy **arg** can be an object name an OID in numeric form. This argument can appear more than once.

-policy_check

Enables certificate policy processing.

-explicit_policy

Set policy variable require-explicit-policy (see RFC5280).

-inhibit_any

Set policy variable inhibit-any-policy (see RFC5280).

-inhibit_map

Set policy variable inhibit-policy-mapping (see RFC5280).

-no_alt_chains

When building a certificate chain, if the first certificate chain found is not trusted, then OpenSSL will continue to check to see if an alternative chain can be found that is trusted. With this option that behaviour is suppressed so that only the first chain found is ever used. Using this option will force the behaviour to match that of previous OpenSSL versions.

-allow_proxy_certs

Allow the verification of proxy certificates.

-policy_print

Print out diagnostics related to policy processing.

-crl_check

Checks end entity certificate validity by attempting to look up a valid CRL. If a valid CRL cannot be found an error occurs.

-crl_check_all

Checks the validity of **all** certificates in the chain by attempting to look up valid CRLs.

-ignore_critical

Normally if an unhandled critical extension is present which is not supported by OpenSSL the certificate is rejected (as required by RFC5280). If this option is set critical extensions are ignored.

-x509_strict

For strict X.509 compliance, disable non-compliant workarounds for broken certificates.

-extended_crl

Enable extended CRL features such as indirect CRLs and alternate CRL signing keys.

-use_deltas

Enable support for delta CRLs.

-check_ss_sig

Verify the signature on the self-signed root CA. This is disabled by default because it doesn't add any security.

-

Indicates the last option. All arguments following this are assumed to be certificate files. This is useful if the first certificate filename begins with a **-**.

certificates

One or more certificates to verify. If no certificates are given, **verify** will attempt to read a certificate from standard input. Certificates must be in PEM format.

VERIFY OPERATION

The **verify** program uses the same functions as the internal SSL and S/MIME verification, therefore this description applies to these verify operations too.

There is one crucial difference between the verify operations performed by the **verify** program: wherever possible an attempt is made to continue after an error whereas normally the verify operation would halt on the first error. This allows all the problems with a certificate chain to be determined.

The verify operation consists of a number of separate steps.

Firstly a certificate chain is built up starting from the supplied certificate and ending in the root CA. It is an error if the whole chain cannot be built up. The chain is built up by looking up the issuers certificate of the current certificate. If a certificate is found which is its own issuer it is assumed to be the root CA.

The process of 'looking up the issuers certificate' itself involves a number of steps. In versions of OpenSSL before 0.9.5a the first certificate whose subject name matched the issuer of the current certificate was assumed to be the issuers certificate. In OpenSSL 0.9.6 and later all certificates whose subject name matches the issuer name of the current certificate are subject to further tests. The relevant authority key identifier components of the current certificate (if present) must match the subject key identifier (if present) and issuer and serial number of the candidate issuer, in addition the keyUsage extension of the candidate issuer (if present) must permit certificate signing.

The lookup first looks in the list of untrusted certificates and if no match is found the remaining lookups are from the trusted certificates. The root CA is always looked up in the trusted certificate list: if the certificate to verify is a root certificate then an exact match must be found in the trusted list.

The second operation is to check every untrusted certificate's extensions for consistency with the supplied purpose. If the **-purpose** option is not included then no checks are done. The supplied or "leaf" certificate must have extensions compatible with the supplied purpose and all other certificates must also be valid CA certificates. The precise extensions required are described in more detail in the **CERTIFICATE EXTENSIONS** section of the **x509** utility.

The third operation is to check the trust settings on the root CA. The root CA should be trusted for the supplied purpose. For compatibility with previous versions of SSLeay and OpenSSL a certificate with no trust settings is considered to be valid for all purposes.

The final operation is to check the validity of the certificate chain. The validity period is checked against the current system time and the notBefore and notAfter dates in the certificate. The certificate signatures are also checked at this point.

If all operations complete successfully then certificate is considered valid. If any operation fails then the certificate is not valid.

DIAGNOSTICS

When a verify operation fails the output messages can be somewhat cryptic. The general form of the error message is:

```
server.pem: /C=AU/ST=Queensland/O=CryptSoft Pty Ltd/CN=Test CA (1024 bit)
error 24 at 1 depth lookup:invalid CA certificate
```

The first line contains the name of the certificate being verified followed by the subject name of the certificate. The second line contains the error number and the depth. The depth is number of the certificate being verified when a problem was detected starting with zero for the certificate being verified itself then 1 for the CA that signed the certificate and so on. Finally a text version of the error number is presented.

An exhaustive list of the error codes and messages is shown below, this also includes the name of the error code as defined in the header file x509_vfy.h Some of the error codes are defined but never returned: these are described as "unused".

0 X509_V_OK: ok

the operation was successful.

2 X509_V_ERR_UNABLE_TO_GET_ISSUER_CERT: unable to get issuer certificate

the issuer certificate of a looked up certificate could not be found. This normally means the list of trusted certificates is not complete.

3 X509_V_ERR_UNABLE_TO_GET_CRL: unable to get certificate CRL

the CRL of a certificate could not be found.

4 X509_V_ERR_UNABLE_TO_DECRYPT_CERT_SIGNATURE: unable to decrypt certificate's signature

the certificate signature could not be decrypted. This means that the actual signature value could not be determined rather than it not matching the expected value, this is only meaningful for RSA keys.

5 X509_V_ERR_UNABLE_TO_DECRYPT_CRL_SIGNATURE: unable to decrypt CRL's signature

the CRL signature could not be decrypted: this means that the actual signature value could not be determined rather than it not matching the expected value. Unused.

6 X509_V_ERR_UNABLE_TO_DECODE_ISSUER_PUBLIC_KEY: unable to decode issuer public key

the public key in the certificate SubjectPublicKeyInfo could not be read.

7 X509_V_ERR_CERT_SIGNATURE_FAILURE: certificate signature failure

the signature of the certificate is invalid.

8 X509_V_ERR_CRL_SIGNATURE_FAILURE: CRL signature failure

the signature of the certificate is invalid.

9 X509_V_ERR_CERT_NOT_YET_VALID: certificate is not yet valid

the certificate is not yet valid: the notBefore date is after the current time.

10 X509_V_ERR_CERT_HAS_EXPIRED: certificate has expired

the certificate has expired: that is the notAfter date is before the current time.

11 X509_V_ERR_CRL_NOT_YET_VALID: CRL is not yet valid

the CRL is not yet valid.

12 X509_V_ERR_CRL_HAS_EXPIRED: CRL has expired

the CRL has expired.

13 X509_V_ERR_ERROR_IN_CERT_NOT_BEFORE_FIELD: format error in certificate's notBefore field

the certificate notBefore field contains an invalid time.

14 X509_V_ERR_ERROR_IN_CERT_NOT_AFTER_FIELD: format error in certificate's notAfter field

the certificate notAfter field contains an invalid time.

15 X509_V_ERR_ERROR_IN_CRL_LAST_UPDATE_FIELD: format error in CRL's lastUpdate field

the CRL lastUpdate field contains an invalid time.

16 X509_V_ERR_ERROR_IN_CRL_NEXT_UPDATE_FIELD: format error in CRL's nextUpdate field

the CRL nextUpdate field contains an invalid time.

17 X509_V_ERR_OUT_OF_MEM: out of memory

an error occurred trying to allocate memory. This should never happen.

18 X509_V_ERR_DEPTH_ZERO_SELF_SIGNED_CERT: self signed certificate

the passed certificate is self signed and the same certificate cannot be found in the list of trusted certificates.

19 X509_V_ERR_SELF_SIGNED_CERT_IN_CHAIN: self signed certificate in certificate chain

the certificate chain could be built up using the untrusted certificates but the root could not be found locally.

20 X509_V_ERR_UNABLE_TO_GET_ISSUER_CERT_LOCALLY: unable to get local issuer certificate

the issuer certificate could not be found: this occurs if the issuer certificate of an untrusted certificate cannot be found.

21 X509_V_ERR_UNABLE_TO_VERIFY_LEAF_SIGNATURE: unable to verify the first certificate

no signatures could be verified because the chain contains only one certificate and it is not self signed.

22 X509_V_ERR_CERT_CHAIN_TOO_LONG: certificate chain too long

the certificate chain length is greater than the supplied maximum depth. Unused.

23 X509_V_ERR_CERT_REVOKED: certificate revoked

the certificate has been revoked.

24 X509_V_ERR_INVALID_CA: invalid CA certificate

a CA certificate is invalid. Either it is not a CA or its extensions are not consistent with the supplied purpose.

25 X509_V_ERR_PATH_LENGTH_EXCEEDED: path length constraint exceeded

the basicConstraints pathlength parameter has been exceeded.

26 X509_V_ERR_INVALID_PURPOSE: unsupported certificate purpose

the supplied certificate cannot be used for the specified purpose.

27 X509_V_ERR_CERT_UNTRUSTED: certificate not trusted

the root CA is not marked as trusted for the specified purpose.

28 X509_V_ERR_CERT_REJECTED: certificate rejected

the root CA is marked to reject the specified purpose.

29 X509_V_ERR_SUBJECT_ISSUER_MISMATCH: subject issuer mismatch

the current candidate issuer certificate was rejected because its subject name did not match the issuer name of the current certificate. Only displayed when the **-issuer_checks** option is set.

30 X509_V_ERR_AKID_SKID_MISMATCH: authority and subject key identifier mismatch

the current candidate issuer certificate was rejected because its subject key identifier was present and did not match the authority key identifier current certificate. Only displayed when the **-issuer_checks** option is set.

31 X509_V_ERR_AKID_ISSUER_SERIAL_MISMATCH: authority and issuer serial number mismatch

the current candidate issuer certificate was rejected because its issuer name and serial number was present and did not match the authority key identifier of the current certificate. Only displayed when the **-issuer_checks** option is set.

32 X509_V_ERR_KEYUSAGE_NO_CERTSIGN:key usage does not include certificate signing

the current candidate issuer certificate was rejected because its keyUsage extension does not permit certificate signing.

50 X509_V_ERR_APPLICATION_VERIFICATION: application verification failure

an application specific error. Unused.

BUGS

Although the issuer checks are a considerable improvement over the old technique they still suffer from limitations in the underlying X509_LOOKUP API. One consequence of this is that trusted certificates with matching subject name must either appear in a file (as specified by the **-CAfile** option) or a directory (as specified by **-CApath**. If they occur in both then only the certificates in the file will be recognised.

Previous versions of OpenSSL assume certificates with matching subject name are identical and mishandled them.

Previous versions of this documentation swapped the meaning of the **X509_V_ERR_UNABLE_TO_GET_ISSUER_CERT** and **20 X509_V_ERR_UNABLE_TO_GET_ISSUER_CERT_LOCALLY** error codes.

SEE ALSO

x509(1)

HISTORY

The -no_alt_chains options was first added to OpenSSL 1.0.1n and 1.0.2b.

Name

version — print OpenSSL version information

Synopsis

```
openssl version
[-a]
[-v]
[-b]
[-o]
[-f]
[-p]
[-d]
```

DESCRIPTION

This command is used to print out version information about OpenSSL.

OPTIONS

-a

all information, this is the same as setting all the other flags.

-v

the current OpenSSL version.

-b

the date the current version of OpenSSL was built.

-o

option information: various options set when the library was built.

-f

compilation flags.

-p

platform setting.

-d

OPENSSLDIR setting.

NOTES

The output of **openssl version -a** would typically be used when sending in a bug report.

HISTORY

The **-d** option was added in OpenSSL 0.9.7.

Name

x509 — Certificate display and signing utility

Synopsis

```
opensslx509
[-inform DER|PEM|NET]
[-outform DER|PEM|NET]
[-keyform DER|PEM]
[-CAform DER|PEM]
[-CAkeyform DER|PEM]
[-in filename]
[-out filename]
[-serial]
[-hash]
[-subject_hash]
[-issuer_hash]
[-ocspid]
[-subject]
[-issuer]
[-nameopt option]
[-email]
[-ocsp_uri]
[-startdate]
[-enddate]
[-purpose]
[-dates]
[-checkend num]
[-modulus]
[-pubkey]
[-fingerprint]
[-alias]
[-noout]
[-trustout]
[-clrtrust]
[-clrreject]
[-addtrust arg]
[-addreject arg]
[-setalias arg]
[-days arg]
[-set_serial n]
[-signkey filename]
[-passin arg]
[-x509toreq]
[-req]
[-CA filename]
[-CAkey filename]
[-CAcreateserial]
[-CAserial filename]
[-text]
[-certopt option]
[-C]
[-md2|-md5|-sha1|-mdc2]
[-clrext]
[-extfile filename]
[-extensions section]
[-engine id]
```

DESCRIPTION

The **x509** command is a multi purpose certificate utility. It can be used to display certificate information, convert certificates to various forms, sign certificate requests like a "mini CA" or edit certificate trust settings.

Since there are a large number of options they will split up into various sections.

OPTIONS

INPUT, OUTPUT AND GENERAL PURPOSE OPTIONS

-inform DER|PEM|NET

This specifies the input format normally the command will expect an X509 certificate but this can change if other options such as **-req** are present. The DER format is the DER encoding of the certificate and PEM is the base64 encoding of the DER encoding with header and footer lines added. The NET option is an obscure Netscape server format that is now obsolete.

-outform DER|PEM|NET

This specifies the output format, the options have the same meaning as the **-inform** option.

-in filename

This specifies the input filename to read a certificate from or standard input if this option is not specified.

-out filename

This specifies the output filename to write to or standard output by default.

-md2|-md5|-sha1|-mdc2

the digest to use. This affects any signing or display option that uses a message digest, such as the **-fingerprint**, **-signkey** and **-CA** options. If not specified then SHA1 is used. If the key being used to sign with is a DSA key then this option has no effect: SHA1 is always used with DSA keys.

-engine id

specifying an engine (by its unique **id** string) will cause **x509** to attempt to obtain a functional reference to the specified engine, thus initialising it if needed. The engine will then be set as the default for all available algorithms.

DISPLAY OPTIONS

Note: the **-alias** and **-purpose** options are also display options but are described in the **TRUST SETTINGS** section.

-text

prints out the certificate in text form. Full details are output including the public key, signature algorithms, issuer and subject names, serial number any extensions present and any trust settings.

-certopt option

customise the output format used with **-text**. The **option** argument can be a single option or multiple options separated by commas. The **-certopt** switch may be also be used more than once to set multiple options. See the **TEXT OPTIONS** section for more information.

-noout

this option prevents output of the encoded version of the request.

-pubkey

outputs the the certificate's SubjectPublicKeyInfo block in PEM format.

-modulus

this option prints out the value of the modulus of the public key contained in the certificate.

-serial

outputs the certificate serial number.

-subject_hash

outputs the "hash" of the certificate subject name. This is used in OpenSSL to form an index to allow certificates in a directory to be looked up by subject name.

-issuer_hash

outputs the "hash" of the certificate issuer name.

-ocspid

outputs the OCSP hash values for the subject name and public key.

-hash

synonym for "-subject_hash" for backward compatibility reasons.

-subject_hash_old

outputs the "hash" of the certificate subject name using the older algorithm as used by OpenSSL versions before 1.0.0.

-issuer_hash_old

outputs the "hash" of the certificate issuer name using the older algorithm as used by OpenSSL versions before 1.0.0.

-subject

outputs the subject name.

-issuer

outputs the issuer name.

-nameopt option

option which determines how the subject or issuer names are displayed. The **option** argument can be a single option or multiple options separated by commas. Alternatively the **-nameopt** switch may be used more than once to set multiple options. See the **NAME OPTIONS** section for more information.

-email

outputs the email address(es) if any.

-ocsp_uri

outputs the OCSP responder address(es) if any.

-startdate

prints out the start date of the certificate, that is the notBefore date.

-enddate

prints out the expiry date of the certificate, that is the notAfter date.

-dates

prints out the start and expiry dates of a certificate.

-checkend arg

checks if the certificate expires within the next **arg** seconds and exits non-zero if yes it will expire or zero if not.

-fingerprint

prints out the digest of the DER encoded version of the whole certificate (see digest options).

-C

this outputs the certificate in the form of a C source file.

TRUST SETTINGS

Please note these options are currently experimental and may well change.

A **trusted certificate** is an ordinary certificate which has several additional pieces of information attached to it such as the permitted and prohibited uses of the certificate and an "alias".

Normally when a certificate is being verified at least one certificate must be "trusted". By default a trusted certificate must be stored locally and must be a root CA: any certificate chain ending in this CA is then usable for any purpose.

Trust settings currently are only used with a root CA. They allow a finer control over the purposes the root CA can be used for. For example a CA may be trusted for SSL client but not SSL server use.

See the description of the **verify** utility for more information on the meaning of trust settings.

Future versions of OpenSSL will recognize trust settings on any certificate: not just root CAs.

-trustout

this causes **x509** to output a **trusted** certificate. An ordinary or trusted certificate can be input but by default an ordinary certificate is output and any trust settings are discarded. With the **-trustout** option a trusted certificate is output. A trusted certificate is automatically output if any trust settings are modified.

-setalias arg

sets the alias of the certificate. This will allow the certificate to be referred to using a nickname for example "Steve's Certificate".

-alias

outputs the certificate alias, if any.

-clrtrust

clears all the permitted or trusted uses of the certificate.

-clrreject

clears all the prohibited or rejected uses of the certificate.

-addtrust arg

adds a trusted certificate use. Any object name can be used here but currently only **clientAuth** (SSL client use), **serverAuth** (SSL server use) and **emailProtection** (S/MIME email) are used. Other OpenSSL applications may define additional uses.

-addreject arg

adds a prohibited use. It accepts the same values as the **-addtrust** option.

-purpose

this option performs tests on the certificate extensions and outputs the results. For a more complete description see the **CERTIFICATE EXTENSIONS** section.

SIGNING OPTIONS

The **x509** utility can be used to sign certificates and requests: it can thus behave like a "mini CA".

-signkey filename

this option causes the input file to be self signed using the supplied private key.

If the input file is a certificate it sets the issuer name to the subject name (i.e. makes it self signed) changes the public key to the supplied value and changes the start and end dates. The start date is set to the current time and the end date is set to a value determined by the **-days** option. Any certificate extensions are retained unless the **-clrext** option is supplied.

If the input is a certificate request then a self signed certificate is created using the supplied private key using the subject name in the request.

-passin arg

the key password source. For more information about the format of **arg** see the **PASS PHRASE ARGUMENTS** section in openssl(1).

-clrext

delete any extensions from a certificate. This option is used when a certificate is being created from another certificate (for example with the **-signkey** or the **-CA** options). Normally all extensions are retained.

-keyform PEM|DER

specifies the format (DER or PEM) of the private key file used in the **-signkey** option.

-days arg

specifies the number of days to make a certificate valid for. The default is 30 days.

-x509toreq

converts a certificate into a certificate request. The **-signkey** option is used to pass the required private key.

-req

by default a certificate is expected on input. With this option a certificate request is expected instead.

-set_serial n

specifies the serial number to use. This option can be used with either the **-signkey** or **-CA** options. If used in conjunction with the **-CA** option the serial number file (as specified by the **-CAserial** or **-CAcreateserial** options) is not used.

The serial number can be decimal or hex (if preceded by **0x**). Negative serial numbers can also be specified but their use is not recommended.

-CA filename

specifies the CA certificate to be used for signing. When this option is present **x509** behaves like a "mini CA". The input file is signed by this CA using this option: that is its issuer name is set to the subject name of the CA and it is digitally signed using the CAs private key.

This option is normally combined with the **-req** option. Without the **-req** option the input is a certificate which must be self signed.

-CAkey filename

sets the CA private key to sign a certificate with. If this option is not specified then it is assumed that the CA private key is present in the CA certificate file.

-CAserial filename

sets the CA serial number file to use.

When the **-CA** option is used to sign a certificate it uses a serial number specified in a file. This file consist of one line containing an even number of hex digits with the serial number to use. After each use the serial number is incremented and written out to the file again.

The default filename consists of the CA certificate file base name with ".srl" appended. For example if the CA certificate file is called "mycacert.pem" it expects to find a serial number file called "mycacert.srl".

-CAcreateserial

with this option the CA serial number file is created if it does not exist: it will contain the serial number "02" and the certificate being signed will have the 1 as its serial number. Normally if the **-CA** option is specified and the serial number file does not exist it is an error.

-extfile filename

file containing certificate extensions to use. If not specified then no extensions are added to the certificate.

-extensions section

the section to add certificate extensions from. If this option is not specified then the extensions should either be contained in the unnamed (default) section or the default section should contain a variable called "extensions" which contains the section to use. See the x509v3_config(5) manual page for details of the extension section format.

NAME OPTIONS

The **nameopt** command line switch determines how the subject and issuer names are displayed. If no **nameopt** switch is present the default "oneline" format is used which is compatible with previous versions of OpenSSL. Each option is described in detail below, all options can be preceded by a **-** to turn the option off. Only the first four will normally be used.

compat

use the old format. This is equivalent to specifying no name options at all.

RFC2253

displays names compatible with RFC2253 equivalent to **esc_2253**, **esc_ctrl**, **esc_msb**, **utf8**, **dump_nostr**, **dump_unknown**, **dump_der**, **sep_comma_plus**, **dn_rev** and **sname**.

oneline

a oneline format which is more readable than RFC2253. It is equivalent to specifying the **esc_2253**, **esc_ctrl**, **esc_msb**, **utf8**, **dump_nostr**, **dump_der**, **use_quote**, **sep_comma_plus_space**, **space_eq** and **sname** options.

multiline

a multiline format. It is equivalent **esc_ctrl**, **esc_msb**, **sep_multiline**, **space_eq**, **lname** and **align**.

esc_2253

escape the "special" characters required by RFC2253 in a field That is **,+"<>;**. Additionally **#** is escaped at the beginning of a string and a space character at the beginning or end of a string.

esc_ctrl

escape control characters. That is those with ASCII values less than 0x20 (space) and the delete (0x7f) character. They are escaped using the RFC2253 \XX notation (where XX are two hex digits representing the character value).

esc_msb

escape characters with the MSB set, that is with ASCII values larger than 127.

use_quote

escapes some characters by surrounding the whole string with **"** characters, without the option all escaping is done with the \ character.

utf8

convert all strings to UTF8 format first. This is required by RFC2253. If you are lucky enough to have a UTF8 compatible terminal then the use of this option (and **not** setting **esc_msb**) may result in the correct display of multibyte (international) characters. Is this option is not present then multibyte characters larger than 0xff will be represented using the format \UXXXX for 16 bits and \WXXXXXXXX for 32 bits. Also if this option is off any UTF8Strings will be converted to their character form first.

ignore_type

this option does not attempt to interpret multibyte characters in any way. That is their content octets are merely dumped as though one octet represents each character. This is useful for diagnostic purposes but will result in rather odd looking output.

show_type

show the type of the ASN1 character string. The type precedes the field contents. For example "BMPSTRING: Hello World".

dump_der

when this option is set any fields that need to be hexdumped will be dumped using the DER encoding of the field. Otherwise just the content octets will be displayed. Both options use the RFC2253 **#XXXX...** format.

dump_nostr

dump non character string types (for example OCTET STRING) if this option is not set then non character string types will be displayed as though each content octet represents a single character.

dump_all

dump all fields. This option when used with **dump_der** allows the DER encoding of the structure to be unambiguously determined.

dump_unknown

dump any field whose OID is not recognised by OpenSSL.

sep_comma_plus, **sep_comma_plus_space**, **sep_semi_plus_space**, **sep_multiline**

these options determine the field separators. The first character is between RDNs and the second between multiple AVAs (multiple AVAs are very rare and their use is discouraged). The options ending in "space" additionally place a space after the separator to make it more readable. The **sep_multiline** uses a linefeed character for the RDN separator and a spaced **+** for the AVA separator. It also indents the fields by four characters. If no field separator is specified then **sep_comma_plus_space** is used by default.

dn_rev

reverse the fields of the DN. This is required by RFC2253. As a side effect this also reverses the order of multiple AVAs but this is permissible.

nofname, **sname**, **lname**, **oid**

these options alter how the field name is displayed. **nofname** does not display the field at all. **sname** uses the "short name" form (CN for commonName for example). **lname** uses the long form. **oid** represents the OID in numerical form and is useful for diagnostic purpose.

align

align field values for a more readable output. Only usable with **sep_multiline**.

space_eq

places spaces round the **=** character which follows the field name.

TEXT OPTIONS

As well as customising the name output format, it is also possible to customise the actual fields printed using the **certopt** options when the **text** option is present. The default behaviour is to print all fields.

compatible

use the old format. This is equivalent to specifying no output options at all.

no_header

don't print header information: that is the lines saying "Certificate" and "Data".

no_version

don't print out the version number.

no_serial

don't print out the serial number.

no_signame

don't print out the signature algorithm used.

no_validity

don't print the validity, that is the **notBefore** and **notAfter** fields.

no_subject

don't print out the subject name.

no_issuer

don't print out the issuer name.

no_pubkey

don't print out the public key.

no_sigdump

don't give a hexadecimal dump of the certificate signature.

no_aux

don't print out certificate trust information.

no_extensions

don't print out any X509V3 extensions.

ext_default

retain default extension behaviour: attempt to print out unsupported certificate extensions.

ext_error

print an error message for unsupported certificate extensions.

ext_parse

ASN1 parse unsupported extensions.

ext_dump

hex dump unsupported extensions.

ca_default

the value used by the **ca** utility, equivalent to **no_issuer**, **no_pubkey**, **no_header**, **no_version**, **no_sigdump** and **no_signame**.

EXAMPLES

Note: in these examples the '\' means the example should be all on one line.

Display the contents of a certificate:

```
openssl x509 -in cert.pem -noout -text
```

Display the certificate serial number:

```
openssl x509 -in cert.pem -noout -serial
```

Display the certificate subject name:

```
openssl x509 -in cert.pem -noout -subject
```

Display the certificate subject name in RFC2253 form:

```
openssl x509 -in cert.pem -noout -subject -nameopt RFC2253
```

Display the certificate subject name in oneline form on a terminal supporting UTF8:

```
openssl x509 -in cert.pem -noout -subject -nameopt oneline,-esc_msb
```

Display the certificate MD5 fingerprint:

```
openssl x509 -in cert.pem -noout -fingerprint
```

Display the certificate SHA1 fingerprint:

```
openssl x509 -sha1 -in cert.pem -noout -fingerprint
```

Convert a certificate from PEM to DER format:

```
openssl x509 -in cert.pem -inform PEM -out cert.der -outform DER
```

Convert a certificate to a certificate request:

```
openssl x509 -x509toreq -in cert.pem -out req.pem -signkey key.pem
```

Convert a certificate request into a self signed certificate using extensions for a CA:

```
openssl x509 -req -in careq.pem -extfile openssl.cnf -extensions v3_ca \
       -signkey key.pem -out cacert.pem
```

Sign a certificate request using the CA certificate above and add user certificate extensions:

```
openssl x509 -req -in req.pem -extfile openssl.cnf -extensions v3_usr \
       -CA cacert.pem -CAkey key.pem -CAcreateserial
```

Set a certificate to be trusted for SSL client use and change set its alias to "Steve's Class 1 CA"

```
openssl x509 -in cert.pem -addtrust clientAuth \
       -setalias "Steve's Class 1 CA" -out trust.pem
```

NOTES

The PEM format uses the header and footer lines:

```
-----BEGIN CERTIFICATE-----
-----END CERTIFICATE-----
```

it will also handle files containing:

```
-----BEGIN X509 CERTIFICATE-----
-----END X509 CERTIFICATE-----
```

Trusted certificates have the lines

```
-----BEGIN TRUSTED CERTIFICATE-----
-----END TRUSTED CERTIFICATE-----
```

The conversion to UTF8 format used with the name options assumes that T61Strings use the ISO8859-1 character set. This is wrong but Netscape and MSIE do this as do many certificates. So although this is incorrect it is more likely to display the majority of certificates correctly.

The **-fingerprint** option takes the digest of the DER encoded certificate. This is commonly called a "fingerprint". Because of the nature of message digests the fingerprint of a certificate is unique to that certificate and two certificates with the same fingerprint can be considered to be the same.

The Netscape fingerprint uses MD5 whereas MSIE uses SHA1.

The **-email** option searches the subject name and the subject alternative name extension. Only unique email addresses will be printed out: it will not print the same address more than once.

CERTIFICATE EXTENSIONS

The **-purpose** option checks the certificate extensions and determines what the certificate can be used for. The actual checks done are rather complex and include various hacks and workarounds to handle broken certificates and software.

The same code is used when verifying untrusted certificates in chains so this section is useful if a chain is rejected by the verify code.

The basicConstraints extension CA flag is used to determine whether the certificate can be used as a CA. If the CA flag is true then it is a CA, if the CA flag is false then it is not a CA. **All** CAs should have the CA flag set to true.

If the basicConstraints extension is absent then the certificate is considered to be a "possible CA" other extensions are checked according to the intended use of the certificate. A warning is given in this case because the certificate should really not be regarded as a CA: however it is allowed to be a CA to work around some broken software.

If the certificate is a V1 certificate (and thus has no extensions) and it is self signed it is also assumed to be a CA but a warning is again given: this is to work around the problem of Verisign roots which are V1 self signed certificates.

If the keyUsage extension is present then additional restraints are made on the uses of the certificate. A CA certificate **must** have the keyCertSign bit set if the keyUsage extension is present.

The extended key usage extension places additional restrictions on the certificate uses. If this extension is present (whether critical or not) the key can only be used for the purposes specified.

A complete description of each test is given below. The comments about basicConstraints and keyUsage and V1 certificates above apply to **all** CA certificates.

SSL Client

> The extended key usage extension must be absent or include the "web client authentication" OID. keyUsage must be absent or it must have the digitalSignature bit set. Netscape certificate type must be absent or it must have the SSL client bit set.

SSL Client CA

> The extended key usage extension must be absent or include the "web client authentication" OID. Netscape certificate type must be absent or it must have the SSL CA bit set: this is used as a work around if the basicConstraints extension is absent.

SSL Server

> The extended key usage extension must be absent or include the "web server authentication" and/or one of the SGC OIDs. keyUsage must be absent or it must have the digitalSignature, the keyEncipherment set or both bits set. Netscape certificate type must be absent or have the SSL server bit set.

SSL Server CA

> The extended key usage extension must be absent or include the "web server authentication" and/or one of the SGC OIDs. Netscape certificate type must be absent or the SSL CA bit must be set: this is used as a work around if the basicConstraints extension is absent.

Netscape SSL Server

> For Netscape SSL clients to connect to an SSL server it must have the keyEncipherment bit set if the keyUsage extension is present. This isn't always valid because some cipher suites use the key for digital signing. Otherwise it is the same as a normal SSL server.

Common S/MIME Client Tests

The extended key usage extension must be absent or include the "email protection" OID. Netscape certificate type must be absent or should have the S/MIME bit set. If the S/MIME bit is not set in netscape certificate type then the SSL client bit is tolerated as an alternative but a warning is shown: this is because some Verisign certificates don't set the S/MIME bit.

S/MIME Signing

In addition to the common S/MIME client tests the digitalSignature bit must be set if the keyUsage extension is present.

S/MIME Encryption

In addition to the common S/MIME tests the keyEncipherment bit must be set if the keyUsage extension is present.

S/MIME CA

The extended key usage extension must be absent or include the "email protection" OID. Netscape certificate type must be absent or must have the S/MIME CA bit set: this is used as a work around if the basicConstraints extension is absent.

CRL Signing

The keyUsage extension must be absent or it must have the CRL signing bit set.

CRL Signing CA

The normal CA tests apply. Except in this case the basicConstraints extension must be present.

BUGS

Extensions in certificates are not transferred to certificate requests and vice versa.

It is possible to produce invalid certificates or requests by specifying the wrong private key or using inconsistent options in some cases: these should be checked.

There should be options to explicitly set such things as start and end dates rather than an offset from the current time.

The code to implement the verify behaviour described in the **TRUST SETTINGS** is currently being developed. It thus describes the intended behaviour rather than the current behaviour. It is hoped that it will represent reality in OpenSSL 0.9.5 and later.

SEE ALSO

req(1), ca(1), genrsa(1), gendsa(1), verify(1), x509v3_config(5)

HISTORY

Before OpenSSL 0.9.8, the default digest for RSA keys was MD5.

The hash algorithm used in the **-subject_hash** and **-issuer_hash** options before OpenSSL 1.0.0 was based on the deprecated MD5 algorithm and the encoding of the distinguished name. In OpenSSL 1.0.0 and later it is based on a canonical version of the DN using SHA1. This means that any directories using the old form must have their links rebuilt using **c_rehash** or similar.

Name

config — OpenSSL CONF library configuration files

DESCRIPTION

The OpenSSL CONF library can be used to read configuration files. It is used for the OpenSSL master configuration file **openssl.cnf** and in a few other places like **SPKAC** files and certificate extension files for the **x509** utility. OpenSSL applications can also use the CONF library for their own purposes.

A configuration file is divided into a number of sections. Each section starts with a line **[section_name]** and ends when a new section is started or end of file is reached. A section name can consist of alphanumeric characters and underscores.

The first section of a configuration file is special and is referred to as the **default** section this is usually unnamed and is from the start of file until the first named section. When a name is being looked up it is first looked up in a named section (if any) and then the default section.

The environment is mapped onto a section called **ENV**.

Comments can be included by preceding them with the **#** character

Each section in a configuration file consists of a number of name and value pairs of the form **name=value**

The **name** string can contain any alphanumeric characters as well as a few punctuation symbols such as **. , ;** and **_**.

The **value** string consists of the string following the **=** character until end of line with any leading and trailing white space removed.

The value string undergoes variable expansion. This can be done by including the form **$var** or **${var}**: this will substitute the value of the named variable in the current section. It is also possible to substitute a value from another section using the syntax **$section::name** or **${section::name}**. By using the form **$ENV::name** environment variables can be substituted. It is also possible to assign values to environment variables by using the name **ENV::name**, this will work if the program looks up environment variables using the **CONF** library instead of calling **getenv()** directly.

It is possible to escape certain characters by using any kind of quote or the \ character. By making the last character of a line a \ a **value** string can be spread across multiple lines. In addition the sequences **\n**, **\r**, **\b** and **\t** are recognized.

OPENSSL LIBRARY CONFIGURATION

In OpenSSL 0.9.7 and later applications can automatically configure certain aspects of OpenSSL using the master OpenSSL configuration file, or optionally an alternative configuration file. The **openssl** utility includes this functionality: any sub command uses the master OpenSSL configuration file unless an option is used in the sub command to use an alternative configuration file.

To enable library configuration the default section needs to contain an appropriate line which points to the main configuration section. The default name is **openssl_conf** which is used by the **openssl** utility. Other applications may use an alternative name such as **myapplicaton_conf**.

The configuration section should consist of a set of name value pairs which contain specific module configuration information. The **name** represents the name of the *configuration module* the meaning of the **value** is module specific: it may, for example, represent a further configuration section containing configuration module specific information. E.g.

```
openssl_conf = openssl_init

[openssl_init]

oid_section = new_oids
engines = engine_section
```

```
[new_oids]

… new oids here …

[engine_section]

… engine stuff here …
```

The features of each configuration module are described below.

ASN1 OBJECT CONFIGURATION MODULE

This module has the name **oid_section**. The value of this variable points to a section containing name value pairs of OIDs: the name is the OID short and long name, the value is the numerical form of the OID. Although some of the **openssl** utility sub commands already have their own ASN1 OBJECT section functionality not all do. By using the ASN1 OBJECT configuration module **all** the **openssl** utility sub commands can see the new objects as well as any compliant applications. For example:

```
[new_oids]

some_new_oid = 1.2.3.4
some_other_oid = 1.2.3.5
```

In OpenSSL 0.9.8 it is also possible to set the value to the long name followed by a comma and the numerical OID form. For example:

```
shortName = some object long name, 1.2.3.4
```

ENGINE CONFIGURATION MODULE

This ENGINE configuration module has the name **engines**. The value of this variable points to a section containing further ENGINE configuration information.

The section pointed to by **engines** is a table of engine names (though see **engine_id** below) and further sections containing configuration information specific to each ENGINE.

Each ENGINE specific section is used to set default algorithms, load dynamic, perform initialization and send ctrls. The actual operation performed depends on the *command* name which is the name of the name value pair. The currently supported commands are listed below.

For example:

```
[engine_section]

# Configure ENGINE named "foo"
foo = foo_section
# Configure ENGINE named "bar"
bar = bar_section

[foo_section]
… foo ENGINE specific commands …

[bar_section]
… "bar" ENGINE specific commands …
```

The command **engine_id** is used to give the ENGINE name. If used this command must be first. For example:

```
[engine_section]
# This would normally handle an ENGINE named "foo"
foo = foo_section

[foo_section]
# Override default name and use "myfoo" instead.
engine_id = myfoo
```

The command **dynamic_path** loads and adds an ENGINE from the given path. It is equivalent to sending the ctrls **SO_PATH** with the path argument followed by **LIST_ADD** with value 2 and **LOAD** to the dynamic ENGINE. If this is not the required behaviour then alternative ctrls can be sent directly to the dynamic ENGINE using ctrl commands.

The command **init** determines whether to initialize the ENGINE. If the value is **0** the ENGINE will not be initialized, if **1** and attempt it made to initialized the ENGINE immediately. If the **init** command is not present then an attempt will be made to initialize the ENGINE after all commands in its section have been processed.

The command **default_algorithms** sets the default algorithms an ENGINE will supply using the functions **ENGINE_set_default_string()**

If the name matches none of the above command names it is assumed to be a ctrl command which is sent to the ENGINE. The value of the command is the argument to the ctrl command. If the value is the string **EMPTY** then no value is sent to the command.

For example:

```
[engine_section]

# Configure ENGINE named "foo"
foo = foo_section

[foo_section]
# Load engine from DSO
dynamic_path = /some/path/fooengine.so
# A foo specific ctrl.
some_ctrl = some_value
# Another ctrl that doesn't take a value.
other_ctrl = EMPTY
# Supply all default algorithms
default_algorithms = ALL
```

EVP CONFIGURATION MODULE

This modules has the name **alg_section** which points to a section containing algorithm commands.

Currently the only algorithm command supported is **fips_mode** whose value should be a boolean string such as **on** or **off**. If the value is **on** this attempt to enter FIPS mode. If the call fails or the library is not FIPS capable then an error occurs.

For example:

```
alg_section = evp_settings

[evp_settings]

fips_mode = on
```

NOTES

If a configuration file attempts to expand a variable that doesn't exist then an error is flagged and the file will not load. This can happen if an attempt is made to expand an environment variable that doesn't exist. For example in a previous version of OpenSSL the default OpenSSL master configuration file used the value of **HOME** which may not be defined on non Unix systems and would cause an error.

This can be worked around by including a **default** section to provide a default value: then if the environment lookup fails the default value will be used instead. For this to work properly the default value must be defined earlier in the configuration file than the expansion. See the **EXAMPLES** section for an example of how to do this.

If the same variable exists in the same section then all but the last value will be silently ignored. In certain circumstances such as with DNs the same field may occur multiple times. This is usually worked around by ignoring any characters before an initial **.** e.g.

```
1.OU="My first OU"
```

```
2.OU="My Second OU"
```

EXAMPLES

Here is a sample configuration file using some of the features mentioned above.

```
# This is the default section.

HOME=/temp
RANDFILE= ${ENV::HOME}/.rnd
configdir=$ENV::HOME/config

[ section_one ]

# We are now in section one.

# Quotes permit leading and trailing whitespace
any = " any variable name "

other = A string that can \
cover several lines \
by including \\ characters

message = Hello World\n

[ section_two ]

greeting = $section_one::message
```

This next example shows how to expand environment variables safely.

Suppose you want a variable called **tmpfile** to refer to a temporary filename. The directory it is placed in can determined by the the **TEMP** or **TMP** environment variables but they may not be set to any value at all. If you just include the environment variable names and the variable doesn't exist then this will cause an error when an attempt is made to load the configuration file. By making use of the default section both values can be looked up with **TEMP** taking priority and **/tmp** used if neither is defined:

```
TMP=/tmp
# The above value is used if TMP isn't in the environment
TEMP=$ENV::TMP
# The above value is used if TEMP isn't in the environment
tmpfile=${ENV::TEMP}/tmp.filename
```

Simple OpenSSL library configuration example to enter FIPS mode:

```
# Default appname: should match "appname" parameter (if any)
# supplied to CONF_modules_load_file et al.
openssl_conf = openssl_conf_section

[openssl_conf_section]
# Configuration module list
alg_section = evp_sect

[evp_sect]
# Set to "yes" to enter FIPS mode if supported
fips_mode = yes
```

Note: in the above example you will get an error in non FIPS capable versions of OpenSSL.

More complex OpenSSL library configuration. Add OID and don't enter FIPS mode:

```
# Default appname: should match "appname" parameter (if any)
# supplied to CONF_modules_load_file et al.
openssl_conf = openssl_conf_section

[openssl_conf_section]
```

```
# Configuration module list
alg_section = evp_sect
oid_section = new_oids

[evp_sect]
# This will have no effect as FIPS mode is off by default.
# Set to "yes" to enter FIPS mode, if supported
fips_mode = no

[new_oids]
# New OID, just short name
newoid1 = 1.2.3.4.1
# New OID shortname and long name
newoid2 = New OID 2 long name, 1.2.3.4.2
```

The above examples can be used with with any application supporting library configuration if "openssl_conf" is modified to match the appropriate "appname".

For example if the second sample file above is saved to "example.cnf" then the command line:

```
OPENSSL_CONF=example.cnf openssl asn1parse -genstr OID:1.2.3.4.1
```

will output:

```
0:d=0  hl=2 l=   4 prim: OBJECT            :newoid1
```

showing that the OID "newoid1" has been added as "1.2.3.4.1".

BUGS

Currently there is no way to include characters using the octal **\nnn** form. Strings are all null terminated so nulls cannot form part of the value.

The escaping isn't quite right: if you want to use sequences like **\n** you can't use any quote escaping on the same line.

Files are loaded in a single pass. This means that an variable expansion will only work if the variables referenced are defined earlier in the file.

SEE ALSO

x509(1), req(1), ca(1)

Name

x509v3_config — X509 V3 certificate extension configuration format

DESCRIPTION

Several of the OpenSSL utilities can add extensions to a certificate or certificate request based on the contents of a configuration file.

Typically the application will contain an option to point to an extension section. Each line of the extension section takes the form:

```
extension_name=[critical,] extension_options
```

If **critical** is present then the extension will be critical.

The format of **extension_options** depends on the value of **extension_name**.

There are four main types of extension: *string* extensions, *multi-valued* extensions, *raw* and *arbitrary* extensions.

String extensions simply have a string which contains either the value itself or how it is obtained.

For example:

```
nsComment="This is a Comment"
```

Multi-valued extensions have a short form and a long form. The short form is a list of names and values:

```
basicConstraints=critical,CA:true,pathlen:1
```

The long form allows the values to be placed in a separate section:

```
basicConstraints=critical,@bs_section

[bs_section]

CA=true
pathlen=1
```

Both forms are equivalent.

The syntax of raw extensions is governed by the extension code: it can for example contain data in multiple sections. The correct syntax to use is defined by the extension code itself: check out the certificate policies extension for an example.

If an extension type is unsupported then the *arbitrary* extension syntax must be used, see the “ARBITRARY EXTENSIONS” section for more details.

STANDARD EXTENSIONS

The following sections describe each supported extension in detail.

Basic Constraints.

This is a multi valued extension which indicates whether a certificate is a CA certificate. The first (mandatory) name is **CA** followed by **TRUE** or **FALSE**. If **CA** is **TRUE** then an optional **pathlen** name followed by an non-negative value can be included.

For example:

```
basicConstraints=CA:TRUE

basicConstraints=CA:FALSE

basicConstraints=critical,CA:TRUE, pathlen:0
```

A CA certificate **must** include the basicConstraints value with the CA field set to TRUE. An end user certificate must either set CA to FALSE or exclude the extension entirely. Some software may require the inclusion of basicConstraints with CA set to FALSE for end entity certificates.

The pathlen parameter indicates the maximum number of CAs that can appear below this one in a chain. So if you have a CA with a pathlen of zero it can only be used to sign end user certificates and not further CAs.

Key Usage.

Key usage is a multi valued extension consisting of a list of names of the permitted key usages.

The supporte names are: digitalSignature, nonRepudiation, keyEncipherment, dataEncipherment, keyAgreement, keyCertSign, cRLSign, encipherOnly and decipherOnly.

Examples:

```
keyUsage=digitalSignature, nonRepudiation

keyUsage=critical, keyCertSign
```

Extended Key Usage.

This extensions consists of a list of usages indicating purposes for which the certificate public key can be used for,

These can either be object short names of the dotted numerical form of OIDs. While any OID can be used only certain values make sense. In particular the following PKIX, NS and MS values are meaningful:

```
Value                  Meaning
-----                  -------
serverAuth             SSL/TLS Web Server Authentication.
clientAuth             SSL/TLS Web Client Authentication.
codeSigning            Code signing.
emailProtection        E-mail Protection (S/MIME).
timeStamping           Trusted Timestamping
msCodeInd              Microsoft Individual Code Signing (authenticode)
msCodeCom              Microsoft Commercial Code Signing (authenticode)
msCTLSign              Microsoft Trust List Signing
msSGC                  Microsoft Server Gated Crypto
msEFS                  Microsoft Encrypted File System
nsSGC                  Netscape Server Gated Crypto
```

Examples:

```
extendedKeyUsage=critical,codeSigning,1.2.3.4
extendedKeyUsage=nsSGC,msSGC
```

Subject Key Identifier.

This is really a string extension and can take two possible values. Either the word **hash** which will automatically follow the guidelines in RFC3280 or a hex string giving the extension value to include. The use of the hex string is strongly discouraged.

Example:

```
subjectKeyIdentifier=hash
```

Authority Key Identifier.

The authority key identifier extension permits two options. keyid and issuer: both can take the optional value "always".

If the keyid option is present an attempt is made to copy the subject key identifier from the parent certificate. If the value "always" is present then an error is returned if the option fails.

The issuer option copies the issuer and serial number from the issuer certificate. This will only be done if the keyid option fails or is not included unless the "always" flag will always include the value.

Example:

```
authorityKeyIdentifier=keyid,issuer
```

Subject Alternative Name.

The subject alternative name extension allows various literal values to be included in the configuration file. These include **email** (an email address) **URI** a uniform resource indicator, **DNS** (a DNS domain name), **RID** (a registered ID: OBJECT IDENTIFIER), **IP** (an IP address), **dirName** (a distinguished name) and otherName.

The email option include a special 'copy' value. This will automatically include and email addresses contained in the certificate subject name in the extension.

The IP address used in the **IP** options can be in either IPv4 or IPv6 format.

The value of **dirName** should point to a section containing the distinguished name to use as a set of name value pairs. Multi values AVAs can be formed by prefacing the name with a **+** character.

otherName can include arbitrary data associated with an OID: the value should be the OID followed by a semicolon and the content in standard ASN1_generate_nconf(3) format.

Examples:

```
subjectAltName=email:copy,email:my@other.address,URI:http://my.url.here/
subjectAltName=IP:192.168.7.1
subjectAltName=IP:13::17
subjectAltName=email:my@other.address,RID:1.2.3.4
subjectAltName=otherName:1.2.3.4;UTF8:some other identifier

subjectAltName=dirName:dir_sect

[dir_sect]
C=UK
O=My Organization
OU=My Unit
CN=My Name
```

Issuer Alternative Name.

The issuer alternative name option supports all the literal options of subject alternative name. It does **not** support the email:copy option because that would not make sense. It does support an additional issuer:copy option that will copy all the subject alternative name values from the issuer certificate (if possible).

Example:

```
issuserAltName = issuer:copy
```

Authority Info Access.

The authority information access extension gives details about how to access certain information relating to the CA. Its syntax is accessOID;location where *location* has the same syntax as subject alternative name (except that email:copy is not supported). accessOID can be any valid OID but only certain values are meaningful, for example OCSP and caIssuers.

Example:

```
authorityInfoAccess = OCSP;URI:http://ocsp.my.host/
authorityInfoAccess = caIssuers;URI:http://my.ca/ca.html
```

CRL distribution points.

This is a multi-valued extension whose options can be either in name:value pair using the same form as subject alternative name or a single value representing a section name containing all the distribution point fields.

For a name:value pair a new DistributionPoint with the fullName field set to the given value both the cRLissuer and reasons fields are omitted in this case.

In the single option case the section indicated contains values for each field. In this section:

If the name is "fullname" the value field should contain the full name of the distribution point in the same format as subject alternative name.

If the name is "relativename" then the value field should contain a section name whose contents represent a DN fragment to be placed in this field.

The name "CRLIssuer" if present should contain a value for this field in subject alternative name format.

If the name is "reasons" the value field should consist of a comma separated field containing the reasons. Valid reasons are: "keyCompromise", "CACompromise", "affiliationChanged", "superseded", "cessationOfOperation", "certificateHold", "privilegeWithdrawn" and "AACompromise".

Simple examples:

```
crlDistributionPoints=URI:http://myhost.com/myca.crl
crlDistributionPoints=URI:http://my.com/my.crl,URI:http://oth.com/my.crl
```

Full distribution point example:

```
crlDistributionPoints=crldp1_section

[crldp1_section]

fullname=URI:http://myhost.com/myca.crl
CRLissuer=dirName:issuer_sect
reasons=keyCompromise, CACompromise

[issuer_sect]
C=UK
O=Organisation
CN=Some Name
```

Issuing Distribution Point

This extension should only appear in CRLs. It is a multi valued extension whose syntax is similar to the "section" pointed to by the CRL distribution points extension with a few differences.

The names "reasons" and "CRLissuer" are not recognized.

The name "onlysomereasons" is accepted which sets this field. The value is in the same format as the CRL distribution point "reasons" field.

The names "onlyuser", "onlyCA", "onlyAA" and "indirectCRL" are also accepted the values should be a boolean value (TRUE or FALSE) to indicate the value of the corresponding field.

Example:

```
issuingDistributionPoint=critical, @idp_section

[idp_section]
```

```
fullname=URI:http://myhost.com/myca.crl
indirectCRL=TRUE
onlysomereasons=keyCompromise, CACompromise

[issuer_sect]
C=UK
O=Organisation
CN=Some Name
```

Certificate Policies.

This is a *raw* extension. All the fields of this extension can be set by using the appropriate syntax.

If you follow the PKIX recommendations and just using one OID then you just include the value of that OID. Multiple OIDs can be set separated by commas, for example:

```
certificatePolicies= 1.2.4.5, 1.1.3.4
```

If you wish to include qualifiers then the policy OID and qualifiers need to be specified in a separate section: this is done by using the @section syntax instead of a literal OID value.

The section referred to must include the policy OID using the name policyIdentifier, cPSuri qualifiers can be included using the syntax:

```
CPS.nnn=value
```

userNotice qualifiers can be set using the syntax:

```
userNotice.nnn=@notice
```

The value of the userNotice qualifier is specified in the relevant section. This section can include explicitText, organization and noticeNumbers options. explicitText and organization are text strings, noticeNumbers is a comma separated list of numbers. The organization and noticeNumbers options (if included) must BOTH be present. If you use the userNotice option with IE5 then you need the 'ia5org' option at the top level to modify the encoding: otherwise it will not be interpreted properly.

Example:

```
certificatePolicies=ia5org,1.2.3.4,1.5.6.7.8,@polsect

[polsect]

policyIdentifier = 1.3.5.8
CPS.1="http://my.host.name/"
CPS.2="http://my.your.name/"
userNotice.1=@notice

[notice]

explicitText="Explicit Text Here"
organization="Organisation Name"
noticeNumbers=1,2,3,4
```

The **ia5org** option changes the type of the *organization* field. In RFC2459 it can only be of type DisplayText. In RFC3280 IA5String is also permissible. Some software (for example some versions of MSIE) may require ia5org.

Policy Constraints

This is a multi-valued extension which consisting of the names **requireExplicitPolicy** or **inhibitPolicyMapping** and a non negative intger value. At least one component must be present.

Example:

```
policyConstraints = requireExplicitPolicy:3
```

Inhibit Any Policy

This is a string extension whose value must be a non negative integer.

Example:

```
inhibitAnyPolicy = 2
```

Name Constraints

The name constraints extension is a multi-valued extension. The name should begin with the word **permitted** or **excluded** followed by a **;**. The rest of the name and the value follows the syntax of subjectAltName except email:copy is not supported and the **IP** form should consist of an IP addresses and subnet mask separated by a **/**.

Examples:

```
nameConstraints=permitted;IP:192.168.0.0/255.255.0.0

nameConstraints=permitted;email:.somedomain.com

nameConstraints=excluded;email:.com
```

OCSP No Check

The OCSP No Check extension is a string extension but its value is ignored.

Example:

```
noCheck = ignored
```

DEPRECATED EXTENSIONS

The following extensions are non standard, Netscape specific and largely obsolete. Their use in new applications is discouraged.

Netscape String extensions.

Netscape Comment (**nsComment**) is a string extension containing a comment which will be displayed when the certificate is viewed in some browsers.

Example:

```
nsComment = "Some Random Comment"
```

Other supported extensions in this category are: **nsBaseUrl**, **nsRevocationUrl**, **nsCaRevocationUrl**, **nsRenewalUrl**, **nsCaPolicyUrl** and **nsSslServerName**.

Netscape Certificate Type

This is a multi-valued extensions which consists of a list of flags to be included. It was used to indicate the purposes for which a certificate could be used. The basicConstraints, keyUsage and extended key usage extensions are now used instead.

Acceptable values for nsCertType are: **client**, **server**, **email**, **objsign**, **reserved**, **sslCA**, **emailCA**, **objCA**.

ARBITRARY EXTENSIONS

If an extension is not supported by the OpenSSL code then it must be encoded using the arbitrary extension format. It is also possible to use the arbitrary format for supported extensions. Extreme care should be taken to ensure that the data is formatted correctly for the given extension type.

There are two ways to encode arbitrary extensions.

The first way is to use the word ASN1 followed by the extension content using the same syntax as ASN1_generate_nconf(3). For example:

```
1.2.3.4=critical,ASN1:UTF8String:Some random data

1.2.3.4=ASN1:SEQUENCE:seq_sect

[seq_sect]

field1 = UTF8:field1
field2 = UTF8:field2
```

It is also possible to use the word DER to include the raw encoded data in any extension.

```
1.2.3.4=critical,DER:01:02:03:04
1.2.3.4=DER:01020304
```

The value following DER is a hex dump of the DER encoding of the extension Any extension can be placed in this form to override the default behaviour. For example:

```
basicConstraints=critical,DER:00:01:02:03
```

WARNING

There is no guarantee that a specific implementation will process a given extension. It may therefore be sometimes possible to use certificates for purposes prohibited by their extensions because a specific application does not recognize or honour the values of the relevant extensions.

The DER and ASN1 options should be used with caution. It is possible to create totally invalid extensions if they are not used carefully.

NOTES

If an extension is multi-value and a field value must contain a comma the long form must be used otherwise the comma would be misinterpreted as a field separator. For example:

```
subjectAltName=URI:ldap://somehost.com/CN=foo,OU=bar
```

will produce an error but the equivalent form:

```
subjectAltName=@subject_alt_section

[subject_alt_section]
subjectAltName=URI:ldap://somehost.com/CN=foo,OU=bar
```

is valid.

Due to the behaviour of the OpenSSL **conf** library the same field name can only occur once in a section. This means that:

```
subjectAltName=@alt_section

[alt_section]

email=steve@here
email=steve@there
```

will only recognize the last value. This can be worked around by using the form:

```
[alt_section]

```

```
email.1=steve@here
email.2=steve@there
```

HISTORY

The X509v3 extension code was first added to OpenSSL 0.9.2.

Policy mappings, inhibit any policy and name constraints support was added in OpenSSL 0.9.8

The **directoryName** and **otherName** option as well as the **ASN1** option for arbitrary extensions was added in OpenSSL 0.9.8

SEE ALSO

req(1), ca(1), x509(1), ASN1_generate_nconf(3)

Cryptographic functions

Name

crypto — OpenSSL cryptographic library

Synopsis

DESCRIPTION

The OpenSSL **crypto** library implements a wide range of cryptographic algorithms used in various Internet standards. The services provided by this library are used by the OpenSSL implementations of SSL, TLS and S/MIME, and they have also been used to implement SSH, OpenPGP, and other cryptographic standards.

OVERVIEW

libcrypto consists of a number of sub-libraries that implement the individual algorithms.

The functionality includes symmetric encryption, public key cryptography and key agreement, certificate handling, cryptographic hash functions and a cryptographic pseudo-random number generator.

SYMMETRIC CIPHERS

blowfish(3), cast(3), des(3), idea(3), rc2(3), rc4(3), rc5(3)

PUBLIC KEY CRYPTOGRAPHY AND KEY AGREEMENT

dsa(3), dh(3), rsa(3)

CERTIFICATES

x509(3), x509v3(3)

AUTHENTICATION CODES
HASH FUNCTIONS

hmac(3), md2(3), md4(3), md5(3), mdc2(3), ripemd(3), sha(3)

AUXILIARY FUNCTIONS

err(3), threads(3), rand(3), OPENSSL_VERSION_NUMBER(3)

INPUT/OUTPUT
DATA ENCODING

asn1(3), bio(3), evp(3), pem(3), pkcs7(3), pkcs12(3)

INTERNAL FUNCTIONS

bn(3), buffer(3), lhash(3), objects(3), stack(3), txt_db(3)

NOTES

Some of the newer functions follow a naming convention using the numbers **0** and **1**. For example the functions:

```
int X509_CRL_add0_revoked(X509_CRL *crl, X509_REVOKED *rev);
int X509_add1_trust_object(X509 *x, ASN1_OBJECT *obj);
```

The **0** version uses the supplied structure pointer directly in the parent and it will be freed up when the parent is freed. In the above example **crl** would be freed but **rev** would not.

The **1** function uses a copy of the supplied structure pointer (or in some cases increases its link count) in the parent and so both (**x** and **obj** above) should be freed up.

SEE ALSO

openssl(1), ssl(3)

Name

ASN1_generate_nconf and ASN1_generate_v3 — ASN1 generation functions

Synopsis

```
#include  <openssl/asn1.h>
```

```
ASN1_TYPE *ASN1_generate_nconf(char *str, CONF *nconf);
ASN1_TYPE *ASN1_generate_v3(char *str, X509V3_CTX *cnf);
```

DESCRIPTION

These functions generate the ASN1 encoding of a string in an **ASN1_TYPE** structure.

str contains the string to encode **nconf** or **cnf** contains the optional configuration information where additional strings will be read from. **nconf** will typically come from a config file wherease **cnf** is obtained from an **X509V3_CTX** structure which will typically be used by X509 v3 certificate extension functions. **cnf** or **nconf** can be set to **NULL** if no additional configuration will be used.

GENERATION STRING FORMAT

The actual data encoded is determined by the string **str** and the configuration information. The general format of the string is:

[modifier,]type[:value]

That is zero or more comma separated modifiers followed by a type followed by an optional colon and a value. The formats of **type**, **value** and **modifier** are explained below.

SUPPORTED TYPES

The supported types are listed below. Unless otherwise specified only the **ASCII** format is permissible.

BOOLEAN
BOOL

This encodes a boolean type. The **value** string is mandatory and should be **TRUE** or **FALSE**. Additionally **TRUE**, **true**, **Y**, **y**, **YES**, **yes**, **FALSE**, **false**, **N**, **n**, **NO** and **no** are acceptable.

NULL

Encode the **NULL** type, the **value** string must not be present.

INTEGER
INT

Encodes an ASN1 **INTEGER** type. The **value** string represents the value of the integer, it can be prefaced by a minus sign and is normally interpreted as a decimal value unless the prefix **0x** is included.

ENUMERATED
ENUM

Encodes the ASN1 **ENUMERATED** type, it is otherwise identical to **INTEGER**.

OBJECT
OID

Encodes an ASN1 **OBJECT IDENTIFIER**, the **value** string can be a short name, a long name or numerical format.

UTCTIME
UTC

Encodes an ASN1 **UTCTime** structure, the value should be in the format **YYMMDDHHMMSSZ**.

GENERALIZEDTIME
GENTIME

Encodes an ASN1 **GeneralizedTime** structure, the value should be in the format **YYYYMMDDHHMMSSZ**.

OCTETSTRING
OCT

Encodes an ASN1 **OCTET STRING**. **value** represents the contents of this structure, the format strings **ASCII** and **HEX** can be used to specify the format of **value**.

BITSTRING
BITSTR

Encodes an ASN1 **BIT STRING**. **value** represents the contents of this structure, the format strings **ASCII**, **HEX** and **BITLIST** can be used to specify the format of **value**.

If the format is anything other than **BITLIST** the number of unused bits is set to zero.

UNIVERSALSTRING, **UNIV**, **IA5**, **IA5STRING**, **UTF8**, **UTF8String**, **BMP**, **BMPSTRING**, **VISIBLESTRING**, **VISIBLE**, **PRINTABLESTRING**, **PRINTABLE**, **T61**, **T61STRING**, **TELETEXSTRING**, **GeneralString**, **NUMERICSTRING**, **NUMERIC**

These encode the corresponding string types. **value** represents the contents of this structure. The format can be **ASCII** or **UTF8**.

SEQUENCE
SEQ
SET

Formats the result as an ASN1 **SEQUENCE** or **SET** type. **value** should be a section name which will contain the contents. The field names in the section are ignored and the values are in the generated string format. If **value** is absent then an empty SEQUENCE will be encoded.

MODIFIERS

Modifiers affect the following structure, they can be used to add EXPLICIT or IMPLICIT tagging, add wrappers or to change the string format of the final type and value. The supported formats are documented below.

EXPLICIT
EXP

Add an explicit tag to the following structure. This string should be followed by a colon and the tag value to use as a decimal value.

By following the number with **U**, **A**, **P** or **C** UNIVERSAL, APPLICATION, PRIVATE or CONTEXT SPECIFIC tagging can be used, the default is CONTEXT SPECIFIC.

IMPLICIT
IMP

This is the same as **EXPLICIT** except IMPLICIT tagging is used instead.

OCTWRAP
SEQWRAP
SETWRAP
BITWRAP

The following structure is surrounded by an OCTET STRING, a SEQUENCE, a SET or a BIT STRING respectively. For a BIT STRING the number of unused bits is set to zero.

FORMAT

This specifies the format of the ultimate value. It should be followed by a colon and one of the strings **ASCII**, **UTF8**, **HEX** or **BITLIST**.

If no format specifier is included then **ASCII** is used. If **UTF8** is specified then the value string must be a valid **UTF8** string. For **HEX** the output must be a set of hex digits. **BITLIST** (which is only valid for a BIT STRING) is a comma separated list of the indices of the set bits, all other bits are zero.

EXAMPLES

A simple IA5String:

```
IA5STRING:Hello World
```

An IA5String explicitly tagged:

```
EXPLICIT:0,IA5STRING:Hello World
```

An IA5String explicitly tagged using APPLICATION tagging:

```
EXPLICIT:0A,IA5STRING:Hello World
```

A BITSTRING with bits 1 and 5 set and all others zero:

```
FORMAT:BITLIST,BITSTRING:1,5
```

A more complex example using a config file to produce a SEQUENCE consiting of a BOOL an OID and a UTF8String:

```
asn1 = SEQUENCE:seq_section

[seq_section]

field1 = BOOLEAN:TRUE
field2 = OID:commonName
field3 = UTF8:Third field
```

This example produces an RSAPrivateKey structure, this is the key contained in the file client.pem in all OpenSSL distributions (note: the field names such as 'coeff' are ignored and are present just for clarity):

```
asn1=SEQUENCE:private_key
[private_key]
version=INTEGER:0

n=INTEGER:0xBB6FE79432CC6EA2D8F970675A5A87BFBE1AFF0BE63E879F2AFFB93644\
D4D2C6D000430DEC66ABF47829E74B8C5108623A1C0EE8BE217B3AD8D36D5EB4FCA1D9

e=INTEGER:0x010001

d=INTEGER:0x6F05EAD2F27FFAEC84BEC360C4B928FD5F3A9865D0FCAAD291E2A52F4A\
F810DC6373278C006A0ABBA27DC8C63BF97F7E666E27C5284D7D3B1FFFE16B7A87B51D

p=INTEGER:0xF3929B9435608F8A22C208D86795271D54EBDFB09DDEF539AB083DA912\
D4BD57
```

```
q=INTEGER:0xC50016F89DFF2561347ED1186A46E150E28BF2D0F539A1594BBD7FE467\
46EC4F

exp1=INTEGER:0x9E7D4326C924AFC1DEA40B45650134966D6F9DFA3A7F9D698CD4ABEA\
9C0A39B9

exp2=INTEGER:0xBA84003BB95355AFB7C50DF140C60513D0BA51D637272E355E397779\
E7B2458F

coeff=INTEGER:0x30B9E4F2AFA5AC679F920FC83F1F2DF1BAF1779CF989447FABC2F5\
628657053A
```

This example is the corresponding public key in a SubjectPublicKeyInfo structure:

```
# Start with a SEQUENCE
asn1=SEQUENCE:pubkeyinfo

# pubkeyinfo contains an algorithm identifier and the public key wrapped
# in a BIT STRING
[pubkeyinfo]
algorithm=SEQUENCE:rsa_alg
pubkey=BITWRAP,SEQUENCE:rsapubkey

# algorithm ID for RSA is just an OID and a NULL
[rsa_alg]
algorithm=OID:rsaEncryption
parameter=NULL

# Actual public key: modulus and exponent
[rsapubkey]
n=INTEGER:0xBB6FE79432CC6EA2D8F970675A5A87BFBE1AFF0BE63E879F2AFFB93644\
D4D2C6D000430DEC66ABF47829E74B8C5108623A1C0EE8BE217B3AD8D36D5EB4FCA1D9

e=INTEGER:0x010001
```

RETURN VALUES

ASN1_generate_nconf() and ASN1_generate_v3() return the encoded data as an **ASN1_TYPE** structure or **NULL** if an error occurred.

The error codes that can be obtained by ERR_get_error(3).

SEE ALSO

ERR_get_error(3)

HISTORY

ASN1_generate_nconf() and ASN1_generate_v3() were added to OpenSSL 0.9.8

Name

ASN1_OBJECT_new and ASN1_OBJECT_free — object allocation functions

Synopsis

```
#include  <openssl/asn1.h>
```

```
ASN1_OBJECT *ASN1_OBJECT_new(void);
void ASN1_OBJECT_free(ASN1_OBJECT *a);
```

DESCRIPTION

The ASN1_OBJECT allocation routines, allocate and free an ASN1_OBJECT structure, which represents an ASN1 OBJECT IDENTIFIER.

ASN1_OBJECT_new() allocates and initializes a ASN1_OBJECT structure.

ASN1_OBJECT_free() frees up the **ASN1_OBJECT** structure **a**.

NOTES

Although ASN1_OBJECT_new() allocates a new ASN1_OBJECT structure it is almost never used in applications. The ASN1 object utility functions such as OBJ_nid2obj() are used instead.

RETURN VALUES

If the allocation fails, ASN1_OBJECT_new() returns **NULL** and sets an error code that can be obtained by ERR_get_error(3). Otherwise it returns a pointer to the newly allocated structure.

ASN1_OBJECT_free() returns no value.

SEE ALSO

ERR_get_error(3), d2i_ASN1_OBJECT(3)

HISTORY

ASN1_OBJECT_new() and ASN1_OBJECT_free() are available in all versions of SSLeay and OpenSSL.

Name

ASN1_STRING_dup, ASN1_STRING_cmp, ASN1_STRING_set, ASN1_STRING_length, ASN1_STRING_length_set, ASN1_STRING_type and ASN1_STRING_data — ASN1_STRING utility functions

Synopsis

```
#include  <openssl/asn1.h>

int ASN1_STRING_length(ASN1_STRING *x);
unsigned char * ASN1_STRING_data(ASN1_STRING *x);

ASN1_STRING * ASN1_STRING_dup(ASN1_STRING *a);

int ASN1_STRING_cmp(ASN1_STRING *a, ASN1_STRING *b);

int ASN1_STRING_set(ASN1_STRING *str, const void *data, int len);

int ASN1_STRING_type(ASN1_STRING *x);

int ASN1_STRING_to_UTF8(unsigned char **out, ASN1_STRING *in);
```

DESCRIPTION

These functions allow an **ASN1_STRING** structure to be manipulated.

ASN1_STRING_length() returns the length of the content of **x**.

ASN1_STRING_data() returns an internal pointer to the data of **x**. Since this is an internal pointer it should **not** be freed or modified in any way.

ASN1_STRING_dup() returns a copy of the structure **a**.

ASN1_STRING_cmp() compares **a** and **b** returning 0 if the two are identical. The string types and content are compared.

ASN1_STRING_set() sets the data of string **str** to the buffer **data** or length **len**. The supplied data is copied. If **len** is -1 then the length is determined by strlen(data).

ASN1_STRING_type() returns the type of **x**, using standard constants such as **V_ASN1_OCTET_STRING**.

ASN1_STRING_to_UTF8() converts the string **in** to UTF8 format, the converted data is allocated in a buffer in ***out**. The length of **out** is returned or a negative error code. The buffer ***out** should be free using OPENSSL_free().

NOTES

Almost all ASN1 types in OpenSSL are represented as an **ASN1_STRING** structure. Other types such as **ASN1_OCTET_STRING** are simply typedefed to **ASN1_STRING** and the functions call the **ASN1_STRING** equivalents. **ASN1_STRING** is also used for some **CHOICE** types which consist entirely of primitive string types such as **DirectoryString** and **Time**.

These functions should **not** be used to examine or modify **ASN1_INTEGER** or **ASN1_ENUMERATED** types: the relevant **INTEGER** or **ENUMERATED** utility functions should be used instead.

In general it cannot be assumed that the data returned by ASN1_STRING_data() is null terminated or does not contain embedded nulls. The actual format of the data will depend on the actual string type itself: for example for and IA5String the data will be ASCII, for a BMPString two bytes per character in big endian format, UTF8String will be in UTF8 format.

Similar care should be take to ensure the data is in the correct format when calling ASN1_STRING_set().

RETURN VALUES

SEE ALSO

ERR_get_error(3)

HISTORY

Name

ASN1_STRING_new, ASN1_STRING_type_new and ASN1_STRING_free — ASN1_STRING allocation functions

Synopsis

```
#include  <openssl/asn1.h>
```

```
ASN1_STRING * ASN1_STRING_new(void);
ASN1_STRING * ASN1_STRING_type_new(int type);
void ASN1_STRING_free(ASN1_STRING *a);
```

DESCRIPTION

ASN1_STRING_new() returns an allocated **ASN1_STRING** structure. Its type is undefined.

ASN1_STRING_type_new() returns an allocated **ASN1_STRING** structure of type **type**.

ASN1_STRING_free() frees up **a**.

NOTES

Other string types call the **ASN1_STRING** functions. For example ASN1_OCTET_STRING_new() calls ASN1_STRING_type(V_ASN1_OCTET_STRING).

RETURN VALUES

ASN1_STRING_new() and ASN1_STRING_type_new() return a valid ASN1_STRING structure or **NULL** if an error occurred.

ASN1_STRING_free() does not return a value.

SEE ALSO

ERR_get_error(3)

HISTORY

TBA

Name

ASN1_STRING_print_ex and ASN1_STRING_print_ex_fp — ASN1_STRING output routines.

Synopsis

```
#include  <openssl/asn1.h>
```

```
int ASN1_STRING_print_ex(BIO *out, ASN1_STRING *str, unsigned long flags);
int ASN1_STRING_print_ex_fp(FILE *fp, ASN1_STRING *str, unsigned long flags);
int ASN1_STRING_print(BIO *out, ASN1_STRING *str);
```

DESCRIPTION

These functions output an **ASN1_STRING** structure. **ASN1_STRING** is used to represent all the ASN1 string types.

ASN1_STRING_print_ex() outputs **str** to **out**, the format is determined by the options **flags**. ASN1_STRING_print_ex_fp() is identical except it outputs to **fp** instead.

ASN1_STRING_print() prints **str** to **out** but using a different format to ASN1_STRING_print_ex(). It replaces unprintable characters (other than CR, LF) with '.'.

NOTES

ASN1_STRING_print() is a legacy function which should be avoided in new applications.

Although there are a large number of options frequently **ASN1_STRFLGS_RFC2253** is suitable, or on UTF8 terminals **ASN1_STRFLGS_RFC2253 & ~ASN1_STRFLGS_ESC_MSB**.

The complete set of supported options for **flags** is listed below.

Various characters can be escaped. If **ASN1_STRFLGS_ESC_2253** is set the characters determined by RFC2253 are escaped. If **ASN1_STRFLGS_ESC_CTRL** is set control characters are escaped. If **ASN1_STRFLGS_ESC_MSB** is set characters with the MSB set are escaped: this option should **not** be used if the terminal correctly interprets UTF8 sequences.

Escaping takes several forms.

If the character being escaped is a 16 bit character then the form "\UXXXX" is used using exactly four characters for the hex representation. If it is 32 bits then "\WXXXXXXXX" is used using eight characters of its hex representation. These forms will only be used if UTF8 conversion is not set (see below).

Printable characters are normally escaped using the backslash '\' character. If **ASN1_STRFLGS_ESC_QUOTE** is set then the whole string is instead surrounded by double quote characters: this is arguably more readable than the backslash notation. Other characters use the "\XX" using exactly two characters of the hex representation.

If **ASN1_STRFLGS_UTF8_CONVERT** is set then characters are converted to UTF8 format first. If the terminal supports the display of UTF8 sequences then this option will correctly display multi byte characters.

If **ASN1_STRFLGS_IGNORE_TYPE** is set then the string type is not interpreted at all: everything is assumed to be one byte per character. This is primarily for debugging purposes and can result in confusing output in multi character strings.

If **ASN1_STRFLGS_SHOW_TYPE** is set then the string type itself is printed out before its value (for example "BMPSTRING"), this actually uses ASN1_tag2str().

The content of a string instead of being interpreted can be "dumped": this just outputs the value of the string using the form #XXXX using hex format for each octet.

If **ASN1_STRFLGS_DUMP_ALL** is set then any type is dumped.

Normally non character string types (such as OCTET STRING) are assumed to be one byte per character, if **ASN1_STRFLGS_DUMP_UNKNOWN** is set then they will be dumped instead.

When a type is dumped normally just the content octets are printed, if **ASN1_STRFLGS_DUMP_DER** is set then the complete encoding is dumped instead (including tag and length octets).

ASN1_STRFLGS_RFC2253 includes all the flags required by RFC2253. It is equivalent to: ASN1_STRFLGS_ESC_2253 | ASN1_STRFLGS_ESC_CTRL | ASN1_STRFLGS_ESC_MSB | ASN1_STRFLGS_UTF8_CONVERT | ASN1_STRFLGS_DUMP_UNKNOWN ASN1_STRFLGS_DUMP_DER

SEE ALSO

X509_NAME_print_ex(3), ASN1_tag2str(3)

HISTORY

TBA

Name

bio — I/O abstraction

Synopsis

```
#include <openssl/bio.h>
```

```
TBA
```

DESCRIPTION

A BIO is an I/O abstraction, it hides many of the underlying I/O details from an application. If an application uses a BIO for its I/O it can transparently handle SSL connections, unencrypted network connections and file I/O.

There are two type of BIO, a source/sink BIO and a filter BIO.

As its name implies a source/sink BIO is a source and/or sink of data, examples include a socket BIO and a file BIO.

A filter BIO takes data from one BIO and passes it through to another, or the application. The data may be left unmodified (for example a message digest BIO) or translated (for example an encryption BIO). The effect of a filter BIO may change according to the I/O operation it is performing: for example an encryption BIO will encrypt data if it is being written to and decrypt data if it is being read from.

BIOs can be joined together to form a chain (a single BIO is a chain with one component). A chain normally consist of one source/sink BIO and one or more filter BIOs. Data read from or written to the first BIO then traverses the chain to the end (normally a source/sink BIO).

SEE ALSO

BIO_ctrl(3), BIO_f_base64(3), BIO_f_buffer(3), BIO_f_cipher(3), BIO_f_md(3), BIO_f_null(3), BIO_f_ssl(3), BIO_find_type(3), BIO_new(3), BIO_new_bio_pair(3), BIO_push(3), BIO_read(3), BIO_s_accept(3), BIO_s_bio(3), BIO_s_connect(3), BIO_s_fd(3), BIO_s_file(3), BIO_s_mem(3), BIO_s_null(3), BIO_s_socket(3), BIO_set_callback(3), BIO_should_retry(3)

Name

BIO_ctrl, BIO_callback_ctrl, BIO_ptr_ctrl, BIO_int_ctrl, BIO_reset, BIO_seek, BIO_tell, BIO_flush, BIO_eof, BIO_set_close, BIO_get_close, BIO_pending, BIO_wpending, BIO_ctrl_pending, BIO_ctrl_wpending, BIO_get_info_callback and BIO_set_info_callback — BIO control operations

Synopsis

```
#include  <openssl/bio.h>
```

```
long BIO_ctrl(BIO *bp,int cmd,long larg,void *parg);
long BIO_callback_ctrl(BIO *b,int cmd,void (*fp)(struct bio_st *,int,const char *,int,long,long));
char * BIO_ptr_ctrl(BIO *bp,int cmd,long larg);
long BIO_int_ctrl(BIO *bp,int cmd,long larg,int iarg);
```

```
int BIO_reset(BIO *b);
int BIO_seek(BIO *b, int ofs);
int BIO_tell(BIO *b);
int BIO_flush(BIO *b);
int BIO_eof(BIO *b);
int BIO_set_close(BIO *b,long flag);
int BIO_get_close(BIO *b);
int BIO_pending(BIO *b);
int BIO_wpending(BIO *b);
size_t BIO_ctrl_pending(BIO *b);
size_t BIO_ctrl_wpending(BIO *b);
```

```
int BIO_get_info_callback(BIO *b,bio_info_cb **cbp);
int BIO_set_info_callback(BIO *b,bio_info_cb *cb);
```

```
typedef void bio_info_cb(BIO *b, int oper, const char *ptr, int arg1, long arg2, long arg3);
```

DESCRIPTION

BIO_ctrl(), BIO_callback_ctrl(), BIO_ptr_ctrl() and BIO_int_ctrl() are BIO "control" operations taking arguments of various types. These functions are not normally called directly, various macros are used instead. The standard macros are described below, macros specific to a particular type of BIO are described in the specific BIOs manual page as well as any special features of the standard calls.

BIO_reset() typically resets a BIO to some initial state, in the case of file related BIOs for example it rewinds the file pointer to the start of the file.

BIO_seek() resets a file related BIO's (that is file descriptor and FILE BIOs) file position pointer to **ofs** bytes from start of file.

BIO_tell() returns the current file position of a file related BIO.

BIO_flush() normally writes out any internally buffered data, in some cases it is used to signal EOF and that no more data will be written.

BIO_eof() returns 1 if the BIO has read EOF, the precise meaning of "EOF" varies according to the BIO type.

BIO_set_close() sets the BIO **b** close flag to **flag**. **flag** can take the value BIO_CLOSE or BIO_NOCLOSE. Typically BIO_CLOSE is used in a source/sink BIO to indicate that the underlying I/O stream should be closed when the BIO is freed.

BIO_get_close() returns the BIOs close flag.

BIO_pending(), BIO_ctrl_pending(), BIO_wpending() and BIO_ctrl_wpending() return the number of pending characters in the BIOs read and write buffers. Not all BIOs support these calls. BIO_ctrl_pending() and BIO_ctrl_wpending() return a size_t type and are functions, BIO_pending() and BIO_wpending() are macros which call BIO_ctrl().

RETURN VALUES

BIO_reset() normally returns 1 for success and 0 or -1 for failure. File BIOs are an exception, they return 0 for success and -1 for failure.

BIO_seek() and BIO_tell() both return the current file position on success and -1 for failure, except file BIOs which for BIO_seek() always return 0 for success and -1 for failure.

BIO_flush() returns 1 for success and 0 or -1 for failure.

BIO_eof() returns 1 if EOF has been reached 0 otherwise.

BIO_set_close() always returns 1.

BIO_get_close() returns the close flag value: BIO_CLOSE or BIO_NOCLOSE.

BIO_pending(), BIO_ctrl_pending(), BIO_wpending() and BIO_ctrl_wpending() return the amount of pending data.

NOTES

BIO_flush(), because it can write data may return 0 or -1 indicating that the call should be retried later in a similar manner to BIO_write(). The BIO_should_retry() call should be used and appropriate action taken is the call fails.

The return values of BIO_pending() and BIO_wpending() may not reliably determine the amount of pending data in all cases. For example in the case of a file BIO some data may be available in the FILE structures internal buffers but it is not possible to determine this in a portably way. For other types of BIO they may not be supported.

Filter BIOs if they do not internally handle a particular BIO_ctrl() operation usually pass the operation to the next BIO in the chain. This often means there is no need to locate the required BIO for a particular operation, it can be called on a chain and it will be automatically passed to the relevant BIO. However this can cause unexpected results: for example no current filter BIOs implement BIO_seek(), but this may still succeed if the chain ends in a FILE or file descriptor BIO.

Source/sink BIOs return an 0 if they do not recognize the BIO_ctrl() operation.

BUGS

Some of the return values are ambiguous and care should be taken. In particular a return value of 0 can be returned if an operation is not supported, if an error occurred, if EOF has not been reached and in the case of BIO_seek() on a file BIO for a successful operation.

SEE ALSO

TBA

Name

BIO_f_base64 — base64 BIO filter

Synopsis

```
#include  <openssl/bio.h>
#include  <openssl/evp.h>
```

```
BIO_METHOD *   BIO_f_base64(void);
```

DESCRIPTION

BIO_f_base64() returns the base64 BIO method. This is a filter BIO that base64 encodes any data written through it and decodes any data read through it.

Base64 BIOs do not support BIO_gets() or BIO_puts().

BIO_flush() on a base64 BIO that is being written through is used to signal that no more data is to be encoded: this is used to flush the final block through the BIO.

The flag BIO_FLAGS_BASE64_NO_NL can be set with BIO_set_flags() to encode the data all on one line or expect the data to be all on one line.

NOTES

Because of the format of base64 encoding the end of the encoded block cannot always be reliably determined.

RETURN VALUES

BIO_f_base64() returns the base64 BIO method.

EXAMPLES

Base64 encode the string "Hello World\n" and write the result to standard output:

```
BIO *bio, *b64;
char message[] = "Hello World \n";

b64 = BIO_new(BIO_f_base64());
bio = BIO_new_fp(stdout, BIO_NOCLOSE);
BIO_push(b64, bio);
BIO_write(b64, message, strlen(message));
BIO_flush(b64);

BIO_free_all(b64);
```

Read Base64 encoded data from standard input and write the decoded data to standard output:

```
BIO *bio, *b64, *bio_out;
char inbuf[512];
int inlen;

b64 = BIO_new(BIO_f_base64());
bio = BIO_new_fp(stdin, BIO_NOCLOSE);
bio_out = BIO_new_fp(stdout, BIO_NOCLOSE);
BIO_push(b64, bio);
while((inlen = BIO_read(b64, inbuf, 512)) > 0)
       BIO_write(bio_out, inbuf, inlen);
```

```
BIO_flush(bio_out);
BIO_free_all(b64);
```

BUGS

The ambiguity of EOF in base64 encoded data can cause additional data following the base64 encoded block to be misinterpreted.

There should be some way of specifying a test that the BIO can perform to reliably determine EOF (for example a MIME boundary).

SEE ALSO

TBA

Name

BIO_f_buffer — buffering BIO

Synopsis

```
#include  <openssl/bio.h>
```

```
BIO_METHOD * BIO_f_buffer(void);
```

```
#define BIO_get_buffer_num_lines(b)          BIO_ctrl(b,BIO_C_GET_BUFF_NUM_LINES,0,NULL)
#define BIO_set_read_buffer_size(b,size)     BIO_int_ctrl(b,BIO_C_SET_BUFF_SIZE,size,0)
#define BIO_set_write_buffer_size(b,size)    BIO_int_ctrl(b,BIO_C_SET_BUFF_SIZE,size,1)
#define BIO_set_buffer_size(b,size)          BIO_ctrl(b,BIO_C_SET_BUFF_SIZE,size,NULL)
#define BIO_set_buffer_read_data(b,buf,num)  BIO_ctrl(b,BIO_C_SET_BUFF_READ_DATA,num,buf)
```

DESCRIPTION

BIO_f_buffer() returns the buffering BIO method.

Data written to a buffering BIO is buffered and periodically written to the next BIO in the chain. Data read from a buffering BIO comes from an internal buffer which is filled from the next BIO in the chain. Both BIO_gets() and BIO_puts() are supported.

Calling BIO_reset() on a buffering BIO clears any buffered data.

BIO_get_buffer_num_lines() returns the number of lines currently buffered.

BIO_set_read_buffer_size(), BIO_set_write_buffer_size() and BIO_set_buffer_size() set the read, write or both read and write buffer sizes to **size**. The initial buffer size is DEFAULT_BUFFER_SIZE, currently 4096. Any attempt to reduce the buffer size below DEFAULT_BUFFER_SIZE is ignored. Any buffered data is cleared when the buffer is resized.

BIO_set_buffer_read_data() clears the read buffer and fills it with **num** bytes of **buf**. If **num** is larger than the current buffer size the buffer is expanded.

NOTES

Buffering BIOs implement BIO_gets() by using BIO_read() operations on the next BIO in the chain. By prepending a buffering BIO to a chain it is therefore possible to provide BIO_gets() functionality if the following BIOs do not support it (for example SSL BIOs).

Data is only written to the next BIO in the chain when the write buffer fills or when BIO_flush() is called. It is therefore important to call BIO_flush() whenever any pending data should be written such as when removing a buffering BIO using BIO_pop(). BIO_flush() may need to be retried if the ultimate source/sink BIO is non blocking.

RETURN VALUES

BIO_f_buffer() returns the buffering BIO method.

BIO_get_buffer_num_lines() returns the number of lines buffered (may be 0).

BIO_set_read_buffer_size(), BIO_set_write_buffer_size() and BIO_set_buffer_size() return 1 if the buffer was successfully resized or 0 for failure.

BIO_set_buffer_read_data() returns 1 if the data was set correctly or 0 if there was an error.

SEE ALSO

BIO(3), BIO_reset(3), BIO_flush(3), BIO_pop(3), BIO_ctrl(3), BIO_int_ctrl(3)

Name

BIO_f_cipher, BIO_set_cipher, BIO_get_cipher_status and BIO_get_cipher_ctx — cipher BIO filter

Synopsis

```
#include  <openssl/bio.h>
#include  <openssl/evp.h>
```

```
BIO_METHOD *   BIO_f_cipher(void);
void BIO_set_cipher(BIO *b,const EVP_CIPHER *cipher,
               unsigned char *key, unsigned char *iv, int enc);
int BIO_get_cipher_status(BIO *b)
int BIO_get_cipher_ctx(BIO *b, EVP_CIPHER_CTX **pctx)
```

DESCRIPTION

BIO_f_cipher() returns the cipher BIO method. This is a filter BIO that encrypts any data written through it, and decrypts any data read from it. It is a BIO wrapper for the cipher routines EVP_CipherInit(), EVP_CipherUpdate() and EVP_CipherFinal().

Cipher BIOs do not support BIO_gets() or BIO_puts().

BIO_flush() on an encryption BIO that is being written through is used to signal that no more data is to be encrypted: this is used to flush and possibly pad the final block through the BIO.

BIO_set_cipher() sets the cipher of BIO **b** to **cipher** using key **key** and IV **iv**. **enc** should be set to 1 for encryption and zero for decryption.

When reading from an encryption BIO the final block is automatically decrypted and checked when EOF is detected. BIO_get_cipher_status() is a BIO_ctrl() macro which can be called to determine whether the decryption operation was successful.

BIO_get_cipher_ctx() is a BIO_ctrl() macro which retrieves the internal BIO cipher context. The retrieved context can be used in conjunction with the standard cipher routines to set it up. This is useful when BIO_set_cipher() is not flexible enough for the applications needs.

NOTES

When encrypting BIO_flush() **must** be called to flush the final block through the BIO. If it is not then the final block will fail a subsequent decrypt.

When decrypting an error on the final block is signalled by a zero return value from the read operation. A successful decrypt followed by EOF will also return zero for the final read. BIO_get_cipher_status() should be called to determine if the decrypt was successful.

As always, if BIO_gets() or BIO_puts() support is needed then it can be achieved by preceding the cipher BIO with a buffering BIO.

RETURN VALUES

BIO_f_cipher() returns the cipher BIO method.

BIO_set_cipher() does not return a value.

BIO_get_cipher_status() returns 1 for a successful decrypt and 0 for failure.

BIO_get_cipher_ctx() currently always returns 1.

EXAMPLES

TBA

SEE ALSO

TBA

Name

BIO_find_type and BIO_next — BIO chain traversal

Synopsis

```
#include  <openssl/bio.h>
```

```
BIO *  BIO_find_type(BIO *b,int bio_type);
BIO *  BIO_next(BIO *b);
```

```
#define BIO_method_type(b)     ((b)->method->type)
```

```
#define BIO_TYPE_NONE          0
#define BIO_TYPE_MEM           (1|0x0400)
#define BIO_TYPE_FILE          (2|0x0400)
```

```
#define BIO_TYPE_FD            (4|0x0400|0x0100)
#define BIO_TYPE_SOCKET        (5|0x0400|0x0100)
#define BIO_TYPE_NULL          (6|0x0400)
#define BIO_TYPE_SSL           (7|0x0200)
#define BIO_TYPE_MD            (8|0x0200)
#define BIO_TYPE_BUFFER        (9|0x0200)
#define BIO_TYPE_CIPHER        (10|0x0200)
#define BIO_TYPE_BASE64        (11|0x0200)
#define BIO_TYPE_CONNECT       (12|0x0400|0x0100)
#define BIO_TYPE_ACCEPT        (13|0x0400|0x0100)
#define BIO_TYPE_PROXY_CLIENT  (14|0x0200)
#define BIO_TYPE_PROXY_SERVER  (15|0x0200)
#define BIO_TYPE_NBIO_TEST     (16|0x0200)
#define BIO_TYPE_NULL_FILTER   (17|0x0200)
#define BIO_TYPE_BER           (18|0x0200)
#define BIO_TYPE_BIO           (19|0x0400)
```

```
#define BIO_TYPE_DESCRIPTOR    0x0100
#define BIO_TYPE_FILTER        0x0200
#define BIO_TYPE_SOURCE_SINK   0x0400
```

DESCRIPTION

The BIO_find_type() searches for a BIO of a given type in a chain, starting at BIO **b**. If **type** is a specific type (such as BIO_TYPE_MEM) then a search is made for a BIO of that type. If **type** is a general type (such as **BIO_TYPE_SOURCE_SINK**) then the next matching BIO of the given general type is searched for. BIO_find_type() returns the next matching BIO or NULL if none is found.

Note: not all the **BIO_TYPE_*** types above have corresponding BIO implementations.

BIO_next() returns the next BIO in a chain. It can be used to traverse all BIOs in a chain or used in conjunction with BIO_find_type() to find all BIOs of a certain type.

BIO_method_type() returns the type of a BIO.

RETURN VALUES

BIO_find_type() returns a matching BIO or NULL for no match.

BIO_next() returns the next BIO in a chain.

BIO_method_type() returns the type of the BIO **b**.

NOTES

BIO_next() was added to OpenSSL 0.9.6 to provide a 'clean' way to traverse a BIO chain or find multiple matches using BIO_find_type(). Previous versions had to use:

```
next = bio->next_bio;
```

BUGS

BIO_find_type() in OpenSSL 0.9.5a and earlier could not be safely passed a NULL pointer for the **b** argument.

EXAMPLE

Traverse a chain looking for digest BIOs:

```
BIO *btmp;
btmp = in_bio; /* in_bio is chain to search through */

do {
       btmp = BIO_find_type(btmp, BIO_TYPE_MD);
       if(btmp == NULL) break; /* Not found */
       /* btmp is a digest BIO, do something with it …*/
       …

       btmp = BIO_next(btmp);
} while(btmp);
```

SEE ALSO

TBA

Name

BIO_f_md, BIO_set_md, BIO_get_md and BIO_get_md_ctx — message digest BIO filter

Synopsis

```
#include  <openssl/bio.h>
#include  <openssl/evp.h>
```

```
BIO_METHOD *   BIO_f_md(void);
int BIO_set_md(BIO *b,EVP_MD *md);
int BIO_get_md(BIO *b,EVP_MD **mdp);
int BIO_get_md_ctx(BIO *b,EVP_MD_CTX **mdcp);
```

DESCRIPTION

BIO_f_md() returns the message digest BIO method. This is a filter BIO that digests any data passed through it, it is a BIO wrapper for the digest routines EVP_DigestInit(), EVP_DigestUpdate() and EVP_DigestFinal().

Any data written or read through a digest BIO using BIO_read() and BIO_write() is digested.

BIO_gets(), if its **size** parameter is large enough finishes the digest calculation and returns the digest value. BIO_puts() is not supported.

BIO_reset() reinitialises a digest BIO.

BIO_set_md() sets the message digest of BIO **b** to **md**: this must be called to initialize a digest BIO before any data is passed through it. It is a BIO_ctrl() macro.

BIO_get_md() places the a pointer to the digest BIOs digest method in **mdp**, it is a BIO_ctrl() macro.

BIO_get_md_ctx() returns the digest BIOs context into **mdcp**.

NOTES

The context returned by BIO_get_md_ctx() can be used in calls to EVP_DigestFinal() and also the signature routines EVP_SignFinal() and EVP_VerifyFinal().

The context returned by BIO_get_md_ctx() is an internal context structure. Changes made to this context will affect the digest BIO itself and the context pointer will become invalid when the digest BIO is freed.

After the digest has been retrieved from a digest BIO it must be reinitialized by calling BIO_reset(), or BIO_set_md() before any more data is passed through it.

If an application needs to call BIO_gets() or BIO_puts() through a chain containing digest BIOs then this can be done by prepending a buffering BIO.

Before OpenSSL 1.0.0 the call to BIO_get_md_ctx() would only work if the BIO had been initialized for example by calling BIO_set_md()). In OpenSSL 1.0.0 and later the context is always returned and the BIO is state is set to initialized. This allows applications to initialize the context externally if the standard calls such as BIO_set_md() are not sufficiently flexible.

RETURN VALUES

BIO_f_md() returns the digest BIO method.

BIO_set_md(), BIO_get_md() and BIO_md_ctx() return 1 for success and 0 for failure.

EXAMPLES

The following example creates a BIO chain containing an SHA1 and MD5 digest BIO and passes the string "Hello World" through it. Error checking has been omitted for clarity.

```
BIO *bio, *mdtmp;
char message[] = "Hello World";
bio = BIO_new(BIO_s_null());
mdtmp = BIO_new(BIO_f_md());
BIO_set_md(mdtmp, EVP_sha1());
/* For BIO_push() we want to append the sink BIO and keep a note of
 * the start of the chain.
 */
bio = BIO_push(mdtmp, bio);
mdtmp = BIO_new(BIO_f_md());
BIO_set_md(mdtmp, EVP_md5());
bio = BIO_push(mdtmp, bio);
/* Note: mdtmp can now be discarded */
BIO_write(bio, message, strlen(message));
```

The next example digests data by reading through a chain instead:

```
BIO *bio, *mdtmp;
char buf[1024];
int rdlen;
bio = BIO_new_file(file, "rb");
mdtmp = BIO_new(BIO_f_md());
BIO_set_md(mdtmp, EVP_sha1());
bio = BIO_push(mdtmp, bio);
mdtmp = BIO_new(BIO_f_md());
BIO_set_md(mdtmp, EVP_md5());
bio = BIO_push(mdtmp, bio);
do {
       rdlen = BIO_read(bio, buf, sizeof(buf));
       /* Might want to do something with the data here */
} while(rdlen > 0);
```

This next example retrieves the message digests from a BIO chain and outputs them. This could be used with the examples above.

```
BIO *mdtmp;
unsigned char mdbuf[EVP_MAX_MD_SIZE];
int mdlen;
int i;
mdtmp = bio;   /* Assume bio has previously been set up */
do {
       EVP_MD *md;
       mdtmp = BIO_find_type(mdtmp, BIO_TYPE_MD);
       if(!mdtmp) break;
       BIO_get_md(mdtmp, &md);
       printf("%s digest", OBJ_nid2sn(EVP_MD_type(md)));
       mdlen = BIO_gets(mdtmp, mdbuf, EVP_MAX_MD_SIZE);
       for(i = 0; i < mdlen; i++) printf(":%02X", mdbuf[i]);
       printf("\n");
       mdtmp = BIO_next(mdtmp);
} while(mdtmp);

BIO_free_all(bio);
```

BUGS

The lack of support for BIO_puts() and the non standard behaviour of BIO_gets() could be regarded as anomalous. It could be argued that BIO_gets() and BIO_puts() should be passed to the next BIO in the chain and digest the data passed through and that digests should be retrieved using a separate BIO_ctrl() call.

SEE ALSO

TBA

Name

BIO_f_null — null filter

Synopsis

```
#include  <openssl/bio.h>
```

```
BIO_METHOD *   BIO_f_null(void);
```

DESCRIPTION

BIO_f_null() returns the null filter BIO method. This is a filter BIO that does nothing.

All requests to a null filter BIO are passed through to the next BIO in the chain: this means that a BIO chain containing a null filter BIO behaves just as though the BIO was not there.

NOTES

As may be apparent a null filter BIO is not particularly useful.

RETURN VALUES

BIO_f_null() returns the null filter BIO method.

SEE ALSO

TBA

Name

BIO_f_ssl, BIO_set_ssl, BIO_get_ssl, BIO_set_ssl_mode, BIO_set_ssl_renegotiate_bytes, BIO_get_num_renegotiates, BIO_set_ssl_renegotiate_timeout, BIO_new_ssl, BIO_new_ssl_connect, BIO_new_buffer_ssl_connect, BIO_ssl_copy_session_id and BIO_ssl_shutdown — SSL BIO

Synopsis

```
#include  <openssl/bio.h>
#include  <openssl/ssl.h>
```

```
BIO_METHOD *BIO_f_ssl(void);
```

```
#define BIO_set_ssl(b,ssl,c)          BIO_ctrl(b,BIO_C_SET_SSL,c,(char *)ssl)
#define BIO_get_ssl(b,sslp)           BIO_ctrl(b,BIO_C_GET_SSL,0,(char *)sslp)
#define BIO_set_ssl_mode(b,client)    BIO_ctrl(b,BIO_C_SSL_MODE,client,NULL)
#define BIO_set_ssl_renegotiate_bytes(b,num) \
       BIO_ctrl(b,BIO_C_SET_SSL_RENEGOTIATE_BYTES,num,NULL);
#define BIO_set_ssl_renegotiate_timeout(b,seconds) \
       BIO_ctrl(b,BIO_C_SET_SSL_RENEGOTIATE_TIMEOUT,seconds,NULL);
#define BIO_get_num_renegotiates(b) \
       BIO_ctrl(b,BIO_C_SET_SSL_NUM_RENEGOTIATES,0,NULL);
```

```
BIO *BIO_new_ssl(SSL_CTX *ctx,int client);
BIO *BIO_new_ssl_connect(SSL_CTX *ctx);
BIO *BIO_new_buffer_ssl_connect(SSL_CTX *ctx);
int BIO_ssl_copy_session_id(BIO *to,BIO *from);
void BIO_ssl_shutdown(BIO *bio);
```

```
#define BIO_do_handshake(b)           BIO_ctrl(b,BIO_C_DO_STATE_MACHINE,0,NULL)
```

DESCRIPTION

BIO_f_ssl() returns the SSL BIO method. This is a filter BIO which is a wrapper round the OpenSSL SSL routines adding a BIO "flavour" to SSL I/O.

I/O performed on an SSL BIO communicates using the SSL protocol with the SSLs read and write BIOs. If an SSL connection is not established then an attempt is made to establish one on the first I/O call.

If a BIO is appended to an SSL BIO using BIO_push() it is automatically used as the SSL BIOs read and write BIOs.

Calling BIO_reset() on an SSL BIO closes down any current SSL connection by calling SSL_shutdown(). BIO_reset() is then sent to the next BIO in the chain: this will typically disconnect the underlying transport. The SSL BIO is then reset to the initial accept or connect state.

If the close flag is set when an SSL BIO is freed then the internal SSL structure is also freed using SSL_free().

BIO_set_ssl() sets the internal SSL pointer of BIO **b** to **ssl** using the close flag **c**.

BIO_get_ssl() retrieves the SSL pointer of BIO **b**, it can then be manipulated using the standard SSL library functions.

BIO_set_ssl_mode() sets the SSL BIO mode to **client**. If **client** is 1 client mode is set. If **client** is 0 server mode is set.

BIO_set_ssl_renegotiate_bytes() sets the renegotiate byte count to **num**. When set after every **num** bytes of I/O (read and write) the SSL session is automatically renegotiated. **num** must be at least 512 bytes.

BIO_set_ssl_renegotiate_timeout() sets the renegotiate timeout to **seconds**. When the renegotiate timeout elapses the session is automatically renegotiated.

BIO_get_num_renegotiates() returns the total number of session renegotiations due to I/O or timeout.

BIO_new_ssl() allocates an SSL BIO using SSL_CTX **ctx** and using client mode if **client** is non zero.

BIO_new_ssl_connect() creates a new BIO chain consisting of an SSL BIO (using **ctx**) followed by a connect BIO.

BIO_new_buffer_ssl_connect() creates a new BIO chain consisting of a buffering BIO, an SSL BIO (using **ctx**) and a connect BIO.

BIO_ssl_copy_session_id() copies an SSL session id between BIO chains **from** and **to**. It does this by locating the SSL BIOs in each chain and calling SSL_copy_session_id() on the internal SSL pointer.

BIO_ssl_shutdown() closes down an SSL connection on BIO chain **bio**. It does this by locating the SSL BIO in the chain and calling SSL_shutdown() on its internal SSL pointer.

BIO_do_handshake() attempts to complete an SSL handshake on the supplied BIO and establish the SSL connection. It returns 1 if the connection was established successfully. A zero or negative value is returned if the connection could not be established, the call BIO_should_retry() should be used for non blocking connect BIOs to determine if the call should be retried. If an SSL connection has already been established this call has no effect.

NOTES

SSL BIOs are exceptional in that if the underlying transport is non blocking they can still request a retry in exceptional circumstances. Specifically this will happen if a session renegotiation takes place during a BIO_read() operation, one case where this happens is when SGC or step up occurs.

In OpenSSL 0.9.6 and later the SSL flag SSL_AUTO_RETRY can be set to disable this behaviour. That is when this flag is set an SSL BIO using a blocking transport will never request a retry.

Since unknown BIO_ctrl() operations are sent through filter BIOs the servers name and port can be set using BIO_set_host() on the BIO returned by BIO_new_ssl_connect() without having to locate the connect BIO first.

Applications do not have to call BIO_do_handshake() but may wish to do so to separate the handshake process from other I/O processing.

RETURN VALUES

TBA

EXAMPLE

This SSL/TLS client example, attempts to retrieve a page from an SSL/TLS web server. The I/O routines are identical to those of the unencrypted example in BIO_s_connect(3).

```
BIO *sbio, *out;
int len;
char tmpbuf[1024];
SSL_CTX *ctx;
SSL *ssl;

ERR_load_crypto_strings();
ERR_load_SSL_strings();
OpenSSL_add_all_algorithms();

/* We would seed the PRNG here if the platform didn't
 * do it automatically
 */

ctx = SSL_CTX_new(SSLv23_client_method());

/* We'd normally set some stuff like the verify paths and
 * mode here because as things stand this will connect to
 * any server whose certificate is signed by any CA.
 */

sbio = BIO_new_ssl_connect(ctx);
```

```
BIO_get_ssl(sbio, &ssl);

if(!ssl) {
  fprintf(stderr, "Can't locate SSL pointer\n");
  /* whatever … */
}

/* Don't want any retries */
SSL_set_mode(ssl, SSL_MODE_AUTO_RETRY);

/* We might want to do other things with ssl here */

BIO_set_conn_hostname(sbio, "localhost:https");

out = BIO_new_fp(stdout, BIO_NOCLOSE);
if(BIO_do_connect(sbio) <= 0) {
       fprintf(stderr, "Error connecting to server\n");
       ERR_print_errors_fp(stderr);
       /* whatever … */
}

if(BIO_do_handshake(sbio) <= 0) {
       fprintf(stderr, "Error establishing SSL connection\n");
       ERR_print_errors_fp(stderr);
       /* whatever … */
}

/* Could examine ssl here to get connection info */

BIO_puts(sbio, "GET / HTTP/1.0\n\n");
for(;;) {
       len = BIO_read(sbio, tmpbuf, 1024);
       if(len <= 0) break;
       BIO_write(out, tmpbuf, len);
}
BIO_free_all(sbio);
BIO_free(out);
```

Here is a simple server example. It makes use of a buffering BIO to allow lines to be read from the SSL BIO using BIO_gets. It creates a pseudo web page containing the actual request from a client and also echoes the request to standard output.

```
BIO *sbio, *bbio, *acpt, *out;
int len;
char tmpbuf[1024];
SSL_CTX *ctx;
SSL *ssl;

ERR_load_crypto_strings();
ERR_load_SSL_strings();
OpenSSL_add_all_algorithms();

/* Might seed PRNG here */

ctx = SSL_CTX_new(SSLv23_server_method());

if (!SSL_CTX_use_certificate_file(ctx,"server.pem",SSL_FILETYPE_PEM)
       || !SSL_CTX_use_PrivateKey_file(ctx,"server.pem",SSL_FILETYPE_PEM)
       || !SSL_CTX_check_private_key(ctx)) {

       fprintf(stderr, "Error setting up SSL_CTX\n");
       ERR_print_errors_fp(stderr);
       return 0;
}

/* Might do other things here like setting verify locations and
 * DH and/or RSA temporary key callbacks
 */
```

```
/* New SSL BIO setup as server */
sbio=BIO_new_ssl(ctx,0);

BIO_get_ssl(sbio, &ssl);

if(!ssl) {
  fprintf(stderr, "Can't locate SSL pointer\n");
  /* whatever … */
}

/* Don't want any retries */
SSL_set_mode(ssl, SSL_MODE_AUTO_RETRY);

/* Create the buffering BIO */

bbio = BIO_new(BIO_f_buffer());

/* Add to chain */
sbio = BIO_push(bbio, sbio);

acpt=BIO_new_accept("4433");

/* By doing this when a new connection is established
 * we automatically have sbio inserted into it. The
 * BIO chain is now 'swallowed' by the accept BIO and
 * will be freed when the accept BIO is freed.
 */

BIO_set_accept_bios(acpt,sbio);

out = BIO_new_fp(stdout, BIO_NOCLOSE);

/* Setup accept BIO */
if(BIO_do_accept(acpt) <= 0) {
       fprintf(stderr, "Error setting up accept BIO\n");
       ERR_print_errors_fp(stderr);
       return 0;
}

/* Now wait for incoming connection */
if(BIO_do_accept(acpt) <= 0) {
       fprintf(stderr, "Error in connection\n");
       ERR_print_errors_fp(stderr);
       return 0;
}

/* We only want one connection so remove and free
 * accept BIO
 */

sbio = BIO_pop(acpt);

BIO_free_all(acpt);

if(BIO_do_handshake(sbio) <= 0) {
       fprintf(stderr, "Error in SSL handshake\n");
       ERR_print_errors_fp(stderr);
       return 0;
}

BIO_puts(sbio, "HTTP/1.0 200 OK\r\nContent-type: text/plain\r\n\r\n");
BIO_puts(sbio, "\r\nConnection Established\r\nRequest headers:\r\n");
BIO_puts(sbio, "--------------------------------------------------\r\n");

for(;;) {
       len = BIO_gets(sbio, tmpbuf, 1024);
       if(len <= 0) break;
       BIO_write(sbio, tmpbuf, len);
       BIO_write(out, tmpbuf, len);
       /* Look for blank line signifying end of headers*/
       if((tmpbuf[0] == '\r') || (tmpbuf[0] == '\n')) break;
}

BIO_puts(sbio, "--------------------------------------------------\r\n");
BIO_puts(sbio, "\r\n");

/* Since there is a buffering BIO present we had better flush it */
BIO_flush(sbio);

BIO_free_all(sbio);
```

BUGS

In OpenSSL versions before 1.0.0 the BIO_pop() call was handled incorrectly, the I/O BIO reference count was incorrectly incremented (instead of decremented) and dissociated with the SSL BIO even if the SSL BIO was not explicitly being popped (e.g. a pop higher up the chain). Applications which included workarounds for this bug (e.g. freeing BIOs more than once) should be modified to handle this fix or they may free up an already freed BIO.

SEE ALSO

TBA

Name

BIO_new_CMS — CMS streaming filter BIO

Synopsis

```
#include  <openssl/cms.h>
```

```
BIO *BIO_new_CMS(BIO *out, CMS_ContentInfo *cms);
```

DESCRIPTION

BIO_new_CMS() returns a streaming filter BIO chain based on **cms**. The output of the filter is written to **out**. Any data written to the chain is automatically translated to a BER format CMS structure of the appropriate type.

NOTES

The chain returned by this function behaves like a standard filter BIO. It supports non blocking I/O. Content is processed and streamed on the fly and not all held in memory at once: so it is possible to encode very large structures. After all content has been written through the chain BIO_flush() must be called to finalise the structure.

The **CMS_STREAM** flag must be included in the corresponding **flags** parameter of the **cms** creation function.

If an application wishes to write additional data to **out** BIOs should be removed from the chain using BIO_pop() and freed with BIO_free() until **out** is reached. If no additional data needs to be written BIO_free_all() can be called to free up the whole chain.

Any content written through the filter is used verbatim: no canonical translation is performed.

It is possible to chain multiple BIOs to, for example, create a triple wrapped signed, enveloped, signed structure. In this case it is the applications responsibility to set the inner content type of any outer CMS_ContentInfo structures.

Large numbers of small writes through the chain should be avoided as this will produce an output consisting of lots of OCTET STRING structures. Prepending a BIO_f_buffer() buffering BIO will prevent this.

BUGS

There is currently no corresponding inverse BIO: i.e. one which can decode a CMS structure on the fly.

RETURN VALUES

BIO_new_CMS() returns a BIO chain when successful or NULL if an error occurred. The error can be obtained from ERR_get_error(3).

SEE ALSO

ERR_get_error(3), CMS_sign(3), CMS_encrypt(3)

HISTORY

BIO_new_CMS() was added to OpenSSL 1.0.0

Name

BIO_new, BIO_set, BIO_free, BIO_vfree and BIO_free_all — BIO allocation and freeing functions

Synopsis

```
#include  <openssl/bio.h>
```

```
BIO *  BIO_new(BIO_METHOD *type);
int    BIO_set(BIO *a,BIO_METHOD *type);
int    BIO_free(BIO *a);
void   BIO_vfree(BIO *a);
void   BIO_free_all(BIO *a);
```

DESCRIPTION

BIO_set() sets the method of an already existing BIO.

BIO_free() frees up a single BIO, BIO_vfree() also frees up a single BIO but it does not return a value. Calling BIO_free() may also have some effect on the underlying I/O structure, for example it may close the file being referred to under certain circumstances. For more details see the individual BIO_METHOD descriptions.

BIO_free_all() frees up an entire BIO chain, it does not halt if an error occurs freeing up an individual BIO in the chain.

RETURN VALUES

BIO_new() returns a newly created BIO or NULL if the call fails.

BIO_set(), BIO_free() return 1 for success and 0 for failure.

BIO_free_all() and BIO_vfree() do not return values.

NOTES

Some BIOs (such as memory BIOs) can be used immediately after calling BIO_new(). Others (such as file BIOs) need some additional initialization, and frequently a utility function exists to create and initialize such BIOs.

If BIO_free() is called on a BIO chain it will only free one BIO resulting in a memory leak.

Calling BIO_free_all() a single BIO has the same effect as calling BIO_free() on it other than the discarded return value.

Normally the **type** argument is supplied by a function which returns a pointer to a BIO_METHOD. There is a naming convention for such functions: a source/sink BIO is normally called BIO_s_*() and a filter BIO BIO_f_*();

EXAMPLE

Create a memory BIO:

```
BIO *mem = BIO_new(BIO_s_mem());
```

SEE ALSO

TBA

Name

BIO_push and BIO_pop — add and remove BIOs from a chain.

Synopsis

```
#include  <openssl/bio.h>
```

```
BIO *  BIO_push(BIO *b,BIO *append);
BIO *  BIO_pop(BIO *b);
```

DESCRIPTION

The BIO_push() function appends the BIO **append** to **b**, it returns **b**.

BIO_pop() removes the BIO **b** from a chain and returns the next BIO in the chain, or NULL if there is no next BIO. The removed BIO then becomes a single BIO with no association with the original chain, it can thus be freed or attached to a different chain.

NOTES

The names of these functions are perhaps a little misleading. BIO_push() joins two BIO chains whereas BIO_pop() deletes a single BIO from a chain, the deleted BIO does not need to be at the end of a chain.

The process of calling BIO_push() and BIO_pop() on a BIO may have additional consequences (a control call is made to the affected BIOs) any effects will be noted in the descriptions of individual BIOs.

EXAMPLES

For these examples suppose **md1** and **md2** are digest BIOs, **b64** is a base64 BIO and **f** is a file BIO.

If the call:

```
BIO_push(b64, f);
```

is made then the new chain will be **b64-f**. After making the calls

```
BIO_push(md2, b64);
BIO_push(md1, md2);
```

the new chain is **md1-md2-b64-f**. Data written to **md1** will be digested by **md1** and **md2**, **base64** encoded and written to **f**.

It should be noted that reading causes data to pass in the reverse direction, that is data is read from **f**, base64 **decoded** and digested by **md1** and **md2**. If the call:

```
BIO_pop(md2);
```

The call will return **b64** and the new chain will be **md1-b64-f** data can be written to **md1** as before.

RETURN VALUES

BIO_push() returns the end of the chain, **b**.

BIO_pop() returns the next BIO in the chain, or NULL if there is no next BIO.

SEE ALSO

TBA

Name

BIO_read, BIO_write, BIO_gets and BIO_puts — BIO I/O functions

Synopsis

```
#include  <openssl/bio.h>
```

```
int    BIO_read(BIO *b, void *buf, int len);
int    BIO_gets(BIO *b, char *buf, int size);
int    BIO_write(BIO *b, const void *buf, int len);
int    BIO_puts(BIO *b, const char *buf);
```

DESCRIPTION

BIO_read() attempts to read **len** bytes from BIO **b** and places the data in **buf**.

BIO_gets() performs the BIOs "gets" operation and places the data in **buf**. Usually this operation will attempt to read a line of data from the BIO of maximum length **len**. There are exceptions to this however, for example BIO_gets() on a digest BIO will calculate and return the digest and other BIOs may not support BIO_gets() at all.

BIO_write() attempts to write **len** bytes from **buf** to BIO **b**.

BIO_puts() attempts to write a null terminated string **buf** to BIO **b**.

RETURN VALUES

All these functions return either the amount of data successfully read or written (if the return value is positive) or that no data was successfully read or written if the result is 0 or -1. If the return value is -2 then the operation is not implemented in the specific BIO type.

NOTES

A 0 or -1 return is not necessarily an indication of an error. In particular when the source/sink is non-blocking or of a certain type it may merely be an indication that no data is currently available and that the application should retry the operation later.

One technique sometimes used with blocking sockets is to use a system call (such as select(), poll() or equivalent) to determine when data is available and then call read() to read the data. The equivalent with BIOs (that is call select() on the underlying I/O structure and then call BIO_read() to read the data) should **not** be used because a single call to BIO_read() can cause several reads (and writes in the case of SSL BIOs) on the underlying I/O structure and may block as a result. Instead select() (or equivalent) should be combined with non blocking I/O so successive reads will request a retry instead of blocking.

See BIO_should_retry(3) for details of how to determine the cause of a retry and other I/O issues.

If the BIO_gets() function is not supported by a BIO then it possible to work around this by adding a buffering BIO BIO_f_buffer(3) to the chain.

SEE ALSO

BIO_should_retry(3)

TBA

Name

BIO_s_accept, BIO_set_accept_port, BIO_get_accept_port, BIO_set_nbio_accept, BIO_set_accept_bios, BIO_set_bind_mode, BIO_get_bind_mode and BIO_do_accept — accept BIO

Synopsis

```
#include  <openssl/bio.h>
```

```
BIO_METHOD *BIO_s_accept(void);
```

```
long BIO_set_accept_port(BIO *b, char *name);
char *BIO_get_accept_port(BIO *b);
```

```
BIO *BIO_new_accept(char *host_port);
```

```
long BIO_set_nbio_accept(BIO *b, int n);
long BIO_set_accept_bios(BIO *b, char *bio);
```

```
long BIO_set_bind_mode(BIO *b, long mode);
long BIO_get_bind_mode(BIO *b, long dummy);
```

```
#define BIO_BIND_NORMAL                 0
#define BIO_BIND_REUSEADDR_IF_UNUSED    1
#define BIO_BIND_REUSEADDR              2
```

```
int BIO_do_accept(BIO *b);
```

DESCRIPTION

BIO_s_accept() returns the accept BIO method. This is a wrapper round the platform's TCP/IP socket accept routines.

Using accept BIOs, TCP/IP connections can be accepted and data transferred using only BIO routines. In this way any platform specific operations are hidden by the BIO abstraction.

Read and write operations on an accept BIO will perform I/O on the underlying connection. If no connection is established and the port (see below) is set up properly then the BIO waits for an incoming connection.

Accept BIOs support BIO_puts() but not BIO_gets().

If the close flag is set on an accept BIO then any active connection on that chain is shutdown and the socket closed when the BIO is freed.

Calling BIO_reset() on a accept BIO will close any active connection and reset the BIO into a state where it awaits another incoming connection.

BIO_get_fd() and BIO_set_fd() can be called to retrieve or set the accept socket. See BIO_s_fd(3)

BIO_set_accept_port() uses the string **name** to set the accept port. The port is represented as a string of the form "host:port", where "host" is the interface to use and "port" is the port. The host can be can be "*" which is interpreted as meaning any interface; "port" has the same syntax as the port specified in BIO_set_conn_port() for connect BIOs, that is it can be a numerical port string or a string to lookup using getservbyname() and a string table.

BIO_new_accept() combines BIO_new() and BIO_set_accept_port() into a single call: that is it creates a new accept BIO with port **host_port**.

BIO_set_nbio_accept() sets the accept socket to blocking mode (the default) if **n** is 0 or non blocking mode if **n** is 1.

BIO_set_accept_bios() can be used to set a chain of BIOs which will be duplicated and prepended to the chain when an incoming connection is received. This is useful if, for example, a buffering or SSL BIO is required for each connection. The chain of BIOs must not be freed after this call, they will be automatically freed when the accept BIO is freed.

BIO_set_bind_mode() and BIO_get_bind_mode() set and retrieve the current bind mode. If BIO_BIND_NORMAL (the default) is set then another socket cannot be bound to the same port. If BIO_BIND_REUSEADDR is set then other sockets can bind to the same port. If BIO_BIND_REUSEADDR_IF_UNUSED is set then and attempt is first made to use BIO_BIN_NORMAL, if this fails and the port is not in use then a second attempt is made using BIO_BIND_REUSEADDR.

BIO_do_accept() serves two functions. When it is first called, after the accept BIO has been setup, it will attempt to create the accept socket and bind an address to it. Second and subsequent calls to BIO_do_accept() will await an incoming connection, or request a retry in non blocking mode.

NOTES

When an accept BIO is at the end of a chain it will await an incoming connection before processing I/O calls. When an accept BIO is not at then end of a chain it passes I/O calls to the next BIO in the chain.

When a connection is established a new socket BIO is created for the connection and appended to the chain. That is the chain is now accept->socket. This effectively means that attempting I/O on an initial accept socket will await an incoming connection then perform I/O on it.

If any additional BIOs have been set using BIO_set_accept_bios() then they are placed between the socket and the accept BIO, that is the chain will be accept->otherbios->socket.

If a server wishes to process multiple connections (as is normally the case) then the accept BIO must be made available for further incoming connections. This can be done by waiting for a connection and then calling:

```
connection = BIO_pop(accept);
```

After this call **connection** will contain a BIO for the recently established connection and **accept** will now be a single BIO again which can be used to await further incoming connections. If no further connections will be accepted the **accept** can be freed using BIO_free().

If only a single connection will be processed it is possible to perform I/O using the accept BIO itself. This is often undesirable however because the accept BIO will still accept additional incoming connections. This can be resolved by using BIO_pop() (see above) and freeing up the accept BIO after the initial connection.

If the underlying accept socket is non-blocking and BIO_do_accept() is called to await an incoming connection it is possible for BIO_should_io_special() with the reason BIO_RR_ACCEPT. If this happens then it is an indication that an accept attempt would block: the application should take appropriate action to wait until the underlying socket has accepted a connection and retry the call.

BIO_set_accept_port(), BIO_get_accept_port(), BIO_set_nbio_accept(), BIO_set_accept_bios(), BIO_set_bind_mode(), BIO_get_bind_mode() and BIO_do_accept() are macros.

RETURN VALUES

TBA

EXAMPLE

This example accepts two connections on port 4444, sends messages down each and finally closes both down.

```
BIO *abio, *cbio, *cbio2;
ERR_load_crypto_strings();
abio = BIO_new_accept("4444");

/* First call to BIO_accept() sets up accept BIO */
if(BIO_do_accept(abio) <= 0) {
       fprintf(stderr, "Error setting up accept\n");
       ERR_print_errors_fp(stderr);
       exit(0);
}

/* Wait for incoming connection */
if(BIO_do_accept(abio) <= 0) {
       fprintf(stderr, "Error accepting connection\n");
       ERR_print_errors_fp(stderr);
       exit(0);
}
fprintf(stderr, "Connection 1 established\n");
/* Retrieve BIO for connection */
cbio = BIO_pop(abio);
BIO_puts(cbio, "Connection 1: Sending out Data on initial connection\n");
fprintf(stderr, "Sent out data on connection 1\n");
/* Wait for another connection */
if(BIO_do_accept(abio) <= 0) {
       fprintf(stderr, "Error accepting connection\n");
       ERR_print_errors_fp(stderr);
       exit(0);
}
fprintf(stderr, "Connection 2 established\n");
/* Close accept BIO to refuse further connections */
cbio2 = BIO_pop(abio);
BIO_free(abio);
BIO_puts(cbio2, "Connection 2: Sending out Data on second\n");
fprintf(stderr, "Sent out data on connection 2\n");

BIO_puts(cbio, "Connection 1: Second connection established\n");
/* Close the two established connections */
BIO_free(cbio);
BIO_free(cbio2);
```

SEE ALSO

TBA

Name

BIO_s_bio, BIO_make_bio_pair, BIO_destroy_bio_pair, BIO_shutdown_wr, BIO_set_write_buf_size, BIO_get_write_buf_size, BIO_new_bio_pair, BIO_get_write_guarantee, BIO_ctrl_get_write_guarantee, BIO_get_read_request, BIO_ctrl_get_read_request and BIO_ctrl_reset_read_request — BIO pair BIO

Synopsis

```
#include  <openssl/bio.h>

BIO_METHOD *BIO_s_bio(void);

#define BIO_make_bio_pair(b1,b2)        (int)BIO_ctrl(b1,BIO_C_MAKE_BIO_PAIR,0,b2)
#define BIO_destroy_bio_pair(b)         (int)BIO_ctrl(b,BIO_C_DESTROY_BIO_PAIR,0,NULL)

#define BIO_shutdown_wr(b)              (int)BIO_ctrl(b, BIO_C_SHUTDOWN_WR, 0, NULL)

#define BIO_set_write_buf_size(b,size)  (int)BIO_ctrl(b,BIO_C_SET_WRITE_BUF_SIZE,size,NULL)
#define BIO_get_write_buf_size(b,size)  (size_t)BIO_ctrl(b,BIO_C_GET_WRITE_BUF_SIZE,size,NULL)

int BIO_new_bio_pair(BIO **bio1, size_t writebuf1, BIO **bio2, size_t writebuf2);

#define BIO_get_write_guarantee(b)      (int)BIO_ctrl(b,BIO_C_GET_WRITE_GUARANTEE,0,NULL)
size_t BIO_ctrl_get_write_guarantee(BIO *b);

#define BIO_get_read_request(b)         (int)BIO_ctrl(b,BIO_C_GET_READ_REQUEST,0,NULL)
size_t BIO_ctrl_get_read_request(BIO *b);

int BIO_ctrl_reset_read_request(BIO *b);
```

DESCRIPTION

BIO_s_bio() returns the method for a BIO pair. A BIO pair is a pair of source/sink BIOs where data written to either half of the pair is buffered and can be read from the other half. Both halves must usually by handled by the same application thread since no locking is done on the internal data structures.

Since BIO chains typically end in a source/sink BIO it is possible to make this one half of a BIO pair and have all the data processed by the chain under application control.

One typical use of BIO pairs is to place TLS/SSL I/O under application control, this can be used when the application wishes to use a non standard transport for TLS/SSL or the normal socket routines are inappropriate.

Calls to BIO_read() will read data from the buffer or request a retry if no data is available.

Calls to BIO_write() will place data in the buffer or request a retry if the buffer is full.

The standard calls BIO_ctrl_pending() and BIO_ctrl_wpending() can be used to determine the amount of pending data in the read or write buffer.

BIO_reset() clears any data in the write buffer.

BIO_make_bio_pair() joins two separate BIOs into a connected pair.

BIO_destroy_pair() destroys the association between two connected BIOs. Freeing up any half of the pair will automatically destroy the association.

BIO_shutdown_wr() is used to close down a BIO **b**. After this call no further writes on BIO **b** are allowed (they will return an error). Reads on the other half of the pair will return any pending data or EOF when all pending data has been read.

BIO_set_write_buf_size() sets the write buffer size of BIO **b** to **size**. If the size is not initialized a default value is used. This is currently 17K, sufficient for a maximum size TLS record.

BIO_get_write_buf_size() returns the size of the write buffer.

BIO_new_bio_pair() combines the calls to BIO_new(), BIO_make_bio_pair() and BIO_set_write_buf_size() to create a connected pair of BIOs **bio1**, **bio2** with write buffer sizes **writebuf1** and **writebuf2**. If either size is zero then the default size is used. BIO_new_bio_pair() does not check whether **bio1** or **bio2** do point to some other BIO, the values are overwritten, BIO_free() is not called.

BIO_get_write_guarantee() and BIO_ctrl_get_write_guarantee() return the maximum length of data that can be currently written to the BIO. Writes larger than this value will return a value from BIO_write() less than the amount requested or if the buffer is full request a retry. BIO_ctrl_get_write_guarantee() is a function whereas BIO_get_write_guarantee() is a macro.

BIO_get_read_request() and BIO_ctrl_get_read_request() return the amount of data requested, or the buffer size if it is less, if the last read attempt at the other half of the BIO pair failed due to an empty buffer. This can be used to determine how much data should be written to the BIO so the next read will succeed: this is most useful in TLS/SSL applications where the amount of data read is usually meaningful rather than just a buffer size. After a successful read this call will return zero. It also will return zero once new data has been written satisfying the read request or part of it. Note that BIO_get_read_request() never returns an amount larger than that returned by BIO_get_write_guarantee().

BIO_ctrl_reset_read_request() can also be used to reset the value returned by BIO_get_read_request() to zero.

NOTES

Both halves of a BIO pair should be freed. That is even if one half is implicit freed due to a BIO_free_all() or SSL_free() call the other half needs to be freed.

When used in bidirectional applications (such as TLS/SSL) care should be taken to flush any data in the write buffer. This can be done by calling BIO_pending() on the other half of the pair and, if any data is pending, reading it and sending it to the underlying transport. This must be done before any normal processing (such as calling select()) due to a request and BIO_should_read() being true.

To see why this is important consider a case where a request is sent using BIO_write() and a response read with BIO_read(), this can occur during an TLS/SSL handshake for example. BIO_write() will succeed and place data in the write buffer. BIO_read() will initially fail and BIO_should_read() will be true. If the application then waits for data to be available on the underlying transport before flushing the write buffer it will never succeed because the request was never sent!

RETURN VALUES

BIO_new_bio_pair() returns 1 on success, with the new BIOs available in **bio1** and **bio2**, or 0 on failure, with NULL pointers stored into the locations for **bio1** and **bio2**. Check the error stack for more information.

[XXXXX: More return values need to be added here]

EXAMPLE

The BIO pair can be used to have full control over the network access of an application. The application can call select() on the socket as required without having to go through the SSL-interface.

```
BIO *internal_bio, *network_bio;
…
BIO_new_bio_pair(internal_bio, 0, network_bio, 0);
SSL_set_bio(ssl, internal_bio, internal_bio);
SSL_operations();
…

application |   TLS-engine
   |        |
   +----------> SSL_operations()
```

```
            |     /\    ||
            |     ||    \/
            |   BIO-pair (internal_bio)
   +----------< BIO-pair (network_bio)
   |        |
 socket     |

…
SSL_free(ssl);                /* implicitly frees internal_bio */
BIO_free(network_bio);
…
```

As the BIO pair will only buffer the data and never directly access the connection, it behaves non-blocking and will return as soon as the write buffer is full or the read buffer is drained. Then the application has to flush the write buffer and/or fill the read buffer.

Use the BIO_ctrl_pending(), to find out whether data is buffered in the BIO and must be transfered to the network. Use BIO_ctrl_get_read_request() to find out, how many bytes must be written into the buffer before the SSL_operation() can successfully be continued.

WARNING

As the data is buffered, SSL_operation() may return with a ERROR_SSL_WANT_READ condition, but there is still data in the write buffer. An application must not rely on the error value of SSL_operation() but must assure that the write buffer is always flushed first. Otherwise a deadlock may occur as the peer might be waiting for the data before being able to continue.

SEE ALSO

SSL_set_bio(3), ssl(3), bio(3), BIO_should_retry(3), BIO_read(3)

Name

BIO_s_connect, BIO_set_conn_hostname, BIO_set_conn_port, BIO_set_conn_ip, BIO_set_conn_int_port, BIO_get_conn_hostname, BIO_get_conn_port, BIO_get_conn_ip, BIO_get_conn_int_port, BIO_set_nbio and BIO_do_connect — connect BIO

Synopsis

```
#include  <openssl/bio.h>
```

```
BIO_METHOD * BIO_s_connect(void);
```

```
BIO *BIO_new_connect(char *name);
```

```
long BIO_set_conn_hostname(BIO *b, char *name);
long BIO_set_conn_port(BIO *b, char *port);
long BIO_set_conn_ip(BIO *b, char *ip);
long BIO_set_conn_int_port(BIO *b, char *port);
char *BIO_get_conn_hostname(BIO *b);
char *BIO_get_conn_port(BIO *b);
char *BIO_get_conn_ip(BIO *b);
long BIO_get_conn_int_port(BIO *b);
```

```
long BIO_set_nbio(BIO *b, long n);
```

```
int BIO_do_connect(BIO *b);
```

DESCRIPTION

BIO_s_connect() returns the connect BIO method. This is a wrapper round the platform's TCP/IP socket connection routines.

Using connect BIOs, TCP/IP connections can be made and data transferred using only BIO routines. In this way any platform specific operations are hidden by the BIO abstraction.

Read and write operations on a connect BIO will perform I/O on the underlying connection. If no connection is established and the port and hostname (see below) is set up properly then a connection is established first.

Connect BIOs support BIO_puts() but not BIO_gets().

If the close flag is set on a connect BIO then any active connection is shutdown and the socket closed when the BIO is freed.

Calling BIO_reset() on a connect BIO will close any active connection and reset the BIO into a state where it can connect to the same host again.

BIO_get_fd() places the underlying socket in **c** if it is not NULL, it also returns the socket . If **c** is not NULL it should be of type (int *).

BIO_set_conn_hostname() uses the string **name** to set the hostname. The hostname can be an IP address. The hostname can also include the port in the form hostname:port . It is also acceptable to use the form "hostname/any/other/path" or "hostname:port/any/other/path".

BIO_set_conn_port() sets the port to **port**. **port** can be the numerical form or a string such as "http". A string will be looked up first using getservbyname() on the host platform but if that fails a standard table of port names will be used. Currently the list is http, telnet, socks, https, ssl, ftp, gopher and wais.

BIO_set_conn_ip() sets the IP address to **ip** using binary form, that is four bytes specifying the IP address in big-endian form.

BIO_set_conn_int_port() sets the port using **port**. **port** should be of type (int *).

BIO_get_conn_hostname() returns the hostname of the connect BIO or NULL if the BIO is initialized but no hostname is set. This return value is an internal pointer which should not be modified.

BIO_get_conn_port() returns the port as a string.

BIO_get_conn_ip() returns the IP address in binary form.

BIO_get_conn_int_port() returns the port as an int.

BIO_set_nbio() sets the non blocking I/O flag to **n**. If **n** is zero then blocking I/O is set. If **n** is 1 then non blocking I/O is set. Blocking I/O is the default. The call to BIO_set_nbio() should be made before the connection is established because non blocking I/O is set during the connect process.

BIO_new_connect() combines BIO_new() and BIO_set_conn_hostname() into a single call: that is it creates a new connect BIO with **name**.

BIO_do_connect() attempts to connect the supplied BIO. It returns 1 if the connection was established successfully. A zero or negative value is returned if the connection could not be established, the call BIO_should_retry() should be used for non blocking connect BIOs to determine if the call should be retried.

NOTES

If blocking I/O is set then a non positive return value from any I/O call is caused by an error condition, although a zero return will normally mean that the connection was closed.

If the port name is supplied as part of the host name then this will override any value set with BIO_set_conn_port(). This may be undesirable if the application does not wish to allow connection to arbitrary ports. This can be avoided by checking for the presence of the ':' character in the passed hostname and either indicating an error or truncating the string at that point.

The values returned by BIO_get_conn_hostname(), BIO_get_conn_port(), BIO_get_conn_ip() and BIO_get_conn_int_port() are updated when a connection attempt is made. Before any connection attempt the values returned are those set by the application itself.

Applications do not have to call BIO_do_connect() but may wish to do so to separate the connection process from other I/O processing.

If non blocking I/O is set then retries will be requested as appropriate.

It addition to BIO_should_read() and BIO_should_write() it is also possible for BIO_should_io_special() to be true during the initial connection process with the reason BIO_RR_CONNECT. If this is returned then this is an indication that a connection attempt would block, the application should then take appropriate action to wait until the underlying socket has connected and retry the call.

BIO_set_conn_hostname(), BIO_set_conn_port(), BIO_set_conn_ip(), BIO_set_conn_int_port(), BIO_get_conn_hostname(), BIO_get_conn_port(), BIO_get_conn_ip(), BIO_get_conn_int_port(), BIO_set_nbio() and BIO_do_connect() are macros.

RETURN VALUES

BIO_s_connect() returns the connect BIO method.

BIO_get_fd() returns the socket or -1 if the BIO has not been initialized.

BIO_set_conn_hostname(), BIO_set_conn_port(), BIO_set_conn_ip() and BIO_set_conn_int_port() always return 1.

BIO_get_conn_hostname() returns the connected hostname or NULL is none was set.

BIO_get_conn_port() returns a string representing the connected port or NULL if not set.

BIO_get_conn_ip() returns a pointer to the connected IP address in binary form or all zeros if not set.

BIO_get_conn_int_port() returns the connected port or 0 if none was set.

BIO_set_nbio() always returns 1.

BIO_do_connect() returns 1 if the connection was successfully established and 0 or -1 if the connection failed.

EXAMPLE

This is example connects to a webserver on the local host and attempts to retrieve a page and copy the result to standard output.

```
BIO *cbio, *out;
int len;
char tmpbuf[1024];
ERR_load_crypto_strings();
cbio = BIO_new_connect("localhost:http");
out = BIO_new_fp(stdout, BIO_NOCLOSE);
if(BIO_do_connect(cbio) <= 0) {
       fprintf(stderr, "Error connecting to server\n");
       ERR_print_errors_fp(stderr);
       /* whatever … */
       }
BIO_puts(cbio, "GET / HTTP/1.0\n\n");
for(;;) {
       len = BIO_read(cbio, tmpbuf, 1024);
       if(len <= 0) break;
       BIO_write(out, tmpbuf, len);
}
BIO_free(cbio);
BIO_free(out);
```

SEE ALSO

TBA

Name

BIO_set_callback, BIO_get_callback, BIO_set_callback_arg, BIO_get_callback_arg and BIO_debug_callback — BIO callback functions

Synopsis

```
#include  <openssl/bio.h>
```

```
#define BIO_set_callback(b,cb)          ((b)->callback=(cb))
#define BIO_get_callback(b)             ((b)->callback)
#define BIO_set_callback_arg(b,arg)     ((b)->cb_arg=(char *)(arg))
#define BIO_get_callback_arg(b)                 ((b)->cb_arg)
```

```
long BIO_debug_callback(BIO *bio,int cmd,const char *argp,int argi,
       long argl,long ret);
```

```
typedef long (*callback)(BIO *b, int oper, const char *argp,
                       int argi, long argl, long retvalue);
```

DESCRIPTION

BIO_set_callback() and BIO_get_callback() set and retrieve the BIO callback, they are both macros. The callback is called during most high level BIO operations. It can be used for debugging purposes to trace operations on a BIO or to modify its operation.

BIO_set_callback_arg() and BIO_get_callback_arg() are macros which can be used to set and retrieve an argument for use in the callback.

BIO_debug_callback() is a standard debugging callback which prints out information relating to each BIO operation. If the callback argument is set if is interpreted as a BIO to send the information to, otherwise stderr is used.

callback() is the callback function itself. The meaning of each argument is described below.

The BIO the callback is attached to is passed in **b**.

oper is set to the operation being performed. For some operations the callback is called twice, once before and once after the actual operation, the latter case has **oper** or'ed with BIO_CB_RETURN.

The meaning of the arguments **argp**, **argi** and **argl** depends on the value of **oper**, that is the operation being performed.

retvalue is the return value that would be returned to the application if no callback were present. The actual value returned is the return value of the callback itself. In the case of callbacks called before the actual BIO operation 1 is placed in retvalue, if the return value is not positive it will be immediately returned to the application and the BIO operation will not be performed.

The callback should normally simply return **retvalue** when it has finished processing, unless if specifically wishes to modify the value returned to the application.

CALLBACK OPERATIONS

BIO_free(b)

callback(b,BIO_CB_FREE,NULL,0L,0L,1L) is called before the free operation.

BIO_read(b, out, outl)

callback(b,BIO_CB_READ,out,outl,0L,1L) is called before the read.

callback(b,BIO_CB_READ|BIO_CB_RETURN,out outl,0L,retvalue) after the read.

BIO_write(b, in, inl)

callback(b,BIO_CB_WRITE,in,inl,0L,1L) is called before the write.

callback(b,BIO_CB_WRITE|BIO_CB_RETURN,in,inl,0L,retvalue) after the write.

BIO_gets(b, out, outl)

callback(b,BIO_CB_GETS,out,outl,0L,1L) is called before the operation.

callback(b,BIO_CB_GETS|BIO_CB_RETURN,out,outl,0L,retvalue) is called after the operation.

BIO_puts(b, in)

callback(b,BIO_CB_WRITE,in,0,0L 1L) is called before the operation.

callback(b, BIO_CB_WRITE|BIO_CB_RETURN, in, 0, 0L, retvalue) is called after the operation.

BIO_ctrl(BIO *b, int cmd, long larg, void *parg)

callback(b,BIO_CB_CTRL,parg,cmd,larg,1L) is called before the call.

callback(b,BIO_CB_CTRL|BIO_CB_RETURN,parg,cmd,larg,ret) is called after the call.

EXAMPLE

The BIO_debug_callback() function is a good example, its source is in crypto/bio/bio_cb.c

SEE ALSO

TBA

Name

BIO_s_fd, BIO_set_fd, BIO_get_fd and BIO_new_fd — file descriptor BIO

Synopsis

```
#include  <openssl/bio.h>
```

```
BIO_METHOD *   BIO_s_fd(void);
```

```
#define BIO_set_fd(b,fd,c)     BIO_int_ctrl(b,BIO_C_SET_FD,c,fd)
#define BIO_get_fd(b,c)        BIO_ctrl(b,BIO_C_GET_FD,0,(char *)c)
```

```
BIO *BIO_new_fd(int fd, int close_flag);
```

DESCRIPTION

BIO_s_fd() returns the file descriptor BIO method. This is a wrapper round the platforms file descriptor routines such as read() and write().

BIO_read() and BIO_write() read or write the underlying descriptor. BIO_puts() is supported but BIO_gets() is not.

If the close flag is set then then close() is called on the underlying file descriptor when the BIO is freed.

BIO_reset() attempts to change the file pointer to the start of file using lseek(fd, 0, 0).

BIO_seek() sets the file pointer to position **ofs** from start of file using lseek(fd, ofs, 0).

BIO_tell() returns the current file position by calling lseek(fd, 0, 1).

BIO_set_fd() sets the file descriptor of BIO **b** to **fd** and the close flag to **c**.

BIO_get_fd() places the file descriptor in **c** if it is not NULL, it also returns the file descriptor. If **c** is not NULL it should be of type (int *).

BIO_new_fd() returns a file descriptor BIO using **fd** and **close_flag**.

NOTES

The behaviour of BIO_read() and BIO_write() depends on the behavior of the platforms read() and write() calls on the descriptor. If the underlying file descriptor is in a non blocking mode then the BIO will behave in the manner described in the BIO_read(3) and BIO_should_retry(3) manual pages.

File descriptor BIOs should not be used for socket I/O. Use socket BIOs instead.

RETURN VALUES

BIO_s_fd() returns the file descriptor BIO method.

BIO_reset() returns zero for success and -1 if an error occurred. BIO_seek() and BIO_tell() return the current file position or -1 is an error occurred. These values reflect the underlying lseek() behaviour.

BIO_set_fd() always returns 1.

BIO_get_fd() returns the file descriptor or -1 if the BIO has not been initialized.

BIO_new_fd() returns the newly allocated BIO or NULL is an error occurred.

EXAMPLE

This is a file descriptor BIO version of "Hello World":

```
BIO *out;
out = BIO_new_fd(fileno(stdout), BIO_NOCLOSE);
BIO_printf(out, "Hello World\n");
BIO_free(out);
```

SEE ALSO

BIO_seek(3), BIO_tell(3), BIO_reset(3), BIO_read(3), BIO_write(3), BIO_puts(3), BIO_gets(3), BIO_printf(3), BIO_set_close(3), BIO_get_close(3)

Name

BIO_s_file, BIO_new_file, BIO_new_fp, BIO_set_fp, BIO_get_fp, BIO_read_filename, BIO_write_filename, BIO_append_filename and BIO_rw_filename — FILE bio

Synopsis

```
#include  <openssl/bio.h>
```

```
BIO_METHOD *   BIO_s_file(void);
BIO *BIO_new_file(const char *filename, const char *mode);
BIO *BIO_new_fp(FILE *stream, int flags);
```

```
BIO_set_fp(BIO *b,FILE *fp, int flags);
BIO_get_fp(BIO *b,FILE **fpp);
```

```
int BIO_read_filename(BIO *b, char *name)
int BIO_write_filename(BIO *b, char *name)
int BIO_append_filename(BIO *b, char *name)
int BIO_rw_filename(BIO *b, char *name)
```

DESCRIPTION

BIO_s_file() returns the BIO file method. As its name implies it is a wrapper round the stdio FILE structure and it is a source/sink BIO.

Calls to BIO_read() and BIO_write() read and write data to the underlying stream. BIO_gets() and BIO_puts() are supported on file BIOs.

BIO_flush() on a file BIO calls the fflush() function on the wrapped stream.

BIO_reset() attempts to change the file pointer to the start of file using fseek(stream, 0, 0).

BIO_seek() sets the file pointer to position **ofs** from start of file using fseek(stream, ofs, 0).

BIO_eof() calls feof().

Setting the BIO_CLOSE flag calls fclose() on the stream when the BIO is freed.

BIO_new_file() creates a new file BIO with mode **mode** the meaning of **mode** is the same as the stdio function fopen(). The BIO_CLOSE flag is set on the returned BIO.

BIO_new_fp() creates a file BIO wrapping **stream**. Flags can be: BIO_CLOSE, BIO_NOCLOSE (the close flag) BIO_FP_TEXT (sets the underlying stream to text mode, default is binary: this only has any effect under Win32).

BIO_set_fp() set the fp of a file BIO to **fp**. **flags** has the same meaning as in BIO_new_fp(), it is a macro.

BIO_get_fp() retrieves the fp of a file BIO, it is a macro.

BIO_seek() is a macro that sets the position pointer to **offset** bytes from the start of file.

BIO_tell() returns the value of the position pointer.

BIO_read_filename(), BIO_write_filename(), BIO_append_filename() and BIO_rw_filename() set the file BIO **b** to use file **name** for reading, writing, append or read write respectively.

NOTES

When wrapping stdout, stdin or stderr the underlying stream should not normally be closed so the BIO_NOCLOSE flag should be set.

Because the file BIO calls the underlying stdio functions any quirks in stdio behaviour will be mirrored by the corresponding BIO.

On Windows BIO_new_files reserves for the filename argument to be UTF-8 encoded. In other words if you have to make it work in multi- lingual environment, encode file names in UTF-8.

EXAMPLES

File BIO "hello world":

```
BIO *bio_out;
bio_out = BIO_new_fp(stdout, BIO_NOCLOSE);
BIO_printf(bio_out, "Hello World\n");
```

Alternative technique:

```
BIO *bio_out;
bio_out = BIO_new(BIO_s_file());
if(bio_out == NULL) /* Error … */
if(!BIO_set_fp(bio_out, stdout, BIO_NOCLOSE)) /* Error … */
BIO_printf(bio_out, "Hello World\n");
```

Write to a file:

```
BIO *out;
out = BIO_new_file("filename.txt", "w");
if(!out) /* Error occurred */
BIO_printf(out, "Hello World\n");
BIO_free(out);
```

Alternative technique:

```
BIO *out;
out = BIO_new(BIO_s_file());
if(out == NULL) /* Error … */
if(!BIO_write_filename(out, "filename.txt")) /* Error … */
BIO_printf(out, "Hello World\n");
BIO_free(out);
```

RETURN VALUES

BIO_s_file() returns the file BIO method.

BIO_new_file() and BIO_new_fp() return a file BIO or NULL if an error occurred.

BIO_set_fp() and BIO_get_fp() return 1 for success or 0 for failure (although the current implementation never return 0).

BIO_seek() returns the same value as the underlying fseek() function: 0 for success or -1 for failure.

BIO_tell() returns the current file position.

BIO_read_filename(), BIO_write_filename(), BIO_append_filename() and BIO_rw_filename() return 1 for success or 0 for failure.

BUGS

BIO_reset() and BIO_seek() are implemented using fseek() on the underlying stream. The return value for fseek() is 0 for success or -1 if an error occurred this differs from other types of BIO which will typically return 1 for success and a non positive value if an error occurred.

SEE ALSO

BIO_seek(3), BIO_tell(3), BIO_reset(3), BIO_flush(3), BIO_read(3), BIO_write(3), BIO_puts(3), BIO_gets(3), BIO_printf(3), BIO_set_close(3), BIO_get_close(3)

Name

BIO_should_retry, BIO_should_read, BIO_should_write, BIO_should_io_special, BIO_retry_type, BIO_should_retry, BIO_get_retry_BIO and BIO_get_retry_reason — BIO retry functions

Synopsis

```
#include  <openssl/bio.h>
```

```
#define BIO_should_read(a)             ((a)->flags & BIO_FLAGS_READ)
#define BIO_should_write(a)            ((a)->flags & BIO_FLAGS_WRITE)
#define BIO_should_io_special(a)       ((a)->flags & BIO_FLAGS_IO_SPECIAL)
#define BIO_retry_type(a)              ((a)->flags & BIO_FLAGS_RWS)
#define BIO_should_retry(a)            ((a)->flags & BIO_FLAGS_SHOULD_RETRY)
```

```
#define BIO_FLAGS_READ         0x01
#define BIO_FLAGS_WRITE        0x02
#define BIO_FLAGS_IO_SPECIAL   0x04
#define BIO_FLAGS_RWS (BIO_FLAGS_READ|BIO_FLAGS_WRITE|BIO_FLAGS_IO_SPECIAL)
#define BIO_FLAGS_SHOULD_RETRY 0x08
```

```
BIO *  BIO_get_retry_BIO(BIO *bio, int *reason);
int    BIO_get_retry_reason(BIO *bio);
```

DESCRIPTION

These functions determine why a BIO is not able to read or write data. They will typically be called after a failed BIO_read() or BIO_write() call.

BIO_should_retry() is true if the call that produced this condition should then be retried at a later time.

If BIO_should_retry() is false then the cause is an error condition.

BIO_should_read() is true if the cause of the condition is that a BIO needs to read data.

BIO_should_write() is true if the cause of the condition is that a BIO needs to read data.

BIO_should_io_special() is true if some "special" condition, that is a reason other than reading or writing is the cause of the condition.

BIO_retry_type() returns a mask of the cause of a retry condition consisting of the values **BIO_FLAGS_READ**, **BIO_FLAGS_WRITE**, **BIO_FLAGS_IO_SPECIAL** though current BIO types will only set one of these.

BIO_get_retry_BIO() determines the precise reason for the special condition, it returns the BIO that caused this condition and if **reason** is not NULL it contains the reason code. The meaning of the reason code and the action that should be taken depends on the type of BIO that resulted in this condition.

BIO_get_retry_reason() returns the reason for a special condition if passed the relevant BIO, for example as returned by BIO_get_retry_BIO().

NOTES

If BIO_should_retry() returns false then the precise "error condition" depends on the BIO type that caused it and the return code of the BIO operation. For example if a call to BIO_read() on a socket BIO returns 0 and BIO_should_retry() is false then the cause will be that the connection closed. A similar condition on a file BIO will mean that it has reached EOF. Some BIO types may place additional information on the error queue. For more details see the individual BIO type manual pages.

If the underlying I/O structure is in a blocking mode almost all current BIO types will not request a retry, because the underlying I/O calls will not. If the application knows that the BIO type will never signal a retry then it need not call BIO_should_retry() after a failed BIO I/O call. This is typically done with file BIOs.

SSL BIOs are the only current exception to this rule: they can request a retry even if the underlying I/O structure is blocking, if a handshake occurs during a call to BIO_read(). An application can retry the failed call immediately or avoid this situation by setting SSL_MODE_AUTO_RETRY on the underlying SSL structure.

While an application may retry a failed non blocking call immediately this is likely to be very inefficient because the call will fail repeatedly until data can be processed or is available. An application will normally wait until the necessary condition is satisfied. How this is done depends on the underlying I/O structure.

For example if the cause is ultimately a socket and BIO_should_read() is true then a call to select() may be made to wait until data is available and then retry the BIO operation. By combining the retry conditions of several non blocking BIOs in a single select() call it is possible to service several BIOs in a single thread, though the performance may be poor if SSL BIOs are present because long delays can occur during the initial handshake process.

It is possible for a BIO to block indefinitely if the underlying I/O structure cannot process or return any data. This depends on the behaviour of the platforms I/O functions. This is often not desirable: one solution is to use non blocking I/O and use a timeout on the select() (or equivalent) call.

BUGS

The OpenSSL ASN1 functions cannot gracefully deal with non blocking I/O: that is they cannot retry after a partial read or write. This is usually worked around by only passing the relevant data to ASN1 functions when the entire structure can be read or written.

SEE ALSO

TBA

Name

BIO_s_mem, BIO_set_mem_eof_return, BIO_get_mem_data, BIO_set_mem_buf, BIO_get_mem_ptr and BIO_new_mem_buf — memory BIO

Synopsis

```
#include  <openssl/bio.h>
```

```
BIO_METHOD *   BIO_s_mem(void);
```

```
BIO_set_mem_eof_return(BIO *b,int v)
long BIO_get_mem_data(BIO *b, char **pp)
BIO_set_mem_buf(BIO *b,BUF_MEM *bm,int c)
BIO_get_mem_ptr(BIO *b,BUF_MEM **pp)
```

```
BIO *BIO_new_mem_buf(void *buf, int len);
```

DESCRIPTION

BIO_s_mem() return the memory BIO method function.

A memory BIO is a source/sink BIO which uses memory for its I/O. Data written to a memory BIO is stored in a BUF_MEM structure which is extended as appropriate to accommodate the stored data.

Any data written to a memory BIO can be recalled by reading from it. Unless the memory BIO is read only any data read from it is deleted from the BIO.

Memory BIOs support BIO_gets() and BIO_puts().

If the BIO_CLOSE flag is set when a memory BIO is freed then the underlying BUF_MEM structure is also freed.

Calling BIO_reset() on a read write memory BIO clears any data in it. On a read only BIO it restores the BIO to its original state and the read only data can be read again.

BIO_eof() is true if no data is in the BIO.

BIO_ctrl_pending() returns the number of bytes currently stored.

BIO_set_mem_eof_return() sets the behaviour of memory BIO **b** when it is empty. If the **v** is zero then an empty memory BIO will return EOF (that is it will return zero and BIO_should_retry(b) will be false. If **v** is non zero then it will return **v** when it is empty and it will set the read retry flag (that is BIO_read_retry(b) is true). To avoid ambiguity with a normal positive return value **v** should be set to a negative value, typically -1.

BIO_get_mem_data() sets **pp** to a pointer to the start of the memory BIOs data and returns the total amount of data available. It is implemented as a macro.

BIO_set_mem_buf() sets the internal BUF_MEM structure to **bm** and sets the close flag to **c**, that is **c** should be either BIO_CLOSE or BIO_NOCLOSE. It is a macro.

BIO_get_mem_ptr() places the underlying BUF_MEM structure in **pp**. It is a macro.

BIO_new_mem_buf() creates a memory BIO using **len** bytes of data at **buf**, if **len** is -1 then the **buf** is assumed to be null terminated and its length is determined by **strlen**. The BIO is set to a read only state and as a result cannot be written to. This is useful when some data needs to be made available from a static area of memory in the form of a BIO. The supplied data is read directly from the supplied buffer: it is **not** copied first, so the supplied area of memory must be unchanged until the BIO is freed.

NOTES

Writes to memory BIOs will always succeed if memory is available: that is their size can grow indefinitely.

Every read from a read write memory BIO will remove the data just read with an internal copy operation, if a BIO contains a lot of data and it is read in small chunks the operation can be very slow. The use of a read only memory BIO avoids this problem. If the BIO must be read write then adding a buffering BIO to the chain will speed up the process.

BUGS

There should be an option to set the maximum size of a memory BIO.

There should be a way to "rewind" a read write BIO without destroying its contents.

The copying operation should not occur after every small read of a large BIO to improve efficiency.

EXAMPLE

Create a memory BIO and write some data to it:

```
BIO *mem = BIO_new(BIO_s_mem());
BIO_puts(mem, "Hello World\n");
```

Create a read only memory BIO:

```
char data[] = "Hello World";
BIO *mem;
mem = BIO_new_mem_buf(data, -1);
```

Extract the BUF_MEM structure from a memory BIO and then free up the BIO:

```
BUF_MEM *bptr;
BIO_get_mem_ptr(mem, &bptr);
BIO_set_close(mem, BIO_NOCLOSE); /* So BIO_free() leaves BUF_MEM alone */
BIO_free(mem);
```

SEE ALSO

TBA

Name

BIO_s_null — null data sink

Synopsis

```
#include  <openssl/bio.h>
```

```
BIO_METHOD *   BIO_s_null(void);
```

DESCRIPTION

BIO_s_null() returns the null sink BIO method. Data written to the null sink is discarded, reads return EOF.

NOTES

A null sink BIO behaves in a similar manner to the Unix /dev/null device.

A null bio can be placed on the end of a chain to discard any data passed through it.

A null sink is useful if, for example, an application wishes to digest some data by writing through a digest bio but not send the digested data anywhere. Since a BIO chain must normally include a source/sink BIO this can be achieved by adding a null sink BIO to the end of the chain

RETURN VALUES

BIO_s_null() returns the null sink BIO method.

SEE ALSO

TBA

Name

BIO_s_socket and BIO_new_socket — socket BIO

Synopsis

```
#include  <openssl/bio.h>
```

```
BIO_METHOD *BIO_s_socket(void);
```

```
long BIO_set_fd(BIO *b, int fd, long close_flag);
long BIO_get_fd(BIO *b, int *c);
```

```
BIO *BIO_new_socket(int sock, int close_flag);
```

DESCRIPTION

BIO_s_socket() returns the socket BIO method. This is a wrapper round the platform's socket routines.

BIO_read() and BIO_write() read or write the underlying socket. BIO_puts() is supported but BIO_gets() is not.

If the close flag is set then the socket is shut down and closed when the BIO is freed.

BIO_set_fd() sets the socket of BIO **b** to **fd** and the close flag to **close_flag**.

BIO_get_fd() places the socket in **c** if it is not NULL, it also returns the socket. If **c** is not NULL it should be of type (int *).

BIO_new_socket() returns a socket BIO using **sock** and **close_flag**.

NOTES

Socket BIOs also support any relevant functionality of file descriptor BIOs.

The reason for having separate file descriptor and socket BIOs is that on some platforms sockets are not file descriptors and use distinct I/O routines, Windows is one such platform. Any code mixing the two will not work on all platforms.

BIO_set_fd() and BIO_get_fd() are macros.

RETURN VALUES

BIO_s_socket() returns the socket BIO method.

BIO_set_fd() always returns 1.

BIO_get_fd() returns the socket or -1 if the BIO has not been initialized.

BIO_new_socket() returns the newly allocated BIO or NULL is an error occurred.

SEE ALSO

TBA

Name

blowfish, BF_set_key, BF_encrypt, BF_decrypt, BF_ecb_encrypt, BF_cbc_encrypt, BF_cfb64_encrypt, BF_ofb64_encrypt and BF_options — Blowfish encryption

Synopsis

```
#include  <openssl/blowfish.h>
```

```
void BF_set_key(BF_KEY *key, int len, const unsigned char *data);
```

```
void BF_ecb_encrypt(const unsigned char *in, unsigned char *out,
        BF_KEY *key, int enc);
void BF_cbc_encrypt(const unsigned char *in, unsigned char *out,
        long length, BF_KEY *schedule, unsigned char *ivec, int enc);
void BF_cfb64_encrypt(const unsigned char *in, unsigned char *out,
        long length, BF_KEY *schedule, unsigned char *ivec, int *num,
        int enc);
void BF_ofb64_encrypt(const unsigned char *in, unsigned char *out,
        long length, BF_KEY *schedule, unsigned char *ivec, int *num);
const char *BF_options(void);
```

```
void BF_encrypt(BF_LONG *data,const BF_KEY *key);
void BF_decrypt(BF_LONG *data,const BF_KEY *key);
```

DESCRIPTION

This library implements the Blowfish cipher, which was invented and described by Counterpane (see http://www.counterpane.com/blowfish.html).

Blowfish is a block cipher that operates on 64 bit (8 byte) blocks of data. It uses a variable size key, but typically, 128 bit (16 byte) keys are considered good for strong encryption. Blowfish can be used in the same modes as DES (see des_modes(7)). Blowfish is currently one of the faster block ciphers. It is quite a bit faster than DES, and much faster than IDEA or RC2.

Blowfish consists of a key setup phase and the actual encryption or decryption phase.

BF_set_key() sets up the **BF_KEY key** using the **len** bytes long key at **data**.

BF_ecb_encrypt() is the basic Blowfish encryption and decryption function. It encrypts or decrypts the first 64 bits of **in** using the key **key**, putting the result in **out**. **enc** decides if encryption (**BF_ENCRYPT**) or decryption (**BF_DECRYPT**) shall be performed. The vector pointed at by **in** and **out** must be 64 bits in length, no less. If they are larger, everything after the first 64 bits is ignored.

The mode functions BF_cbc_encrypt(), BF_cfb64_encrypt() and BF_ofb64_encrypt() all operate on variable length data. They all take an initialization vector **ivec** which needs to be passed along into the next call of the same function for the same message. **ivec** may be initialized with anything, but the recipient needs to know what it was initialized with, or it won't be able to decrypt. Some programs and protocols simplify this, like SSH, where **ivec** is simply initialized to zero. BF_cbc_encrypt() operates on data that is a multiple of 8 bytes long, while BF_cfb64_encrypt() and BF_ofb64_encrypt() are used to encrypt an variable number of bytes (the amount does not have to be an exact multiple of 8). The purpose of the latter two is to simulate stream ciphers, and therefore, they need the parameter **num**, which is a pointer to an integer where the current offset in **ivec** is stored between calls. This integer must be initialized to zero when **ivec** is initialized.

BF_cbc_encrypt() is the Cipher Block Chaining function for Blowfish. It encrypts or decrypts the 64 bits chunks of **in** using the key **schedule**, putting the result in **out**. **enc** decides if encryption (BF_ENCRYPT) or decryption (BF_DECRYPT) shall be performed. **ivec** must point at an 8 byte long initialization vector.

BF_cfb64_encrypt() is the CFB mode for Blowfish with 64 bit feedback. It encrypts or decrypts the bytes in **in** using the key **schedule**, putting the result in **out**. **enc** decides if encryption (**BF_ENCRYPT**) or decryption (**BF_DECRYPT**) shall be performed. **ivec** must point at an 8 byte long initialization vector. **num** must point at an integer which must be initially zero.

BF_ofb64_encrypt() is the OFB mode for Blowfish with 64 bit feedback. It uses the same parameters as BF_cfb64_encrypt(), which must be initialized the same way.

BF_encrypt() and BF_decrypt() are the lowest level functions for Blowfish encryption. They encrypt/decrypt the first 64 bits of the vector pointed by **data**, using the key **key**. These functions should not be used unless you implement 'modes' of Blowfish. The alternative is to use BF_ecb_encrypt(). If you still want to use these functions, you should be aware that they take each 32-bit chunk in host-byte order, which is little-endian on little-endian platforms and big-endian on big-endian ones.

RETURN VALUES

None of the functions presented here return any value.

NOTE

Applications should use the higher level functions EVP_EncryptInit(3) etc. instead of calling the blowfish functions directly.

SEE ALSO

des_modes(7)

HISTORY

The Blowfish functions are available in all versions of SSLeay and OpenSSL.

Name

bn — multiprecision integer arithmetics

Synopsis

```
#include  <openssl/bn.h>
```

```
BIGNUM *BN_new(void);
void BN_free(BIGNUM *a);
void BN_init(BIGNUM *);
void BN_clear(BIGNUM *a);
void BN_clear_free(BIGNUM *a);
```

```
BN_CTX *BN_CTX_new(void);
void BN_CTX_init(BN_CTX *c);
void BN_CTX_free(BN_CTX *c);
```

```
BIGNUM *BN_copy(BIGNUM *a, const BIGNUM *b);
BIGNUM *BN_dup(const BIGNUM *a);
```

```
BIGNUM *BN_swap(BIGNUM *a, BIGNUM *b);
```

```
int BN_num_bytes(const BIGNUM *a);
int BN_num_bits(const BIGNUM *a);
int BN_num_bits_word(BN_ULONG w);
```

```
void BN_set_negative(BIGNUM *a, int n);
int  BN_is_negative(const BIGNUM *a);
```

```
int BN_add(BIGNUM *r, const BIGNUM *a, const BIGNUM *b);
int BN_sub(BIGNUM *r, const BIGNUM *a, const BIGNUM *b);
int BN_mul(BIGNUM *r, BIGNUM *a, BIGNUM *b, BN_CTX *ctx);
int BN_sqr(BIGNUM *r, BIGNUM *a, BN_CTX *ctx);
int BN_div(BIGNUM *dv, BIGNUM *rem, const BIGNUM *a, const BIGNUM *d,
        BN_CTX *ctx);
int BN_mod(BIGNUM *rem, const BIGNUM *a, const BIGNUM *m, BN_CTX *ctx);
int BN_nnmod(BIGNUM *rem, const BIGNUM *a, const BIGNUM *m, BN_CTX *ctx);
int BN_mod_add(BIGNUM *ret, BIGNUM *a, BIGNUM *b, const BIGNUM *m,
        BN_CTX *ctx);
int BN_mod_sub(BIGNUM *ret, BIGNUM *a, BIGNUM *b, const BIGNUM *m,
        BN_CTX *ctx);
int BN_mod_mul(BIGNUM *ret, BIGNUM *a, BIGNUM *b, const BIGNUM *m,
        BN_CTX *ctx);
int BN_mod_sqr(BIGNUM *ret, BIGNUM *a, const BIGNUM *m, BN_CTX *ctx);
int BN_exp(BIGNUM *r, BIGNUM *a, BIGNUM *p, BN_CTX *ctx);
int BN_mod_exp(BIGNUM *r, BIGNUM *a, const BIGNUM *p,
        const BIGNUM *m, BN_CTX *ctx);
int BN_gcd(BIGNUM *r, BIGNUM *a, BIGNUM *b, BN_CTX *ctx);
```

```
int BN_add_word(BIGNUM *a, BN_ULONG w);
int BN_sub_word(BIGNUM *a, BN_ULONG w);
int BN_mul_word(BIGNUM *a, BN_ULONG w);
BN_ULONG BN_div_word(BIGNUM *a, BN_ULONG w);
BN_ULONG BN_mod_word(const BIGNUM *a, BN_ULONG w);
```

```
int BN_cmp(BIGNUM *a, BIGNUM *b);
int BN_ucmp(BIGNUM *a, BIGNUM *b);
int BN_is_zero(BIGNUM *a);
int BN_is_one(BIGNUM *a);
int BN_is_word(BIGNUM *a, BN_ULONG w);
int BN_is_odd(BIGNUM *a);
```

```
int BN_zero(BIGNUM *a);
int BN_one(BIGNUM *a);
const BIGNUM *BN_value_one(void);
int BN_set_word(BIGNUM *a, unsigned long w);
unsigned long BN_get_word(BIGNUM *a);
```

```
int BN_rand(BIGNUM *rnd, int bits, int top, int bottom);
int BN_pseudo_rand(BIGNUM *rnd, int bits, int top, int bottom);
int BN_rand_range(BIGNUM *rnd, BIGNUM *range);
int BN_pseudo_rand_range(BIGNUM *rnd, BIGNUM *range);
```

```
BIGNUM *BN_generate_prime(BIGNUM *ret, int bits,int safe, BIGNUM *add,
        BIGNUM *rem, void (*callback)(int, int, void *), void *cb_arg);
int BN_is_prime(const BIGNUM *p, int nchecks,
        void (*callback)(int, int, void *), BN_CTX *ctx, void *cb_arg);
```

```
int BN_set_bit(BIGNUM *a, int n);
int BN_clear_bit(BIGNUM *a, int n);
int BN_is_bit_set(const BIGNUM *a, int n);
int BN_mask_bits(BIGNUM *a, int n);
int BN_lshift(BIGNUM *r, const BIGNUM *a, int n);
int BN_lshift1(BIGNUM *r, BIGNUM *a);
int BN_rshift(BIGNUM *r, BIGNUM *a, int n);
int BN_rshift1(BIGNUM *r, BIGNUM *a);
```

```
int BN_bn2bin(const BIGNUM *a, unsigned char *to);
BIGNUM *BN_bin2bn(const unsigned char *s, int len, BIGNUM *ret);
char *BN_bn2hex(const BIGNUM *a);
char *BN_bn2dec(const BIGNUM *a);
int BN_hex2bn(BIGNUM **a, const char *str);
int BN_dec2bn(BIGNUM **a, const char *str);
int BN_print(BIO *fp, const BIGNUM *a);
int BN_print_fp(FILE *fp, const BIGNUM *a);
int BN_bn2mpi(const BIGNUM *a, unsigned char *to);
BIGNUM *BN_mpi2bn(unsigned char *s, int len, BIGNUM *ret);
```

```
BIGNUM *BN_mod_inverse(BIGNUM *r, BIGNUM *a, const BIGNUM *n,
    BN_CTX *ctx);
```

```
BN_RECP_CTX *BN_RECP_CTX_new(void);
void BN_RECP_CTX_init(BN_RECP_CTX *recp);
void BN_RECP_CTX_free(BN_RECP_CTX *recp);
int BN_RECP_CTX_set(BN_RECP_CTX *recp, const BIGNUM *m, BN_CTX *ctx);
int BN_mod_mul_reciprocal(BIGNUM *r, BIGNUM *a, BIGNUM *b,
       BN_RECP_CTX *recp, BN_CTX *ctx);
```

```
BN_MONT_CTX *BN_MONT_CTX_new(void);
void BN_MONT_CTX_init(BN_MONT_CTX *ctx);
void BN_MONT_CTX_free(BN_MONT_CTX *mont);
int BN_MONT_CTX_set(BN_MONT_CTX *mont, const BIGNUM *m, BN_CTX *ctx);
BN_MONT_CTX *BN_MONT_CTX_copy(BN_MONT_CTX *to, BN_MONT_CTX *from);
int BN_mod_mul_montgomery(BIGNUM *r, BIGNUM *a, BIGNUM *b,
        BN_MONT_CTX *mont, BN_CTX *ctx);
int BN_from_montgomery(BIGNUM *r, BIGNUM *a, BN_MONT_CTX *mont,
        BN_CTX *ctx);
int BN_to_montgomery(BIGNUM *r, BIGNUM *a, BN_MONT_CTX *mont,
        BN_CTX *ctx);
```

```
BN_BLINDING *BN_BLINDING_new(const BIGNUM *A, const BIGNUM *Ai,
       BIGNUM *mod);
void BN_BLINDING_free(BN_BLINDING *b);
int BN_BLINDING_update(BN_BLINDING *b,BN_CTX *ctx);
int BN_BLINDING_convert(BIGNUM *n, BN_BLINDING *b, BN_CTX *ctx);
int BN_BLINDING_invert(BIGNUM *n, BN_BLINDING *b, BN_CTX *ctx);
int BN_BLINDING_convert_ex(BIGNUM *n, BIGNUM *r, BN_BLINDING *b,
       BN_CTX *ctx);
int BN_BLINDING_invert_ex(BIGNUM *n,const BIGNUM *r,BN_BLINDING *b,
       BN_CTX *ctx);
unsigned long BN_BLINDING_get_thread_id(const BN_BLINDING *);
void BN_BLINDING_set_thread_id(BN_BLINDING *, unsigned long);
unsigned long BN_BLINDING_get_flags(const BN_BLINDING *);
void BN_BLINDING_set_flags(BN_BLINDING *, unsigned long);
BN_BLINDING *BN_BLINDING_create_param(BN_BLINDING *b,
       const BIGNUM *e, BIGNUM *m, BN_CTX *ctx,
       int (*bn_mod_exp)(BIGNUM *r, const BIGNUM *a, const BIGNUM *p,
                         const BIGNUM *m, BN_CTX *ctx, BN_MONT_CTX *m_ctx),
       BN_MONT_CTX *m_ctx);
```

DESCRIPTION

This library performs arithmetic operations on integers of arbitrary size. It was written for use in public key cryptography, such as RSA and Diffie-Hellman.

It uses dynamic memory allocation for storing its data structures. That means that there is no limit on the size of the numbers manipulated by these functions, but return values must always be checked in case a memory allocation error has occurred.

The basic object in this library is a **BIGNUM**. It is used to hold a single large integer. This type should be considered opaque and fields should not be modified or accessed directly.

The creation of **BIGNUM** objects is described in BN_new(3); BN_add(3) describes most of the arithmetic operations. Comparison is described in BN_cmp(3); BN_zero(3) describes certain assignments, BN_rand(3) the generation of random numbers, BN_generate_prime(3) deals with prime numbers and BN_set_bit(3) with bit operations. The conversion of **BIGNUM**s to external formats is described in BN_bn2bin(3).

SEE ALSO

bn_internal(3), dh(3), err(3), rand(3), rsa(3), BN_new(3), BN_CTX_new(3), BN_copy(3), BN_swap(3), BN_num_bytes(3), BN_add(3), BN_add_word(3), BN_cmp(3), BN_zero(3), BN_rand(3), BN_generate_prime(3), BN_set_bit(3), BN_bn2bin(3), BN_mod_inverse(3), BN_mod_mul_reciprocal(3), BN_mod_mul_montgomery(3), BN_BLINDING_new(3)

Name

bn_mul_words, bn_mul_add_words, bn_sqr_words, bn_div_words, bn_add_words, bn_sub_words, bn_mul_comba4, bn_mul_comba8, bn_sqr_comba4, bn_sqr_comba8, bn_cmp_words, bn_mul_normal, bn_mul_low_normal, bn_mul_recursive, bn_mul_part_recursive, bn_mul_low_recursive, bn_mul_high, bn_sqr_normal, bn_sqr_recursive, bn_expand, bn_wexpand, bn_expand2, bn_fix_top, bn_check_top, bn_print, bn_dump, bn_set_max, bn_set_high and bn_set_low — BIGNUM library internal functions

Synopsis

```
#include  <openssl/bn.h>
```

```
BN_ULONG bn_mul_words(BN_ULONG *rp, BN_ULONG *ap, int num, BN_ULONG w);
BN_ULONG bn_mul_add_words(BN_ULONG *rp, BN_ULONG *ap, int num,
  BN_ULONG w);
void     bn_sqr_words(BN_ULONG *rp, BN_ULONG *ap, int num);
BN_ULONG bn_div_words(BN_ULONG h, BN_ULONG l, BN_ULONG d);
BN_ULONG bn_add_words(BN_ULONG *rp, BN_ULONG *ap, BN_ULONG *bp,
  int num);
BN_ULONG bn_sub_words(BN_ULONG *rp, BN_ULONG *ap, BN_ULONG *bp,
  int num);
```

```
void bn_mul_comba4(BN_ULONG *r, BN_ULONG *a, BN_ULONG *b);
void bn_mul_comba8(BN_ULONG *r, BN_ULONG *a, BN_ULONG *b);
void bn_sqr_comba4(BN_ULONG *r, BN_ULONG *a);
void bn_sqr_comba8(BN_ULONG *r, BN_ULONG *a);
```

```
int bn_cmp_words(BN_ULONG *a, BN_ULONG *b, int n);
```

```
void bn_mul_normal(BN_ULONG *r, BN_ULONG *a, int na, BN_ULONG *b,
  int nb);
void bn_mul_low_normal(BN_ULONG *r, BN_ULONG *a, BN_ULONG *b, int n);
void bn_mul_recursive(BN_ULONG *r, BN_ULONG *a, BN_ULONG *b, int n2,
  int dna,int dnb,BN_ULONG *tmp);
void bn_mul_part_recursive(BN_ULONG *r, BN_ULONG *a, BN_ULONG *b,
  int n, int tna,int tnb, BN_ULONG *tmp);
void bn_mul_low_recursive(BN_ULONG *r, BN_ULONG *a, BN_ULONG *b,
  int n2, BN_ULONG *tmp);
void bn_mul_high(BN_ULONG *r, BN_ULONG *a, BN_ULONG *b, BN_ULONG *l,
  int n2, BN_ULONG *tmp);
```

```
void bn_sqr_normal(BN_ULONG *r, BN_ULONG *a, int n, BN_ULONG *tmp);
void bn_sqr_recursive(BN_ULONG *r, BN_ULONG *a, int n2, BN_ULONG *tmp);
```

```
void mul(BN_ULONG r, BN_ULONG a, BN_ULONG w, BN_ULONG c);
void mul_add(BN_ULONG r, BN_ULONG a, BN_ULONG w, BN_ULONG c);
void sqr(BN_ULONG r0, BN_ULONG r1, BN_ULONG a);
```

```
BIGNUM *bn_expand(BIGNUM *a, int bits);
BIGNUM *bn_wexpand(BIGNUM *a, int n);
BIGNUM *bn_expand2(BIGNUM *a, int n);
void bn_fix_top(BIGNUM *a);
```

```
void bn_check_top(BIGNUM *a);
void bn_print(BIGNUM *a);
void bn_dump(BN_ULONG *d, int n);
void bn_set_max(BIGNUM *a);
void bn_set_high(BIGNUM *r, BIGNUM *a, int n);
void bn_set_low(BIGNUM *r, BIGNUM *a, int n);
```

DESCRIPTION

This page documents the internal functions used by the OpenSSL **BIGNUM** implementation. They are described here to facilitate debugging and extending the library. They are *not* to be used by applications.

The BIGNUM structure

```
typedef struct bignum_st BIGNUM;

struct bignum_st
       {
       BN_ULONG *d;    /* Pointer to an array of 'BN_BITS2' bit chunks. */
       int top;        /* Index of last used d +1. */
       /* The next are internal book keeping for bn_expand. */
       int dmax;       /* Size of the d array. */
       int neg;        /* one if the number is negative */
       int flags;
       };
```

The integer value is stored in **d**, a malloc()ed array of words (**BN_ULONG**), least significant word first. A **BN_ULONG** can be either 16, 32 or 64 bits in size, depending on the 'number of bits' (**BITS2**) specified in `openssl/bn.h`.

dmax is the size of the **d** array that has been allocated. **top** is the number of words being used, so for a value of 4, bn.d[0]=4 and bn.top=1. **neg** is 1 if the number is negative. When a **BIGNUM** is **0**, the **d** field can be **NULL** and **top** == **0**.

flags is a bit field of flags which are defined in `openssl/bn.h`. The flags begin with **BN_FLG_**. The macros BN_set_flags(b,n) and BN_get_flags(b,n) exist to enable or fetch flag(s) **n** from **BIGNUM** structure **b**.

Various routines in this library require the use of temporary **BIGNUM** variables during their execution. Since dynamic memory allocation to create **BIGNUM**s is rather expensive when used in conjunction with repeated subroutine calls, the **BN_CTX** structure is used. This structure contains **BN_CTX_NUM** **BIGNUM**s, see BN_CTX_start(3).

Low-level arithmetic operations

These functions are implemented in C and for several platforms in assembly language:

bn_mul_words(**rp**, **ap**, **num**, **w**) operates on the **num** word arrays **rp** and **ap**. It computes **ap** * **w**, places the result in **rp**, and returns the high word (carry).

bn_mul_add_words(**rp**, **ap**, **num**, **w**) operates on the **num** word arrays **rp** and **ap**. It computes **ap** * **w** + **rp**, places the result in **rp**, and returns the high word (carry).

bn_sqr_words(**rp**, **ap**, **n**) operates on the **num** word array **ap** and the 2***num** word array **ap**. It computes **ap** * **ap** word-wise, and places the low and high bytes of the result in **rp**.

bn_div_words(**h**, **l**, **d**) divides the two word number (**h**,**l**) by **d** and returns the result.

bn_add_words(**rp**, **ap**, **bp**, **num**) operates on the **num** word arrays **ap**, **bp** and **rp**. It computes **ap** + **bp**, places the result in **rp**, and returns the high word (carry).

bn_sub_words(**rp**, **ap**, **bp**, **num**) operates on the **num** word arrays **ap**, **bp** and **rp**. It computes **ap** - **bp**, places the result in **rp**, and returns the carry (1 if **bp** > **ap**, 0 otherwise).

bn_mul_comba4(**r**, **a**, **b**) operates on the 4 word arrays **a** and **b** and the 8 word array **r**. It computes **a***__b__ and places the result in **r**.

bn_mul_comba8(**r**, **a**, **b**) operates on the 8 word arrays **a** and **b** and the 16 word array **r**. It computes **a***__b__ and places the result in **r**.

bn_sqr_comba4(**r**, **a**, **b**) operates on the 4 word arrays **a** and **b** and the 8 word array **r**.

bn_sqr_comba8(**r**, **a**, **b**) operates on the 8 word arrays **a** and **b** and the 16 word array **r**.

The following functions are implemented in C:

bn_cmp_words(**a**, **b**, **n**) operates on the **n** word arrays **a** and **b**. It returns 1, 0 and -1 if **a** is greater than, equal and less than **b**.

bn_mul_normal(**r**, **a**, **na**, **b**, **nb**) operates on the **na** word array **a**, the **nb** word array **b** and the **na**+**nb** word array **r**. It computes **a*****b** and places the result in **r**.

bn_mul_low_normal(**r**, **a**, **b**, **n**) operates on the **n** word arrays **r**, **a** and **b**. It computes the **n** low words of **a*****b** and places the result in **r**.

bn_mul_recursive(**r**, **a**, **b**, **n2**, **dna**, **dnb**, **t**) operates on the word arrays **a** and **b** of length **n2**+**dna** and **n2**+**dnb** (**dna** and **dnb** are currently allowed to be 0 or negative) and the 2***n2** word arrays **r** and **t**. **n2** must be a power of 2. It computes **a*****b** and places the result in **r**.

bn_mul_part_recursive(**r**, **a**, **b**, **n**, **tna**, **tnb**, **tmp**) operates on the word arrays **a** and **b** of length **n**+**tna** and **n**+**tnb** and the 4***n** word arrays **r** and **tmp**.

bn_mul_low_recursive(**r**, **a**, **b**, **n2**, **tmp**) operates on the **n2** word arrays **r** and **tmp** and the **n2**/2 word arrays **a** and **b**.

bn_mul_high(**r**, **a**, **b**, **l**, **n2**, **tmp**) operates on the **n2** word arrays **r**, **a**, **b** and **l** (?) and the 3***n2** word array **tmp**.

BN_mul() calls bn_mul_normal(), or an optimized implementation if the factors have the same size: bn_mul_comba8() is used if they are 8 words long, bn_mul_recursive() if they are larger than **BN_MULL_SIZE_NORMAL** and the size is an exact multiple of the word size, and bn_mul_part_recursive() for others that are larger than **BN_MULL_SIZE_NORMAL**.

bn_sqr_normal(**r**, **a**, **n**, **tmp**) operates on the **n** word array **a** and the 2***n** word arrays **tmp** and **r**.

The implementations use the following macros which, depending on the architecture, may use "long long" C operations or inline assembler. They are defined in `bn_lcl.h`.

mul(**r**, **a**, **w**, **c**) computes **w*****a**+**c** and places the low word of the result in **r** and the high word in **c**.

mul_add(**r**, **a**, **w**, **c**) computes **w*****a**+**r**+**c** and places the low word of the result in **r** and the high word in **c**.

sqr(**r0**, **r1**, **a**) computes **a*****a** and places the low word of the result in **r0** and the high word in **r1**.

Size changes

bn_expand() ensures that **b** has enough space for a **bits** bit number. bn_wexpand() ensures that **b** has enough space for an **n** word number. If the number has to be expanded, both macros call bn_expand2(), which allocates a new **d** array and copies the data. They return **NULL** on error, **b** otherwise.

The bn_fix_top() macro reduces **a->top** to point to the most significant non-zero word plus one when **a** has shrunk.

Debugging

bn_check_top() verifies that `((a)-&gt;top &gt;= 0 && (a)-&gt;top &lt;= (a)-&gt;dmax)`. A violation will cause the program to abort.

bn_print() prints **a** to stderr. bn_dump() prints **n** words at **d** (in reverse order, i.e. most significant word first) to stderr.

bn_set_max() makes **a** a static number with a **dmax** of its current size. This is used by bn_set_low() and bn_set_high() to make **r** a read-only **BIGNUM** that contains the **n** low or high words of **a**.

If **BN_DEBUG** is not defined, bn_check_top(), bn_print(), bn_dump() and bn_set_max() are defined as empty macros.

SEE ALSO

bn(3)

Name

BN_add, BN_sub, BN_mul, BN_sqr, BN_div, BN_mod, BN_nnmod, BN_mod_add, BN_mod_sub, BN_mod_mul, BN_mod_sqr, BN_exp, BN_mod_exp and BN_gcd — arithmetic operations on BIGNUMs

Synopsis

```
#include  <openssl/bn.h>
```

```
int BN_add(BIGNUM *r, const BIGNUM *a, const BIGNUM *b);
```

```
int BN_sub(BIGNUM *r, const BIGNUM *a, const BIGNUM *b);
```

```
int BN_mul(BIGNUM *r, BIGNUM *a, BIGNUM *b, BN_CTX *ctx);
```

```
int BN_sqr(BIGNUM *r, BIGNUM *a, BN_CTX *ctx);
```

```
int BN_div(BIGNUM *dv, BIGNUM *rem, const BIGNUM *a, const BIGNUM *d,
        BN_CTX *ctx);
```

```
int BN_mod(BIGNUM *rem, const BIGNUM *a, const BIGNUM *m, BN_CTX *ctx);
```

```
int BN_nnmod(BIGNUM *r, const BIGNUM *a, const BIGNUM *m, BN_CTX *ctx);
```

```
int BN_mod_add(BIGNUM *r, BIGNUM *a, BIGNUM *b, const BIGNUM *m,
        BN_CTX *ctx);
```

```
int BN_mod_sub(BIGNUM *r, BIGNUM *a, BIGNUM *b, const BIGNUM *m,
        BN_CTX *ctx);
```

```
int BN_mod_mul(BIGNUM *r, BIGNUM *a, BIGNUM *b, const BIGNUM *m,
        BN_CTX *ctx);
```

```
int BN_mod_sqr(BIGNUM *r, BIGNUM *a, const BIGNUM *m, BN_CTX *ctx);
```

```
int BN_exp(BIGNUM *r, BIGNUM *a, BIGNUM *p, BN_CTX *ctx);
```

```
int BN_mod_exp(BIGNUM *r, BIGNUM *a, const BIGNUM *p,
        const BIGNUM *m, BN_CTX *ctx);
```

```
int BN_gcd(BIGNUM *r, BIGNUM *a, BIGNUM *b, BN_CTX *ctx);
```

DESCRIPTION

BN_add() adds *a* and *b* and places the result in *r* (`r=a+b`). *r* may be the same **BIGNUM** as *a* or *b*.

BN_sub() subtracts *b* from *a* and places the result in *r* (`r=a-b`).

BN_mul() multiplies *a* and *b* and places the result in *r* (`r=a*b`). *r* may be the same **BIGNUM** as *a* or *b*. For multiplication by powers of 2, use BN_lshift(3).

BN_sqr() takes the square of *a* and places the result in *r* (`r=a^2`). *r* and *a* may be the same **BIGNUM**. This function is faster than BN_mul(r,a,a).

BN_div() divides *a* by *d* and places the result in *dv* and the remainder in *rem* (`dv=a/d, rem=a%d`). Either of *dv* and *rem* may be **NULL**, in which case the respective value is not returned. The result is rounded towards zero; thus if *a* is negative, the remainder will be zero or negative. For division by powers of 2, use BN_rshift(3).

BN_mod() corresponds to BN_div() with *dv* set to **NULL**.

BN_nnmod() reduces *a* modulo *m* and places the non-negative remainder in *r*.

BN_mod_add() adds *a* to *b* modulo *m* and places the non-negative result in *r*.

BN_mod_sub() subtracts *b* from *a* modulo *m* and places the non-negative result in *r*.

BN_mod_mul() multiplies *a* by *b* and finds the non-negative remainder respective to modulus *m* (`r=(a*b) mod m`). *r* may be the same **BIGNUM** as *a* or *b*. For more efficient algorithms for repeated computations using the same modulus, see BN_mod_mul_montgomery(3) and BN_mod_mul_reciprocal(3).

BN_mod_sqr() takes the square of *a* modulo **m** and places the result in *r*.

BN_exp() raises *a* to the *p*-th power and places the result in *r* (`r=a^p`). This function is faster than repeated applications of BN_mul().

BN_mod_exp() computes *a* to the *p*-th power modulo *m* (`r=a^p % m`). This function uses less time and space than BN_exp().

BN_gcd() computes the greatest common divisor of *a* and *b* and places the result in *r*. *r* may be the same **BIGNUM** as *a* or *b*.

For all functions, *ctx* is a previously allocated **BN_CTX** used for temporary variables; see BN_CTX_new(3).

Unless noted otherwise, the result **BIGNUM** must be different from the arguments.

RETURN VALUES

For all functions, 1 is returned for success, 0 on error. The return value should always be checked (e.g., `if (!BN_add(r,a,b)) goto err;`). The error codes can be obtained by ERR_get_error(3).

SEE ALSO

bn(3), ERR_get_error(3), BN_CTX_new(3), BN_add_word(3), BN_set_bit(3)

HISTORY

BN_add(), BN_sub(), BN_sqr(), BN_div(), BN_mod(), BN_mod_mul(), BN_mod_exp() and BN_gcd() are available in all versions of SSLeay and OpenSSL. The *ctx* argument to BN_mul() was added in SSLeay 0.9.1b. BN_exp() appeared in SSLeay 0.9.0. BN_nnmod(), BN_mod_add(), BN_mod_sub(), and BN_mod_sqr() were added in OpenSSL 0.9.7.

Name

BN_add_word, BN_sub_word, BN_mul_word, BN_div_word and BN_mod_word — arithmetic functions on BIGNUMs with integers

Synopsis

```
#include  <openssl/bn.h>
```

```
int BN_add_word(BIGNUM *a, BN_ULONG w);
```

```
int BN_sub_word(BIGNUM *a, BN_ULONG w);
```

```
int BN_mul_word(BIGNUM *a, BN_ULONG w);
```

```
BN_ULONG BN_div_word(BIGNUM *a, BN_ULONG w);
```

```
BN_ULONG BN_mod_word(const BIGNUM *a, BN_ULONG w);
```

DESCRIPTION

These functions perform arithmetic operations on BIGNUMs with unsigned integers. They are much more efficient than the normal BIGNUM arithmetic operations.

BN_add_word() adds **w** to **a** (`a+=w`).

BN_sub_word() subtracts **w** from **a** (`a-=w`).

BN_mul_word() multiplies **a** and **w** (`a*=w`).

BN_div_word() divides **a** by **w** (`a/=w`) and returns the remainder.

BN_mod_word() returns the remainder of **a** divided by **w** (`a%w`).

For BN_div_word() and BN_mod_word(), **w** must not be 0.

RETURN VALUES

BN_add_word(), BN_sub_word() and BN_mul_word() return 1 for success, 0 on error. The error codes can be obtained by ERR_get_error(3).

BN_mod_word() and BN_div_word() return **a**%**w** on success and **(BN_ULONG)-1** if an error occurred.

SEE ALSO

bn(3), ERR_get_error(3), BN_add(3)

HISTORY

BN_add_word() and BN_mod_word() are available in all versions of SSLeay and OpenSSL. BN_div_word() was added in SSLeay 0.8, and BN_sub_word() and BN_mul_word() in SSLeay 0.9.0.

Before 0.9.8a the return value for BN_div_word() and BN_mod_word() in case of an error was 0.

Name

BN_BLINDING_new, BN_BLINDING_free, BN_BLINDING_update, BN_BLINDING_convert, BN_BLINDING_invert, BN_BLINDING_convert_ex, BN_BLINDING_invert_ex, BN_BLINDING_get_thread_id, BN_BLINDING_set_thread_id, BN_BLINDING_get_flags, BN_BLINDING_set_flags and BN_BLINDING_create_param — blinding related BIGNUM functions.

Synopsis

```
#include  <openssl/bn.h>
```

```
BN_BLINDING *BN_BLINDING_new(const BIGNUM *A, const BIGNUM *Ai,
       BIGNUM *mod);
void BN_BLINDING_free(BN_BLINDING *b);
int BN_BLINDING_update(BN_BLINDING *b,BN_CTX *ctx);
int BN_BLINDING_convert(BIGNUM *n, BN_BLINDING *b, BN_CTX *ctx);
int BN_BLINDING_invert(BIGNUM *n, BN_BLINDING *b, BN_CTX *ctx);
int BN_BLINDING_convert_ex(BIGNUM *n, BIGNUM *r, BN_BLINDING *b,
       BN_CTX *ctx);
int BN_BLINDING_invert_ex(BIGNUM *n, const BIGNUM *r, BN_BLINDING *b,
       BN_CTX *ctx);
#ifndef OPENSSL_NO_DEPRECATED
unsigned long BN_BLINDING_get_thread_id(const BN_BLINDING *);
void BN_BLINDING_set_thread_id(BN_BLINDING *, unsigned long);
#endif
CRYPTO_THREADID *BN_BLINDING_thread_id(BN_BLINDING *);
unsigned long BN_BLINDING_get_flags(const BN_BLINDING *);
void BN_BLINDING_set_flags(BN_BLINDING *, unsigned long);
BN_BLINDING *BN_BLINDING_create_param(BN_BLINDING *b,
       const BIGNUM *e, BIGNUM *m, BN_CTX *ctx,
       int (*bn_mod_exp)(BIGNUM *r, const BIGNUM *a, const BIGNUM *p,
                         const BIGNUM *m, BN_CTX *ctx, BN_MONT_CTX *m_ctx),
       BN_MONT_CTX *m_ctx);
```

DESCRIPTION

BN_BLINDING_new() allocates a new **BN_BLINDING** structure and copies the **A** and **Ai** values into the newly created **BN_BLINDING** object.

BN_BLINDING_free() frees the **BN_BLINDING** structure.

BN_BLINDING_update() updates the **BN_BLINDING** parameters by squaring the **A** and **Ai** or, after specific number of uses and if the necessary parameters are set, by re-creating the blinding parameters.

BN_BLINDING_convert_ex() multiplies **n** with the blinding factor **A**. If **r** is not NULL a copy the inverse blinding factor **Ai** will be returned in **r** (this is useful if a **RSA** object is shared among several threads). BN_BLINDING_invert_ex() multiplies **n** with the inverse blinding factor **Ai**. If **r** is not NULL it will be used as the inverse blinding.

BN_BLINDING_convert() and BN_BLINDING_invert() are wrapper functions for BN_BLINDING_convert_ex() and BN_BLINDING_invert_ex() with **r** set to NULL.

BN_BLINDING_thread_id() provides access to the **CRYPTO_THREADID** object within the **BN_BLINDING** structure. This is to help users provide proper locking if needed for multi-threaded use. The "thread id" object of a newly allocated **BN_BLINDING** structure is initialised to the thread id in which BN_BLINDING_new() was called.

BN_BLINDING_get_flags() returns the BN_BLINDING flags. Currently there are two supported flags: **BN_BLINDING_NO_UPDATE** and **BN_BLINDING_NO_RECREATE**. **BN_BLINDING_NO_UPDATE** inhibits the automatic update of the **BN_BLINDING** parameters after each use and **BN_BLINDING_NO_RECREATE** inhibits the automatic re-creation of the **BN_BLINDING** parameters after a fixed number of uses (currently 32). In newly allocated **BN_BLINDING** objects no flags are set. BN_BLINDING_set_flags() sets the **BN_BLINDING** parameters flags.

BN_BLINDING_create_param() creates new **BN_BLINDING** parameters using the exponent **e** and the modulus **m**. **bn_mod_exp** and **m_ctx** can be used to pass special functions for exponentiation (normally BN_mod_exp_mont() and **BN_MONT_CTX**).

RETURN VALUES

BN_BLINDING_new() returns the newly allocated **BN_BLINDING** structure or NULL in case of an error.

BN_BLINDING_update(), BN_BLINDING_convert(), BN_BLINDING_invert(), BN_BLINDING_convert_ex() and BN_BLINDING_invert_ex() return 1 on success and 0 if an error occured.

BN_BLINDING_thread_id() returns a pointer to the thread id object within a **BN_BLINDING** object.

BN_BLINDING_get_flags() returns the currently set **BN_BLINDING** flags (a **unsigned long** value).

BN_BLINDING_create_param() returns the newly created **BN_BLINDING** parameters or NULL on error.

SEE ALSO

bn(3)

HISTORY

BN_BLINDING_thread_id was first introduced in OpenSSL 1.0.0, and it deprecates BN_BLINDING_set_thread_id and BN_BLINDING_get_thread_id.

BN_BLINDING_convert_ex, BN_BLINDIND_invert_ex, BN_BLINDING_get_thread_id, BN_BLINDING_set_thread_id, BN_BLINDING_set_flags, BN_BLINDING_get_flags and BN_BLINDING_create_param were first introduced in OpenSSL 0.9.8

AUTHOR

Nils Larsch for the OpenSSL project (http://www.openssl.org).

Name

BN_bn2bin, BN_bin2bn, BN_bn2hex, BN_bn2dec, BN_hex2bn, BN_dec2bn, BN_print, BN_print_fp, BN_bn2mpi and BN_mpi2bn — format conversions

Synopsis

```
#include  <openssl/bn.h>
```

```
int BN_bn2bin(const BIGNUM *a, unsigned char *to);
BIGNUM *BN_bin2bn(const unsigned char *s, int len, BIGNUM *ret);
```

```
char *BN_bn2hex(const BIGNUM *a);
char *BN_bn2dec(const BIGNUM *a);
int BN_hex2bn(BIGNUM **a, const char *str);
int BN_dec2bn(BIGNUM **a, const char *str);
```

```
int BN_print(BIO *fp, const BIGNUM *a);
int BN_print_fp(FILE *fp, const BIGNUM *a);
```

```
int BN_bn2mpi(const BIGNUM *a, unsigned char *to);
BIGNUM *BN_mpi2bn(unsigned char *s, int len, BIGNUM *ret);
```

DESCRIPTION

BN_bn2bin() converts the absolute value of **a** into big-endian form and stores it at **to**. **to** must point to BN_num_bytes(**a**) bytes of memory.

BN_bin2bn() converts the positive integer in big-endian form of length **len** at **s** into a **BIGNUM** and places it in **ret**. If **ret** is NULL, a new **BIGNUM** is created.

BN_bn2hex() and BN_bn2dec() return printable strings containing the hexadecimal and decimal encoding of **a** respectively. For negative numbers, the string is prefaced with a leading '-'. The string must be freed later using OPENSSL_free().

BN_hex2bn() converts the string **str** containing a hexadecimal number to a **BIGNUM** and stores it in ****bn**. If ***bn** is NULL, a new **BIGNUM** is created. If **bn** is NULL, it only computes the number's length in hexadecimal digits. If the string starts with '-', the number is negative. BN_dec2bn() is the same using the decimal system.

BN_print() and BN_print_fp() write the hexadecimal encoding of **a**, with a leading '-' for negative numbers, to the **BIO** or **FILE fp**.

BN_bn2mpi() and BN_mpi2bn() convert **BIGNUM**s from and to a format that consists of the number's length in bytes represented as a 4-byte big-endian number, and the number itself in big-endian format, where the most significant bit signals a negative number (the representation of numbers with the MSB set is prefixed with null byte).

BN_bn2mpi() stores the representation of **a** at **to**, where **to** must be large enough to hold the result. The size can be determined by calling BN_bn2mpi(**a**, NULL).

BN_mpi2bn() converts the **len** bytes long representation at **s** to a **BIGNUM** and stores it at **ret**, or in a newly allocated **BIGNUM** if **ret** is NULL.

RETURN VALUES

BN_bn2bin() returns the length of the big-endian number placed at **to**. BN_bin2bn() returns the **BIGNUM**, NULL on error.

BN_bn2hex() and BN_bn2dec() return a null-terminated string, or NULL on error. BN_hex2bn() and BN_dec2bn() return the number's length in hexadecimal or decimal digits, and 0 on error.

BN_print_fp() and BN_print() return 1 on success, 0 on write errors.

BN_bn2mpi() returns the length of the representation. BN_mpi2bn() returns the **BIGNUM**, and NULL on error.

The error codes can be obtained by ERR_get_error(3).

SEE ALSO

bn(3), ERR_get_error(3), BN_zero(3), ASN1_INTEGER_to_BN(3), BN_num_bytes(3)

HISTORY

BN_bn2bin(), BN_bin2bn(), BN_print_fp() and BN_print() are available in all versions of SSLeay and OpenSSL.

BN_bn2hex(), BN_bn2dec(), BN_hex2bn(), BN_dec2bn(), BN_bn2mpi() and BN_mpi2bn() were added in SSLeay 0.9.0.

Name

BN_cmp, BN_ucmp, BN_is_zero, BN_is_one, BN_is_word and BN_is_odd — BIGNUM comparison and test functions

Synopsis

```
#include  <openssl/bn.h>
```

```
int BN_cmp(BIGNUM *a, BIGNUM *b);
int BN_ucmp(BIGNUM *a, BIGNUM *b);
```

```
int BN_is_zero(BIGNUM *a);
int BN_is_one(BIGNUM *a);
int BN_is_word(BIGNUM *a, BN_ULONG w);
int BN_is_odd(BIGNUM *a);
```

DESCRIPTION

BN_cmp() compares the numbers **a** and **b**. BN_ucmp() compares their absolute values.

BN_is_zero(), BN_is_one() and BN_is_word() test if **a** equals 0, 1, or **w** respectively. BN_is_odd() tests if a is odd.

BN_is_zero(), BN_is_one(), BN_is_word() and BN_is_odd() are macros.

RETURN VALUES

BN_cmp() returns -1 if **a** < **b**, 0 if **a** == **b** and 1 if **a** > **b**. BN_ucmp() is the same using the absolute values of **a** and **b**.

BN_is_zero(), BN_is_one() BN_is_word() and BN_is_odd() return 1 if the condition is true, 0 otherwise.

SEE ALSO

bn(3)

HISTORY

BN_cmp(), BN_ucmp(), BN_is_zero(), BN_is_one() and BN_is_word() are available in all versions of SSLeay and OpenSSL. BN_is_odd() was added in SSLeay 0.8.

Name

BN_copy and BN_dup — copy BIGNUMs

Synopsis

```
#include  <openssl/bn.h>
```

```
BIGNUM *BN_copy(BIGNUM *to, const BIGNUM *from);
```

```
BIGNUM *BN_dup(const BIGNUM *from);
```

DESCRIPTION

BN_copy() copies **from** to **to**. BN_dup() creates a new **BIGNUM** containing the value **from**.

RETURN VALUES

BN_copy() returns **to** on success, NULL on error. BN_dup() returns the new **BIGNUM**, and NULL on error. The error codes can be obtained by ERR_get_error(3).

SEE ALSO

bn(3), ERR_get_error(3)

HISTORY

BN_copy() and BN_dup() are available in all versions of SSLeay and OpenSSL.

Name

BN_CTX_new, BN_CTX_init and BN_CTX_free — allocate and free BN_CTX structures

Synopsis

```
#include  <openssl/bn.h>
```

```
BN_CTX *BN_CTX_new(void);
```

```
void BN_CTX_init(BN_CTX *c);
```

```
void BN_CTX_free(BN_CTX *c);
```

DESCRIPTION

A **BN_CTX** is a structure that holds **BIGNUM** temporary variables used by library functions. Since dynamic memory allocation to create **BIGNUM**s is rather expensive when used in conjunction with repeated subroutine calls, the **BN_CTX** structure is used.

BN_CTX_new() allocates and initializes a **BN_CTX** structure. BN_CTX_init() initializes an existing uninitialized **BN_CTX**.

BN_CTX_free() frees the components of the **BN_CTX**, and if it was created by BN_CTX_new(), also the structure itself. If BN_CTX_start(3) has been used on the **BN_CTX**, BN_CTX_end(3) must be called before the **BN_CTX** may be freed by BN_CTX_free().

RETURN VALUES

BN_CTX_new() returns a pointer to the **BN_CTX**. If the allocation fails, it returns **NULL** and sets an error code that can be obtained by ERR_get_error(3).

BN_CTX_init() and BN_CTX_free() have no return values.

SEE ALSO

bn(3), ERR_get_error(3), BN_add(3), BN_CTX_start(3)

HISTORY

BN_CTX_new() and BN_CTX_free() are available in all versions on SSLeay and OpenSSL. BN_CTX_init() was added in SSLeay 0.9.1b.

Name

BN_CTX_start, BN_CTX_get and BN_CTX_end — use temporary BIGNUM variables

Synopsis

```
#include  <openssl/bn.h>
```

```
void BN_CTX_start(BN_CTX *ctx);
```

```
BIGNUM *BN_CTX_get(BN_CTX *ctx);
```

```
void BN_CTX_end(BN_CTX *ctx);
```

DESCRIPTION

These functions are used to obtain temporary **BIGNUM** variables from a **BN_CTX** (which can been created by using BN_CTX_new(3)) in order to save the overhead of repeatedly creating and freeing **BIGNUM**s in functions that are called from inside a loop.

A function must call BN_CTX_start() first. Then, BN_CTX_get() may be called repeatedly to obtain temporary **BIGNUM**s. All BN_CTX_get() calls must be made before calling any other functions that use the **ctx** as an argument.

Finally, BN_CTX_end() must be called before returning from the function. When BN_CTX_end() is called, the **BIGNUM** pointers obtained from BN_CTX_get() become invalid.

RETURN VALUES

BN_CTX_start() and BN_CTX_end() return no values.

BN_CTX_get() returns a pointer to the **BIGNUM**, or **NULL** on error. Once BN_CTX_get() has failed, the subsequent calls will return **NULL** as well, so it is sufficient to check the return value of the last BN_CTX_get() call. In case of an error, an error code is set, which can be obtained by ERR_get_error(3).

SEE ALSO

BN_CTX_new(3)

HISTORY

BN_CTX_start(), BN_CTX_get() and BN_CTX_end() were added in OpenSSL 0.9.5.

Name

BN_generate_prime, BN_is_prime and BN_is_prime_fasttest — generate primes and test for primality

Synopsis

```
#include  <openssl/bn.h>
```

```
BIGNUM *BN_generate_prime(BIGNUM *ret, int num, int safe, BIGNUM *add,
    BIGNUM *rem, void (*callback)(int, int, void *), void *cb_arg);
```

```
int BN_is_prime(const BIGNUM *a, int checks, void (*callback)(int, int,
    void *), BN_CTX *ctx, void *cb_arg);
```

```
int BN_is_prime_fasttest(const BIGNUM *a, int checks,
    void (*callback)(int, int, void *), BN_CTX *ctx, void *cb_arg,
    int do_trial_division);
```

DESCRIPTION

BN_generate_prime() generates a pseudo-random prime number of **num** bits. If **ret** is not **NULL**, it will be used to store the number.

If **callback** is not **NULL**, it is called as follows:

- **callback(0, i, cb_arg)** is called after generating the i-th potential prime number.

- While the number is being tested for primality, **callback(1, j, cb_arg)** is called as described below.

- When a prime has been found, **callback(2, i, cb_arg)** is called.

The prime may have to fulfill additional requirements for use in Diffie-Hellman key exchange:

If **add** is not **NULL**, the prime will fulfill the condition p % **add** == **rem** (p % **add** == 1 if **rem** == **NULL**) in order to suit a given generator.

If **safe** is true, it will be a safe prime (i.e. a prime p so that (p-1)/2 is also prime).

The PRNG must be seeded prior to calling BN_generate_prime(). The prime number generation has a negligible error probability.

BN_is_prime() and BN_is_prime_fasttest() test if the number **a** is prime. The following tests are performed until one of them shows that **a** is composite; if **a** passes all these tests, it is considered prime.

BN_is_prime_fasttest(), when called with **do_trial_division == 1**, first attempts trial division by a number of small primes; if no divisors are found by this test and **callback** is not **NULL**, **callback(1, -1, cb_arg)** is called. If **do_trial_division == 0**, this test is skipped.

Both BN_is_prime() and BN_is_prime_fasttest() perform a Miller-Rabin probabilistic primality test with **checks** iterations. If **checks == BN_prime_checks**, a number of iterations is used that yields a false positive rate of at most 2^-80 for random input.

If **callback** is not **NULL**, **callback(1, j, cb_arg)** is called after the j-th iteration (j = 0, 1, …). **ctx** is a pre-allocated **BN_CTX** (to save the overhead of allocating and freeing the structure in a loop), or **NULL**.

RETURN VALUES

BN_generate_prime() returns the prime number on success, **NULL** otherwise.

BN_is_prime() returns 0 if the number is composite, 1 if it is prime with an error probability of less than 0.25^**checks**, and -1 on error.

The error codes can be obtained by ERR_get_error(3).

SEE ALSO

bn(3), ERR_get_error(3), rand(3)

HISTORY

The **cb_arg** arguments to BN_generate_prime() and to BN_is_prime() were added in SSLeay 0.9.0. The **ret** argument to BN_generate_prime() was added in SSLeay 0.9.1. BN_is_prime_fasttest() was added in OpenSSL 0.9.5.

Name

BN_mod_inverse — compute inverse modulo n

Synopsis

```
#include  <openssl/bn.h>
```

```
BIGNUM *BN_mod_inverse(BIGNUM *r, BIGNUM *a, const BIGNUM *n,
          BN_CTX *ctx);
```

DESCRIPTION

BN_mod_inverse() computes the inverse of **a** modulo **n** places the result in **r** (`(a*r)%n==1`). If **r** is NULL, a new **BIGNUM** is created.

ctx is a previously allocated **BN_CTX** used for temporary variables. **r** may be the same **BIGNUM** as **a** or **n**.

RETURN VALUES

BN_mod_inverse() returns the **BIGNUM** containing the inverse, and NULL on error. The error codes can be obtained by ERR_get_error(3).

SEE ALSO

bn(3), ERR_get_error(3), BN_add(3)

HISTORY

BN_mod_inverse() is available in all versions of SSLeay and OpenSSL.

Name

BN_mod_mul_montgomery, BN_MONT_CTX_new, BN_MONT_CTX_init, BN_MONT_CTX_free, BN_MONT_CTX_set, BN_MONT_CTX_copy, BN_from_montgomery and BN_to_montgomery — Montgomery multiplication

Synopsis

```
#include  <openssl/bn.h>
```

```
BN_MONT_CTX *BN_MONT_CTX_new(void);
void BN_MONT_CTX_init(BN_MONT_CTX *ctx);
void BN_MONT_CTX_free(BN_MONT_CTX *mont);
```

```
int BN_MONT_CTX_set(BN_MONT_CTX *mont, const BIGNUM *m, BN_CTX *ctx);
BN_MONT_CTX *BN_MONT_CTX_copy(BN_MONT_CTX *to, BN_MONT_CTX *from);
```

```
int BN_mod_mul_montgomery(BIGNUM *r, BIGNUM *a, BIGNUM *b,
        BN_MONT_CTX *mont, BN_CTX *ctx);
```

```
int BN_from_montgomery(BIGNUM *r, BIGNUM *a, BN_MONT_CTX *mont,
        BN_CTX *ctx);
```

```
int BN_to_montgomery(BIGNUM *r, BIGNUM *a, BN_MONT_CTX *mont,
        BN_CTX *ctx);
```

DESCRIPTION

These functions implement Montgomery multiplication. They are used automatically when BN_mod_exp(3) is called with suitable input, but they may be useful when several operations are to be performed using the same modulus.

BN_MONT_CTX_new() allocates and initializes a **BN_MONT_CTX** structure. BN_MONT_CTX_init() initializes an existing uninitialized **BN_MONT_CTX**.

BN_MONT_CTX_set() sets up the *mont* structure from the modulus *m* by precomputing its inverse and a value R.

BN_MONT_CTX_copy() copies the **BN_MONT_CTX** *from* to *to*.

BN_MONT_CTX_free() frees the components of the **BN_MONT_CTX**, and, if it was created by BN_MONT_CTX_new(), also the structure itself.

BN_mod_mul_montgomery() computes Mont(*a*,*b*):=*a***b**R^-1 and places the result in *r*.

BN_from_montgomery() performs the Montgomery reduction *r* = *a**R^-1.

BN_to_montgomery() computes Mont(*a*,R^2), i.e. *a**R. Note that *a* must be non-negative and smaller than the modulus.

For all functions, *ctx* is a previously allocated **BN_CTX** used for temporary variables.

The **BN_MONT_CTX** structure is defined as follows:

```
typedef struct bn_mont_ctx_st
       {
       int ri;         /* number of bits in R */
       BIGNUM RR;      /* R^2 (used to convert to Montgomery form) */
       BIGNUM N;       /* The modulus */
       BIGNUM Ni;      /* R*(1/R mod N) - N*Ni = 1
                        * (Ni is only stored for bignum algorithm) */
       BN_ULONG n0;    /* least significant word of Ni */
       int flags;
       } BN_MONT_CTX;
```

BN_to_montgomery() is a macro.

RETURN VALUES

BN_MONT_CTX_new() returns the newly allocated **BN_MONT_CTX**, and NULL on error.

BN_MONT_CTX_init() and BN_MONT_CTX_free() have no return values.

For the other functions, 1 is returned for success, 0 on error. The error codes can be obtained by ERR_get_error(3).

WARNING

The inputs must be reduced modulo **m**, otherwise the result will be outside the expected range.

SEE ALSO

bn(3), ERR_get_error(3), BN_add(3), BN_CTX_new(3)

HISTORY

BN_MONT_CTX_new(), BN_MONT_CTX_free(), BN_MONT_CTX_set(), BN_mod_mul_montgomery(), BN_from_montgomery() and BN_to_montgomery() are available in all versions of SSLeay and OpenSSL.

BN_MONT_CTX_init() and BN_MONT_CTX_copy() were added in SSLeay 0.9.1b.

Name

BN_mod_mul_reciprocal, BN_div_recp, BN_RECP_CTX_new, BN_RECP_CTX_init, BN_RECP_CTX_free and BN_RECP_CTX_set — modular multiplication using reciprocal

Synopsis

```
#include  <openssl/bn.h>
```

```
BN_RECP_CTX *BN_RECP_CTX_new(void);
void BN_RECP_CTX_init(BN_RECP_CTX *recp);
void BN_RECP_CTX_free(BN_RECP_CTX *recp);
```

```
int BN_RECP_CTX_set(BN_RECP_CTX *recp, const BIGNUM *m, BN_CTX *ctx);
```

```
int BN_div_recp(BIGNUM *dv, BIGNUM *rem, BIGNUM *a, BN_RECP_CTX *recp,
       BN_CTX *ctx);
```

```
int BN_mod_mul_reciprocal(BIGNUM *r, BIGNUM *a, BIGNUM *b,
       BN_RECP_CTX *recp, BN_CTX *ctx);
```

DESCRIPTION

BN_mod_mul_reciprocal() can be used to perform an efficient BN_mod_mul(3) operation when the operation will be performed repeatedly with the same modulus. It computes **r**=(**a*****b**)%**m** using **recp**=1/**m**, which is set as described below. **ctx** is a previously allocated **BN_CTX** used for temporary variables.

BN_RECP_CTX_new() allocates and initializes a **BN_RECP** structure. BN_RECP_CTX_init() initializes an existing uninitialized **BN_RECP**.

BN_RECP_CTX_free() frees the components of the **BN_RECP**, and, if it was created by BN_RECP_CTX_new(), also the structure itself.

BN_RECP_CTX_set() stores **m** in **recp** and sets it up for computing 1/**m** and shifting it left by BN_num_bits(**m**)+1 to make it an integer. The result and the number of bits it was shifted left will later be stored in **recp**.

BN_div_recp() divides **a** by **m** using **recp**. It places the quotient in **dv** and the remainder in **rem**.

The **BN_RECP_CTX** structure is defined as follows:

```
typedef struct bn_recp_ctx_st
       {
       BIGNUM N;       /* the divisor */
       BIGNUM Nr;      /* the reciprocal */
       int num_bits;
       int shift;
       int flags;
       } BN_RECP_CTX;
```

It cannot be shared between threads.

RETURN VALUES

BN_RECP_CTX_new() returns the newly allocated **BN_RECP_CTX**, and NULL on error.

BN_RECP_CTX_init() and BN_RECP_CTX_free() have no return values.

For the other functions, 1 is returned for success, 0 on error. The error codes can be obtained by ERR_get_error(3).

SEE ALSO

bn(3), ERR_get_error(3), BN_add(3), BN_CTX_new(3)

HISTORY

BN_RECP_CTX was added in SSLeay 0.9.0. Before that, the function BN_reciprocal() was used instead, and the BN_mod_mul_reciprocal() arguments were different.

Name

BN_new, BN_init, BN_clear, BN_free and BN_clear_free — allocate and free BIGNUMs

Synopsis

```
#include  <openssl/bn.h>
```

```
BIGNUM *BN_new(void);
```

```
void BN_init(BIGNUM *);
```

```
void BN_clear(BIGNUM *a);
```

```
void BN_free(BIGNUM *a);
```

```
void BN_clear_free(BIGNUM *a);
```

DESCRIPTION

BN_new() allocates and initializes a **BIGNUM** structure. BN_init() initializes an existing uninitialized **BIGNUM**.

BN_clear() is used to destroy sensitive data such as keys when they are no longer needed. It erases the memory used by **a** and sets it to the value 0.

BN_free() frees the components of the **BIGNUM**, and if it was created by BN_new(), also the structure itself. BN_clear_free() additionally overwrites the data before the memory is returned to the system.

RETURN VALUES

BN_new() returns a pointer to the **BIGNUM**. If the allocation fails, it returns **NULL** and sets an error code that can be obtained by ERR_get_error(3).

BN_init(), BN_clear(), BN_free() and BN_clear_free() have no return values.

SEE ALSO

bn(3), ERR_get_error(3)

HISTORY

BN_new(), BN_clear(), BN_free() and BN_clear_free() are available in all versions on SSLeay and OpenSSL. BN_init() was added in SSLeay 0.9.1b.

Name

BN_num_bits, BN_num_bytes and BN_num_bits_word — get BIGNUM size

Synopsis

```
#include  <openssl/bn.h>
```

```
int BN_num_bytes(const BIGNUM *a);
```

```
int BN_num_bits(const BIGNUM *a);
```

```
int BN_num_bits_word(BN_ULONG w);
```

DESCRIPTION

BN_num_bytes() returns the size of a **BIGNUM** in bytes.

BN_num_bits_word() returns the number of significant bits in a word. If we take 0x00000432 as an example, it returns 11, not 16, not 32. Basically, except for a zero, it returns *floor(log2(w))+1*.

BN_num_bits() returns the number of significant bits in a **BIGNUM**, following the same principle as BN_num_bits_word().

BN_num_bytes() is a macro.

RETURN VALUES

The size.

NOTES

Some have tried using BN_num_bits() on individual numbers in RSA keys, DH keys and DSA keys, and found that they don't always come up with the number of bits they expected (something like 512, 1024, 2048, …). This is because generating a number with some specific number of bits doesn't always set the highest bits, thereby making the number of *significant* bits a little lower. If you want to know the "key size" of such a key, either use functions like RSA_size(), DH_size() and DSA_size(), or use BN_num_bytes() and multiply with 8 (although there's no real guarantee that will match the "key size", just a lot more probability).

SEE ALSO

bn(3), DH_size(3), DSA_size(3), RSA_size(3)

HISTORY

BN_num_bytes(), BN_num_bits() and BN_num_bits_word() are available in all versions of SSLeay and OpenSSL.

Name

BN_rand and BN_pseudo_rand — generate pseudo-random number

Synopsis

```
#include  <openssl/bn.h>
```

```
int BN_rand(BIGNUM *rnd, int bits, int top, int bottom);
```

```
int BN_pseudo_rand(BIGNUM *rnd, int bits, int top, int bottom);
```

```
int BN_rand_range(BIGNUM *rnd, BIGNUM *range);
```

```
int BN_pseudo_rand_range(BIGNUM *rnd, BIGNUM *range);
```

DESCRIPTION

BN_rand() generates a cryptographically strong pseudo-random number of **bits** in length and stores it in **rnd**. If **top** is -1, the most significant bit of the random number can be zero. If **top** is 0, it is set to 1, and if **top** is 1, the two most significant bits of the number will be set to 1, so that the product of two such random numbers will always have 2***bits** length. If **bottom** is true, the number will be odd. The value of **bits** must be zero or greater. If **bits** is 1 then **top** cannot also be 1.

BN_pseudo_rand() does the same, but pseudo-random numbers generated by this function are not necessarily unpredictable. They can be used for non-cryptographic purposes and for certain purposes in cryptographic protocols, but usually not for key generation etc.

BN_rand_range() generates a cryptographically strong pseudo-random number **rnd** in the range 0 <= **rnd** < **range**. BN_pseudo_rand_range() does the same, but is based on BN_pseudo_rand(), and hence numbers generated by it are not necessarily unpredictable.

The PRNG must be seeded prior to calling BN_rand() or BN_rand_range().

RETURN VALUES

The functions return 1 on success, 0 on error. The error codes can be obtained by ERR_get_error(3).

SEE ALSO

bn(3), ERR_get_error(3), rand(3), RAND_add(3), RAND_bytes(3)

HISTORY

BN_rand() is available in all versions of SSLeay and OpenSSL. BN_pseudo_rand() was added in OpenSSL 0.9.5. The **top** == -1 case and the function BN_rand_range() were added in OpenSSL 0.9.6a. BN_pseudo_rand_range() was added in OpenSSL 0.9.6c.

Name

BN_set_bit, BN_clear_bit, BN_is_bit_set, BN_mask_bits, BN_lshift, BN_lshift1, BN_rshift and BN_rshift1 — bit operations on BIGNUMs

Synopsis

```
#include  <openssl/bn.h>
```

```
int BN_set_bit(BIGNUM *a, int n);
int BN_clear_bit(BIGNUM *a, int n);
```

```
int BN_is_bit_set(const BIGNUM *a, int n);
```

```
int BN_mask_bits(BIGNUM *a, int n);
```

```
int BN_lshift(BIGNUM *r, const BIGNUM *a, int n);
int BN_lshift1(BIGNUM *r, BIGNUM *a);
```

```
int BN_rshift(BIGNUM *r, BIGNUM *a, int n);
int BN_rshift1(BIGNUM *r, BIGNUM *a);
```

DESCRIPTION

BN_set_bit() sets bit **n** in **a** to 1 (`a|=(1&lt;&lt;n)`). The number is expanded if necessary.

BN_clear_bit() sets bit **n** in **a** to 0 (`a&=~(1&lt;&lt;n)`). An error occurs if **a** is shorter than **n** bits.

BN_is_bit_set() tests if bit **n** in **a** is set.

BN_mask_bits() truncates **a** to an **n** bit number (`a&=~((~0)&gt;&gt;n)`). An error occurs if **a** already is shorter than **n** bits.

BN_lshift() shifts **a** left by **n** bits and places the result in **r** (`r=a*2^n`). Note that **n** must be non-negative. BN_lshift1() shifts **a** left by one and places the result in **r** (`r=2*a`).

BN_rshift() shifts **a** right by **n** bits and places the result in **r** (`r=a/2^n`). Note that **n** must be non-negative. BN_rshift1() shifts **a** right by one and places the result in **r** (`r=a/2`).

For the shift functions, **r** and **a** may be the same variable.

RETURN VALUES

BN_is_bit_set() returns 1 if the bit is set, 0 otherwise.

All other functions return 1 for success, 0 on error. The error codes can be obtained by ERR_get_error(3).

SEE ALSO

bn(3), BN_num_bytes(3), BN_add(3)

HISTORY

BN_set_bit(), BN_clear_bit(), BN_is_bit_set(), BN_mask_bits(), BN_lshift(), BN_lshift1(), BN_rshift(), and BN_rshift1() are available in all versions of SSLeay and OpenSSL.

Name

BN_swap — exchange BIGNUMs

Synopsis

```
#include  <openssl/bn.h>
```

```
void BN_swap(BIGNUM *a, BIGNUM *b);
```

DESCRIPTION

BN_swap() exchanges the values of *a* and *b*.

bn(3)

HISTORY

BN_swap was added in OpenSSL 0.9.7.

Name

BN_zero, BN_one, BN_value_one, BN_set_word and BN_get_word — BIGNUM assignment operations

Synopsis

```
#include  <openssl/bn.h>
```

```
int BN_zero(BIGNUM *a);
int BN_one(BIGNUM *a);
```

```
const BIGNUM *BN_value_one(void);
```

```
int BN_set_word(BIGNUM *a, unsigned long w);
unsigned long BN_get_word(BIGNUM *a);
```

DESCRIPTION

BN_zero(), BN_one() and BN_set_word() set **a** to the values 0, 1 and **w** respectively. BN_zero() and BN_one() are macros.

BN_value_one() returns a **BIGNUM** constant of value 1. This constant is useful for use in comparisons and assignment.

BN_get_word() returns **a**, if it can be represented as an unsigned long.

RETURN VALUES

BN_get_word() returns the value **a**, and 0xffffffffL if **a** cannot be represented as an unsigned long.

BN_zero(), BN_one() and BN_set_word() return 1 on success, 0 otherwise. BN_value_one() returns the constant.

BUGS

Someone might change the constant.

If a **BIGNUM** is equal to 0xffffffffL it can be represented as an unsigned long but this value is also returned on error.

SEE ALSO

bn(3), BN_bn2bin(3)

HISTORY

BN_zero(), BN_one() and BN_set_word() are available in all versions of SSLeay and OpenSSL. BN_value_one() and BN_get_word() were added in SSLeay 0.8.

BN_value_one() was changed to return a true const BIGNUM * in OpenSSL 0.9.7.

Name

BUF_MEM_new, BUF_MEM_new_ex, BUF_MEM_free, BUF_MEM_grow, BUF_strdup, BUF_strndup, BUF_memdup, BUF_strlcpy and BUF_strlcat — simple character array structure, with some standard C library equivalents

Synopsis

```
#include  <openssl/buffer.h>
```

```
BUF_MEM *BUF_MEM_new(void);
```

```
void   BUF_MEM_free(BUF_MEM *a);
```

```
int    BUF_MEM_grow(BUF_MEM *str, int len);
```

```
char *BUF_strdup(const char *str);
```

```
char *BUF_strndup(const char *str, size_t siz);
```

```
void *BUF_memdup(const void *data, size_t siz);
```

```
size_t BUF_strlcpy(char *dst, const char *src, size_t size);
```

```
size_t BUF_strlcat(char *dst, const char *src, size_t size);
```

DESCRIPTION

The buffer library handles simple character arrays. Buffers are used for various purposes in the library, most notably memory BIOs.

BUF_MEM_new() allocates a new buffer of zero size.

BUF_MEM_free() frees up an already existing buffer. The data is zeroed before freeing up in case the buffer contains sensitive data.

BUF_MEM_grow() changes the size of an already existing buffer to **len**. Any data already in the buffer is preserved if it increases in size.

BUF_strdup(), BUF_strndup(), BUF_memdup(), BUF_strlcpy() and BUF_strlcat() are equivalents of the standard C library functions. The dup() functions use OPENSSL_malloc() underneath and so should be used in preference to the standard library for memory leak checking or replacing the malloc() function.

Memory allocated from these functions should be freed up using the OPENSSL_free() function.

BUF_strndup makes the explicit guarantee that it will never read past the first **siz** bytes of **str**.

RETURN VALUES

BUF_MEM_new() returns the buffer or NULL on error.

BUF_MEM_free() has no return value.

BUF_MEM_grow() returns zero on error or the new size (i.e. **len**).

SEE ALSO

bio(3)

HISTORY

BUF_MEM_new(), BUF_MEM_free() and BUF_MEM_grow() are available in all versions of SSLeay and OpenSSL. BUF_strdup() was added in SSLeay 0.8.

Name

CMS_add0_cert, CMS_add1_cert, CMS_get1_certs, CMS_add0_crl and CMS_get1_crls — CMS certificate and CRL utility functions

Synopsis

```
#include  <openssl/cms.h>
```

```
int CMS_add0_cert(CMS_ContentInfo *cms, X509 *cert);
int CMS_add1_cert(CMS_ContentInfo *cms, X509 *cert);
STACK_OF(X509) *CMS_get1_certs(CMS_ContentInfo *cms);
```

```
int CMS_add0_crl(CMS_ContentInfo *cms, X509_CRL *crl);
int CMS_add1_crl(CMS_ContentInfo *cms, X509_CRL *crl);
STACK_OF(X509_CRL) *CMS_get1_crls(CMS_ContentInfo *cms);
```

DESCRIPTION

CMS_add0_cert() and CMS_add1_cert() add certificate **cert** to **cms**. must be of type signed data or enveloped data.

CMS_get1_certs() returns all certificates in **cms**.

CMS_add0_crl() and CMS_add1_crl() add CRL **crl** to **cms**. CMS_get1_crls() returns any CRLs in **cms**.

NOTES

The CMS_ContentInfo structure **cms** must be of type signed data or enveloped data or an error will be returned.

For signed data certificates and CRLs are added to the **certificates** and **crls** fields of SignedData structure. For enveloped data they are added to **OriginatorInfo**.

As the **0** implies CMS_add0_cert() adds **cert** internally to **cms** and it must not be freed up after the call as opposed to CMS_add1_cert() where **cert** must be freed up.

The same certificate or CRL must not be added to the same cms structure more than once.

RETURN VALUES

CMS_add0_cert(), CMS_add1_cert() and CMS_add0_crl() and CMS_add1_crl() return 1 for success and 0 for failure.

CMS_get1_certs() and CMS_get1_crls() return the STACK of certificates or CRLs or NULL if there are none or an error occurs. The only error which will occur in practice is if the **cms** type is invalid.

SEE ALSO

ERR_get_error(3), CMS_sign(3), CMS_encrypt(3)

HISTORY

CMS_add0_cert(), CMS_add1_cert(), CMS_get1_certs(), CMS_add0_crl() and CMS_get1_crls() were all first added to OpenSSL 0.9.8

Name

CMS_add1_recipient_cert and CMS_add0_recipient_key — add recipients to a CMS enveloped data structure

Synopsis

```
#include  <openssl/cms.h>
```

```
CMS_RecipientInfo *CMS_add1_recipient_cert(CMS_ContentInfo *cms, X509 *recip, unsigned int flags);
```

```
CMS_RecipientInfo *CMS_add0_recipient_key(CMS_ContentInfo *cms, int nid, unsigned char *key,
    size_t keylen, unsigned char *id, size_t idlen, ASN1_GENERALIZEDTIME *date,
    WASN1_OBJECT *otherTypeId, ASN1_TYPE *otherType);
```

DESCRIPTION

CMS_add1_recipient_cert() adds recipient **recip** to CMS_ContentInfo enveloped data structure **cms** as a KeyTransRecipientInfo structure.

CMS_add0_recipient_key() adds symmetric key **key** of length **keylen** using wrapping algorithm **nid**, identifier **id** of length **idlen** and optional values **date**, **otherTypeId** and **otherType** to CMS_ContentInfo enveloped data structure **cms** as a KEKRecipientInfo structure.

The CMS_ContentInfo structure should be obtained from an initial call to CMS_encrypt() with the flag **CMS_PARTIAL** set.

NOTES

The main purpose of this function is to provide finer control over a CMS enveloped data structure where the simpler CMS_encrypt() function defaults are not appropriate. For example if one or more KEKRecipientInfo structures need to be added. New attributes can also be added using the returned CMS_RecipientInfo structure and the CMS attribute utility functions.

OpenSSL will by default identify recipient certificates using issuer name and serial number. If **CMS_USE_KEYID** is set it will use the subject key identifier value instead. An error occurs if all recipient certificates do not have a subject key identifier extension.

Currently only AES based key wrapping algorithms are supported for **nid**, specifically: NID_id_aes128_wrap, NID_id_aes192_wrap and NID_id_aes256_wrap. If **nid** is set to **NID_undef** then an AES wrap algorithm will be used consistent with **keylen**.

RETURN VALUES

CMS_add1_recipient_cert() and CMS_add0_recipient_key() return an internal pointer to the CMS_RecipientInfo structure just added or NULL if an error occurs.

SEE ALSO

ERR_get_error(3), CMS_decrypt(3), CMS_final(3)

HISTORY

CMS_add1_recipient_cert() and CMS_add0_recipient_key() were added to OpenSSL 0.9.8

Name

CMS_compress — create a CMS CompressedData structure

Synopsis

```
#include  <openssl/cms.h>
```

```
CMS_ContentInfo *CMS_compress(BIO *in, int comp_nid, unsigned int flags);
```

DESCRIPTION

CMS_compress() creates and returns a CMS CompressedData structure. **comp_nid** is the compression algorithm to use or **NID_undef** to use the default algorithm (zlib compression). **in** is the content to be compressed. **flags** is an optional set of flags.

NOTES

The only currently supported compression algorithm is zlib using the NID NID_zlib_compression.

If zlib support is not compiled into OpenSSL then CMS_compress() will return an error.

If the **CMS_TEXT** flag is set MIME headers for type **text/plain** are prepended to the data.

Normally the supplied content is translated into MIME canonical format (as required by the S/MIME specifications) if **CMS_BINARY** is set no translation occurs. This option should be used if the supplied data is in binary format otherwise the translation will corrupt it. If **CMS_BINARY** is set then **CMS_TEXT** is ignored.

If the **CMS_STREAM** flag is set a partial **CMS_ContentInfo** structure is returned suitable for streaming I/O: no data is read from the BIO **in**.

The compressed data is included in the CMS_ContentInfo structure, unless **CMS_DETACHED** is set in which case it is omitted. This is rarely used in practice and is not supported by SMIME_write_CMS().

NOTES

If the flag **CMS_STREAM** is set the returned **CMS_ContentInfo** structure is **not** complete and outputting its contents via a function that does not properly finalize the **CMS_ContentInfo** structure will give unpredictable results.

Several functions including SMIME_write_CMS(), i2d_CMS_bio_stream(), PEM_write_bio_CMS_stream() finalize the structure. Alternatively finalization can be performed by obtaining the streaming ASN1 **BIO** directly using BIO_new_CMS().

Additional compression parameters such as the zlib compression level cannot currently be set.

RETURN VALUES

CMS_compress() returns either a CMS_ContentInfo structure or NULL if an error occurred. The error can be obtained from ERR_get_error(3).

SEE ALSO

ERR_get_error(3), CMS_uncompress(3)

HISTORY

CMS_compress() was added to OpenSSL 0.9.8 The **CMS_STREAM** flag was first supported in OpenSSL 1.0.0.

Name

CMS_decrypt — decrypt content from a CMS envelopedData structure

Synopsis

```
#include  <openssl/cms.h>
```

```
int CMS_decrypt(CMS_ContentInfo *cms, EVP_PKEY *pkey, X509 *cert, BIO *dcont, BIO *out,
    unsigned int flags);
```

DESCRIPTION

CMS_decrypt() extracts and decrypts the content from a CMS EnvelopedData structure. **pkey** is the private key of the recipient, **cert** is the recipient's certificate, **out** is a BIO to write the content to and **flags** is an optional set of flags.

The **dcont** parameter is used in the rare case where the encrypted content is detached. It will normally be set to NULL.

NOTES

OpenSSL_add_all_algorithms() (or equivalent) should be called before using this function or errors about unknown algorithms will occur.

Although the recipients certificate is not needed to decrypt the data it is needed to locate the appropriate (of possible several) recipients in the CMS structure.

If **cert** is set to NULL all possible recipients are tried. This case however is problematic. To thwart the MMA attack (Bleichenbacher's attack on PKCS #1 v1.5 RSA padding) all recipients are tried whether they succeed or not. If no recipient succeeds then a random symmetric key is used to decrypt the content: this will typically output garbage and may (but is not guaranteed to) ultimately return a padding error only. If CMS_decrypt() just returned an error when all recipient encrypted keys failed to decrypt an attacker could use this in a timing attack. If the special flag **CMS_DEBUG_DECRYPT** is set then the above behaviour is modified and an error **is** returned if no recipient encrypted key can be decrypted **without** generating a random content encryption key. Applications should use this flag with **extreme caution** especially in automated gateways as it can leave them open to attack.

It is possible to determine the correct recipient key by other means (for example looking them up in a database) and setting them in the CMS structure in advance using the CMS utility functions such as CMS_set1_pkey(). In this case both **cert** and **pkey** should be set to NULL.

To process KEKRecipientInfo types CMS_set1_key() or CMS_RecipientInfo_set0_key() and CMS_ReceipientInfo_decrypt() should be called before CMS_decrypt() and **cert** and **pkey** set to NULL.

The following flags can be passed in the **flags** parameter.

If the **CMS_TEXT** flag is set MIME headers for type **text/plain** are deleted from the content. If the content is not of type **text/plain** then an error is returned.

RETURN VALUES

CMS_decrypt() returns either 1 for success or 0 for failure. The error can be obtained from ERR_get_error(3)

BUGS

The lack of single pass processing and the need to hold all data in memory as mentioned in CMS_verify() also applies to CMS_decrypt().

SEE ALSO

ERR_get_error(3), CMS_encrypt(3)

HISTORY

CMS_decrypt() was added to OpenSSL 0.9.8

Name

CMS_encrypt — create a CMS envelopedData structure

Synopsis

```
#include  <openssl/cms.h>
```

```
CMS_ContentInfo *CMS_encrypt(STACK_OF(X509) *certs, BIO *in, const EVP_CIPHER *cipher,
      unsigned int flags);
```

DESCRIPTION

CMS_encrypt() creates and returns a CMS EnvelopedData structure. **certs** is a list of recipient certificates. **in** is the content to be encrypted. **cipher** is the symmetric cipher to use. **flags** is an optional set of flags.

NOTES

Only certificates carrying RSA keys are supported so the recipient certificates supplied to this function must all contain RSA public keys, though they do not have to be signed using the RSA algorithm.

EVP_des_ede3_cbc() (triple DES) is the algorithm of choice for S/MIME use because most clients will support it.

The algorithm passed in the **cipher** parameter must support ASN1 encoding of its parameters.

Many browsers implement a "sign and encrypt" option which is simply an S/MIME envelopedData containing an S/MIME signed message. This can be readily produced by storing the S/MIME signed message in a memory BIO and passing it to CMS_encrypt().

The following flags can be passed in the **flags** parameter.

If the **CMS_TEXT** flag is set MIME headers for type **text/plain** are prepended to the data.

Normally the supplied content is translated into MIME canonical format (as required by the S/MIME specifications) if **CMS_BINARY** is set no translation occurs. This option should be used if the supplied data is in binary format otherwise the translation will corrupt it. If **CMS_BINARY** is set then **CMS_TEXT** is ignored.

OpenSSL will by default identify recipient certificates using issuer name and serial number. If **CMS_USE_KEYID** is set it will use the subject key identifier value instead. An error occurs if all recipient certificates do not have a subject key identifier extension.

If the **CMS_STREAM** flag is set a partial **CMS_ContentInfo** structure is returned suitable for streaming I/O: no data is read from the BIO **in**.

If the **CMS_PARTIAL** flag is set a partial **CMS_ContentInfo** structure is returned to which additional recipients and attributes can be added before finalization.

The data being encrypted is included in the CMS_ContentInfo structure, unless **CMS_DETACHED** is set in which case it is omitted. This is rarely used in practice and is not supported by SMIME_write_CMS().

NOTES

If the flag **CMS_STREAM** is set the returned **CMS_ContentInfo** structure is **not** complete and outputting its contents via a function that does not properly finalize the **CMS_ContentInfo** structure will give unpredictable results.

Several functions including SMIME_write_CMS(), i2d_CMS_bio_stream(), PEM_write_bio_CMS_stream() finalize the structure. Alternatively finalization can be performed by obtaining the streaming ASN1 **BIO** directly using BIO_new_CMS().

The recipients specified in **certs** use a CMS KeyTransRecipientInfo info structure. KEKRecipientInfo is also supported using the flag **CMS_PARTIAL** and CMS_add0_recipient_key().

The parameter **certs** may be NULL if **CMS_PARTIAL** is set and recipients added later using CMS_add1_recipient_cert() or CMS_add0_recipient_key().

RETURN VALUES

CMS_encrypt() returns either a CMS_ContentInfo structure or NULL if an error occurred. The error can be obtained from ERR_get_error(3).

SEE ALSO

ERR_get_error(3), CMS_decrypt(3)

HISTORY

CMS_decrypt() was added to OpenSSL 0.9.8 The **CMS_STREAM** flag was first supported in OpenSSL 1.0.0.

Name

CMS_final — finalise a CMS_ContentInfo structure

Synopsis

```
#include  <openssl/cms.h>
```

```
int CMS_final(CMS_ContentInfo *cms, BIO *data, BIO *dcont, unsigned int flags);
```

DESCRIPTION

CMS_final() finalises the structure **cms**. It's purpose is to perform any operations necessary on **cms** (digest computation for example) and set the appropriate fields. The parameter **data** contains the content to be processed. The **dcont** parameter contains a BIO to write content to after processing: this is only used with detached data and will usually be set to NULL.

NOTES

This function will normally be called when the **CMS_PARTIAL** flag is used. It should only be used when streaming is not performed because the streaming I/O functions perform finalisation operations internally.

RETURN VALUES

CMS_final() returns 1 for success or 0 for failure.

SEE ALSO

ERR_get_error(3), CMS_sign(3), CMS_encrypt(3)

HISTORY

CMS_final() was added to OpenSSL 0.9.8

Name

CMS_get0_RecipientInfos, CMS_RecipientInfo_type, CMS_RecipientInfo_ktri_get0_signer_id, CMS_RecipientInfo_ktri_cert_cmp, CMS_RecipientInfo_set0_pkey, CMS_RecipientInfo_kekri_get0_id, CMS_RecipientInfo_kekri_id_cmp, CMS_RecipientInfo_set0_key and CMS_RecipientInfo_decrypt — CMS envelopedData RecipientInfo routines

Synopsis

```
#include  <openssl/cms.h>
```

```
STACK_OF(CMS_RecipientInfo) *CMS_get0_RecipientInfos(CMS_ContentInfo *cms);
int CMS_RecipientInfo_type(CMS_RecipientInfo *ri);
```

```
int CMS_RecipientInfo_ktri_get0_signer_id(CMS_RecipientInfo *ri, ASN1_OCTET_STRING **keyid,
    X509_NAME **issuer, ASN1_INTEGER **sno);
int CMS_RecipientInfo_ktri_cert_cmp(CMS_RecipientInfo *ri, X509 *cert);
int CMS_RecipientInfo_set0_pkey(CMS_RecipientInfo *ri, EVP_PKEY *pkey);
```

```
int CMS_RecipientInfo_kekri_get0_id(CMS_RecipientInfo *ri, X509_ALGOR **palg, ASN1_OCTET_STRING **pid,
    ASN1_GENERALIZEDTIME **pdate, ASN1_OBJECT **potherid, ASN1_TYPE **pothertype);
int CMS_RecipientInfo_kekri_id_cmp(CMS_RecipientInfo *ri, const unsigned char *id, size_t idlen);
int CMS_RecipientInfo_set0_key(CMS_RecipientInfo *ri, unsigned char *key, size_t keylen);
```

```
int CMS_RecipientInfo_decrypt(CMS_ContentInfo *cms, CMS_RecipientInfo *ri);
```

DESCRIPTION

The function CMS_get0_RecipientInfos() returns all the CMS_RecipientInfo structures associated with a CMS EnvelopedData structure.

CMS_RecipientInfo_type() returns the type of CMS_RecipientInfo structure **ri**. It will currently return CMS_RECIPINFO_TRANS, CMS_RECIPINFO_AGREE, CMS_RECIPINFO_KEK, CMS_RECIPINFO_PASS, or CMS_RECIPINFO_OTHER.

CMS_RecipientInfo_ktri_get0_signer_id() retrieves the certificate recipient identifier associated with a specific CMS_RecipientInfo structure **ri**, which must be of type CMS_RECIPINFO_TRANS. Either the keyidentifier will be set in **keyid** or **both** issuer name and serial number in **issuer** and **sno**.

CMS_RecipientInfo_ktri_cert_cmp() compares the certificate **cert** against the CMS_RecipientInfo structure **ri**, which must be of type CMS_RECIPINFO_TRANS. It returns zero if the comparison is successful and non zero if not.

CMS_RecipientInfo_set0_pkey() associates the private key **pkey** with the CMS_RecipientInfo structure **ri**, which must be of type CMS_RECIPINFO_TRANS.

CMS_RecipientInfo_kekri_get0_id() retrieves the key information from the CMS_RecipientInfo structure **ri** which must be of type CMS_RECIPINFO_KEK. Any of the remaining parameters can be NULL if the application is not interested in the value of a field. Where a field is optional and absent NULL will be written to the corresponding parameter. The keyEncryptionAlgorithm field is written to **palg**, the **keyIdentifier** field is written to **pid**, the **date** field if present is written to **pdate**, if the **other** field is present the components **keyAttrId** and **keyAttr** are written to parameters **potherid** and **pothertype**.

CMS_RecipientInfo_kekri_id_cmp() compares the ID in the **id** and **idlen** parameters against the **keyIdentifier** CMS_RecipientInfo structure **ri**, which must be of type CMS_RECIPINFO_KEK. It returns zero if the comparison is successful and non zero if not.

CMS_RecipientInfo_set0_key() associates the symmetric key **key** of length **keylen** with the CMS_RecipientInfo structure **ri**, which must be of type CMS_RECIPINFO_KEK.

CMS_RecipientInfo_decrypt() attempts to decrypt CMS_RecipientInfo structure **ri** in structure **cms**. A key must have been associated with the structure first.

NOTES

The main purpose of these functions is to enable an application to lookup recipient keys using any appropriate technique when the simpler method of CMS_decrypt() is not appropriate.

In typical usage and application will retrieve all CMS_RecipientInfo structures using CMS_get0_RecipientInfos() and check the type of each using CMS_RecpientInfo_type(). Depending on the type the CMS_RecipientInfo structure can be ignored or its key identifier data retrieved using an appropriate function. Then if the corresponding secret or private key can be obtained by any appropriate means it can then associated with the structure and CMS_RecpientInfo_decrypt() called. If successful CMS_decrypt() can be called with a NULL key to decrypt the enveloped content.

RETURN VALUES

CMS_get0_RecipientInfos() returns all CMS_RecipientInfo structures, or NULL if an error occurs.

CMS_RecipientInfo_ktri_get0_signer_id(), CMS_RecipientInfo_set0_pkey(), CMS_RecipientInfo_kekri_get0_id(), CMS_RecipientInfo_set0_key() and CMS_RecipientInfo_decrypt() return 1 for success or 0 if an error occurs.

CMS_RecipientInfo_ktri_cert_cmp() and CMS_RecipientInfo_kekri_cmp() return 0 for a successful comparison and non zero otherwise.

Any error can be obtained from ERR_get_error(3).

SEE ALSO

ERR_get_error(3), CMS_decrypt(3)

HISTORY

These functions were first was added to OpenSSL 0.9.8

Name

CMS_get0_SignerInfos, CMS_SignerInfo_get0_signer_id, CMS_SignerInfo_cert_cmp and CMS_set1_signer_certs — CMS signedData signer functions.

Synopsis

```
#include  <openssl/cms.h>
```

```
STACK_OF(CMS_SignerInfo) *CMS_get0_SignerInfos(CMS_ContentInfo *cms);
```

```
int CMS_SignerInfo_get0_signer_id(CMS_SignerInfo *si, ASN1_OCTET_STRING **keyid, X509_NAME **issuer,
    ASN1_INTEGER **sno);
int CMS_SignerInfo_cert_cmp(CMS_SignerInfo *si, X509 *cert);
void CMS_SignerInfo_set1_signer_cert(CMS_SignerInfo *si, X509 *signer);
```

DESCRIPTION

The function CMS_get0_SignerInfos() returns all the CMS_SignerInfo structures associated with a CMS signedData structure.

CMS_SignerInfo_get0_signer_id() retrieves the certificate signer identifier associated with a specific CMS_SignerInfo structure **si**. Either the keyidentifier will be set in **keyid** or **both** issuer name and serial number in **issuer** and **sno**.

CMS_SignerInfo_cert_cmp() compares the certificate **cert** against the signer identifier **si**. It returns zero if the comparison is successful and non zero if not.

CMS_SignerInfo_set1_signer_cert() sets the signers certificate of **si** to **signer**.

NOTES

The main purpose of these functions is to enable an application to lookup signers certificates using any appropriate technique when the simpler method of CMS_verify() is not appropriate.

In typical usage and application will retrieve all CMS_SignerInfo structures using CMS_get0_SignerInfo() and retrieve the identifier information using CMS. It will then obtain the signer certificate by some unspecified means (or return and error if it cannot be found) and set it using CMS_SignerInfo_set1_signer_cert().

Once all signer certificates have been set CMS_verify() can be used.

Although CMS_get0_SignerInfos() can return NULL is an error occur **or** if there are no signers this is not a problem in practice because the only error which can occur is if the **cms** structure is not of type signedData due to application error.

RETURN VALUES

CMS_get0_SignerInfos() returns all CMS_SignerInfo structures, or NULL there are no signers or an error occurs.

CMS_SignerInfo_get0_signer_id() returns 1 for success and 0 for failure.

CMS_SignerInfo_cert_cmp() returns 0 for a successful comparison and non zero otherwise.

CMS_SignerInfo_set1_signer_cert() does not return a value.

Any error can be obtained from ERR_get_error(3)

SEE ALSO

ERR_get_error(3), CMS_verify(3)

HISTORY

These functions were first was added to OpenSSL 0.9.8

Name

CMS_get0_type, CMS_set1_eContentType, CMS_get0_eContentType and CMS_get0_content — get and set CMS content types and content

Synopsis

```
#include  <openssl/cms.h>
```

```
const ASN1_OBJECT *CMS_get0_type(CMS_ContentInfo *cms);
int CMS_set1_eContentType(CMS_ContentInfo *cms, const ASN1_OBJECT *oid);
const ASN1_OBJECT *CMS_get0_eContentType(CMS_ContentInfo *cms);
ASN1_OCTET_STRING **CMS_get0_content(CMS_ContentInfo *cms);
```

DESCRIPTION

CMS_get0_type() returns the content type of a CMS_ContentInfo structure as and ASN1_OBJECT pointer. An application can then decide how to process the CMS_ContentInfo structure based on this value.

CMS_set1_eContentType() sets the embedded content type of a CMS_ContentInfo structure. It should be called with CMS functions with the **CMS_PARTIAL** flag and **before** the structure is finalised, otherwise the results are undefined.

ASN1_OBJECT *CMS_get0_eContentType() returns a pointer to the embedded content type.

CMS_get0_content() returns a pointer to the **ASN1_OCTET_STRING** pointer containing the embedded content.

NOTES

As the **0** implies CMS_get0_type(), CMS_get0_eContentType() and CMS_get0_content() return internal pointers which should **not** be freed up. CMS_set1_eContentType() copies the supplied OID and it **should** be freed up after use.

The **ASN1_OBJECT** values returned can be converted to an integer **NID** value using OBJ_obj2nid(). For the currently supported content types the following values are returned:

```
NID_pkcs7_data
NID_pkcs7_signed
NID_pkcs7_digest
NID_id_smime_ct_compressedData:
NID_pkcs7_encrypted
NID_pkcs7_enveloped
```

The return value of CMS_get0_content() is a pointer to the **ASN1_OCTET_STRING** content pointer. That means that for example:

```
ASN1_OCTET_STRING **pconf = CMS_get0_content(cms);
```

***pconf** could be NULL if there is no embedded content. Applications can access, modify or create the embedded content in a **CMS_ContentInfo** structure using this function. Applications usually will not need to modify the embedded content as it is normally set by higher level functions.

RETURN VALUES

CMS_get0_type() and CMS_get0_eContentType() return and ASN1_OBJECT structure.

CMS_set1_eContentType() returns 1 for success or 0 if an error occurred. The error can be obtained from ERR_get_error(3).

SEE ALSO

ERR_get_error(3)

HISTORY

CMS_get0_type(), CMS_set1_eContentType() and CMS_get0_eContentType() were all first added to OpenSSL 0.9.8

Name

CMS_ReceiptRequest_create0, CMS_add1_ReceiptRequest, CMS_get1_ReceiptRequest and CMS_ReceiptRequest_get0_values — CMS signed receipt request functions.

Synopsis

```
#include  <openssl/cms.h>
```

```
CMS_ReceiptRequest *CMS_ReceiptRequest_create0(unsigned char *id, int idlen, int allorfirst,
    STACK_OF(GENERAL_NAMES) *receiptList, STACK_OF(GENERAL_NAMES) *receiptsTo);
int CMS_add1_ReceiptRequest(CMS_SignerInfo *si, CMS_ReceiptRequest *rr);
int CMS_get1_ReceiptRequest(CMS_SignerInfo *si, CMS_ReceiptRequest **prr);
void CMS_ReceiptRequest_get0_values(CMS_ReceiptRequest *rr, ASN1_STRING **pcid, int *pallorfirst,
    STACK_OF(GENERAL_NAMES) **plist, STACK_OF(GENERAL_NAMES) **prto);
```

DESCRIPTION

CMS_ReceiptRequest_create0() creates a signed receipt request structure. The **signedContentIdentifier** field is set using **id** and **idlen**, or it is set to 32 bytes of pseudo random data if **id** is NULL. If **receiptList** is NULL the allOrFirstTier option in **receiptsFrom** is used and set to the value of the **allorfirst** parameter. If **receiptList** is not NULL the **receiptList** option in **receiptsFrom** is used. The **receiptsTo** parameter specifies the **receiptsTo** field value.

The CMS_add1_ReceiptRequest() function adds a signed receipt request **rr** to SignerInfo structure **si**.

int CMS_get1_ReceiptRequest() looks for a signed receipt request in **si**, if any is found it is decoded and written to **prr**.

CMS_ReceiptRequest_get0_values() retrieves the values of a receipt request. The signedContentIdentifier is copied to **pcid**. If the **allOrFirstTier** option of **receiptsFrom** is used its value is copied to **pallorfirst** otherwise the **receiptList** field is copied to **plist**. The **receiptsTo** parameter is copied to **prto**.

NOTES

For more details of the meaning of the fields see RFC2634.

The contents of a signed receipt should only be considered meaningful if the corresponding CMS_ContentInfo structure can be successfully verified using CMS_verify().

RETURN VALUES

CMS_ReceiptRequest_create0() returns a signed receipt request structure or NULL if an error occurred.

CMS_add1_ReceiptRequest() returns 1 for success or 0 is an error occurred.

CMS_get1_ReceiptRequest() returns 1 is a signed receipt request is found and decoded. It returns 0 if a signed receipt request is not present and -1 if it is present but malformed.

SEE ALSO

ERR_get_error(3), CMS_sign(3), CMS_sign_receipt(3), CMS_verify(3), CMS_verify_receipt(3)

HISTORY

CMS_ReceiptRequest_create0(), CMS_add1_ReceiptRequest(), CMS_get1_ReceiptRequest() and CMS_ReceiptRequest_get0_values() were added to OpenSSL 0.9.8

Name

CMS_add1_signer and CMS_SignerInfo_sign — add a signer to a CMS_ContentInfo signed data structure.

Synopsis

```
#include  <openssl/cms.h>
```

```
CMS_SignerInfo *CMS_add1_signer(CMS_ContentInfo *cms, X509 *signcert, EVP_PKEY *pkey, const EVP_MD *md,
    unsigned int flags);
```

```
int CMS_SignerInfo_sign(CMS_SignerInfo *si);
```

DESCRIPTION

CMS_add1_signer() adds a signer with certificate **signcert** and private key **pkey** using message digest **md** to CMS_ContentInfo SignedData structure **cms**.

The CMS_ContentInfo structure should be obtained from an initial call to CMS_sign() with the flag **CMS_PARTIAL** set or in the case or re-signing a valid CMS_ContentInfo SignedData structure.

If the **md** parameter is **NULL** then the default digest for the public key algorithm will be used.

Unless the **CMS_REUSE_DIGEST** flag is set the returned CMS_ContentInfo structure is not complete and must be finalized either by streaming (if applicable) or a call to CMS_final().

The CMS_SignerInfo_sign() function will explicitly sign a CMS_SignerInfo structure, its main use is when **CMS_REUSE_DIGEST** and **CMS_PARTIAL** flags are both set.

NOTES

The main purpose of CMS_add1_signer() is to provide finer control over a CMS signed data structure where the simpler CMS_sign() function defaults are not appropriate. For example if multiple signers or non default digest algorithms are needed. New attributes can also be added using the returned CMS_SignerInfo structure and the CMS attribute utility functions or the CMS signed receipt request functions.

Any of the following flags (ored together) can be passed in the **flags** parameter.

If **CMS_REUSE_DIGEST** is set then an attempt is made to copy the content digest value from the CMS_ContentInfo structure: to add a signer to an existing structure. An error occurs if a matching digest value cannot be found to copy. The returned CMS_ContentInfo structure will be valid and finalized when this flag is set.

If **CMS_PARTIAL** is set in addition to **CMS_REUSE_DIGEST** then the CMS_SignerInfo structure will not be finalized so additional attributes can be added. In this case an explicit call to CMS_SignerInfo_sign() is needed to finalize it.

If **CMS_NOCERTS** is set the signer's certificate will not be included in the CMS_ContentInfo structure, the signer's certificate must still be supplied in the **signcert** parameter though. This can reduce the size of the signature if the signers certificate can be obtained by other means: for example a previously signed message.

The SignedData structure includes several CMS signedAttributes including the signing time, the CMS content type and the supported list of ciphers in an SMIMECapabilities attribute. If **CMS_NOATTR** is set then no signedAttributes will be used. If **CMS_NOSMIMECAP** is set then just the SMIMECapabilities are omitted.

OpenSSL will by default identify signing certificates using issuer name and serial number. If **CMS_USE_KEYID** is set it will use the subject key identifier value instead. An error occurs if the signing certificate does not have a subject key identifier extension.

If present the SMIMECapabilities attribute indicates support for the following algorithms in preference order: 256 bit AES, Gost R3411-94, Gost 28147-89, 192 bit AES, 128 bit AES, triple DES, 128 bit RC2, 64 bit RC2, DES and 40 bit RC2. If any of these algorithms is not available then it will not be included: for example the GOST algorithms will not be included if the GOST ENGINE is not loaded.

CMS_add1_signer() returns an internal pointer to the CMS_SignerInfo structure just added, this can be used to set additional attributes before it is finalized.

RETURN VALUES

CMS_add1_signer() returns an internal pointer to the CMS_SignerInfo structure just added or NULL if an error occurs.

SEE ALSO

ERR_get_error(3), CMS_sign(3), CMS_final(3)

HISTORY

CMS_add1_signer() was added to OpenSSL 0.9.8

Name

CMS_sign — create a CMS SignedData structure

Synopsis

```
#include  <openssl/cms.h>
```

```
CMS_ContentInfo *CMS_sign(X509 *signcert, EVP_PKEY *pkey, STACK_OF(X509) *certs, BIO *data,
    unsigned int flags);
```

DESCRIPTION

CMS_sign() creates and returns a CMS SignedData structure. **signcert** is the certificate to sign with, **pkey** is the corresponding private key. **certs** is an optional additional set of certificates to include in the CMS structure (for example any intermediate CAs in the chain). Any or all of these parameters can be **NULL**, see **NOTES** below.

The data to be signed is read from BIO **data**.

flags is an optional set of flags.

NOTES

Any of the following flags (ored together) can be passed in the **flags** parameter.

Many S/MIME clients expect the signed content to include valid MIME headers. If the **CMS_TEXT** flag is set MIME headers for type **text/plain** are prepended to the data.

If **CMS_NOCERTS** is set the signer's certificate will not be included in the CMS_ContentInfo structure, the signer's certificate must still be supplied in the **signcert** parameter though. This can reduce the size of the signature if the signers certificate can be obtained by other means: for example a previously signed message.

The data being signed is included in the CMS_ContentInfo structure, unless **CMS_DETACHED** is set in which case it is omitted. This is used for CMS_ContentInfo detached signatures which are used in S/MIME plaintext signed messages for example.

Normally the supplied content is translated into MIME canonical format (as required by the S/MIME specifications) if **CMS_BINARY** is set no translation occurs. This option should be used if the supplied data is in binary format otherwise the translation will corrupt it.

The SignedData structure includes several CMS signedAttributes including the signing time, the CMS content type and the supported list of ciphers in an SMIMECapabilities attribute. If **CMS_NOATTR** is set then no signedAttributes will be used. If **CMS_NOSMIMECAP** is set then just the SMIMECapabilities are omitted.

If present the SMIMECapabilities attribute indicates support for the following algorithms in preference order: 256 bit AES, Gost R3411-94, Gost 28147-89, 192 bit AES, 128 bit AES, triple DES, 128 bit RC2, 64 bit RC2, DES and 40 bit RC2. If any of these algorithms is not available then it will not be included: for example the GOST algorithms will not be included if the GOST ENGINE is not loaded.

OpenSSL will by default identify signing certificates using issuer name and serial number. If **CMS_USE_KEYID** is set it will use the subject key identifier value instead. An error occurs if the signing certificate does not have a subject key identifier extension.

If the flags **CMS_STREAM** is set then the returned **CMS_ContentInfo** structure is just initialized ready to perform the signing operation. The signing is however **not** performed and the data to be signed is not read from the **data** parameter. Signing is deferred until after the data has been written. In this way data can be signed in a single pass.

If the **CMS_PARTIAL** flag is set a partial **CMS_ContentInfo** structure is output to which additional signers and capabilities can be added before finalization.

If the flag **CMS_STREAM** is set the returned **CMS_ContentInfo** structure is **not** complete and outputting its contents via a function that does not properly finalize the **CMS_ContentInfo** structure will give unpredictable results.

Several functions including SMIME_write_CMS(), i2d_CMS_bio_stream(), PEM_write_bio_CMS_stream() finalize the structure. Alternatively finalization can be performed by obtaining the streaming ASN1 **BIO** directly using BIO_new_CMS().

If a signer is specified it will use the default digest for the signing algorithm. This is **SHA1** for both RSA and DSA keys.

If **signcert** and **pkey** are NULL then a certificates only CMS structure is output.

The function CMS_sign() is a basic CMS signing function whose output will be suitable for many purposes. For finer control of the output format the **certs**, **signcert** and **pkey** parameters can all be **NULL** and the **CMS_PARTIAL** flag set. Then one or more signers can be added using the function CMS_sign_add1_signer(), non default digests can be used and custom attributes added. **CMS_final()** must then be called to finalize the structure if streaming is not enabled.

BUGS

Some attributes such as counter signatures are not supported.

RETURN VALUES

CMS_sign() returns either a valid CMS_ContentInfo structure or NULL if an error occurred. The error can be obtained from ERR_get_error(3).

SEE ALSO

ERR_get_error(3), CMS_verify(3)

HISTORY

CMS_sign() was added to OpenSSL 0.9.8

The **CMS_STREAM** flag is only supported for detached data in OpenSSL 0.9.8, it is supported for embedded data in OpenSSL 1.0.0 and later.

Name

CMS_sign_receipt — create a CMS signed receipt

Synopsis

```
#include  <openssl/cms.h>
```

```
CMS_ContentInfo *CMS_sign_receipt(CMS_SignerInfo *si, X509 *signcert, EVP_PKEY *pkey,
    STACK_OF(X509) *certs, unsigned int flags);
```

DESCRIPTION

CMS_sign_receipt() creates and returns a CMS signed receipt structure. **si** is the **CMS_SignerInfo** structure containing the signed receipt request. **signcert** is the certificate to sign with, **pkey** is the corresponding private key. **certs** is an optional additional set of certificates to include in the CMS structure (for example any intermediate CAs in the chain).

flags is an optional set of flags.

NOTES

This functions behaves in a similar way to CMS_sign() except the flag values **CMS_DETACHED**, **CMS_BINARY**, **CMS_NOATTR**, **CMS_TEXT** and **CMS_STREAM** are not supported since they do not make sense in the context of signed receipts.

RETURN VALUES

CMS_sign_receipt() returns either a valid CMS_ContentInfo structure or NULL if an error occurred. The error can be obtained from ERR_get_error(3).

SEE ALSO

ERR_get_error(3), CMS_verify_receipt(3), CMS_sign(3)

HISTORY

CMS_sign_receipt() was added to OpenSSL 0.9.8

Name

CMS_uncompress — uncompress a CMS CompressedData structure

Synopsis

```
#include  <openssl/cms.h>
```

```
int CMS_uncompress(CMS_ContentInfo *cms, BIO *dcont, BIO *out, unsigned int flags);
```

DESCRIPTION

CMS_uncompress() extracts and uncompresses the content from a CMS CompressedData structure **cms**. **data** is a BIO to write the content to and **flags** is an optional set of flags.

The **dcont** parameter is used in the rare case where the compressed content is detached. It will normally be set to NULL.

NOTES

The only currently supported compression algorithm is zlib: if the structure indicates the use of any other algorithm an error is returned.

If zlib support is not compiled into OpenSSL then CMS_uncompress() will always return an error.

The following flags can be passed in the **flags** parameter.

If the **CMS_TEXT** flag is set MIME headers for type **text/plain** are deleted from the content. If the content is not of type **text/plain** then an error is returned.

RETURN VALUES

CMS_uncompress() returns either 1 for success or 0 for failure. The error can be obtained from ERR_get_error(3)

BUGS

The lack of single pass processing and the need to hold all data in memory as mentioned in CMS_verify() also applies to CMS_decompress().

SEE ALSO

ERR_get_error(3), CMS_compress(3)

HISTORY

CMS_uncompress() was added to OpenSSL 0.9.8

Name

CMS_verify — verify a CMS SignedData structure

Synopsis

```
#include  <openssl/cms.h>
```

```
int CMS_verify(CMS_ContentInfo *cms, STACK_OF(X509) *certs, X509_STORE *store, BIO *indata,
    BIO *out, unsigned int flags);
```

```
STACK_OF(X509) *CMS_get0_signers(CMS_ContentInfo *cms);
```

DESCRIPTION

CMS_verify() verifies a CMS SignedData structure. **cms** is the CMS_ContentInfo structure to verify. **certs** is a set of certificates in which to search for the signing certificate(s). **store** is a trusted certificate store used for chain verification. **indata** is the detached content if the content is not present in **cms**. The content is written to **out** if it is not NULL.

flags is an optional set of flags, which can be used to modify the verify operation.

CMS_get0_signers() retrieves the signing certificate(s) from **cms**, it must be called after a successful CMS_verify() operation.

VERIFY PROCESS

Normally the verify process proceeds as follows.

Initially some sanity checks are performed on **cms**. The type of **cms** must be SignedData. There must be at least one signature on the data and if the content is detached **indata** cannot be **NULL**.

An attempt is made to locate all the signing certificate(s), first looking in the **certs** parameter (if it is not NULL) and then looking in any certificates contained in the **cms** structure itself. If any signing certificate cannot be located the operation fails.

Each signing certificate is chain verified using the **smimesign** purpose and the supplied trusted certificate store. Any internal certificates in the message are used as untrusted CAs. If CRL checking is enabled in **store** any internal CRLs are used in addition to attempting to look them up in **store**. If any chain verify fails an error code is returned.

Finally the signed content is read (and written to **out** is it is not NULL) and the signature's checked.

If all signature's verify correctly then the function is successful.

Any of the following flags (ored together) can be passed in the **flags** parameter to change the default verify behaviour.

If **CMS_NOINTERN** is set the certificates in the message itself are not searched when locating the signing certificate(s). This means that all the signing certificates must be in the **certs** parameter.

If **CMS_NOCRL** is set and CRL checking is enabled in **store** then any CRLs in the message itself are ignored.

If the **CMS_TEXT** flag is set MIME headers for type **text/plain** are deleted from the content. If the content is not of type **text/plain** then an error is returned.

If **CMS_NO_SIGNER_CERT_VERIFY** is set the signing certificates are not verified.

If **CMS_NO_ATTR_VERIFY** is set the signed attributes signature is not verified.

If **CMS_NO_CONTENT_VERIFY** is set then the content digest is not checked.

NOTES

One application of **CMS_NOINTERN** is to only accept messages signed by a small number of certificates. The acceptable certificates would be passed in the **certs** parameter. In this case if the signer is not one of the certificates supplied in **certs** then the verify will fail because the signer cannot be found.

In some cases the standard techniques for looking up and validating certificates are not appropriate: for example an application may wish to lookup certificates in a database or perform customised verification. This can be achieved by setting and verifying the signers certificates manually using the signed data utility functions.

Care should be taken when modifying the default verify behaviour, for example setting **CMS_NO_CONTENT_VERIFY** will totally disable all content verification and any modified content will be considered valid. This combination is however useful if one merely wishes to write the content to **out** and its validity is not considered important.

Chain verification should arguably be performed using the signing time rather than the current time. However since the signing time is supplied by the signer it cannot be trusted without additional evidence (such as a trusted timestamp).

RETURN VALUES

CMS_verify() returns 1 for a successful verification and zero if an error occurred.

CMS_get0_signers() returns all signers or NULL if an error occurred.

The error can be obtained from ERR_get_error(3)

BUGS

The trusted certificate store is not searched for the signing certificate, this is primarily due to the inadequacies of the current **X509_STORE** functionality.

The lack of single pass processing means that the signed content must all be held in memory if it is not detached.

SEE ALSO

ERR_get_error(3), CMS_sign(3)

HISTORY

CMS_verify() was added to OpenSSL 0.9.8

Name

CMS_verify_receipt — verify a CMS signed receipt

Synopsis

```
#include  <openssl/cms.h>
```

```
int CMS_verify_receipt(CMS_ContentInfo *rcms, CMS_ContentInfo *ocms, STACK_OF(X509) *certs,
    X509_STORE *store, unsigned int flags);
```

DESCRIPTION

CMS_verify_receipt() verifies a CMS signed receipt. **rcms** is the signed receipt to verify. **ocms** is the original SignedData structure containing the receipt request. **certs** is a set of certificates in which to search for the signing certificate. **store** is a trusted certificate store (used for chain verification).

flags is an optional set of flags, which can be used to modify the verify operation.

NOTES

This functions behaves in a similar way to CMS_verify() except the flag values **CMS_DETACHED**, **CMS_BINARY**, **CMS_TEXT** and **CMS_STREAM** are not supported since they do not make sense in the context of signed receipts.

RETURN VALUES

CMS_verify_receipt() returns 1 for a successful verification and zero if an error occurred.

The error can be obtained from ERR_get_error(3)

SEE ALSO

ERR_get_error(3), CMS_sign_receipt(3), CMS_verify(3)

HISTORY

CMS_verify_receipt() was added to OpenSSL 0.9.8

Name

CONF_modules_free, CONF_modules_finish and CONF_modules_unload — OpenSSL configuration cleanup functions

Synopsis

```
#include  <openssl/conf.h>
```

```
void CONF_modules_free(void);
void CONF_modules_finish(void);
void CONF_modules_unload(int all);
```

DESCRIPTION

CONF_modules_free() closes down and frees up all memory allocated by all configuration modules.

CONF_modules_finish() calls each configuration modules **finish** handler to free up any configuration that module may have performed.

CONF_modules_unload() finishes and unloads configuration modules. If **all** is set to **0** only modules loaded from DSOs will be unloads. If **all** is **1** all modules, including builtin modules will be unloaded.

NOTES

Normally applications will only call CONF_modules_free() at application to tidy up any configuration performed.

RETURN VALUE

None of the functions return a value.

SEE ALSO

conf(5), OPENSSL_config(3), CONF_modules_load_file(3)

HISTORY

CONF_modules_free(), CONF_modules_unload(), and CONF_modules_finish() first appeared in OpenSSL 0.9.7.

Name

CONF_modules_load_file and CONF_modules_load — OpenSSL configuration functions

Synopsis

```
#include  <openssl/conf.h>
```

```
int CONF_modules_load_file(const char *filename, const char *appname,
                                       unsigned long flags);
int CONF_modules_load(const CONF *cnf, const char *appname,
                              unsigned long flags);
```

DESCRIPTION

The function CONF_modules_load_file() configures OpenSSL using file **filename** and application name **appname**. If **filename** is NULL the standard OpenSSL configuration file is used. If **appname** is NULL the standard OpenSSL application name **openssl_conf** is used. The behaviour can be cutomized using **flags**.

CONF_modules_load() is idential to CONF_modules_load_file() except it reads configuration information from **cnf**.

NOTES

The following **flags** are currently recognized:

CONF_MFLAGS_IGNORE_ERRORS if set errors returned by individual configuration modules are ignored. If not set the first module error is considered fatal and no further modules are loaded.

Normally any modules errors will add error information to the error queue. If **CONF_MFLAGS_SILENT** is set no error information is added.

If **CONF_MFLAGS_NO_DSO** is set configuration module loading from DSOs is disabled.

CONF_MFLAGS_IGNORE_MISSING_FILE if set will make CONF_load_modules_file() ignore missing configuration files. Normally a missing configuration file return an error.

CONF_MFLAGS_DEFAULT_SECTION if set and **appname** is not NULL will use the default section pointed to by **openssl_conf** if **appname** does not exist.

Applications should call these functions after loading builtin modules using OPENSSL_load_builtin_modules(), any ENGINEs for example using ENGINE_load_builtin_engines(), any algorithms for example OPENSSL_add_all_algorithms() and (if the application uses libssl) SSL_library_init().

By using CONF_modules_load_file() with appropriate flags an application can customise application configuration to best suit its needs. In some cases the use of a configuration file is optional and its absence is not an error: in this case **CONF_MFLAGS_IGNORE_MISSING_FILE** would be set.

Errors during configuration may also be handled differently by different applications. For example in some cases an error may simply print out a warning message and the application continue. In other cases an application might consider a configuration file error as fatal and exit immediately.

Applications can use the CONF_modules_load() function if they wish to load a configuration file themselves and have finer control over how errors are treated.

EXAMPLES

Load a configuration file and print out any errors and exit (missing file considered fatal):

```
if (CONF_modules_load_file(NULL, NULL, 0) <= 0) {
   fprintf(stderr, "FATAL: error loading configuration file\n");
   ERR_print_errors_fp(stderr);
   exit(1);
}
```

Load default configuration file using the section indicated by "myapp", tolerate missing files, but exit on other errors:

```
if (CONF_modules_load_file(NULL, "myapp",
                           CONF_MFLAGS_IGNORE_MISSING_FILE) <= 0) {
   fprintf(stderr, "FATAL: error loading configuration file\n");
   ERR_print_errors_fp(stderr);
   exit(1);
}
```

Load custom configuration file and section, only print warnings on error, missing configuration file ignored:

```
if (CONF_modules_load_file("/something/app.cnf", "myapp",
                           CONF_MFLAGS_IGNORE_MISSING_FILE) <= 0) {
   fprintf(stderr, "WARNING: error loading configuration file\n");
   ERR_print_errors_fp(stderr);
}
```

Load and parse configuration file manually, custom error handling:

```
FILE *fp;
CONF *cnf = NULL;
long eline;
fp = fopen("/somepath/app.cnf", "r");
if (fp == NULL) {
   fprintf(stderr, "Error opening configuration file\n");
   /* Other missing configuration file behaviour */
} else {
   cnf = NCONF_new(NULL);
   if (NCONF_load_fp(cnf, fp, &eline) == 0) {
       fprintf(stderr, "Error on line %ld of configuration file\n", eline);
       ERR_print_errors_fp(stderr);
       /* Other malformed configuration file behaviour */
   } else if (CONF_modules_load(cnf, "appname", 0) <= 0) {
     fprintf(stderr, "Error configuring application\n");
     ERR_print_errors_fp(stderr);
     /* Other configuration error behaviour */
   }
   fclose(fp);
   NCONF_free(cnf);
 }
```

RETURN VALUES

These functions return 1 for success and a zero or negative value for failure. If module errors are not ignored the return code will reflect the return value of the failing module (this will always be zero or negative).

SEE ALSO

conf(5), OPENSSL_config(3), CONF_free(3), err(3)

HISTORY

CONF_modules_load_file and CONF_modules_load first appeared in OpenSSL 0.9.7.

Name

CRYPTO_set_ex_data and CRYPTO_get_ex_data — internal application specific data functions

Synopsis

```
#include  <openssl/crypto.h>
```

```
int CRYPTO_set_ex_data(CRYPTO_EX_DATA *r, int idx, void *arg);
```

```
void *CRYPTO_get_ex_data(CRYPTO_EX_DATA *r, int idx);
```

DESCRIPTION

Several OpenSSL structures can have application specific data attached to them. These functions are used internally by OpenSSL to manipulate application specific data attached to a specific structure.

These functions should only be used by applications to manipulate **CRYPTO_EX_DATA** structures passed to the **new_func()**, **free_func()** and **dup_func()** callbacks: as passed to **RSA_get_ex_new_index()** for example.

CRYPTO_set_ex_data() is used to set application specific data, the data is supplied in the **arg** parameter and its precise meaning is up to the application.

CRYPTO_get_ex_data() is used to retrieve application specific data. The data is returned to the application, this will be the same value as supplied to a previous **CRYPTO_set_ex_data()** call.

RETURN VALUES

CRYPTO_set_ex_data() returns 1 on success or 0 on failure.

CRYPTO_get_ex_data() returns the application data or 0 on failure. 0 may also be valid application data but currently it can only fail if given an invalid **idx** parameter.

On failure an error code can be obtained from ERR_get_error(3).

SEE ALSO

RSA_get_ex_new_index(3), DSA_get_ex_new_index(3), DH_get_ex_new_index(3)

HISTORY

CRYPTO_set_ex_data() and CRYPTO_get_ex_data() have been available since SSLeay 0.9.0.

Name

d2i_ASN1_OBJECT and i2d_ASN1_OBJECT — ASN1 OBJECT IDENTIFIER functions

Synopsis

```
#include  <openssl/objects.h>
```

```
ASN1_OBJECT *d2i_ASN1_OBJECT(ASN1_OBJECT **a, unsigned char **pp, long length);
int i2d_ASN1_OBJECT(ASN1_OBJECT *a, unsigned char **pp);
```

DESCRIPTION

These functions decode and encode an ASN1 OBJECT IDENTIFIER.

Othewise these behave in a similar way to d2i_X509() and i2d_X509() described in the d2i_X509(3) manual page.

SEE ALSO

d2i_X509(3)

HISTORY

TBA

Name

d2i_CMS_ContentInfo and i2d_CMS_ContentInfo — CMS ContentInfo functions

Synopsis

```
#include  <openssl/cms.h>
```

```
CMS_ContentInfo *d2i_CMS_ContentInfo(CMS_ContentInfo **a, unsigned char **pp, long length);
int i2d_CMS_ContentInfo(CMS_ContentInfo *a, unsigned char **pp);
```

DESCRIPTION

These functions decode and encode an CMS ContentInfo structure.

Otherwise they behave in a similar way to d2i_X509() and i2d_X509() described in the d2i_X509(3) manual page.

SEE ALSO

d2i_X509(3)

HISTORY

These functions were first added to OpenSSL 0.9.8

Name

d2i_DHparams and i2d_DHparams — PKCS#3 DH parameter functions.

Synopsis

```
#include  <openssl/dh.h>
```

```
DH *d2i_DHparams(DH **a, unsigned char **pp, long length);
int i2d_DHparams(DH *a, unsigned char **pp);
```

DESCRIPTION

These functions decode and encode PKCS#3 DH parameters using the DHparameter structure described in PKCS#3.

Othewise these behave in a similar way to d2i_X509() and i2d_X509() described in the d2i_X509(3) manual page.

SEE ALSO

d2i_X509(3)

HISTORY

TBA

Name

d2i_DSAPrivateKey, i2d_DSAPrivateKey, d2i_DSAPrivateKey, i2d_DSAPrivateKey, d2i_DSA_PUBKEY, i2d_DSA_PUBKEY, d2i_DSA_SIG and i2d_DSA_SIG — DSA key encoding and parsing functions.

Synopsis

```
#include  <openssl/dsa.h>
#include  <openssl/x509.h>
```

```
DSA * d2i_DSAPublicKey(DSA **a, const unsigned char **pp, long length);
```

```
int i2d_DSAPublicKey(const DSA *a, unsigned char **pp);
```

```
DSA * d2i_DSA_PUBKEY(DSA **a, const unsigned char **pp, long length);
```

```
int i2d_DSA_PUBKEY(const DSA *a, unsigned char **pp);
```

```
DSA * d2i_DSAPrivateKey(DSA **a, const unsigned char **pp, long length);
```

```
int i2d_DSAPrivateKey(const DSA *a, unsigned char **pp);
```

```
DSA * d2i_DSAparams(DSA **a, const unsigned char **pp, long length);
```

```
int i2d_DSAparams(const DSA *a, unsigned char **pp);
```

```
DSA * d2i_DSA_SIG(DSA_SIG **a, const unsigned char **pp, long length);
```

```
int i2d_DSA_SIG(const DSA_SIG *a, unsigned char **pp);
```

DESCRIPTION

d2i_DSAPublicKey() and i2d_DSAPublicKey() decode and encode the DSA public key components structure.

d2i_DSA_PUBKEY() and i2d_DSA_PUBKEY() decode and encode an DSA public key using a SubjectPublicKeyInfo (certificate public key) structure.

d2i_DSAPrivateKey(), i2d_DSAPrivateKey() decode and encode the DSA private key components.

d2i_DSAparams(), i2d_DSAparams() decode and encode the DSA parameters using a **Dss-Parms** structure as defined in RFC2459.

d2i_DSA_SIG(), i2d_DSA_SIG() decode and encode a DSA signature using a **Dss-Sig-Value** structure as defined in RFC2459.

The usage of all of these functions is similar to the d2i_X509() and i2d_X509() described in the d2i_X509(3) manual page.

NOTES

The **DSA** structure passed to the private key encoding functions should have all the private key components present.

The data encoded by the private key functions is unencrypted and therefore offers no private key security.

The **DSA_PUBKEY** functions should be used in preference to the **DSAPublicKey** functions when encoding public keys because they use a standard format.

The **DSAPublicKey** functions use an non standard format the actual data encoded depends on the value of the **write_params** field of the **a** key parameter. If **write_params** is zero then only the **pub_key** field is encoded as an **INTEGER**. If **write_params** is 1 then a **SEQUENCE** consisting of the **p**, **q**, **g** and **pub_key** respectively fields are encoded.

The **DSAPrivateKey** functions also use a non standard structure consiting consisting of a SEQUENCE containing the **p**, **q**, **g** and **pub_key** and **priv_key** fields respectively.

SEE ALSO

d2i_X509(3)

HISTORY

TBA

Name

i2d_ECPrivateKey and d2i_ECPrivate_key — Encode and decode functions for saving and reading EC_KEY structures

Synopsis

```
#include  <openssl/ec.h>
```

```
EC_KEY *d2i_ECPrivateKey(EC_KEY **key, const unsigned char **in, long len);
int i2d_ECPrivateKey(EC_KEY *key, unsigned char **out);
```

```
unsigned int EC_KEY_get_enc_flags(const EC_KEY *key);
void EC_KEY_set_enc_flags(EC_KEY *eckey, unsigned int flags);
```

DESCRIPTION

The ECPrivateKey encode and decode routines encode and parse an **EC_KEY** structure into a binary format (ASN.1 DER) and back again.

These functions are similar to the d2i_X509() functions, and you should refer to that page for a detailed description (see d2i_X509(3)).

The format of the external representation of the public key written by i2d_ECPrivateKey (such as whether it is stored in a compressed form or not) is described by the point_conversion_form. See EC_GROUP_copy(3) for a description of point_conversion_form.

When reading a private key encoded without an associated public key (e.g. if EC_PKEY_NO_PUBKEY has been used - see below), then d2i_ECPrivateKey generates the missing public key automatically. Private keys encoded without parameters (e.g. if EC_PKEY_NO_PARAMETERS has been used - see below) cannot be loaded using d2i_ECPrivateKey.

The functions EC_KEY_get_enc_flags and EC_KEY_set_enc_flags get and set the value of the encoding flags for the **key**. There are two encoding flags currently defined - EC_PKEY_NO_PARAMETERS and EC_PKEY_NO_PUBKEY. These flags define the behaviour of how the **key** is converted into ASN1 in a call to i2d_ECPrivateKey. If EC_PKEY_NO_PARAMETERS is set then the public parameters for the curve are not encoded along with the private key. If EC_PKEY_NO_PUBKEY is set then the public key is not encoded along with the private key.

RETURN VALUES

d2i_ECPrivateKey() returns a valid **EC_KEY** structure or **NULL** if an error occurs. The error code that can be obtained by ERR_get_error(3).

i2d_ECPrivateKey() returns the number of bytes successfully encoded or a negative value if an error occurs. The error code can be obtained by ERR_get_error(3).

EC_KEY_get_enc_flags returns the value of the current encoding flags for the EC_KEY.

SEE ALSO

crypto(3), ec(3), EC_GROUP_new(3), EC_GROUP_copy(3), EC_POINT_new(3), EC_POINT_add(3), EC_GFp_simple_method(3), d2i_ECPKParameters(3), d2i_ECPrivateKey(3)

Name

d2i_PKCS8PrivateKey_bio, d2i_PKCS8PrivateKey_fp, i2d_PKCS8PrivateKey_bio, i2d_PKCS8PrivateKey_fp, i2d_PKCS8PrivateKey_nid_bio and i2d_PKCS8PrivateKey_nid_fp — PKCS#8 format private key functions

Synopsis

```
#include  <openssl/evp.h>
```

```
EVP_PKEY *d2i_PKCS8PrivateKey_bio(BIO *bp, EVP_PKEY **x, pem_password_cb *cb, void *u);
EVP_PKEY *d2i_PKCS8PrivateKey_fp(FILE *fp, EVP_PKEY **x, pem_password_cb *cb, void *u);
```

```
int i2d_PKCS8PrivateKey_bio(BIO *bp, EVP_PKEY *x, const EVP_CIPHER *enc,
                                 char *kstr, int klen,
                                 pem_password_cb *cb, void *u);
```

```
int i2d_PKCS8PrivateKey_fp(FILE *fp, EVP_PKEY *x, const EVP_CIPHER *enc,
                                 char *kstr, int klen,
                                 pem_password_cb *cb, void *u);
```

```
int i2d_PKCS8PrivateKey_nid_bio(BIO *bp, EVP_PKEY *x, int nid,
                                 char *kstr, int klen,
                                 pem_password_cb *cb, void *u);
```

```
int i2d_PKCS8PrivateKey_nid_fp(FILE *fp, EVP_PKEY *x, int nid,
                                 char *kstr, int klen,
                                 pem_password_cb *cb, void *u);
```

DESCRIPTION

The PKCS#8 functions encode and decode private keys in PKCS#8 format using both PKCS#5 v1.5 and PKCS#5 v2.0 password based encryption algorithms.

Other than the use of DER as opposed to PEM these functions are identical to the corresponding **PEM** function as described in the pem(3) manual page.

NOTES

Before using these functions OpenSSL_add_all_algorithms(3) should be called to initialize the internal algorithm lookup tables otherwise errors about unknown algorithms will occur if an attempt is made to decrypt a private key.

These functions are currently the only way to store encrypted private keys using DER format.

Currently all the functions use BIOs or FILE pointers, there are no functions which work directly on memory: this can be readily worked around by converting the buffers to memory BIOs, see BIO_s_mem(3) for details.

SEE ALSO

pem(3)

Name

d2i_Private_key, d2i_AutoPrivateKey and i2d_PrivateKey — decode and encode functions for reading and saving EVP_PKEY structures.

Synopsis

```
#include  <openssl/evp.h>
```

```
EVP_PKEY *d2i_PrivateKey(int type, EVP_PKEY **a, const unsigned char **pp,
                         long length);
EVP_PKEY *d2i_AutoPrivateKey(EVP_PKEY **a, const unsigned char **pp,
                             long length);
int i2d_PrivateKey(EVP_PKEY *a, unsigned char **pp);
```

DESCRIPTION

d2i_PrivateKey() decodes a private key using algorithm **type**. It attempts to use any key specific format or PKCS#8 unencrypted PrivateKeyInfo format. The **type** parameter should be a public key algorithm constant such as **EVP_PKEY_RSA**. An error occurs if the decoded key does not match **type**.

d2i_AutoPrivateKey() is similar to d2i_PrivateKey() except it attempts to automatically detect the private key format.

i2d_PrivateKey() encodes **key**. It uses a key specific format or, if none is defined for that key type, PKCS#8 unencrypted PrivateKeyInfo format.

These functions are similar to the d2i_X509() functions, and you should refer to that page for a detailed description (see d2i_X509(3)).

NOTES

All these functions use DER format and unencrypted keys. Applications wishing to encrypt or decrypt private keys should use other functions such as d2i_PKC8PrivateKey() instead.

If the ***a** is not NULL when calling d2i_PrivateKey() or d2i_AutoPrivateKey() (i.e. an existing structure is being reused) and the key format is PKCS#8 then ***a** will be freed and replaced on a successful call.

RETURN VALUES

d2i_PrivateKey() and d2i_AutoPrivateKey() return a valid **EVP_KEY** structure or **NULL** if an error occurs. The error code can be obtained by calling ERR_get_error(3).

i2d_PrivateKey() returns the number of bytes successfully encoded or a negative value if an error occurs. The error code can be obtained by calling ERR_get_error(3).

SEE ALSO

crypto(3), d2i_PKCS8PrivateKey(3)

Name

d2i_RSAPublicKey, i2d_RSAPublicKey, d2i_RSAPrivateKey, i2d_RSAPrivateKey, d2i_RSA_PUBKEY, i2d_RSA_PUBKEY, i2d_Netscape_RSA and d2i_Netscape_RSA — RSA public and private key encoding functions.

Synopsis

```
#include  <openssl/rsa.h>
#include  <openssl/x509.h>
```

```
RSA * d2i_RSAPublicKey(RSA **a, const unsigned char **pp, long length);
```

```
int i2d_RSAPublicKey(RSA *a, unsigned char **pp);
```

```
RSA * d2i_RSA_PUBKEY(RSA **a, const unsigned char **pp, long length);
```

```
int i2d_RSA_PUBKEY(RSA *a, unsigned char **pp);
```

```
RSA * d2i_RSAPrivateKey(RSA **a, const unsigned char **pp, long length);
```

```
int i2d_RSAPrivateKey(RSA *a, unsigned char **pp);
```

```
int i2d_Netscape_RSA(RSA *a, unsigned char **pp, int (*cb)());
```

```
RSA * d2i_Netscape_RSA(RSA **a, const unsigned char **pp, long length, int (*cb)());
```

DESCRIPTION

d2i_RSAPublicKey() and i2d_RSAPublicKey() decode and encode a PKCS#1 RSAPublicKey structure.

d2i_RSA_PUBKEY() and i2d_RSA_PUBKEY() decode and encode an RSA public key using a SubjectPublicKeyInfo (certificate public key) structure.

d2i_RSAPrivateKey(), i2d_RSAPrivateKey() decode and encode a PKCS#1 RSAPrivateKey structure.

d2i_Netscape_RSA(), i2d_Netscape_RSA() decode and encode an RSA private key in NET format.

The usage of all of these functions is similar to the d2i_X509() and i2d_X509() described in the d2i_X509(3) manual page.

NOTES

The **RSA** structure passed to the private key encoding functions should have all the PKCS#1 private key components present.

The data encoded by the private key functions is unencrypted and therefore offers no private key security.

The NET format functions are present to provide compatibility with certain very old software. This format has some severe security weaknesses and should be avoided if possible.

SEE ALSO

d2i_X509(3)

HISTORY

TBA

Name

d2i_X509_ALGOR and i2d_X509_ALGOR — AlgorithmIdentifier functions.

Synopsis

```
#include  <openssl/x509.h>
```

```
X509_ALGOR *d2i_X509_ALGOR(X509_ALGOR **a, unsigned char **pp, long length);
int i2d_X509_ALGOR(X509_ALGOR *a, unsigned char **pp);
```

DESCRIPTION

These functions decode and encode an **X509_ALGOR** structure which is equivalent to the **AlgorithmIdentifier** structure.

Othewise these behave in a similar way to d2i_X509() and i2d_X509() described in the d2i_X509(3) manual page.

SEE ALSO

d2i_X509(3)

HISTORY

TBA

Name

d2i_X509_CRL, i2d_X509_CRL, d2i_X509_CRL_bio, d2i_509_CRL_fp, i2d_X509_CRL_bio and i2d_X509_CRL_fp — PKCS#10 certificate request functions.

Synopsis

```
#include  <openssl/x509.h>
```

```
X509_CRL *d2i_X509_CRL(X509_CRL **a, const unsigned char **pp, long length);
int i2d_X509_CRL(X509_CRL *a, unsigned char **pp);
```

```
X509_CRL *d2i_X509_CRL_bio(BIO *bp, X509_CRL **x);
X509_CRL *d2i_X509_CRL_fp(FILE *fp, X509_CRL **x);
```

```
int i2d_X509_CRL_bio(BIO *bp, X509_CRL *x);
int i2d_X509_CRL_fp(FILE *fp, X509_CRL *x);
```

DESCRIPTION

These functions decode and encode an X509 CRL (certificate revocation list).

Othewise the functions behave in a similar way to d2i_X509() and i2d_X509() described in the d2i_X509(3) manual page.

SEE ALSO

d2i_X509(3)

HISTORY

TBA

Name

d2i_X509_NAME and i2d_X509_NAME — X509_NAME encoding functions

Synopsis

```
#include  <openssl/x509.h>
```

```
X509_NAME *d2i_X509_NAME(X509_NAME **a, unsigned char **pp, long length);
int i2d_X509_NAME(X509_NAME *a, unsigned char **pp);
```

DESCRIPTION

These functions decode and encode an **X509_NAME** structure which is the same as the **Name** type defined in RFC2459 (and elsewhere) and used for example in certificate subject and issuer names.

Othewise the functions behave in a similar way to d2i_X509() and i2d_X509() described in the d2i_X509(3) manual page.

SEE ALSO

d2i_X509(3)

HISTORY

TBA

Name

d2i_X509, i2d_X509, d2i_X509_bio, d2i_X509_fp, i2d_X509_bio and i2d_X509_fp — X509 encode and decode functions

Synopsis

```
#include  <openssl/x509.h>
```

```
X509 *d2i_X509(X509 **px, const unsigned char **in, int len);
int i2d_X509(X509 *x, unsigned char **out);
```

```
X509 *d2i_X509_bio(BIO *bp, X509 **x);
X509 *d2i_X509_fp(FILE *fp, X509 **x);
```

```
int i2d_X509_bio(BIO *bp, X509 *x);
int i2d_X509_fp(FILE *fp, X509 *x);
```

DESCRIPTION

The X509 encode and decode routines encode and parse an **X509** structure, which represents an X509 certificate.

d2i_X509() attempts to decode **len** bytes at ***in**. If successful a pointer to the **X509** structure is returned. If an error occurred then **NULL** is returned. If **px** is not **NULL** then the returned structure is written to ***px**. If ***px** is not **NULL** then it is assumed that ***px** contains a valid **X509** structure and an attempt is made to reuse it. This "reuse" capability is present for historical compatibility but its use is **strongly discouraged** (see BUGS below, and the discussion in the RETURN VALUES section).

If the call is successful ***in** is incremented to the byte following the parsed data.

i2d_X509() encodes the structure pointed to by **x** into DER format. If **out** is not **NULL** is writes the DER encoded data to the buffer at ***out**, and increments it to point after the data just written. If the return value is negative an error occurred, otherwise it returns the length of the encoded data.

For OpenSSL 0.9.7 and later if ***out** is **NULL** memory will be allocated for a buffer and the encoded data written to it. In this case ***out** is not incremented and it points to the start of the data just written.

d2i_X509_bio() is similar to d2i_X509() except it attempts to parse data from BIO **bp**.

d2i_X509_fp() is similar to d2i_X509() except it attempts to parse data from FILE pointer **fp**.

i2d_X509_bio() is similar to i2d_X509() except it writes the encoding of the structure **x** to BIO **bp** and it returns 1 for success and 0 for failure.

i2d_X509_fp() is similar to i2d_X509() except it writes the encoding of the structure **x** to BIO **bp** and it returns 1 for success and 0 for failure.

NOTES

The letters **i** and **d** in for example **i2d_X509** stand for "internal" (that is an internal C structure) and "DER". So that **i2d_X509** converts from internal to DER.

The functions can also understand **BER** forms.

The actual X509 structure passed to i2d_X509() must be a valid populated **X509** structure it can **not** simply be fed with an empty structure such as that returned by X509_new().

The encoded data is in binary form and may contain embedded zeroes. Therefore any FILE pointers or BIOs should be opened in binary mode. Functions such as **strlen()** will **not** return the correct length of the encoded structure.

The ways that ***in** and ***out** are incremented after the operation can trap the unwary. See the **WARNINGS** section for some common errors.

The reason for the auto increment behaviour is to reflect a typical usage of ASN1 functions: after one structure is encoded or decoded another will processed after it.

EXAMPLES

Allocate and encode the DER encoding of an X509 structure:

```
int len;
unsigned char *buf, *p;

len = i2d_X509(x, NULL);

buf = OPENSSL_malloc(len);

if (buf == NULL)
       /* error */

p = buf;

i2d_X509(x, &p);
```

If you are using OpenSSL 0.9.7 or later then this can be simplified to:

```
int len;
unsigned char *buf;

buf = NULL;

len = i2d_X509(x, &buf);

if (len < 0)
       /* error */
```

Attempt to decode a buffer:

```
X509 *x;

unsigned char *buf, *p;

int len;

/* Something to setup buf and len */

p = buf;

x = d2i_X509(NULL, &p, len);

if (x == NULL)
   /* Some error */
```

Alternative technique:

```
X509 *x;

unsigned char *buf, *p;

int len;

/* Something to setup buf and len */

p = buf;
```

```
x = NULL;

if(!d2i_X509(&x, &p, len))
   /* Some error */
```

WARNINGS

The use of temporary variable is mandatory. A common mistake is to attempt to use a buffer directly as follows:

```
int len;
unsigned char *buf;

len = i2d_X509(x, NULL);

buf = OPENSSL_malloc(len);

if (buf == NULL)
       /* error */

i2d_X509(x, &buf);

/* Other stuff … */

OPENSSL_free(buf);
```

This code will result in **buf** apparently containing garbage because it was incremented after the call to point after the data just written. Also **buf** will no longer contain the pointer allocated by **OPENSSL_malloc()** and the subsequent call to **OPENSSL_free()** may well crash.

The auto allocation feature (setting buf to NULL) only works on OpenSSL 0.9.7 and later. Attempts to use it on earlier versions will typically cause a segmentation violation.

Another trap to avoid is misuse of the **xp** argument to **d2i_X509()**:

```
X509 *x;

if (!d2i_X509(&x, &p, len))
       /* Some error */
```

This will probably crash somewhere in **d2i_X509()**. The reason for this is that the variable **x** is uninitialized and an attempt will be made to interpret its (invalid) value as an **X509** structure, typically causing a segmentation violation. If **x** is set to NULL first then this will not happen.

BUGS

In some versions of OpenSSL the "reuse" behaviour of d2i_X509() when ***px** is valid is broken and some parts of the reused structure may persist if they are not present in the new one. As a result the use of this "reuse" behaviour is strongly discouraged.

i2d_X509() will not return an error in many versions of OpenSSL, if mandatory fields are not initialized due to a programming error then the encoded structure may contain invalid data or omit the fields entirely and will not be parsed by d2i_X509(). This may be fixed in future so code should not assume that i2d_X509() will always succeed.

RETURN VALUES

d2i_X509(), d2i_X509_bio() and d2i_X509_fp() return a valid **X509** structure or **NULL** if an error occurs. The error code that can be obtained by ERR_get_error(3). If the "reuse" capability has been used with a valid X509 structure being passed in via **px** then the object is not freed in the event of error but may be in a potentially invalid or inconsistent state.

i2d_X509() returns the number of bytes successfully encoded or a negative value if an error occurs. The error code can be obtained by ERR_get_error(3).

i2d_X509_bio() and i2d_X509_fp() return 1 for success and 0 if an error occurs The error code can be obtained by ERR_get_error(3).

SEE ALSO

ERR_get_error(3)

HISTORY

d2i_X509, i2d_X509, d2i_X509_bio, d2i_X509_fp, i2d_X509_bio and i2d_X509_fp are available in all versions of SSLeay and OpenSSL.

Name

d2i_X509_REQ, i2d_X509_REQ, d2i_X509_REQ_bio, d2i_X509_REQ_fp, i2d_X509_REQ_bio and i2d_X509_REQ_fp — PKCS#10 certificate request functions.

Synopsis

```
#include  <openssl/x509.h>
```

```
X509_REQ *d2i_X509_REQ(X509_REQ **a, const unsigned char **pp, long length);
int i2d_X509_REQ(X509_REQ *a, unsigned char **pp);
```

```
X509_REQ *d2i_X509_REQ_bio(BIO *bp, X509_REQ **x);
X509_REQ *d2i_X509_REQ_fp(FILE *fp, X509_REQ **x);
```

```
int i2d_X509_REQ_bio(BIO *bp, X509_REQ *x);
int i2d_X509_REQ_fp(FILE *fp, X509_REQ *x);
```

DESCRIPTION

These functions decode and encode a PKCS#10 certificate request.

Othewise these behave in a similar way to d2i_X509() and i2d_X509() described in the d2i_X509(3) manual page.

SEE ALSO

d2i_X509(3)

HISTORY

TBA

Name

d2i_X509_SIG and i2d_X509_SIG — DigestInfo functions.

Synopsis

```
#include  <openssl/x509.h>
```

```
X509_SIG *d2i_X509_SIG(X509_SIG **a, unsigned char **pp, long length);
int i2d_X509_SIG(X509_SIG *a, unsigned char **pp);
```

DESCRIPTION

These functions decode and encode an X509_SIG structure which is equivalent to the **DigestInfo** structure defined in PKCS#1 and PKCS#7.

Othewise these behave in a similar way to d2i_X509() and i2d_X509() described in the d2i_X509(3) manual page.

SEE ALSO

d2i_X509(3)

HISTORY

TBA

Name

DES_random_key, DES_set_key, DES_key_sched, DES_set_key_checked, DES_set_key_unchecked, DES_set_odd_parity, DES_is_weak_key, DES_ecb_encrypt, DES_ecb2_encrypt, DES_ecb3_encrypt, DES_ncbc_encrypt, DES_cfb_encrypt, DES_ofb_encrypt, DES_pcbc_encrypt, DES_cfb64_encrypt, DES_ofb64_encrypt, DES_xcbc_encrypt, DES_ede2_cbc_encrypt, DES_ede2_cfb64_encrypt, DES_ede2_ofb64_encrypt, DES_ede3_cbc_encrypt, DES_ede3_cbcm_encrypt, DES_ede3_cfb64_encrypt, DES_ede3_ofb64_encrypt, DES_cbc_cksum, DES_quad_cksum, DES_string_to_key, DES_string_to_2keys, DES_fcrypt, DES_crypt, DES_enc_read and DES_enc_write — DES encryption

Synopsis

```
#include  <openssl/des.h>
```

```
void DES_random_key(DES_cblock *ret);
```

```
int DES_set_key(const_DES_cblock *key, DES_key_schedule *schedule);
int DES_key_sched(const_DES_cblock *key, DES_key_schedule *schedule);
int DES_set_key_checked(const_DES_cblock *key,
       DES_key_schedule *schedule);
void DES_set_key_unchecked(const_DES_cblock *key,
       DES_key_schedule *schedule);
```

```
void DES_set_odd_parity(DES_cblock *key);
int DES_is_weak_key(const_DES_cblock *key);
```

```
void DES_ecb_encrypt(const_DES_cblock *input, DES_cblock *output,
       DES_key_schedule *ks, int enc);
void DES_ecb2_encrypt(const_DES_cblock *input, DES_cblock *output,
       DES_key_schedule *ks1, DES_key_schedule *ks2, int enc);
void DES_ecb3_encrypt(const_DES_cblock *input, DES_cblock *output,
       DES_key_schedule *ks1, DES_key_schedule *ks2,
       DES_key_schedule *ks3, int enc);
```

```
void DES_ncbc_encrypt(const unsigned char *input, unsigned char *output,
       long length, DES_key_schedule *schedule, DES_cblock *ivec,
       int enc);
void DES_cfb_encrypt(const unsigned char *in, unsigned char *out,
       int numbits, long length, DES_key_schedule *schedule,
       DES_cblock *ivec, int enc);
void DES_ofb_encrypt(const unsigned char *in, unsigned char *out,
       int numbits, long length, DES_key_schedule *schedule,
       DES_cblock *ivec);
void DES_pcbc_encrypt(const unsigned char *input, unsigned char *output,
       long length, DES_key_schedule *schedule, DES_cblock *ivec,
       int enc);
void DES_cfb64_encrypt(const unsigned char *in, unsigned char *out,
       long length, DES_key_schedule *schedule, DES_cblock *ivec,
       int *num, int enc);
void DES_ofb64_encrypt(const unsigned char *in, unsigned char *out,
       long length, DES_key_schedule *schedule, DES_cblock *ivec,
       int *num);
```

```
void DES_xcbc_encrypt(const unsigned char *input, unsigned char *output,
       long length, DES_key_schedule *schedule, DES_cblock *ivec,
       const_DES_cblock *inw, const_DES_cblock *outw, int enc);
```

```
void DES_ede2_cbc_encrypt(const unsigned char *input,
       unsigned char *output, long length, DES_key_schedule *ks1,
       DES_key_schedule *ks2, DES_cblock *ivec, int enc);
void DES_ede2_cfb64_encrypt(const unsigned char *in,
       unsigned char *out, long length, DES_key_schedule *ks1,
       DES_key_schedule *ks2, DES_cblock *ivec, int *num, int enc);
void DES_ede2_ofb64_encrypt(const unsigned char *in,
       unsigned char *out, long length, DES_key_schedule *ks1,
       DES_key_schedule *ks2, DES_cblock *ivec, int *num);
```

```
void DES_ede3_cbc_encrypt(const unsigned char *input,
       unsigned char *output, long length, DES_key_schedule *ks1,
       DES_key_schedule *ks2, DES_key_schedule *ks3, DES_cblock *ivec,
       int enc);
void DES_ede3_cbcm_encrypt(const unsigned char *in, unsigned char *out,
       long length, DES_key_schedule *ks1, DES_key_schedule *ks2,
       DES_key_schedule *ks3, DES_cblock *ivec1, DES_cblock *ivec2,
       int enc);
void DES_ede3_cfb64_encrypt(const unsigned char *in, unsigned char *out,
       long length, DES_key_schedule *ks1, DES_key_schedule *ks2,
       DES_key_schedule *ks3, DES_cblock *ivec, int *num, int enc);
void DES_ede3_ofb64_encrypt(const unsigned char *in, unsigned char *out,
       long length, DES_key_schedule *ks1,
       DES_key_schedule *ks2, DES_key_schedule *ks3,
       DES_cblock *ivec, int *num);
```

```
DES_LONG DES_cbc_cksum(const unsigned char *input, DES_cblock *output,
       long length, DES_key_schedule *schedule,
       const_DES_cblock *ivec);
DES_LONG DES_quad_cksum(const unsigned char *input, DES_cblock output[],
       long length, int out_count, DES_cblock *seed);
void DES_string_to_key(const char *str, DES_cblock *key);
void DES_string_to_2keys(const char *str, DES_cblock *key1,
       DES_cblock *key2);
```

```
char *DES_fcrypt(const char *buf, const char *salt, char *ret);
char *DES_crypt(const char *buf, const char *salt);
```

```
int DES_enc_read(int fd, void *buf, int len, DES_key_schedule *sched,
       DES_cblock *iv);
int DES_enc_write(int fd, const void *buf, int len,
       DES_key_schedule *sched, DES_cblock *iv);
```

DESCRIPTION

This library contains a fast implementation of the DES encryption algorithm.

There are two phases to the use of DES encryption. The first is the generation of a *DES_key_schedule* from a key, the second is the actual encryption. A DES key is of type *DES_cblock*. This type is consists of 8 bytes with odd parity. The least significant bit in each byte is the parity bit. The key schedule is an expanded form of the key; it is used to speed the encryption process.

DES_random_key() generates a random key. The PRNG must be seeded prior to using this function (see rand(3)). If the PRNG could not generate a secure key, 0 is returned.

Before a DES key can be used, it must be converted into the architecture dependent *DES_key_schedule* via the DES_set_key_checked() or DES_set_key_unchecked() function.

DES_set_key_checked() will check that the key passed is of odd parity and is not a week or semi-weak key. If the parity is wrong, then -1 is returned. If the key is a weak key, then -2 is returned. If an error is returned, the key schedule is not generated.

DES_set_key() works like DES_set_key_checked() if the *DES_check_key* flag is non-zero, otherwise like DES_set_key_unchecked(). These functions are available for compatibility; it is recommended to use a function that does not depend on a global variable.

DES_set_odd_parity() sets the parity of the passed *key* to odd.

DES_is_weak_key() returns 1 if the passed key is a weak key, 0 if it is ok.

The following routines mostly operate on an input and output stream of *DES_cblock*s.

DES_ecb_encrypt() is the basic DES encryption routine that encrypts or decrypts a single 8-byte *DES_cblock* in *electronic code book* (ECB) mode. It always transforms the input data, pointed to by *input*, into the output data, pointed to by the *output* argument. If the *encrypt* argument is non-zero (DES_ENCRYPT), the *input* (cleartext) is encrypted in to the *output* (ciphertext) using the key_schedule specified by the *schedule* argument, previously set via *DES_set_key*. If *encrypt* is zero (DES_DECRYPT), the *input* (now ciphertext) is decrypted into the *output* (now cleartext). Input and output may overlap. DES_ecb_encrypt() does not return a value.

DES_ecb3_encrypt() encrypts/decrypts the *input* block by using three-key Triple-DES encryption in ECB mode. This involves encrypting the input with *ks1*, decrypting with the key schedule *ks2*, and then encrypting with *ks3*. This routine greatly reduces the chances of brute force breaking of DES and has the advantage of if *ks1*, *ks2* and *ks3* are the same, it is equivalent to just encryption using ECB mode and *ks1* as the key.

The macro DES_ecb2_encrypt() is provided to perform two-key Triple-DES encryption by using *ks1* for the final encryption.

DES_ncbc_encrypt() encrypts/decrypts using the *cipher-block-chaining* (CBC) mode of DES. If the *encrypt* argument is non-zero, the routine cipher-block-chain encrypts the cleartext data pointed to by the *input* argument into the ciphertext pointed to by the *output* argument, using the key schedule provided by the *schedule* argument, and initialization vector provided by the *ivec* argument. If the *length* argument is not an integral multiple of eight bytes, the last block is copied to a temporary area and zero filled. The output is always an integral multiple of eight bytes.

DES_xcbc_encrypt() is RSA's DESX mode of DES. It uses *inw* and *outw* to 'whiten' the encryption. *inw* and *outw* are secret (unlike the iv) and are as such, part of the key. So the key is sort of 24 bytes. This is much better than CBC DES.

DES_ede3_cbc_encrypt() implements outer triple CBC DES encryption with three keys. This means that each DES operation inside the CBC mode is an `C=E(ks3,D(ks2,E(ks1,M)))`. This mode is used by SSL.

The DES_ede2_cbc_encrypt() macro implements two-key Triple-DES by reusing *ks1* for the final encryption. `C=E(ks1,D(ks2,E(ks1,M)))`. This form of Triple-DES is used by the RSAREF library.

DES_pcbc_encrypt() encrypt/decrypts using the propagating cipher block chaining mode used by Kerberos v4. Its parameters are the same as DES_ncbc_encrypt().

DES_cfb_encrypt() encrypt/decrypts using cipher feedback mode. This method takes an array of characters as input and outputs and array of characters. It does not require any padding to 8 character groups. Note: the *ivec* variable is changed and the new changed value needs to be passed to the next call to this function. Since this function runs a complete DES ECB encryption per *numbits*, this function is only suggested for use when sending small numbers of characters.

DES_cfb64_encrypt() implements CFB mode of DES with 64bit feedback. Why is this useful you ask? Because this routine will allow you to encrypt an arbitrary number of bytes, no 8 byte padding. Each call to this routine will encrypt the input bytes to output and then update ivec and num. num contains 'how far' we are though ivec. If this does not make much sense, read more about cfb mode of DES :-).

DES_ede3_cfb64_encrypt() and DES_ede2_cfb64_encrypt() is the same as DES_cfb64_encrypt() except that Triple-DES is used.

DES_ofb_encrypt() encrypts using output feedback mode. This method takes an array of characters as input and outputs and array of characters. It does not require any padding to 8 character groups. Note: the *ivec* variable is changed and the new changed value needs to be passed to the next call to this function. Since this function runs a complete DES ECB encryption per numbits, this function is only suggested for use when sending small numbers of characters.

DES_ofb64_encrypt() is the same as DES_cfb64_encrypt() using Output Feed Back mode.

DES_ede3_ofb64_encrypt() and DES_ede2_ofb64_encrypt() is the same as DES_ofb64_encrypt(), using Triple-DES.

The following functions are included in the DES library for compatibility with the MIT Kerberos library.

DES_cbc_cksum() produces an 8 byte checksum based on the input stream (via CBC encryption). The last 4 bytes of the checksum are returned and the complete 8 bytes are placed in *output*. This function is used by Kerberos v4. Other applications should use EVP_DigestInit(3) etc. instead.

DES_quad_cksum() is a Kerberos v4 function. It returns a 4 byte checksum from the input bytes. The algorithm can be iterated over the input, depending on *out_count*, 1, 2, 3 or 4 times. If *output* is non-NULL, the 8 bytes generated by each pass are written into *output*.

The following are DES-based transformations:

DES_fcrypt() is a fast version of the Unix crypt(3) function. This version takes only a small amount of space relative to other fast crypt() implementations. This is different to the normal crypt in that the third parameter is the buffer that the return value is written into. It needs to be at least 14 bytes long. This function is thread safe, unlike the normal crypt.

DES_crypt() is a faster replacement for the normal system crypt(). This function calls DES_fcrypt() with a static array passed as the third parameter. This emulates the normal non-thread safe semantics of crypt(3).

DES_enc_write() writes *len* bytes to file descriptor *fd* from buffer *buf*. The data is encrypted via *pcbc_encrypt* (default) using *sched* for the key and *iv* as a starting vector. The actual data send down *fd* consists of 4 bytes (in network byte order) containing the length of the following encrypted data. The encrypted data then follows, padded with random data out to a multiple of 8 bytes.

DES_enc_read() is used to read *len* bytes from file descriptor *fd* into buffer *buf*. The data being read from *fd* is assumed to have come from DES_enc_write() and is decrypted using *sched* for the key schedule and *iv* for the initial vector.

Warning: The data format used by DES_enc_write() and DES_enc_read() has a cryptographic weakness: When asked to write more than MAXWRITE bytes, DES_enc_write() will split the data into several chunks that are all encrypted using the same IV. So don't use these functions unless you are sure you know what you do (in which case you might not want to use them anyway). They cannot handle non-blocking sockets. DES_enc_read() uses an internal state and thus cannot be used on multiple files.

DES_rw_mode is used to specify the encryption mode to use with DES_enc_read() and DES_end_write(). If set to *DES_PCBC_MODE* (the default), DES_pcbc_encrypt is used. If set to *DES_CBC_MODE* DES_cbc_encrypt is used.

NOTES

Single-key DES is insecure due to its short key size. ECB mode is not suitable for most applications; see des_modes(7).

The evp(3) library provides higher-level encryption functions.

BUGS

DES_3cbc_encrypt() is flawed and must not be used in applications.

DES_cbc_encrypt() does not modify **ivec**; use DES_ncbc_encrypt() instead.

DES_cfb_encrypt() and DES_ofb_encrypt() operates on input of 8 bits. What this means is that if you set numbits to 12, and length to 2, the first 12 bits will come from the 1st input byte and the low half of the second input byte. The second 12 bits will have the low 8 bits taken from the 3rd input byte and the top 4 bits taken from the 4th input byte. The same holds for output. This function has been implemented this way because most people will be using a multiple of 8 and because once you get into pulling bytes input bytes apart things get ugly!

DES_string_to_key() is available for backward compatibility with the MIT library. New applications should use a cryptographic hash function. The same applies for DES_string_to_2key().

CONFORMING TO

ANSI X3.106

The **des** library was written to be source code compatible with the MIT Kerberos library.

SEE ALSO

crypt(3), des_modes(7), evp(3), rand(3)

HISTORY

In OpenSSL 0.9.7, all des_ functions were renamed to DES_ to avoid clashes with older versions of libdes. Compatibility des_ functions are provided for a short while, as well as crypt(). Declarations for these are in <openssl/des_old.h>. There is no DES_ variant for des_random_seed(). This will happen to other functions as well if they are deemed redundant (des_random_seed() just calls RAND_seed() and is present for backward compatibility only), buggy or already scheduled for removal.

des_cbc_cksum(), des_cbc_encrypt(), des_ecb_encrypt(), des_is_weak_key(), des_key_sched(), des_pcbc_encrypt(), des_quad_cksum(), des_random_key() and des_string_to_key() are available in the MIT Kerberos library; des_check_key_parity(), des_fixup_key_parity() and des_is_weak_key() are available in newer versions of that library.

des_set_key_checked() and des_set_key_unchecked() were added in OpenSSL 0.9.5.

des_generate_random_block(), des_init_random_number_generator(), des_new_random_key(), des_set_random_generator_seed() and des_set_sequence_number() and des_rand_data() are used in newer versions of Kerberos but are not implemented here.

des_random_key() generated cryptographically weak random data in SSLeay and in OpenSSL prior version 0.9.5, as well as in the original MIT library.

AUTHOR

Eric Young (eay@cryptsoft.com). Modified for the OpenSSL project (http://www.openssl.org).

Name

des_modes — the variants of DES and other crypto algorithms of OpenSSL

DESCRIPTION

Several crypto algorithms for OpenSSL can be used in a number of modes. Those are used for using block ciphers in a way similar to stream ciphers, among other things.

OVERVIEW

Electronic Codebook Mode (ECB)

Normally, this is found as the function *algorithm*_ecb_encrypt().

- 64 bits are enciphered at a time.
- The order of the blocks can be rearranged without detection.
- The same plaintext block always produces the same ciphertext block (for the same key) making it vulnerable to a 'dictionary attack'.
- An error will only affect one ciphertext block.

Cipher Block Chaining Mode (CBC)

Normally, this is found as the function *algorithm*_cbc_encrypt(). Be aware that des_cbc_encrypt() is not really DES CBC (it does not update the IV); use des_ncbc_encrypt() instead.

- a multiple of 64 bits are enciphered at a time.
- The CBC mode produces the same ciphertext whenever the same plaintext is encrypted using the same key and starting variable.
- The chaining operation makes the ciphertext blocks dependent on the current and all preceding plaintext blocks and therefore blocks can not be rearranged.
- The use of different starting variables prevents the same plaintext enciphering to the same ciphertext.
- An error will affect the current and the following ciphertext blocks.

Cipher Feedback Mode (CFB)

Normally, this is found as the function *algorithm*_cfb_encrypt().

- a number of bits (j) <= 64 are enciphered at a time.
- The CFB mode produces the same ciphertext whenever the same plaintext is encrypted using the same key and starting variable.
- The chaining operation makes the ciphertext variables dependent on the current and all preceding variables and therefore j-bit variables are chained together and can not be rearranged.
- The use of different starting variables prevents the same plaintext enciphering to the same ciphertext.
- The strength of the CFB mode depends on the size of k (maximal if j == k). In my implementation this is always the case.
- Selection of a small value for j will require more cycles through the encipherment algorithm per unit of plaintext and thus cause greater processing overheads.

- Only multiples of j bits can be enciphered.
- An error will affect the current and the following ciphertext variables.

Output Feedback Mode (OFB)

Normally, this is found as the function *algorithm*_ofb_encrypt().

- a number of bits (j) <= 64 are enciphered at a time.
- The OFB mode produces the same ciphertext whenever the same plaintext enciphered using the same key and starting variable. More over, in the OFB mode the same key stream is produced when the same key and start variable are used. Consequently, for security reasons a specific start variable should be used only once for a given key.
- The absence of chaining makes the OFB more vulnerable to specific attacks.
- The use of different start variables values prevents the same plaintext enciphering to the same ciphertext, by producing different key streams.
- Selection of a small value for j will require more cycles through the encipherment algorithm per unit of plaintext and thus cause greater processing overheads.
- Only multiples of j bits can be enciphered.
- OFB mode of operation does not extend ciphertext errors in the resultant plaintext output. Every bit error in the ciphertext causes only one bit to be in error in the deciphered plaintext.
- OFB mode is not self-synchronizing. If the two operation of encipherment and decipherment get out of synchronism, the system needs to be re-initialized.
- Each re-initialization should use a value of the start variable different from the start variable values used before with the same key. The reason for this is that an identical bit stream would be produced each time from the same parameters. This would be susceptible to a 'known plaintext' attack.

Triple ECB Mode

Normally, this is found as the function *algorithm*_ecb3_encrypt().

- Encrypt with key1, decrypt with key2 and encrypt with key3 again.
- As for ECB encryption but increases the key length to 168 bits. There are theoretic attacks that can be used that make the effective key length 112 bits, but this attack also requires 2^56 blocks of memory, not very likely, even for the NSA.
- If both keys are the same it is equivalent to encrypting once with just one key.
- If the first and last key are the same, the key length is 112 bits. There are attacks that could reduce the effective key strength to only slightly more than 56 bits, but these require a lot of memory.
- If all 3 keys are the same, this is effectively the same as normal ecb mode.

Triple CBC Mode

Normally, this is found as the function *algorithm*_ede3_cbc_encrypt().

- Encrypt with key1, decrypt with key2 and then encrypt with key3.
- As for CBC encryption but increases the key length to 168 bits with the same restrictions as for triple ecb mode.

NOTES

This text was been written in large parts by Eric Young in his original documentation for SSLeay, the predecessor of OpenSSL. In turn, he attributed it to:

```
AS 2805.5.2
Australian Standard
Electronic funds transfer - Requirements for interfaces,
Part 5.2: Modes of operation for an n-bit block cipher algorithm
Appendix A
```

SEE ALSO

blowfish(3), des(3), idea(3), rc2(3)

Name

dh — Diffie-Hellman key agreement

Synopsis

```
#include  <openssl/dh.h>
#include  <openssl/engine.h>
```

```
DH *   DH_new(void);
void   DH_free(DH *dh);
```

```
int    DH_size(const DH *dh);
```

```
DH *   DH_generate_parameters(int prime_len, int generator,
               void (*callback)(int, int, void *), void *cb_arg);
int    DH_check(const DH *dh, int *codes);
```

```
int    DH_generate_key(DH *dh);
int    DH_compute_key(unsigned char *key, BIGNUM *pub_key, DH *dh);
```

```
void DH_set_default_method(const DH_METHOD *meth);
const DH_METHOD *DH_get_default_method(void);
int DH_set_method(DH *dh, const DH_METHOD *meth);
DH *DH_new_method(ENGINE *engine);
const DH_METHOD *DH_OpenSSL(void);
```

```
int DH_get_ex_new_index(long argl, char *argp, int (*new_func)(),
            int (*dup_func)(), void (*free_func)());
int DH_set_ex_data(DH *d, int idx, char *arg);
char *DH_get_ex_data(DH *d, int idx);
```

```
DH *   d2i_DHparams(DH **a, unsigned char **pp, long length);
int    i2d_DHparams(const DH *a, unsigned char **pp);
```

```
int    DHparams_print_fp(FILE *fp, const DH *x);
int    DHparams_print(BIO *bp, const DH *x);
```

DESCRIPTION

These functions implement the Diffie-Hellman key agreement protocol. The generation of shared DH parameters is described in DH_generate_parameters(3); DH_generate_key(3) describes how to perform a key agreement.

The **DH** structure consists of several BIGNUM components.

```
struct
       {
       BIGNUM *p;              // prime number (shared)
       BIGNUM *g;              // generator of Z_p (shared)
       BIGNUM *priv_key;       // private DH value x
       BIGNUM *pub_key;        // public DH value g^x
       // ...
       };
DH
```

Note that DH keys may use non-standard **DH_METHOD** implementations, either directly or by the use of **ENGINE** modules. In some cases (eg. an ENGINE providing support for hardware-embedded keys), these BIGNUM values will not be used by the implementation or may be used for alternative data storage. For this reason, applications should generally avoid using DH structure elements directly and instead use API functions to query or modify keys.

SEE ALSO

dhparam(1), bn(3), dsa(3), err(3), rand(3), rsa(3), engine(3), DH_set_method(3), DH_new(3), DH_get_ex_new_index(3), DH_generate_parameters(3), DH_compute_key(3), d2i_DHparams(3), RSA_print(3)

Name

DH_generate_key and DH_compute_key — perform Diffie-Hellman key exchange

Synopsis

```
#include  <openssl/dh.h>
```

```
int DH_generate_key(DH *dh);
```

```
int DH_compute_key(unsigned char *key, BIGNUM *pub_key, DH *dh);
```

DESCRIPTION

DH_generate_key() performs the first step of a Diffie-Hellman key exchange by generating private and public DH values. By calling DH_compute_key(), these are combined with the other party's public value to compute the shared key.

DH_generate_key() expects **dh** to contain the shared parameters **dh->p** and **dh->g**. It generates a random private DH value unless **dh->priv_key** is already set, and computes the corresponding public value **dh->pub_key**, which can then be published.

DH_compute_key() computes the shared secret from the private DH value in **dh** and the other party's public value in **pub_key** and stores it in **key**. **key** must point to **DH_size(dh)** bytes of memory.

RETURN VALUES

DH_generate_key() returns 1 on success, 0 otherwise.

DH_compute_key() returns the size of the shared secret on success, -1 on error.

The error codes can be obtained by ERR_get_error(3).

SEE ALSO

dh(3), ERR_get_error(3), rand(3), DH_size(3)

HISTORY

DH_generate_key() and DH_compute_key() are available in all versions of SSLeay and OpenSSL.

Name

DH_generate_parameters and DH_check — generate and check Diffie-Hellman parameters

Synopsis

```
#include  <openssl/dh.h>
```

```
DH *DH_generate_parameters(int prime_len, int generator,
    void (*callback)(int, int, void *), void *cb_arg);
```

```
int DH_check(DH *dh, int *codes);
```

DESCRIPTION

DH_generate_parameters() generates Diffie-Hellman parameters that can be shared among a group of users, and returns them in a newly allocated **DH** structure. The pseudo-random number generator must be seeded prior to calling DH_generate_parameters().

prime_len is the length in bits of the safe prime to be generated. **generator** is a small number > 1, typically 2 or 5.

A callback function may be used to provide feedback about the progress of the key generation. If **callback** is not **NULL**, it will be called as described in BN_generate_prime(3) while a random prime number is generated, and when a prime has been found, **callback(3, 0, cb_arg)** is called.

DH_check() validates Diffie-Hellman parameters. It checks that **p** is a safe prime, and that **g** is a suitable generator. In the case of an error, the bit flags DH_CHECK_P_NOT_SAFE_PRIME or DH_NOT_SUITABLE_GENERATOR are set in ***codes**. DH_UNABLE_TO_CHECK_GENERATOR is set if the generator cannot be checked, i.e. it does not equal 2 or 5.

RETURN VALUES

DH_generate_parameters() returns a pointer to the DH structure, or NULL if the parameter generation fails. The error codes can be obtained by ERR_get_error(3).

DH_check() returns 1 if the check could be performed, 0 otherwise.

NOTES

DH_generate_parameters() may run for several hours before finding a suitable prime.

The parameters generated by DH_generate_parameters() are not to be used in signature schemes.

BUGS

If **generator** is not 2 or 5, **dh->g=generator** is not a usable generator.

SEE ALSO

dh(3), ERR_get_error(3), rand(3), DH_free(3)

HISTORY

DH_check() is available in all versions of SSLeay and OpenSSL. The **cb_arg** argument to DH_generate_parameters() was added in SSLeay 0.9.0.

In versions before OpenSSL 0.9.5, DH_CHECK_P_NOT_STRONG_PRIME is used instead of DH_CHECK_P_NOT_SAFE_PRIME.

Name

DH_get_ex_new_index, DH_set_ex_data and DH_get_ex_data — add application specific data to DH structures

Synopsis

```
#include  <openssl/dh.h>
```

```
int DH_get_ex_new_index(long argl, void *argp,
               CRYPTO_EX_new *new_func,
               CRYPTO_EX_dup *dup_func,
               CRYPTO_EX_free *free_func);
```

```
int DH_set_ex_data(DH *d, int idx, void *arg);
```

```
char *DH_get_ex_data(DH *d, int idx);
```

DESCRIPTION

These functions handle application specific data in DH structures. Their usage is identical to that of RSA_get_ex_new_index(), RSA_set_ex_data() and RSA_get_ex_data() as described in RSA_get_ex_new_index(3).

SEE ALSO

RSA_get_ex_new_index(3), dh(3)

HISTORY

DH_get_ex_new_index(), DH_set_ex_data() and DH_get_ex_data() are available since OpenSSL 0.9.5.

Name

DH_new and DH_free — allocate and free DH objects

Synopsis

```
#include  <openssl/dh.h>
```

```
DH* DH_new(void);
```

```
void DH_free(DH *dh);
```

DESCRIPTION

DH_new() allocates and initializes a **DH** structure.

DH_free() frees the **DH** structure and its components. The values are erased before the memory is returned to the system.

RETURN VALUES

If the allocation fails, DH_new() returns **NULL** and sets an error code that can be obtained by ERR_get_error(3). Otherwise it returns a pointer to the newly allocated structure.

DH_free() returns no value.

SEE ALSO

dh(3), ERR_get_error(3), DH_generate_parameters(3), DH_generate_key(3)

HISTORY

DH_new() and DH_free() are available in all versions of SSLeay and OpenSSL.

Name

DH_set_default_method, DH_get_default_method, DH_set_method, DH_new_method and DH_OpenSSL — select DH method

Synopsis

```
#include  <openssl/dh.h>
#include  <openssl/engine.h>
```

```
void DH_set_default_method(const DH_METHOD *meth);
```

```
const DH_METHOD *DH_get_default_method(void);
```

```
int DH_set_method(DH *dh, const DH_METHOD *meth);
```

```
DH *DH_new_method(ENGINE *engine);
```

```
const DH_METHOD *DH_OpenSSL(void);
```

DESCRIPTION

A **DH_METHOD** specifies the functions that OpenSSL uses for Diffie-Hellman operations. By modifying the method, alternative implementations such as hardware accelerators may be used. IMPORTANT: See the NOTES section for important information about how these DH API functions are affected by the use of **ENGINE** API calls.

Initially, the default DH_METHOD is the OpenSSL internal implementation, as returned by DH_OpenSSL().

DH_set_default_method() makes **meth** the default method for all DH structures created later. **NB**: This is true only whilst no ENGINE has been set as a default for DH, so this function is no longer recommended.

DH_get_default_method() returns a pointer to the current default DH_METHOD. However, the meaningfulness of this result is dependent on whether the ENGINE API is being used, so this function is no longer recommended.

DH_set_method() selects **meth** to perform all operations using the key **dh**. This will replace the DH_METHOD used by the DH key and if the previous method was supplied by an ENGINE, the handle to that ENGINE will be released during the change. It is possible to have DH keys that only work with certain DH_METHOD implementations (eg. from an ENGINE module that supports embedded hardware-protected keys), and in such cases attempting to change the DH_METHOD for the key can have unexpected results.

DH_new_method() allocates and initializes a DH structure so that **engine** will be used for the DH operations. If **engine** is NULL, the default ENGINE for DH operations is used, and if no default ENGINE is set, the DH_METHOD controlled by DH_set_default_method() is used.

THE DH_METHOD STRUCTURE

```
typedef struct dh_meth_st
{
    /* name of the implementation */
       const char *name;

/* generate private and public DH values for key agreement */
   int (*generate_key)(DH *dh);

/* compute shared secret */
   int (*compute_key)(unsigned char *key, BIGNUM *pub_key, DH *dh);

/* compute r = a ^ p mod m (May be NULL for some implementations) */
   int (*bn_mod_exp)(DH *dh, BIGNUM *r, BIGNUM *a, const BIGNUM *p,
                           const BIGNUM *m, BN_CTX *ctx,
                           BN_MONT_CTX *m_ctx);
/* called at DH_new */
   int (*init)(DH *dh);

/* called at DH_free */
   int (*finish)(DH *dh);

int flags;

char *app_data; /* ?? */

} DH_METHOD;
```

RETURN VALUES

DH_OpenSSL() and DH_get_default_method() return pointers to the respective **DH_METHOD**s.

DH_set_default_method() returns no value.

DH_set_method() returns non-zero if the provided **meth** was successfully set as the method for **dh** (including unloading the ENGINE handle if the previous method was supplied by an ENGINE).

DH_new_method() returns NULL and sets an error code that can be obtained by ERR_get_error(3) if the allocation fails. Otherwise it returns a pointer to the newly allocated structure.

NOTES

As of version 0.9.7, DH_METHOD implementations are grouped together with other algorithmic APIs (eg. RSA_METHOD, EVP_CIPHER, etc) in **ENGINE** modules. If a default ENGINE is specified for DH functionality using an ENGINE API function, that will override any DH defaults set using the DH API (ie. DH_set_default_method()). For this reason, the ENGINE API is the recommended way to control default implementations for use in DH and other cryptographic algorithms.

SEE ALSO

dh(3), DH_new(3)

HISTORY

DH_set_default_method(), DH_get_default_method(), DH_set_method(), DH_new_method() and DH_OpenSSL() were added in OpenSSL 0.9.4.

DH_set_default_openssl_method() and DH_get_default_openssl_method() replaced DH_set_default_method() and DH_get_default_method() respectively, and DH_set_method() and DH_new_method() were altered to use **ENGINE**s rather than **DH_METHOD**s during development of the engine version of OpenSSL 0.9.6. For 0.9.7, the handling of defaults in the ENGINE API was restructured so that this change was reversed, and behaviour of the other functions resembled more closely the previous behaviour. The behaviour of defaults in the ENGINE API now transparently overrides the behaviour of defaults in the DH API without requiring changing these function prototypes.

Name

DH_size — get Diffie-Hellman prime size

Synopsis

```
#include  <openssl/dh.h>
```

```
int DH_size(DH *dh);
```

DESCRIPTION

This function returns the Diffie-Hellman size in bytes. It can be used to determine how much memory must be allocated for the shared secret computed by DH_compute_key().

dh->p must not be **NULL**.

RETURN VALUE

The size in bytes.

SEE ALSO

dh(3), DH_generate_key(3)

HISTORY

DH_size() is available in all versions of SSLeay and OpenSSL.

Name

dsa — Digital Signature Algorithm

Synopsis

```
#include  <openssl/dsa.h>
#include  <openssl/engine.h>
```

```
DSA *   DSA_new(void);
void    DSA_free(DSA *dsa);
```

```
int     DSA_size(const DSA *dsa);
```

```
DSA *   DSA_generate_parameters(int bits, unsigned char *seed,
               int seed_len, int *counter_ret, unsigned long *h_ret,
               void (*callback)(int, int, void *), void *cb_arg);
```

```
DH *    DSA_dup_DH(const DSA *r);
```

```
int     DSA_generate_key(DSA *dsa);
```

```
int     DSA_sign(int dummy, const unsigned char *dgst, int len,
               unsigned char *sigret, unsigned int *siglen, DSA *dsa);
int     DSA_sign_setup(DSA *dsa, BN_CTX *ctx, BIGNUM **kinvp,
               BIGNUM **rp);
int     DSA_verify(int dummy, const unsigned char *dgst, int len,
               const unsigned char *sigbuf, int siglen, DSA *dsa);
```

```
void DSA_set_default_method(const DSA_METHOD *meth);
const DSA_METHOD *DSA_get_default_method(void);
int DSA_set_method(DSA *dsa, const DSA_METHOD *meth);
DSA *DSA_new_method(ENGINE *engine);
const DSA_METHOD *DSA_OpenSSL(void);
```

```
int DSA_get_ex_new_index(long argl, char *argp, int (*new_func)(),
            int (*dup_func)(), void (*free_func)());
int DSA_set_ex_data(DSA *d, int idx, char *arg);
char *DSA_get_ex_data(DSA *d, int idx);
```

```
DSA_SIG *DSA_SIG_new(void);
void    DSA_SIG_free(DSA_SIG *a);
int     i2d_DSA_SIG(const DSA_SIG *a, unsigned char **pp);
DSA_SIG *d2i_DSA_SIG(DSA_SIG **v, unsigned char **pp, long length);
```

```
DSA_SIG *DSA_do_sign(const unsigned char *dgst, int dlen, DSA *dsa);
int     DSA_do_verify(const unsigned char *dgst, int dgst_len,
            DSA_SIG *sig, DSA *dsa);
```

```
DSA *   d2i_DSAPublicKey(DSA **a, unsigned char **pp, long length);
DSA *   d2i_DSAPrivateKey(DSA **a, unsigned char **pp, long length);
DSA *   d2i_DSAparams(DSA **a, unsigned char **pp, long length);
int     i2d_DSAPublicKey(const DSA *a, unsigned char **pp);
int     i2d_DSAPrivateKey(const DSA *a, unsigned char **pp);
int     i2d_DSAparams(const DSA *a,unsigned char **pp);
```

```
int     DSAparams_print(BIO *bp, const DSA *x);
int     DSAparams_print_fp(FILE *fp, const DSA *x);
int     DSA_print(BIO *bp, const DSA *x, int off);
int     DSA_print_fp(FILE *bp, const DSA *x, int off);
```

DESCRIPTION

These functions implement the Digital Signature Algorithm (DSA). The generation of shared DSA parameters is described in DSA_generate_parameters(3); DSA_generate_key(3) describes how to generate a signature key. Signature generation and verification are described in DSA_sign(3).

The **DSA** structure consists of several BIGNUM components.

```
struct
       {
       BIGNUM *p;              // prime number (public)
       BIGNUM *q;              // 160-bit subprime, q | p-1 (public)
       BIGNUM *g;              // generator of subgroup (public)
       BIGNUM *priv_key;       // private key x
       BIGNUM *pub_key;        // public key y = g^x
       // ...
       }
DSA;
```

In public keys, **priv_key** is NULL.

Note that DSA keys may use non-standard **DSA_METHOD** implementations, either directly or by the use of **ENGINE** modules. In some cases (eg. an ENGINE providing support for hardware-embedded keys), these BIGNUM values will not be used by the implementation or may be used for alternative data storage. For this reason, applications should generally avoid using DSA structure elements directly and instead use API functions to query or modify keys.

CONFORMING TO

US Federal Information Processing Standard FIPS 186 (Digital Signature Standard, DSS), ANSI X9.30

SEE ALSO

bn(3), dh(3), err(3), rand(3), rsa(3), sha(3), engine(3), DSA_new(3), DSA_size(3), DSA_generate_parameters(3), DSA_dup_DH(3), DSA_generate_key(3), DSA_sign(3), DSA_set_method(3), DSA_get_ex_new_index(3), RSA_print(3)

Name

DSA_do_sign and DSA_do_verify — raw DSA signature operations

Synopsis

```
#include  <openssl/dsa.h>
```

```
DSA_SIG *DSA_do_sign(const unsigned char *dgst, int dlen, DSA *dsa);
```

```
int DSA_do_verify(const unsigned char *dgst, int dgst_len,
            DSA_SIG *sig, DSA *dsa);
```

DESCRIPTION

DSA_do_sign() computes a digital signature on the **len** byte message digest **dgst** using the private key **dsa** and returns it in a newly allocated **DSA_SIG** structure.

DSA_sign_setup(3) may be used to precompute part of the signing operation in case signature generation is time-critical.

DSA_do_verify() verifies that the signature **sig** matches a given message digest **dgst** of size **len**. **dsa** is the signer's public key.

RETURN VALUES

DSA_do_sign() returns the signature, NULL on error. DSA_do_verify() returns 1 for a valid signature, 0 for an incorrect signature and -1 on error. The error codes can be obtained by ERR_get_error(3).

SEE ALSO

dsa(3), ERR_get_error(3), rand(3), DSA_SIG_new(3), DSA_sign(3)

HISTORY

DSA_do_sign() and DSA_do_verify() were added in OpenSSL 0.9.3.

Name

DSA_dup_DH — create a DH structure out of DSA structure

Synopsis

```
#include  <openssl/dsa.h>
```

```
DH * DSA_dup_DH(const DSA *r);
```

DESCRIPTION

DSA_dup_DH() duplicates DSA parameters/keys as DH parameters/keys. q is lost during that conversion, but the resulting DH parameters contain its length.

RETURN VALUE

DSA_dup_DH() returns the new **DH** structure, and NULL on error. The error codes can be obtained by ERR_get_error(3).

NOTE

Be careful to avoid small subgroup attacks when using this.

SEE ALSO

dh(3), dsa(3), ERR_get_error(3)

HISTORY

DSA_dup_DH() was added in OpenSSL 0.9.4.

Name

DSA_generate_key — generate DSA key pair

Synopsis

```
#include  <openssl/dsa.h>
```

```
int DSA_generate_key(DSA *a);
```

DESCRIPTION

DSA_generate_key() expects **a** to contain DSA parameters. It generates a new key pair and stores it in **a->pub_key** and **a->priv_key**.

The PRNG must be seeded prior to calling DSA_generate_key().

RETURN VALUE

DSA_generate_key() returns 1 on success, 0 otherwise. The error codes can be obtained by ERR_get_error(3).

SEE ALSO

dsa(3), ERR_get_error(3), rand(3), DSA_generate_parameters(3)

HISTORY

DSA_generate_key() is available since SSLeay 0.8.

Name

DSA_generate_parameters — generate DSA parameters

Synopsis

```
#include  <openssl/dsa.h>
```

```
DSA *DSA_generate_parameters(int bits, unsigned char *seed,
               int seed_len, int *counter_ret, unsigned long *h_ret,
               void (*callback)(int, int, void *), void *cb_arg);
```

DESCRIPTION

DSA_generate_parameters() generates primes p and q and a generator g for use in the DSA.

bits is the length of the prime to be generated; the DSS allows a maximum of 1024 bits.

If **seed** is **NULL** or **seed_len** < 20, the primes will be generated at random. Otherwise, the seed is used to generate them. If the given seed does not yield a prime q, a new random seed is chosen.

DSA_generate_parameters() places the iteration count in ***counter_ret** and a counter used for finding a generator in ***h_ret**, unless these are **NULL**.

A callback function may be used to provide feedback about the progress of the key generation. If **callback** is not **NULL**, it will be called as follows:

- When a candidate for q is generated, **callback(0, m++, cb_arg)** is called (m is 0 for the first candidate).
- When a candidate for q has passed a test by trial division, **callback(1, -1, cb_arg)** is called. While a candidate for q is tested by Miller-Rabin primality tests, **callback(1, i, cb_arg)** is called in the outer loop (once for each witness that confirms that the candidate may be prime); i is the loop counter (starting at 0).
- When a prime q has been found, **callback(2, 0, cb_arg)** and **callback(3, 0, cb_arg)** are called.
- Before a candidate for p (other than the first) is generated and tested, **callback(0, counter, cb_arg)** is called.
- When a candidate for p has passed the test by trial division, **callback(1, -1, cb_arg)** is called. While it is tested by the Miller-Rabin primality test, **callback(1, i, cb_arg)** is called in the outer loop (once for each witness that confirms that the candidate may be prime). i is the loop counter (starting at 0).
- When p has been found, **callback(2, 1, cb_arg)** is called.
- When the generator has been found, **callback(3, 1, cb_arg)** is called.

RETURN VALUE

DSA_generate_parameters() returns a pointer to the DSA structure, or **NULL** if the parameter generation fails. The error codes can be obtained by ERR_get_error(3).

BUGS

Seed lengths > 20 are not supported.

SEE ALSO

dsa(3), ERR_get_error(3), rand(3), DSA_free(3)

HISTORY

DSA_generate_parameters() appeared in SSLeay 0.8. The **cb_arg** argument was added in SSLeay 0.9.0. In versions up to OpenSSL 0.9.4, **callback(1, ...)** was called in the inner loop of the Miller-Rabin test whenever it reached the squaring step (the parameters to **callback** did not reveal how many witnesses had been tested); since OpenSSL 0.9.5, **callback(1, ...)** is called as in BN_is_prime(3), i.e. once for each witness. =cut

Name

DSA_get_ex_new_index, DSA_set_ex_data and DSA_get_ex_data — add application specific data to DSA structures

Synopsis

```
#include  <openssl/dsa.h>
```

```
int DSA_get_ex_new_index(long argl, void *argp,
               CRYPTO_EX_new *new_func,
               CRYPTO_EX_dup *dup_func,
               CRYPTO_EX_free *free_func);
```

```
int DSA_set_ex_data(DSA *d, int idx, void *arg);
```

```
char *DSA_get_ex_data(DSA *d, int idx);
```

DESCRIPTION

These functions handle application specific data in DSA structures. Their usage is identical to that of RSA_get_ex_new_index(), RSA_set_ex_data() and RSA_get_ex_data() as described in RSA_get_ex_new_index(3).

SEE ALSO

RSA_get_ex_new_index(3), dsa(3)

HISTORY

DSA_get_ex_new_index(), DSA_set_ex_data() and DSA_get_ex_data() are available since OpenSSL 0.9.5.

Name

DSA_new and DSA_free — allocate and free DSA objects

Synopsis

```
#include  <openssl/dsa.h>
```

```
DSA* DSA_new(void);
```

```
void DSA_free(DSA *dsa);
```

DESCRIPTION

DSA_new() allocates and initializes a **DSA** structure. It is equivalent to calling DSA_new_method(NULL).

DSA_free() frees the **DSA** structure and its components. The values are erased before the memory is returned to the system.

RETURN VALUES

If the allocation fails, DSA_new() returns **NULL** and sets an error code that can be obtained by ERR_get_error(3). Otherwise it returns a pointer to the newly allocated structure.

DSA_free() returns no value.

SEE ALSO

dsa(3), ERR_get_error(3), DSA_generate_parameters(3), DSA_generate_key(3)

HISTORY

DSA_new() and DSA_free() are available in all versions of SSLeay and OpenSSL.

Name

DSA_set_default_method, DSA_get_default_method, DSA_set_method, DSA_new_method and DSA_OpenSSL — select DSA method

Synopsis

```
#include  <openssl/dsa.h>
#include  <openssl/engine.h>
```

```
void DSA_set_default_method(const DSA_METHOD *meth);
```

```
const DSA_METHOD *DSA_get_default_method(void);
```

```
int DSA_set_method(DSA *dsa, const DSA_METHOD *meth);
```

```
DSA *DSA_new_method(ENGINE *engine);
```

```
DSA_METHOD *DSA_OpenSSL(void);
```

DESCRIPTION

A **DSA_METHOD** specifies the functions that OpenSSL uses for DSA operations. By modifying the method, alternative implementations such as hardware accelerators may be used. IMPORTANT: See the NOTES section for important information about how these DSA API functions are affected by the use of **ENGINE** API calls.

Initially, the default DSA_METHOD is the OpenSSL internal implementation, as returned by DSA_OpenSSL().

DSA_set_default_method() makes **meth** the default method for all DSA structures created later. **NB**: This is true only whilst no ENGINE has been set as a default for DSA, so this function is no longer recommended.

DSA_get_default_method() returns a pointer to the current default DSA_METHOD. However, the meaningfulness of this result is dependent on whether the ENGINE API is being used, so this function is no longer recommended.

DSA_set_method() selects **meth** to perform all operations using the key **rsa**. This will replace the DSA_METHOD used by the DSA key and if the previous method was supplied by an ENGINE, the handle to that ENGINE will be released during the change. It is possible to have DSA keys that only work with certain DSA_METHOD implementations (eg. from an ENGINE module that supports embedded hardware-protected keys), and in such cases attempting to change the DSA_METHOD for the key can have unexpected results.

DSA_new_method() allocates and initializes a DSA structure so that **engine** will be used for the DSA operations. If **engine** is NULL, the default engine for DSA operations is used, and if no default ENGINE is set, the DSA_METHOD controlled by DSA_set_default_method() is used.

THE DSA_METHOD STRUCTURE

struct { /* name of the implementation */ const char *name;

```
/* sign */
   DSA_SIG *(*dsa_do_sign)(const unsigned char *dgst, int dlen,
                            DSA *dsa);

/* pre-compute k^-1 and r */
   int (*dsa_sign_setup)(DSA *dsa, BN_CTX *ctx_in, BIGNUM **kinvp,
                            BIGNUM **rp);

/* verify */
   int (*dsa_do_verify)(const unsigned char *dgst, int dgst_len,
                            DSA_SIG *sig, DSA *dsa);

/* compute rr = a1^p1 * a2^p2 mod m (May be NULL for some
                                     implementations) */
   int (*dsa_mod_exp)(DSA *dsa, BIGNUM *rr, BIGNUM *a1, BIGNUM *p1,
                            BIGNUM *a2, BIGNUM *p2, BIGNUM *m,
                            BN_CTX *ctx, BN_MONT_CTX *in_mont);

/* compute r = a ^ p mod m (May be NULL for some implementations) */
   int (*bn_mod_exp)(DSA *dsa, BIGNUM *r, BIGNUM *a,
                            const BIGNUM *p, const BIGNUM *m,
                            BN_CTX *ctx, BN_MONT_CTX *m_ctx);

/* called at DSA_new */
   int (*init)(DSA *DSA);

/* called at DSA_free */
   int (*finish)(DSA *DSA);

int flags;

char *app_data; /* ?? */

} DSA_METHOD;
```

RETURN VALUES

DSA_OpenSSL() and DSA_get_default_method() return pointers to the respective **DSA_METHOD**s.

DSA_set_default_method() returns no value.

DSA_set_method() returns non-zero if the provided **meth** was successfully set as the method for **dsa** (including unloading the ENGINE handle if the previous method was supplied by an ENGINE).

DSA_new_method() returns NULL and sets an error code that can be obtained by ERR_get_error(3) if the allocation fails. Otherwise it returns a pointer to the newly allocated structure.

NOTES

As of version 0.9.7, DSA_METHOD implementations are grouped together with other algorithmic APIs (eg. RSA_METHOD, EVP_CIPHER, etc) in **ENGINE** modules. If a default ENGINE is specified for DSA functionality using an ENGINE API function, that will override any DSA defaults set using the DSA API (ie. DSA_set_default_method()). For this reason, the ENGINE API is the recommended way to control default implementations for use in DSA and other cryptographic algorithms.

SEE ALSO

dsa(3), DSA_new(3)

HISTORY

DSA_set_default_method(), DSA_get_default_method(), DSA_set_method(), DSA_new_method() and DSA_OpenSSL() were added in OpenSSL 0.9.4.

DSA_set_default_openssl_method() and DSA_get_default_openssl_method() replaced DSA_set_default_method() and DSA_get_default_method() respectively, and DSA_set_method() and DSA_new_method() were altered to use **ENGINE**s rather than **DSA_METHOD**s during development of the engine version of OpenSSL 0.9.6. For 0.9.7, the handling of defaults in the ENGINE API was restructured so that this change was reversed, and behaviour of the other functions resembled more closely the previous behaviour. The behaviour of defaults in the ENGINE API now transparently overrides the behaviour of defaults in the DSA API without requiring changing these function prototypes.

Name

DSA_SIG_new and DSA_SIG_free — allocate and free DSA signature objects

Synopsis

```
#include  <openssl/dsa.h>
```

```
DSA_SIG *DSA_SIG_new(void);
```

```
void   DSA_SIG_free(DSA_SIG *a);
```

DESCRIPTION

DSA_SIG_new() allocates and initializes a **DSA_SIG** structure.

DSA_SIG_free() frees the **DSA_SIG** structure and its components. The values are erased before the memory is returned to the system.

RETURN VALUES

If the allocation fails, DSA_SIG_new() returns **NULL** and sets an error code that can be obtained by ERR_get_error(3). Otherwise it returns a pointer to the newly allocated structure.

DSA_SIG_free() returns no value.

SEE ALSO

dsa(3), ERR_get_error(3), DSA_do_sign(3)

HISTORY

DSA_SIG_new() and DSA_SIG_free() were added in OpenSSL 0.9.3.

Name

DSA_sign, DSA_sign_setup and DSA_verify — DSA signatures

Synopsis

```
#include  <openssl/dsa.h>

int    DSA_sign(int type, const unsigned char *dgst, int len,
               unsigned char *sigret, unsigned int *siglen, DSA *dsa);

int    DSA_sign_setup(DSA *dsa, BN_CTX *ctx, BIGNUM **kinvp,
               BIGNUM **rp);

int    DSA_verify(int type, const unsigned char *dgst, int len,
               unsigned char *sigbuf, int siglen, DSA *dsa);
```

DESCRIPTION

DSA_sign() computes a digital signature on the **len** byte message digest **dgst** using the private key **dsa** and places its ASN.1 DER encoding at **sigret**. The length of the signature is places in ***siglen**. **sigret** must point to DSA_size(**dsa**) bytes of memory.

DSA_sign_setup() may be used to precompute part of the signing operation in case signature generation is time-critical. It expects **dsa** to contain DSA parameters. It places the precomputed values in newly allocated **BIGNUM**s at ***kinvp** and ***rp**, after freeing the old ones unless ***kinvp** and ***rp** are NULL. These values may be passed to DSA_sign() in **dsa->kinv** and **dsa->r**. **ctx** is a pre-allocated **BN_CTX** or NULL.

DSA_verify() verifies that the signature **sigbuf** of size **siglen** matches a given message digest **dgst** of size **len**. **dsa** is the signer's public key.

The **type** parameter is ignored.

The PRNG must be seeded before DSA_sign() (or DSA_sign_setup()) is called.

RETURN VALUES

DSA_sign() and DSA_sign_setup() return 1 on success, 0 on error. DSA_verify() returns 1 for a valid signature, 0 for an incorrect signature and -1 on error. The error codes can be obtained by ERR_get_error(3).

CONFORMING TO

US Federal Information Processing Standard FIPS 186 (Digital Signature Standard, DSS), ANSI X9.30

SEE ALSO

dsa(3), ERR_get_error(3), rand(3), DSA_do_sign(3)

HISTORY

DSA_sign() and DSA_verify() are available in all versions of SSLeay. DSA_sign_setup() was added in SSLeay 0.8.

Name

DSA_size — get DSA signature size

Synopsis

```
#include  <openssl/dsa.h>
```

```
int DSA_size(const DSA *dsa);
```

DESCRIPTION

This function returns the size of an ASN.1 encoded DSA signature in bytes. It can be used to determine how much memory must be allocated for a DSA signature.

dsa->q must not be **NULL**.

RETURN VALUE

The size in bytes.

SEE ALSO

dsa(3), DSA_sign(3)

HISTORY

DSA_size() is available in all versions of SSLeay and OpenSSL.

Name

ecdsa — Elliptic Curve Digital Signature Algorithm

Synopsis

```
#include  <openssl/ecdsa.h>
```

```
ECDSA_SIG*     ECDSA_SIG_new(void);
void           ECDSA_SIG_free(ECDSA_SIG *sig);
int            i2d_ECDSA_SIG(const ECDSA_SIG *sig, unsigned char **pp);
ECDSA_SIG*     d2i_ECDSA_SIG(ECDSA_SIG **sig, const unsigned char **pp,
               long len);
```

```
ECDSA_SIG*     ECDSA_do_sign(const unsigned char *dgst, int dgst_len,
                       EC_KEY *eckey);
ECDSA_SIG*     ECDSA_do_sign_ex(const unsigned char *dgst, int dgstlen,
                       const BIGNUM *kinv, const BIGNUM *rp,
                       EC_KEY *eckey);
int            ECDSA_do_verify(const unsigned char *dgst, int dgst_len,
                       const ECDSA_SIG *sig, EC_KEY* eckey);
int            ECDSA_sign_setup(EC_KEY *eckey, BN_CTX *ctx,
                       BIGNUM **kinv, BIGNUM **rp);
int            ECDSA_sign(int type, const unsigned char *dgst,
                       int dgstlen, unsigned char *sig,
                       unsigned int *siglen, EC_KEY *eckey);
int            ECDSA_sign_ex(int type, const unsigned char *dgst,
                       int dgstlen, unsigned char *sig,
                       unsigned int *siglen, const BIGNUM *kinv,
                       const BIGNUM *rp, EC_KEY *eckey);
int            ECDSA_verify(int type, const unsigned char *dgst,
                       int dgstlen, const unsigned char *sig,
                       int siglen, EC_KEY *eckey);
int            ECDSA_size(const EC_KEY *eckey);
```

```
const ECDSA_METHOD*    ECDSA_OpenSSL(void);
void           ECDSA_set_default_method(const ECDSA_METHOD *meth);
const ECDSA_METHOD*    ECDSA_get_default_method(void);
int            ECDSA_set_method(EC_KEY *eckey,const ECDSA_METHOD *meth);
```

```
int            ECDSA_get_ex_new_index(long argl, void *argp,
                       CRYPTO_EX_new *new_func,
                       CRYPTO_EX_dup *dup_func,
                       CRYPTO_EX_free *free_func);
int            ECDSA_set_ex_data(EC_KEY *d, int idx, void *arg);
void*          ECDSA_get_ex_data(EC_KEY *d, int idx);
```

DESCRIPTION

The **ECDSA_SIG** structure consists of two BIGNUMs for the r and s value of a ECDSA signature (see X9.62 or FIPS 186-2).

```
struct
       {
       BIGNUM *r;
       BIGNUM *s;
} ECDSA_SIG;
```

ECDSA_SIG_new() allocates a new **ECDSA_SIG** structure (note: this function also allocates the BIGNUMs) and initialize it.

ECDSA_SIG_free() frees the **ECDSA_SIG** structure **sig**.

i2d_ECDSA_SIG() creates the DER encoding of the ECDSA signature **sig** and writes the encoded signature to ***pp** (note: if **pp** is NULL **i2d_ECDSA_SIG** returns the expected length in bytes of the DER encoded signature). **i2d_ECDSA_SIG** returns the length of the DER encoded signature (or 0 on error).

d2i_ECDSA_SIG() decodes a DER encoded ECDSA signature and returns the decoded signature in a newly allocated **ECDSA_SIG** structure. ***sig** points to the buffer containing the DER encoded signature of size **len**.

ECDSA_size() returns the maximum length of a DER encoded ECDSA signature created with the private EC key **eckey**.

ECDSA_sign_setup() may be used to precompute parts of the signing operation. **eckey** is the private EC key and **ctx** is a pointer to **BN_CTX** structure (or NULL). The precomputed values or returned in **kinv** and **rp** and can be used in a later call to **ECDSA_sign_ex** or **ECDSA_do_sign_ex**.

ECDSA_sign() is wrapper function for ECDSA_sign_ex with **kinv** and **rp** set to NULL.

ECDSA_sign_ex() computes a digital signature of the **dgstlen** bytes hash value **dgst** using the private EC key **eckey** and the optional pre-computed values **kinv** and **rp**. The DER encoded signatures is stored in **sig** and it's length is returned in **sig_len**. Note: **sig** must point to **ECDSA_size** bytes of memory. The parameter **type** is ignored.

ECDSA_verify() verifies that the signature in **sig** of size **siglen** is a valid ECDSA signature of the hash value **dgst** of size **dgstlen** using the public key **eckey**. The parameter **type** is ignored.

ECDSA_do_sign() is wrapper function for ECDSA_do_sign_ex with **kinv** and **rp** set to NULL.

ECDSA_do_sign_ex() computes a digital signature of the **dgst_len** bytes hash value **dgst** using the private key **eckey** and the optional pre-computed values **kinv** and **rp**. The signature is returned in a newly allocated **ECDSA_SIG** structure (or NULL on error).

ECDSA_do_verify() verifies that the signature **sig** is a valid ECDSA signature of the hash value **dgst** of size **dgst_len** using the public key **eckey**.

RETURN VALUES

ECDSA_size() returns the maximum length signature or 0 on error.

ECDSA_sign_setup() and ECDSA_sign() return 1 if successful or 0 on error.

ECDSA_verify() and ECDSA_do_verify() return 1 for a valid signature, 0 for an invalid signature and -1 on error. The error codes can be obtained by ERR_get_error(3).

EXAMPLES

Creating a ECDSA signature of given SHA-1 hash value using the named curve secp192k1.

First step: create a EC_KEY object (note: this part is **not** ECDSA specific)

```
int        ret;
ECDSA_SIG *sig;
EC_KEY    *eckey;
eckey = EC_KEY_new_by_curve_name(NID_secp192k1);
if (eckey == NULL)
       {
       /* error */
       }
if (!EC_KEY_generate_key(eckey))
       {
       /* error */
       }
```

Second step: compute the ECDSA signature of a SHA-1 hash value using **ECDSA_do_sign**

```
sig = ECDSA_do_sign(digest, 20, eckey);
if (sig == NULL)
```

```
       {
       /* error */
       }
```

or using **ECDSA_sign**

```
unsigned char *buffer, *pp;
int            buf_len;
buf_len = ECDSA_size(eckey);
buffer  = OPENSSL_malloc(buf_len);
pp = buffer;
if (!ECDSA_sign(0, dgst, dgstlen, pp, &buf_len, eckey);
       {
       /* error */
       }
```

Third step: verify the created ECDSA signature using **ECDSA_do_verify**

```
ret = ECDSA_do_verify(digest, 20, sig, eckey);
```

or using **ECDSA_verify**

```
ret = ECDSA_verify(0, digest, 20, buffer, buf_len, eckey);
```

and finally evaluate the return value:

```
if (ret == -1)
       {
       /* error */
       }
else if (ret == 0)
       {
       /* incorrect signature */
       }
else   /* ret == 1 */
       {
       /* signature ok */
       }
```

CONFORMING TO

ANSI X9.62, US Federal Information Processing Standard FIPS 186-2 (Digital Signature Standard, DSS)

SEE ALSO

dsa(3), rsa(3)

HISTORY

The ecdsa implementation was first introduced in OpenSSL 0.9.8

AUTHOR

Nils Larsch for the OpenSSL project (http://www.openssl.org).

Name

engine — ENGINE cryptographic module support

Synopsis

```
#include  <openssl/engine.h>
```

```
ENGINE *ENGINE_get_first(void);
ENGINE *ENGINE_get_last(void);
ENGINE *ENGINE_get_next(ENGINE *e);
ENGINE *ENGINE_get_prev(ENGINE *e);
```

```
int ENGINE_add(ENGINE *e);
int ENGINE_remove(ENGINE *e);
```

```
ENGINE *ENGINE_by_id(const char *id);
```

```
int ENGINE_init(ENGINE *e);
int ENGINE_finish(ENGINE *e);
```

```
void ENGINE_load_openssl(void);
void ENGINE_load_dynamic(void);
#ifndef OPENSSL_NO_STATIC_ENGINE
void ENGINE_load_4758cca(void);
void ENGINE_load_aep(void);
void ENGINE_load_atalla(void);
void ENGINE_load_chil(void);
void ENGINE_load_cswift(void);
void ENGINE_load_gmp(void);
void ENGINE_load_nuron(void);
void ENGINE_load_sureware(void);
void ENGINE_load_ubsec(void);
#endif
void ENGINE_load_cryptodev(void);
void ENGINE_load_builtin_engines(void);
```

```
void ENGINE_cleanup(void);
```

```
ENGINE *ENGINE_get_default_RSA(void);
ENGINE *ENGINE_get_default_DSA(void);
ENGINE *ENGINE_get_default_ECDH(void);
ENGINE *ENGINE_get_default_ECDSA(void);
ENGINE *ENGINE_get_default_DH(void);
ENGINE *ENGINE_get_default_RAND(void);
ENGINE *ENGINE_get_cipher_engine(int nid);
ENGINE *ENGINE_get_digest_engine(int nid);
```

```
int ENGINE_set_default_RSA(ENGINE *e);
int ENGINE_set_default_DSA(ENGINE *e);
int ENGINE_set_default_ECDH(ENGINE *e);
int ENGINE_set_default_ECDSA(ENGINE *e);
int ENGINE_set_default_DH(ENGINE *e);
int ENGINE_set_default_RAND(ENGINE *e);
int ENGINE_set_default_ciphers(ENGINE *e);
int ENGINE_set_default_digests(ENGINE *e);
int ENGINE_set_default_string(ENGINE *e, const char *list);
```

```
int ENGINE_set_default(ENGINE *e, unsigned int flags);
```

```
unsigned int ENGINE_get_table_flags(void);
void ENGINE_set_table_flags(unsigned int flags);
```

```
int ENGINE_register_RSA(ENGINE *e);
void ENGINE_unregister_RSA(ENGINE *e);
void ENGINE_register_all_RSA(void);
int ENGINE_register_DSA(ENGINE *e);
void ENGINE_unregister_DSA(ENGINE *e);
void ENGINE_register_all_DSA(void);
int ENGINE_register_ECDH(ENGINE *e);
void ENGINE_unregister_ECDH(ENGINE *e);
void ENGINE_register_all_ECDH(void);
int ENGINE_register_ECDSA(ENGINE *e);
void ENGINE_unregister_ECDSA(ENGINE *e);
void ENGINE_register_all_ECDSA(void);
int ENGINE_register_DH(ENGINE *e);
void ENGINE_unregister_DH(ENGINE *e);
void ENGINE_register_all_DH(void);
int ENGINE_register_RAND(ENGINE *e);
void ENGINE_unregister_RAND(ENGINE *e);
void ENGINE_register_all_RAND(void);
int ENGINE_register_STORE(ENGINE *e);
void ENGINE_unregister_STORE(ENGINE *e);
void ENGINE_register_all_STORE(void);
int ENGINE_register_ciphers(ENGINE *e);
void ENGINE_unregister_ciphers(ENGINE *e);
void ENGINE_register_all_ciphers(void);
int ENGINE_register_digests(ENGINE *e);
void ENGINE_unregister_digests(ENGINE *e);
void ENGINE_register_all_digests(void);
int ENGINE_register_complete(ENGINE *e);
int ENGINE_register_all_complete(void);
```

```
int ENGINE_ctrl(ENGINE *e, int cmd, long i, void *p, void (*f)(void));
int ENGINE_cmd_is_executable(ENGINE *e, int cmd);
int ENGINE_ctrl_cmd(ENGINE *e, const char *cmd_name,
        long i, void *p, void (*f)(void), int cmd_optional);
int ENGINE_ctrl_cmd_string(ENGINE *e, const char *cmd_name, const char *arg,
        int cmd_optional);
```

```
int ENGINE_set_ex_data(ENGINE *e, int idx, void *arg);
void *ENGINE_get_ex_data(const ENGINE *e, int idx);
```

```
int ENGINE_get_ex_new_index(long argl, void *argp, CRYPTO_EX_new *new_func,
        CRYPTO_EX_dup *dup_func, CRYPTO_EX_free *free_func);
```

```
ENGINE *ENGINE_new(void);
int ENGINE_free(ENGINE *e);
int ENGINE_up_ref(ENGINE *e);
```

```
int ENGINE_set_id(ENGINE *e, const char *id);
int ENGINE_set_name(ENGINE *e, const char *name);
int ENGINE_set_RSA(ENGINE *e, const RSA_METHOD *rsa_meth);
int ENGINE_set_DSA(ENGINE *e, const DSA_METHOD *dsa_meth);
int ENGINE_set_ECDH(ENGINE *e, const ECDH_METHOD *dh_meth);
int ENGINE_set_ECDSA(ENGINE *e, const ECDSA_METHOD *dh_meth);
int ENGINE_set_DH(ENGINE *e, const DH_METHOD *dh_meth);
int ENGINE_set_RAND(ENGINE *e, const RAND_METHOD *rand_meth);
int ENGINE_set_STORE(ENGINE *e, const STORE_METHOD *rand_meth);
int ENGINE_set_destroy_function(ENGINE *e, ENGINE_GEN_INT_FUNC_PTR destroy_f);
int ENGINE_set_init_function(ENGINE *e, ENGINE_GEN_INT_FUNC_PTR init_f);
int ENGINE_set_finish_function(ENGINE *e, ENGINE_GEN_INT_FUNC_PTR finish_f);
int ENGINE_set_ctrl_function(ENGINE *e, ENGINE_CTRL_FUNC_PTR ctrl_f);
int ENGINE_set_load_privkey_function(ENGINE *e, ENGINE_LOAD_KEY_PTR loadpriv_f);
int ENGINE_set_load_pubkey_function(ENGINE *e, ENGINE_LOAD_KEY_PTR loadpub_f);
int ENGINE_set_ciphers(ENGINE *e, ENGINE_CIPHERS_PTR f);
int ENGINE_set_digests(ENGINE *e, ENGINE_DIGESTS_PTR f);
int ENGINE_set_flags(ENGINE *e, int flags);
int ENGINE_set_cmd_defns(ENGINE *e, const ENGINE_CMD_DEFN *defns);
```

```
const char *ENGINE_get_id(const ENGINE *e);
const char *ENGINE_get_name(const ENGINE *e);
const RSA_METHOD *ENGINE_get_RSA(const ENGINE *e);
const DSA_METHOD *ENGINE_get_DSA(const ENGINE *e);
const ECDH_METHOD *ENGINE_get_ECDH(const ENGINE *e);
const ECDSA_METHOD *ENGINE_get_ECDSA(const ENGINE *e);
const DH_METHOD *ENGINE_get_DH(const ENGINE *e);
const RAND_METHOD *ENGINE_get_RAND(const ENGINE *e);
const STORE_METHOD *ENGINE_get_STORE(const ENGINE *e);
ENGINE_GEN_INT_FUNC_PTR ENGINE_get_destroy_function(const ENGINE *e);
ENGINE_GEN_INT_FUNC_PTR ENGINE_get_init_function(const ENGINE *e);
ENGINE_GEN_INT_FUNC_PTR ENGINE_get_finish_function(const ENGINE *e);
ENGINE_CTRL_FUNC_PTR ENGINE_get_ctrl_function(const ENGINE *e);
ENGINE_LOAD_KEY_PTR ENGINE_get_load_privkey_function(const ENGINE *e);
ENGINE_LOAD_KEY_PTR ENGINE_get_load_pubkey_function(const ENGINE *e);
ENGINE_CIPHERS_PTR ENGINE_get_ciphers(const ENGINE *e);
ENGINE_DIGESTS_PTR ENGINE_get_digests(const ENGINE *e);
const EVP_CIPHER *ENGINE_get_cipher(ENGINE *e, int nid);
const EVP_MD *ENGINE_get_digest(ENGINE *e, int nid);
int ENGINE_get_flags(const ENGINE *e);
const ENGINE_CMD_DEFN *ENGINE_get_cmd_defns(const ENGINE *e);
```

```
EVP_PKEY *ENGINE_load_private_key(ENGINE *e, const char *key_id,
    UI_METHOD *ui_method, void *callback_data);
EVP_PKEY *ENGINE_load_public_key(ENGINE *e, const char *key_id,
    UI_METHOD *ui_method, void *callback_data);
```

```
void ENGINE_add_conf_module(void);
```

DESCRIPTION

These functions create, manipulate, and use cryptographic modules in the form of **ENGINE** objects. These objects act as containers for implementations of cryptographic algorithms, and support a reference-counted mechanism to allow them to be dynamically loaded in and out of the running application.

The cryptographic functionality that can be provided by an **ENGINE** implementation includes the following abstractions;

```
RSA_METHOD - for providing alternative RSA implementations
DSA_METHOD, DH_METHOD, RAND_METHOD, ECDH_METHOD, ECDSA_METHOD,
      STORE_METHOD - similarly for other OpenSSL APIs
EVP_CIPHER - potentially multiple cipher algorithms (indexed by 'nid')
EVP_DIGEST - potentially multiple hash algorithms (indexed by 'nid')
key-loading - loading public and/or private EVP_PKEY keys
```

Reference counting and handles

Due to the modular nature of the ENGINE API, pointers to ENGINEs need to be treated as handles - ie. not only as pointers, but also as references to the underlying ENGINE object. Ie. one should obtain a new reference when making copies of an ENGINE pointer if the copies will be used (and released) independently.

ENGINE objects have two levels of reference-counting to match the way in which the objects are used. At the most basic level, each ENGINE pointer is inherently a **structural** reference - a structural reference is required to use the pointer value at all, as this kind of reference is a guarantee that the structure can not be deallocated until the reference is released.

However, a structural reference provides no guarantee that the ENGINE is initiliased and able to use any of its cryptographic implementations. Indeed it's quite possible that most ENGINEs will not initialise at all in typical environments, as ENGINEs are typically used to support specialised hardware. To use an ENGINE's functionality, you need a **functional** reference. This kind of reference can be considered a specialised form of structural reference, because each functional reference implicitly contains a structural reference as well - however to avoid difficult-to-find programming bugs, it is recommended to treat the two kinds of reference independently. If you have a functional reference to an ENGINE, you have a guarantee that the ENGINE has been initialised ready to perform cryptographic operations and will remain uninitialised until after you have released your reference.

Structural references

This basic type of reference is used for instantiating new ENGINEs, iterating across OpenSSL's internal linked-list of loaded ENGINEs, reading information about an ENGINE, etc. Essentially a structural reference is sufficient if you only need to query or manipulate the data of an ENGINE implementation rather than use its functionality.

The ENGINE_new() function returns a structural reference to a new (empty) ENGINE object. There are other ENGINE API functions that return structural references such as; ENGINE_by_id(), ENGINE_get_first(), ENGINE_get_last(), ENGINE_get_next(), ENGINE_get_prev(). All structural references should be released by a corresponding to call to the ENGINE_free() function - the ENGINE object itself will only actually be cleaned up and deallocated when the last structural reference is released.

It should also be noted that many ENGINE API function calls that accept a structural reference will internally obtain another reference - typically this happens whenever the supplied ENGINE will be needed by OpenSSL after the function has returned. Eg. the function to add a new ENGINE to OpenSSL's internal list is ENGINE_add() - if this function returns success, then OpenSSL will have stored a new structural reference internally so the caller is still responsible for freeing their own reference with ENGINE_free() when they are finished with it. In a similar way, some functions will automatically release the structural reference passed to it if part of the function's job is to do so. Eg. the ENGINE_get_next() and ENGINE_get_prev() functions are used for iterating across the internal ENGINE list - they will return a new structural reference to the next (or previous) ENGINE in the list or NULL if at the end (or beginning) of the list, but in either case the structural reference passed to the function is released on behalf of the caller.

To clarify a particular function's handling of references, one should always consult that function's documentation "man" page, or failing that the openssl/engine.h header file includes some hints.

Functional references

As mentioned, functional references exist when the cryptographic functionality of an ENGINE is required to be available. A functional reference can be obtained in one of two ways; from an existing structural reference to the required ENGINE, or by asking OpenSSL for the default operational ENGINE for a given cryptographic purpose.

To obtain a functional reference from an existing structural reference, call the ENGINE_init() function. This returns zero if the ENGINE was not already operational and couldn't be successfully initialised (eg. lack of system drivers, no special hardware attached, etc), otherwise it will return non-zero to indicate that the ENGINE is now operational and will have allocated a new **functional** reference to the ENGINE. All functional references are released by calling ENGINE_finish() (which removes the implicit structural reference as well).

The second way to get a functional reference is by asking OpenSSL for a default implementation for a given task, eg. by ENGINE_get_default_RSA(), ENGINE_get_default_cipher_engine(), etc. These are discussed in the next section, though they are not usually required by application programmers as they are used automatically when creating and using the relevant algorithm-specific types in OpenSSL, such as RSA, DSA, EVP_CIPHER_CTX, etc.

Default implementations

For each supported abstraction, the ENGINE code maintains an internal table of state to control which implementations are available for a given abstraction and which should be used by default. These implementations are registered in the tables and indexed by an 'nid' value, because abstractions like EVP_CIPHER and EVP_DIGEST support many distinct algorithms and modes, and ENGINEs can support arbitrarily many of them. In the case of other abstractions like RSA, DSA, etc, there is only one "algorithm" so all implementations implicitly register using the same 'nid' index.

When a default ENGINE is requested for a given abstraction/algorithm/mode, (eg. when calling RSA_new_method(NULL)), a "get_default" call will be made to the ENGINE subsystem to process the corresponding state table and return a functional reference to an initialised ENGINE whose implementation should be used. If no ENGINE should (or can) be used, it will return NULL and the caller will operate with a NULL ENGINE handle - this usually equates to using the conventional software implementation. In the latter case, OpenSSL will from then on behave the way it used to before the ENGINE API existed.

Each state table has a flag to note whether it has processed this "get_default" query since the table was last modified, because to process this question it must iterate across all the registered ENGINEs in the table trying to initialise each of them in turn, in case one of them is operational. If it returns a functional reference to an ENGINE, it will also cache another reference to speed up processing future queries (without needing to iterate across the table). Likewise, it will cache a NULL response if no ENGINE was available so that future queries won't repeat the same iteration unless the state table changes. This behaviour can also be changed; if the ENGINE_TABLE_FLAG_NOINIT flag is set (using ENGINE_set_table_flags()), no attempted initialisations will take place, instead the only way for the state table to return a non-NULL ENGINE to the "get_default" query will be if one is expressly set in the table. Eg. ENGINE_set_default_RSA() does the same job as ENGINE_register_RSA() except that it also sets the state table's cached response for the "get_default" query. In the case of abstractions like EVP_CIPHER, where implementations are indexed by 'nid', these flags and cached-responses are distinct for each 'nid' value.

Application requirements

This section will explain the basic things an application programmer should support to make the most useful elements of the ENGINE functionality available to the user. The first thing to consider is whether the programmer wishes to make alternative ENGINE modules available to the application and user. OpenSSL maintains an internal linked list of "visible" ENGINEs from which it has to operate - at start-up, this list is empty and in fact if an application does not call any ENGINE API calls and it uses static linking against openssl, then the resulting application binary will not contain any alternative ENGINE code at all. So the first consideration is whether any/all available ENGINE implementations should be made visible to OpenSSL - this is controlled by calling the various "load" functions, eg.

```
/* Make the "dynamic" ENGINE available */
void ENGINE_load_dynamic(void);
/* Make the CryptoSwift hardware acceleration support available */
void ENGINE_load_cswift(void);
/* Make support for nCipher's "CHIL" hardware available */
void ENGINE_load_chil(void);
…
/* Make ALL ENGINE implementations bundled with OpenSSL available */
void ENGINE_load_builtin_engines(void);
```

Having called any of these functions, ENGINE objects would have been dynamically allocated and populated with these implementations and linked into OpenSSL's internal linked list. At this point it is important to mention an important API function;

```
void ENGINE_cleanup(void);
```

If no ENGINE API functions are called at all in an application, then there are no inherent memory leaks to worry about from the ENGINE functionality, however if any ENGINEs are loaded, even if they are never registered or used, it is necessary to use the ENGINE_cleanup() function to correspondingly cleanup before program exit, if the caller wishes to avoid memory leaks. This mechanism uses an internal callback registration table so that any ENGINE API functionality that knows it requires cleanup can register its cleanup details to be called during ENGINE_cleanup(). This approach allows ENGINE_cleanup() to clean up after any ENGINE functionality at all that your program uses, yet doesn't automatically create linker dependencies to all possible ENGINE functionality - only the cleanup callbacks required by the functionality you do use will be required by the linker.

The fact that ENGINEs are made visible to OpenSSL (and thus are linked into the program and loaded into memory at run-time) does not mean they are "registered" or called into use by OpenSSL automatically - that behaviour is something for the application to control. Some applications will want to allow the user to specify exactly which ENGINE they want used if any is to be used at all. Others may prefer to load all support and have OpenSSL automatically use at run-time any ENGINE that is able to successfully initialise - ie. to assume that this corresponds to acceleration hardware attached to the machine or some such thing. There are probably numerous other ways in which applications may prefer to handle things, so we will simply illustrate the consequences as they apply to a couple of simple cases and leave developers to consider these and the source code to openssl's builtin utilities as guides.

Using a specific ENGINE implementation

Here we'll assume an application has been configured by its user or admin to want to use the "ACME" ENGINE if it is available in the version of OpenSSL the application was compiled with. If it is available, it should be used by default for all RSA, DSA, and symmetric cipher operation, otherwise OpenSSL should use its builtin software as per usual. The following code illustrates how to approach this;

```
ENGINE *e;
const char *engine_id = "ACME";
ENGINE_load_builtin_engines();
e = ENGINE_by_id(engine_id);
if(!e)
```

```
    /* the engine isn't available */
    return;
if(!ENGINE_init(e)) {
    /* the engine couldn't initialise, release 'e' */
    ENGINE_free(e);
    return;
}
if(!ENGINE_set_default_RSA(e))
    /* This should only happen when 'e' can't initialise, but the previous
     * statement suggests it did. */
    abort();
ENGINE_set_default_DSA(e);
ENGINE_set_default_ciphers(e);
/* Release the functional reference from ENGINE_init() */
ENGINE_finish(e);
/* Release the structural reference from ENGINE_by_id() */
ENGINE_free(e);
```

Automatically using builtin ENGINE implementations

Here we'll assume we want to load and register all ENGINE implementations bundled with OpenSSL, such that for any cryptographic algorithm required by OpenSSL - if there is an ENGINE that implements it and can be initialise, it should be used. The following code illustrates how this can work;

```
/* Load all bundled ENGINEs into memory and make them visible */
ENGINE_load_builtin_engines();
/* Register all of them for every algorithm they collectively implement */
ENGINE_register_all_complete();
```

That's all that's required. Eg. the next time OpenSSL tries to set up an RSA key, any bundled ENGINEs that implement RSA_METHOD will be passed to ENGINE_init() and if any of those succeed, that ENGINE will be set as the default for RSA use from then on.

Advanced configuration support

There is a mechanism supported by the ENGINE framework that allows each ENGINE implementation to define an arbitrary set of configuration "commands" and expose them to OpenSSL and any applications based on OpenSSL. This mechanism is entirely based on the use of name-value pairs and assumes ASCII input (no unicode or UTF for now!), so it is ideal if applications want to provide a transparent way for users to provide arbitrary configuration "directives" directly to such ENGINEs. It is also possible for the application to dynamically interrogate the loaded ENGINE implementations for the names, descriptions, and input flags of their available "control commands", providing a more flexible configuration scheme. However, if the user is expected to know which ENGINE device he/she is using (in the case of specialised hardware, this goes without saying) then applications may not need to concern themselves with discovering the supported control commands and simply prefer to pass settings into ENGINEs exactly as they are provided by the user.

Before illustrating how control commands work, it is worth mentioning what they are typically used for. Broadly speaking there are two uses for control commands; the first is to provide the necessary details to the implementation (which may know nothing at all specific to the host system) so that it can be initialised for use. This could include the path to any driver or config files it needs to load, required network addresses, smart-card identifiers, passwords to initialise protected devices, logging information, etc etc. This class of commands typically needs to be passed to an ENGINE **before** attempting to initialise it, ie. before calling ENGINE_init(). The other class of commands consist of settings or operations that tweak certain behaviour or cause certain operations to take place, and these commands may work either before or after ENGINE_init(), or in some cases both. ENGINE implementations should provide indications of this in the descriptions attached to builtin control commands and/or in external product documentation.

Issuing control commands to an ENGINE

Let's illustrate by example; a function for which the caller supplies the name of the ENGINE it wishes to use, a table of string-pairs for use before initialisation, and another table for use after initialisation. Note that the string-pairs used for control commands consist of a command "name" followed by the command "parameter" - the parameter could be NULL in some cases but the name can not. This function should initialise the ENGINE (issuing the "pre" commands beforehand and the "post" commands afterwards) and set it as the default for everything except RAND and then return a boolean success or failure.

```
int generic_load_engine_fn(const char *engine_id,
                           const char **pre_cmds, int pre_num,
                           const char **post_cmds, int post_num)
{
    ENGINE *e = ENGINE_by_id(engine_id);
    if(!e) return 0;
    while(pre_num--) {
        if(!ENGINE_ctrl_cmd_string(e, pre_cmds[0], pre_cmds[1], 0)) {
            fprintf(stderr, "Failed command (%s - %s:%s)\n", engine_id,
                pre_cmds[0], pre_cmds[1] ? pre_cmds[1] : "(NULL)");
            ENGINE_free(e);
            return 0;
        }
        pre_cmds += 2;
    }
    if(!ENGINE_init(e)) {
        fprintf(stderr, "Failed initialisation\n");
        ENGINE_free(e);
        return 0;
    }
    /* ENGINE_init() returned a functional reference, so free the structural
     * reference from ENGINE_by_id(). */
    ENGINE_free(e);
    while(post_num--) {
        if(!ENGINE_ctrl_cmd_string(e, post_cmds[0], post_cmds[1], 0)) {
            fprintf(stderr, "Failed command (%s - %s:%s)\n", engine_id,
                post_cmds[0], post_cmds[1] ? post_cmds[1] : "(NULL)");
            ENGINE_finish(e);
            return 0;
        }
        post_cmds += 2;
    }
    ENGINE_set_default(e, ENGINE_METHOD_ALL & ~ENGINE_METHOD_RAND);
    /* Success */
    return 1;
}
```

Note that ENGINE_ctrl_cmd_string() accepts a boolean argument that can relax the semantics of the function - if set non-zero it will only return failure if the ENGINE supported the given command name but failed while executing it, if the ENGINE doesn't support the command name it will simply return success without doing anything. In this case we assume the user is only supplying commands specific to the given ENGINE so we set this to FALSE.

Discovering supported control commands

It is possible to discover at run-time the names, numerical-ids, descriptions and input parameters of the control commands supported by an ENGINE using a structural reference. Note that some control commands are defined by OpenSSL itself and it will intercept and handle these control commands on behalf of the ENGINE, ie. the ENGINE's ctrl() handler is not used for the control command. openssl/engine.h defines an index, ENGINE_CMD_BASE, that all control commands implemented by ENGINEs should be numbered from. Any command value lower than this symbol is considered a "generic" command is handled directly by the OpenSSL core routines.

It is using these "core" control commands that one can discover the the control commands implemented by a given ENGINE, specifically the commands;

```
#define ENGINE_HAS_CTRL_FUNCTION                10
#define ENGINE_CTRL_GET_FIRST_CMD_TYPE          11
#define ENGINE_CTRL_GET_NEXT_CMD_TYPE           12
#define ENGINE_CTRL_GET_CMD_FROM_NAME           13
#define ENGINE_CTRL_GET_NAME_LEN_FROM_CMD       14
#define ENGINE_CTRL_GET_NAME_FROM_CMD           15
#define ENGINE_CTRL_GET_DESC_LEN_FROM_CMD       16
```

```
#define ENGINE_CTRL_GET_DESC_FROM_CMD           17
#define ENGINE_CTRL_GET_CMD_FLAGS               18
```

Whilst these commands are automatically processed by the OpenSSL framework code, they use various properties exposed by each ENGINE to process these queries. An ENGINE has 3 properties it exposes that can affect how this behaves; it can supply a ctrl() handler, it can specify ENGINE_FLAGS_MANUAL_CMD_CTRL in the ENGINE's flags, and it can expose an array of control command descriptions. If an ENGINE specifies the ENGINE_FLAGS_MANUAL_CMD_CTRL flag, then it will simply pass all these "core" control commands directly to the ENGINE's ctrl() handler (and thus, it must have supplied one), so it is up to the ENGINE to reply to these "discovery" commands itself. If that flag is not set, then the OpenSSL framework code will work with the following rules;

```
if no ctrl() handler supplied;
    ENGINE_HAS_CTRL_FUNCTION returns FALSE (zero),
    all other commands fail.
if a ctrl() handler was supplied but no array of control commands;
    ENGINE_HAS_CTRL_FUNCTION returns TRUE,
    all other commands fail.
if a ctrl() handler and array of control commands was supplied;
    ENGINE_HAS_CTRL_FUNCTION returns TRUE,
    all other commands proceed processing …
```

If the ENGINE's array of control commands is empty then all other commands will fail, otherwise; ENGINE_CTRL_GET_FIRST_CMD_TYPE returns the identifier of the first command supported by the ENGINE, ENGINE_GET_NEXT_CMD_TYPE takes the identifier of a command supported by the ENGINE and returns the next command identifier or fails if there are no more, ENGINE_CMD_FROM_NAME takes a string name for a command and returns the corresponding identifier or fails if no such command name exists, and the remaining commands take a command identifier and return properties of the corresponding commands. All except ENGINE_CTRL_GET_FLAGS return the string length of a command name or description, or populate a supplied character buffer with a copy of the command name or description. ENGINE_CTRL_GET_FLAGS returns a bitwise-OR'd mask of the following possible values;

```
#define ENGINE_CMD_FLAG_NUMERIC                 (unsigned int)0x0001
#define ENGINE_CMD_FLAG_STRING                  (unsigned int)0x0002
#define ENGINE_CMD_FLAG_NO_INPUT                (unsigned int)0x0004
#define ENGINE_CMD_FLAG_INTERNAL                (unsigned int)0x0008
```

If the ENGINE_CMD_FLAG_INTERNAL flag is set, then any other flags are purely informational to the caller - this flag will prevent the command being usable for any higher-level ENGINE functions such as ENGINE_ctrl_cmd_string(). "INTERNAL" commands are not intended to be exposed to text-based configuration by applications, administrations, users, etc. These can support arbitrary operations via ENGINE_ctrl(), including passing to and/or from the control commands data of any arbitrary type. These commands are supported in the discovery mechanisms simply to allow applications determinie if an ENGINE supports certain specific commands it might want to use (eg. application "foo" might query various ENGINEs to see if they implement "FOO_GET_VENDOR_LOGO_GIF" - and ENGINE could therefore decide whether or not to support this "foo"-specific extension).

Future developments

The ENGINE API and internal architecture is currently being reviewed. Slated for possible release in 0.9.8 is support for transparent loading of "dynamic" ENGINEs (built as self-contained shared-libraries). This would allow ENGINE implementations to be provided independently of OpenSSL libraries and/or OpenSSL-based applications, and would also remove any requirement for applications to explicitly use the "dynamic" ENGINE to bind to shared-library implementations.

SEE ALSO

rsa(3), dsa(3), dh(3), rand(3)

Name

err — error codes

Synopsis

```
#include  <openssl/err.h>
```

```
unsigned long ERR_get_error(void);
unsigned long ERR_peek_error(void);
unsigned long ERR_get_error_line(const char **file, int *line);
unsigned long ERR_peek_error_line(const char **file, int *line);
unsigned long ERR_get_error_line_data(const char **file, int *line,
        const char **data, int *flags);
unsigned long ERR_peek_error_line_data(const char **file, int *line,
        const char **data, int *flags);
```

```
int ERR_GET_LIB(unsigned long e);
int ERR_GET_FUNC(unsigned long e);
int ERR_GET_REASON(unsigned long e);
```

```
void ERR_clear_error(void);
```

```
char *ERR_error_string(unsigned long e, char *buf);
const char *ERR_lib_error_string(unsigned long e);
const char *ERR_func_error_string(unsigned long e);
const char *ERR_reason_error_string(unsigned long e);
```

```
void ERR_print_errors(BIO *bp);
void ERR_print_errors_fp(FILE *fp);
```

```
void ERR_load_crypto_strings(void);
void ERR_free_strings(void);
```

```
void ERR_remove_state(unsigned long pid);
```

```
void ERR_put_error(int lib, int func, int reason, const char *file,
        int line);
void ERR_add_error_data(int num, …);
```

```
void ERR_load_strings(int lib,ERR_STRING_DATA str[]);
unsigned long ERR_PACK(int lib, int func, int reason);
int ERR_get_next_error_library(void);
```

DESCRIPTION

When a call to the OpenSSL library fails, this is usually signalled by the return value, and an error code is stored in an error queue associated with the current thread. The **err** library provides functions to obtain these error codes and textual error messages.

The ERR_get_error(3) manpage describes how to access error codes.

Error codes contain information about where the error occurred, and what went wrong. ERR_GET_LIB(3) describes how to extract this information. A method to obtain human-readable error messages is described in ERR_error_string(3).

ERR_clear_error(3) can be used to clear the error queue.

Note that ERR_remove_state(3) should be used to avoid memory leaks when threads are terminated.

ADDING NEW ERROR CODES TO OPENSSL

See ERR_put_error(3) if you want to record error codes in the OpenSSL error system from within your application.

The remainder of this section is of interest only if you want to add new error codes to OpenSSL or add error codes from external libraries.

Reporting errors

Each sub-library has a specific macro XXXerr() that is used to report errors. Its first argument is a function code **XXX_F_...**, the second argument is a reason code **XXX_R_...**. Function codes are derived from the function names; reason codes consist of textual error descriptions. For example, the function ssl23_read() reports a "handshake failure" as follows:

```
SSLerr(SSL_F_SSL23_READ, SSL_R_SSL_HANDSHAKE_FAILURE);
```

Function and reason codes should consist of upper case characters, numbers and underscores only. The error file generation script translates function codes into function names by looking in the header files for an appropriate function name, if none is found it just uses the capitalized form such as "SSL23_READ" in the above example.

The trailing section of a reason code (after the "_R_") is translated into lower case and underscores changed to spaces.

When you are using new function or reason codes, run **make errors**. The necessary **#define**s will then automatically be added to the sub-library's header file.

Although a library will normally report errors using its own specific XXXerr macro, another library's macro can be used. This is normally only done when a library wants to include ASN1 code which must use the ASN1err() macro.

Adding new libraries

When adding a new sub-library to OpenSSL, assign it a library number **ERR_LIB_XXX**, define a macro XXXerr() (both in **err.h**), add its name to **ERR_str_libraries[]** (in **crypto/err/err.c**), and add `ERR_load_XXX_strings()` to the ERR_load_crypto_strings() function (in **crypto/err/err_all.c**). Finally, add an entry

```
L       XXX     xxx.h   xxx_err.c
```

to **crypto/err/openssl.ec**, and add **xxx_err.c** to the Makefile. Running **make errors** will then generate a file **xxx_err.c**, and add all error codes used in the library to **xxx.h**.

Additionally the library include file must have a certain form. Typically it will initially look like this:

```
#ifndef HEADER_XXX_H
#define HEADER_XXX_H

#ifdef __cplusplus
extern "C" {
#endif

/* Include files */

#include <openssl/bio.h>
#include <openssl/x509.h>

/* Macros, structures and function prototypes */

/* BEGIN ERROR CODES */
```

The **BEGIN ERROR CODES** sequence is used by the error code generation script as the point to place new error codes, any text after this point will be overwritten when **make errors** is run. The closing #endif etc will be automatically added by the script.

The generated C error code file **xxx_err.c** will load the header files **stdio.h**, **openssl/err.h** and **openssl/xxx.h** so the header file must load any additional header files containing any definitions it uses.

USING ERROR CODES IN EXTERNAL LIBRARIES

It is also possible to use OpenSSL's error code scheme in external libraries. The library needs to load its own codes and call the OpenSSL error code insertion script **mkerr.pl** explicitly to add codes to the header file and generate the C error code file. This will normally be done if the external library needs to generate new ASN1 structures but it can also be used to add more general purpose error code handling.

TBA more details

INTERNALS

The error queues are stored in a hash table with one **ERR_STATE** entry for each pid. ERR_get_state() returns the current thread's **ERR_STATE**. An **ERR_STATE** can hold up to **ERR_NUM_ERRORS** error codes. When more error codes are added, the old ones are overwritten, on the assumption that the most recent errors are most important.

Error strings are also stored in hash table. The hash tables can be obtained by calling ERR_get_err_state_table(void) and ERR_get_string_table(void) respectively.

SEE ALSO

CRYPTO_set_locking_callback(3), ERR_get_error(3), ERR_GET_LIB(3), ERR_clear_error(3), ERR_error_string(3), ERR_print_errors(3), ERR_load_crypto_strings(3), ERR_remove_state(3), ERR_put_error(3), ERR_load_strings(3), SSL_get_error(3)

Name

ERR_clear_error — clear the error queue

Synopsis

```
#include  <openssl/err.h>
```

```
void ERR_clear_error(void);
```

DESCRIPTION

ERR_clear_error() empties the current thread's error queue.

RETURN VALUES

ERR_clear_error() has no return value.

SEE ALSO

err(3), ERR_get_error(3)

HISTORY

ERR_clear_error() is available in all versions of SSLeay and OpenSSL.

Name

ERR_error_string, ERR_error_string_n, ERR_lib_error_string, ERR_func_error_string and ERR_reason_error_string — obtain human-readable error message

Synopsis

```
#include  <openssl/err.h>
```

```
char *ERR_error_string(unsigned long e, char *buf);
void ERR_error_string_n(unsigned long e, char *buf, size_t len);
```

```
const char *ERR_lib_error_string(unsigned long e);
const char *ERR_func_error_string(unsigned long e);
const char *ERR_reason_error_string(unsigned long e);
```

DESCRIPTION

ERR_error_string() generates a human-readable string representing the error code *e*, and places it at *buf*. *buf* must be at least 120 bytes long. If *buf* is **NULL**, the error string is placed in a static buffer. ERR_error_string_n() is a variant of ERR_error_string() that writes at most *len* characters (including the terminating 0) and truncates the string if necessary. For ERR_error_string_n(), *buf* may not be **NULL**.

The string will have the following format:

```
error:[error code]:[library name]:[function name]:[reason string]
```

error code is an 8 digit hexadecimal number, *library name*, *function name* and *reason string* are ASCII text.

ERR_lib_error_string(), ERR_func_error_string() and ERR_reason_error_string() return the library name, function name and reason string respectively.

The OpenSSL error strings should be loaded by calling ERR_load_crypto_strings(3) or, for SSL applications, SSL_load_error_strings(3) first. If there is no text string registered for the given error code, the error string will contain the numeric code.

ERR_print_errors(3) can be used to print all error codes currently in the queue.

RETURN VALUES

ERR_error_string() returns a pointer to a static buffer containing the string if *buf* **== NULL**, *buf* otherwise.

ERR_lib_error_string(), ERR_func_error_string() and ERR_reason_error_string() return the strings, and **NULL** if none is registered for the error code.

SEE ALSO

err(3), ERR_get_error(3), ERR_load_crypto_strings(3), SSL_load_error_strings(3), ERR_print_errors(3)

HISTORY

ERR_error_string() is available in all versions of SSLeay and OpenSSL. ERR_error_string_n() was added in OpenSSL 0.9.6.

Name

ERR_get_error, ERR_peek_error, ERR_peek_last_error, ERR_get_error_line, ERR_peek_error_line, ERR_peek_last_error_line, ERR_get_error_line_data, ERR_peek_error_line_data and ERR_peek_last_error_line_data — obtain error code and data

Synopsis

```
#include  <openssl/err.h>
```

```
unsigned long ERR_get_error(void);
unsigned long ERR_peek_error(void);
unsigned long ERR_peek_last_error(void);
```

```
unsigned long ERR_get_error_line(const char **file, int *line);
unsigned long ERR_peek_error_line(const char **file, int *line);
unsigned long ERR_peek_last_error_line(const char **file, int *line);
```

```
unsigned long ERR_get_error_line_data(const char **file, int *line,
        const char **data, int *flags);
unsigned long ERR_peek_error_line_data(const char **file, int *line,
        const char **data, int *flags);
unsigned long ERR_peek_last_error_line_data(const char **file, int *line,
        const char **data, int *flags);
```

DESCRIPTION

ERR_get_error() returns the earliest error code from the thread's error queue and removes the entry. This function can be called repeatedly until there are no more error codes to return.

ERR_peek_error() returns the earliest error code from the thread's error queue without modifying it.

ERR_peek_last_error() returns the latest error code from the thread's error queue without modifying it.

See ERR_GET_LIB(3) for obtaining information about location and reason of the error, and ERR_error_string(3) for human-readable error messages.

ERR_get_error_line(), ERR_peek_error_line() and ERR_peek_last_error_line() are the same as the above, but they additionally store the file name and line number where the error occurred in ***file** and ***line**, unless these are **NULL**.

ERR_get_error_line_data(), ERR_peek_error_line_data() and ERR_peek_last_error_line_data() store additional data and flags associated with the error code in ***data** and ***flags**, unless these are **NULL**. ***data** contains a string if ***flags&ERR_TXT_STRING** is true.

An application **MUST NOT** free the ***data** pointer (or any other pointers returned by these functions) with OPENSSL_free() as freeing is handled automatically by the error library.

RETURN VALUES

The error code, or 0 if there is no error in the queue.

SEE ALSO

err(3), ERR_error_string(3), ERR_GET_LIB(3)

HISTORY

ERR_get_error(), ERR_peek_error(), ERR_get_error_line() and ERR_peek_error_line() are available in all versions of SSLeay and OpenSSL. ERR_get_error_line_data() and ERR_peek_error_line_data() were added in SSLeay 0.9.0. ERR_peek_last_error(), ERR_peek_last_error_line() and ERR_peek_last_error_line_data() were added in OpenSSL 0.9.7.

Name

ERR_GET_LIB, ERR_GET_FUNC and ERR_GET_REASON — get library, function and reason code

Synopsis

```
#include  <openssl/err.h>
```

```
int ERR_GET_LIB(unsigned long e);
```

```
int ERR_GET_FUNC(unsigned long e);
```

```
int ERR_GET_REASON(unsigned long e);
```

DESCRIPTION

The error code returned by ERR_get_error() consists of a library number, function code and reason code. ERR_GET_LIB(), ERR_GET_FUNC() and ERR_GET_REASON() can be used to extract these.

The library number and function code describe where the error occurred, the reason code is the information about what went wrong.

Each sub-library of OpenSSL has a unique library number; function and reason codes are unique within each sub-library. Note that different libraries may use the same value to signal different functions and reasons.

ERR_R_... reason codes such as **ERR_R_MALLOC_FAILURE** are globally unique. However, when checking for sub-library specific reason codes, be sure to also compare the library number.

ERR_GET_LIB(), ERR_GET_FUNC() and ERR_GET_REASON() are macros.

RETURN VALUES

The library number, function code and reason code respectively.

SEE ALSO

err(3), ERR_get_error(3)

HISTORY

ERR_GET_LIB(), ERR_GET_FUNC() and ERR_GET_REASON() are available in all versions of SSLeay and OpenSSL.

Name

ERR_load_crypto_strings, SSL_load_error_strings and ERR_free_strings — load and free error strings

Synopsis

```
#include  <openssl/err.h>
```

```
void ERR_load_crypto_strings(void);
void ERR_free_strings(void);
```

```
#include  <openssl/ssl.h>
```

```
void SSL_load_error_strings(void);
```

DESCRIPTION

ERR_load_crypto_strings() registers the error strings for all **libcrypto** functions. SSL_load_error_strings() does the same, but also registers the **libssl** error strings.

One of these functions should be called before generating textual error messages. However, this is not required when memory usage is an issue.

ERR_free_strings() frees all previously loaded error strings.

RETURN VALUES

ERR_load_crypto_strings(), SSL_load_error_strings() and ERR_free_strings() return no values.

SEE ALSO

err(3), ERR_error_string(3)

HISTORY

ERR_load_error_strings(), SSL_load_error_strings() and ERR_free_strings() are available in all versions of SSLeay and OpenSSL.

Name

ERR_load_strings, ERR_PACK and ERR_get_next_error_library — load arbitrary error strings

Synopsis

```
#include  <openssl/err.h>
```

```
void ERR_load_strings(int lib, ERR_STRING_DATA str[]);
```

```
int ERR_get_next_error_library(void);
```

```
unsigned long ERR_PACK(int lib, int func, int reason);
```

DESCRIPTION

ERR_load_strings() registers error strings for library number **lib**.

str is an array of error string data:

```
typedef struct ERR_string_data_st
{
       unsigned long error;
       char *string;
} ERR_STRING_DATA;
```

and reason code: **error** = ERR_PACK(**lib**, **func**, **reason**). ERR_PACK() is a macro.

The last entry in the array is {0,0}.

ERR_get_next_error_library() can be used to assign library numbers to user libraries at runtime.

RETURN VALUE

ERR_load_strings() returns no value. ERR_PACK() return the error code. ERR_get_next_error_library() returns a new library number.

SEE ALSO

err(3), ERR_load_strings(3)

HISTORY

ERR_load_error_strings() and ERR_PACK() are available in all versions of SSLeay and OpenSSL. ERR_get_next_error_library() was added in SSLeay 0.9.0.

Name

ERR_print_errors and ERR_print_errors_fp — print error messages

Synopsis

```
#include  <openssl/err.h>
```

```
void ERR_print_errors(BIO *bp);
void ERR_print_errors_fp(FILE *fp);
```

DESCRIPTION

ERR_print_errors() is a convenience function that prints the error strings for all errors that OpenSSL has recorded to **bp**, thus emptying the error queue.

ERR_print_errors_fp() is the same, except that the output goes to a **FILE**.

The error strings will have the following format:

```
[pid]:error:[error code]:[library name]:[function name]:[reason string]:[file name]:[line]:[optional text message]
```

error code is an 8 digit hexadecimal number. *library name*, *function name* and *reason string* are ASCII text, as is *optional text message* if one was set for the respective error code.

If there is no text string registered for the given error code, the error string will contain the numeric code.

RETURN VALUES

ERR_print_errors() and ERR_print_errors_fp() return no values.

SEE ALSO

err(3), ERR_error_string(3), ERR_get_error(3), ERR_load_crypto_strings(3), SSL_load_error_strings(3)

HISTORY

ERR_print_errors() and ERR_print_errors_fp() are available in all versions of SSLeay and OpenSSL.

Name

ERR_put_error and ERR_add_error_data — record an error

Synopsis

```
#include  <openssl/err.h>
```

```
void ERR_put_error(int lib, int func, int reason, const char *file,
        int line);
```

```
void ERR_add_error_data(int num, …);
```

DESCRIPTION

ERR_put_error() adds an error code to the thread's error queue. It signals that the error of reason code **reason** occurred in function **func** of library **lib**, in line number **line** of **file**. This function is usually called by a macro.

ERR_add_error_data() associates the concatenation of its **num** string arguments with the error code added last.

ERR_load_strings(3) can be used to register error strings so that the application can a generate human-readable error messages for the error code.

RETURN VALUES

ERR_put_error() and ERR_add_error_data() return no values.

SEE ALSO

err(3), ERR_load_strings(3)

HISTORY

ERR_put_error() is available in all versions of SSLeay and OpenSSL. ERR_add_error_data() was added in SSLeay 0.9.0.

Name

ERR_remove_state — free a thread's error queue

Synopsis

```
#include  <openssl/err.h>
```

```
void ERR_remove_state(unsigned long pid);
```

DESCRIPTION

ERR_remove_state() frees the error queue associated with thread **pid**. If **pid** == 0, the current thread will have its error queue removed.

Since error queue data structures are allocated automatically for new threads, they must be freed when threads are terminated in order to avoid memory leaks.

RETURN VALUE

ERR_remove_state() returns no value.

SEE ALSO

err(3)

HISTORY

ERR_remove_state() is available in all versions of SSLeay and OpenSSL.

Name

ERR_set_mark and ERR_pop_to_mark — set marks and pop errors until mark

Synopsis

```
#include  <openssl/err.h>
```

```
int ERR_set_mark(void);
```

```
int ERR_pop_to_mark(void);
```

DESCRIPTION

ERR_set_mark() sets a mark on the current topmost error record if there is one.

ERR_pop_to_mark() will pop the top of the error stack until a mark is found. The mark is then removed. If there is no mark, the whole stack is removed.

RETURN VALUES

ERR_set_mark() returns 0 if the error stack is empty, otherwise 1.

ERR_pop_to_mark() returns 0 if there was no mark in the error stack, which implies that the stack became empty, otherwise 1.

SEE ALSO

err(3)

HISTORY

ERR_set_mark() and ERR_pop_to_mark() were added in OpenSSL 0.9.8.

Name

evp — high-level cryptographic functions

Synopsis

```
#include  <openssl/evp.h>
```

DESCRIPTION

The EVP library provides a high-level interface to cryptographic functions.

EVP_Seal… and **EVP_Open**… provide public key encryption and decryption to implement digital "envelopes".

The **EVP_Sign**… and **EVP_Verify**… functions implement digital signatures.

Symmetric encryption is available with the **EVP_Encrypt**… functions. The **EVP_Digest**… functions provide message digests.

The **EVP_PKEY**… functions provide a high level interface to asymmetric algorithms.

The **EVP_Encode**… and **EVP_Decode**… functions implement base 64 encoding and decoding.

Algorithms are loaded with OpenSSL_add_all_algorithms(3).

All the symmetric algorithms (ciphers), digests and asymmetric algorithms (public key algorithms) can be replaced by ENGINE modules providing alternative implementations. If ENGINE implementations of ciphers or digests are registered as defaults, then the various EVP functions will automatically use those implementations automatically in preference to built in software implementations. For more information, consult the engine(3) man page.

Although low level algorithm specific functions exist for many algorithms their use is discouraged. They cannot be used with an ENGINE and ENGINE versions of new algorithms cannot be accessed using the low level functions. Also makes code harder to adapt to new algorithms and some options are not cleanly supported at the low level and some operations are more efficient using the high level interface.

SEE ALSO

EVP_DigestInit(3), EVP_EncryptInit(3), EVP_OpenInit(3), EVP_SealInit(3), EVP_SignInit(3), EVP_VerifyInit(3), EVP_EncodeInit(3), OpenSSL_add_all_algorithms(3), engine(3)

Name

EVP_BytesToKey — password based encryption routine

Synopsis

```
#include  <openssl/evp.h>
```

```
int EVP_BytesToKey(const EVP_CIPHER *type,const EVP_MD *md,
                      const unsigned char *salt,
                      const unsigned char *data, int datal, int count,
                      unsigned char *key,unsigned char *iv);
```

DESCRIPTION

EVP_BytesToKey() derives a key and IV from various parameters. **type** is the cipher to derive the key and IV for. **md** is the message digest to use. The **salt** parameter is used as a salt in the derivation: it should point to an 8 byte buffer or NULL if no salt is used. **data** is a buffer containing **datal** bytes which is used to derive the keying data. **count** is the iteration count to use. The derived key and IV will be written to **key** and **iv** respectively.

NOTES

A typical application of this function is to derive keying material for an encryption algorithm from a password in the **data** parameter.

Increasing the **count** parameter slows down the algorithm which makes it harder for an attacker to peform a brute force attack using a large number of candidate passwords.

If the total key and IV length is less than the digest length and **MD5** is used then the derivation algorithm is compatible with PKCS#5 v1.5 otherwise a non standard extension is used to derive the extra data.

Newer applications should use more standard algorithms such as PKCS#5 v2.0 for key derivation.

KEY DERIVATION ALGORITHM

The key and IV is derived by concatenating D_1, D_2, etc until enough data is available for the key and IV. D_i is defined as:

```
D_i = HASH^count(D_(i-1) || data || salt)
```

where || denotes concatentaion, D_0 is empty, HASH is the digest algorithm in use, HASH^1(data) is simply HASH(data), HASH^2(data) is HASH(HASH(data)) and so on.

The initial bytes are used for the key and the subsequent bytes for the IV.

RETURN VALUES

EVP_BytesToKey() returns the size of the derived key in bytes.

SEE ALSO

evp(3), rand(3), EVP_EncryptInit(3)

HISTORY

Name

EVP_MD_CTX_init, EVP_MD_CTX_create, EVP_DigestInit_ex, EVP_DigestUpdate, EVP_DigestFinal_ex, EVP_MD_CTX_cleanup, EVP_MD_CTX_destroy, EVP_MAX_MD_SIZE, EVP_MD_CTX_copy_ex, EVP_MD_CTX_copy, EVP_MD_type, EVP_MD_pkey_type, EVP_MD_size, EVP_MD_block_size, EVP_MD_CTX_md, EVP_MD_CTX_size, EVP_MD_CTX_block_size, EVP_MD_CTX_type, EVP_md_null, EVP_md2, EVP_md5, EVP_sha, EVP_sha1, EVP_sha224, EVP_sha256, EVP_sha384, EVP_sha512, EVP_dss, EVP_dss1, EVP_mdc2, EVP_ripemd160, EVP_get_digestbyname, EVP_get_digestbynid and EVP_get_digestbyobj — EVP digest routines

Synopsis

```
#include  <openssl/evp.h>
```

```
void EVP_MD_CTX_init(EVP_MD_CTX *ctx);
EVP_MD_CTX *EVP_MD_CTX_create(void);
```

```
int EVP_DigestInit_ex(EVP_MD_CTX *ctx, const EVP_MD *type, ENGINE *impl);
int EVP_DigestUpdate(EVP_MD_CTX *ctx, const void *d, size_t cnt);
int EVP_DigestFinal_ex(EVP_MD_CTX *ctx, unsigned char *md,
       unsigned int *s);
```

```
int EVP_MD_CTX_cleanup(EVP_MD_CTX *ctx);
void EVP_MD_CTX_destroy(EVP_MD_CTX *ctx);
```

```
int EVP_MD_CTX_copy_ex(EVP_MD_CTX *out,const EVP_MD_CTX *in);
```

```
int EVP_DigestInit(EVP_MD_CTX *ctx, const EVP_MD *type);
int EVP_DigestFinal(EVP_MD_CTX *ctx, unsigned char *md,
       unsigned int *s);
```

```
int EVP_MD_CTX_copy(EVP_MD_CTX *out,EVP_MD_CTX *in);
```

```
#define EVP_MAX_MD_SIZE 64     /* SHA512 */
```

```
int EVP_MD_type(const EVP_MD *md);
int EVP_MD_pkey_type(const EVP_MD *md);
int EVP_MD_size(const EVP_MD *md);
int EVP_MD_block_size(const EVP_MD *md);
```

```
const EVP_MD *EVP_MD_CTX_md(const EVP_MD_CTX *ctx);
#define EVP_MD_CTX_size(e)             EVP_MD_size(EVP_MD_CTX_md(e))
#define EVP_MD_CTX_block_size(e)       EVP_MD_block_size((e)->digest)
#define EVP_MD_CTX_type(e)             EVP_MD_type((e)->digest)
```

```
const EVP_MD *EVP_md_null(void);
const EVP_MD *EVP_md2(void);
const EVP_MD *EVP_md5(void);
const EVP_MD *EVP_sha(void);
const EVP_MD *EVP_sha1(void);
const EVP_MD *EVP_dss(void);
const EVP_MD *EVP_dss1(void);
const EVP_MD *EVP_mdc2(void);
const EVP_MD *EVP_ripemd160(void);
```

```
const EVP_MD *EVP_sha224(void);
const EVP_MD *EVP_sha256(void);
const EVP_MD *EVP_sha384(void);
const EVP_MD *EVP_sha512(void);
```

```
const EVP_MD *EVP_get_digestbyname(const char *name);
#define EVP_get_digestbynid(a) EVP_get_digestbyname(OBJ_nid2sn(a))
#define EVP_get_digestbyobj(a) EVP_get_digestbynid(OBJ_obj2nid(a))
```

DESCRIPTION

The EVP digest routines are a high level interface to message digests.

EVP_MD_CTX_init() initializes digest context **ctx**.

EVP_MD_CTX_create() allocates, initializes and returns a digest context.

EVP_DigestInit_ex() sets up digest context **ctx** to use a digest **type** from ENGINE **impl**. **ctx** must be initialized before calling this function. **type** will typically be supplied by a functionsuch as EVP_sha1(). If **impl** is NULL then the default implementation of digest **type** is used.

EVP_DigestUpdate() hashes **cnt** bytes of data at **d** into the digest context **ctx**. This function can be called several times on the same **ctx** to hash additional data.

EVP_DigestFinal_ex() retrieves the digest value from **ctx** and places it in **md**. If the **s** parameter is not NULL then the number of bytes of data written (i.e. the length of the digest) will be written to the integer at **s**, at most **EVP_MAX_MD_SIZE** bytes will be written. After calling EVP_DigestFinal_ex() no additional calls to EVP_DigestUpdate() can be made, but EVP_DigestInit_ex() can be called to initialize a new digest operation.

EVP_MD_CTX_cleanup() cleans up digest context **ctx**, it should be called after a digest context is no longer needed.

EVP_MD_CTX_destroy() cleans up digest context **ctx** and frees up the space allocated to it, it should be called only on a context created using EVP_MD_CTX_create().

EVP_MD_CTX_copy_ex() can be used to copy the message digest state from **in** to **out**. This is useful if large amounts of data are to be hashed which only differ in the last few bytes. **out** must be initialized before calling this function.

EVP_DigestInit() behaves in the same way as EVP_DigestInit_ex() except the passed context **ctx** does not have to be initialized, and it always uses the default digest implementation.

EVP_DigestFinal() is similar to EVP_DigestFinal_ex() except the digest context **ctx** is automatically cleaned up.

EVP_MD_CTX_copy() is similar to EVP_MD_CTX_copy_ex() except the destination **out** does not have to be initialized.

EVP_MD_size() and EVP_MD_CTX_size() return the size of the message digest when passed an **EVP_MD** or an **EVP_MD_CTX** structure, i.e. the size of the hash.

EVP_MD_block_size() and EVP_MD_CTX_block_size() return the block size of the message digest when passed an **EVP_MD** or an **EVP_MD_CTX** structure.

EVP_MD_type() and EVP_MD_CTX_type() return the NID of the OBJECT IDENTIFIER representing the given message digest when passed an **EVP_MD** structure. For example EVP_MD_type(EVP_sha1()) returns **NID_sha1**. This function is normally used when setting ASN1 OIDs.

EVP_MD_CTX_md() returns the **EVP_MD** structure corresponding to the passed **EVP_MD_CTX**.

EVP_MD_pkey_type() returns the NID of the public key signing algorithm associated with this digest. For example EVP_sha1() is associated with RSA so this will return **NID_sha1WithRSAEncryption**. Since digests and signature algorithms are no longer linked this function is only retained for compatibility reasons.

EVP_md2(), EVP_md5(), EVP_sha(), EVP_sha1(), EVP_sha224(), EVP_sha256(), EVP_sha384(), EVP_sha512(), EVP_mdc2() and EVP_ripemd160() return **EVP_MD** structures for the MD2, MD5, SHA, SHA1, SHA224, SHA256, SHA384, SHA512, MDC2 and RIPEMD160 digest algorithms respectively.

EVP_dss() and EVP_dss1() return **EVP_MD** structures for SHA and SHA1 digest algorithms but using DSS (DSA) for the signature algorithm. Note: there is no need to use these pseudo-digests in OpenSSL 1.0.0 and later, they are however retained for compatibility.

EVP_md_null() is a "null" message digest that does nothing: i.e. the hash it returns is of zero length.

EVP_get_digestbyname(), EVP_get_digestbynid() and EVP_get_digestbyobj() return an **EVP_MD** structure when passed a digest name, a digest NID or an ASN1_OBJECT structure respectively. The digest table must be initialized using, for example, OpenSSL_add_all_digests() for these functions to work.

RETURN VALUES

EVP_DigestInit_ex(), EVP_DigestUpdate() and EVP_DigestFinal_ex() return 1 for success and 0 for failure.

EVP_MD_CTX_copy_ex() returns 1 if successful or 0 for failure.

EVP_MD_type(), EVP_MD_pkey_type() and EVP_MD_type() return the NID of the corresponding OBJECT IDENTIFIER or NID_undef if none exists.

EVP_MD_size(), EVP_MD_block_size(), EVP_MD_CTX_size() and EVP_MD_CTX_block_size() return the digest or block size in bytes.

EVP_md_null(), EVP_md2(), EVP_md5(), EVP_sha(), EVP_sha1(), EVP_dss(), EVP_dss1(), EVP_mdc2() and EVP_ripemd160() return pointers to the corresponding EVP_MD structures.

EVP_get_digestbyname(), EVP_get_digestbynid() and EVP_get_digestbyobj() return either an **EVP_MD** structure or NULL if an error occurs.

NOTES

The **EVP** interface to message digests should almost always be used in preference to the low level interfaces. This is because the code then becomes transparent to the digest used and much more flexible.

New applications should use the SHA2 digest algorithms such as SHA256. The other digest algorithms are still in common use.

For most applications the **impl** parameter to EVP_DigestInit_ex() will be set to NULL to use the default digest implementation.

The functions EVP_DigestInit(), EVP_DigestFinal() and EVP_MD_CTX_copy() are obsolete but are retained to maintain compatibility with existing code. New applications should use EVP_DigestInit_ex(), EVP_DigestFinal_ex() and EVP_MD_CTX_copy_ex() because they can efficiently reuse a digest context instead of initializing and cleaning it up on each call and allow non default implementations of digests to be specified.

In OpenSSL 0.9.7 and later if digest contexts are not cleaned up after use memory leaks will occur.

Stack allocation of EVP_MD_CTX structures is common, for example:

```
EVP_MD_CTX mctx;
EVP_MD_CTX_init(&mctx);
```

This will cause binary compatibility issues if the size of EVP_MD_CTX structure changes (this will only happen with a major release of OpenSSL). Applications wishing to avoid this should use EVP_MD_CTX_create() instead:

```
EVP_MD_CTX *mctx;
mctx = EVP_MD_CTX_create();
```

EXAMPLE

This example digests the data "Test Message\n" and "Hello World\n", using the digest name passed on the command line.

```
#include <stdio.h>
#include <openssl/evp.h>

main(int argc, char *argv[])
{
```

```
EVP_MD_CTX *mdctx;
const EVP_MD *md;
char mess1[] = "Test Message\n";
char mess2[] = "Hello World\n";
unsigned char md_value[EVP_MAX_MD_SIZE];
int md_len, i;

OpenSSL_add_all_digests();

if(!argv[1]) {
       printf("Usage: mdtest digestname\n");
       exit(1);
}

md = EVP_get_digestbyname(argv[1]);

if(!md) {
       printf("Unknown message digest %s\n", argv[1]);
       exit(1);
}

mdctx = EVP_MD_CTX_create();
EVP_DigestInit_ex(mdctx, md, NULL);
EVP_DigestUpdate(mdctx, mess1, strlen(mess1));
EVP_DigestUpdate(mdctx, mess2, strlen(mess2));
EVP_DigestFinal_ex(mdctx, md_value, &md_len);
EVP_MD_CTX_destroy(mdctx);

printf("Digest is: ");
for(i = 0; i < md_len; i++)
       printf("%02x", md_value[i]);
printf("\n");

/* Call this once before exit. */
EVP_cleanup();
exit(0);
}
```

SEE ALSO

dgst(1), evp(3)

HISTORY

EVP_DigestInit(), EVP_DigestUpdate() and EVP_DigestFinal() are available in all versions of SSLeay and OpenSSL.

EVP_MD_CTX_init(), EVP_MD_CTX_create(), EVP_MD_CTX_copy_ex(), EVP_MD_CTX_cleanup(), EVP_MD_CTX_destroy(), EVP_DigestInit_ex() and EVP_DigestFinal_ex() were added in OpenSSL 0.9.7.

EVP_md_null(), EVP_md2(), EVP_md5(), EVP_sha(), EVP_sha1(), EVP_dss(), EVP_dss1(), EVP_mdc2() and EVP_ripemd160() were changed to return truely const EVP_MD * in OpenSSL 0.9.7.

The link between digests and signing algorithms was fixed in OpenSSL 1.0 and later, so now EVP_sha1() can be used with RSA and DSA; there is no need to use EVP_dss1() any more.

OpenSSL 1.0 and later does not include the MD2 digest algorithm in the default configuration due to its security weaknesses.

Name

EVP_DigestSignInit, EVP_DigestSignUpdate and EVP_DigestSignFinal — EVP signing functions

Synopsis

```
#include  <openssl/evp.h>
```

```
int EVP_DigestSignInit(EVP_MD_CTX *ctx, EVP_PKEY_CTX **pctx,
                       const EVP_MD *type, ENGINE *e, EVP_PKEY *pkey);
int EVP_DigestSignUpdate(EVP_MD_CTX *ctx, const void *d, unsigned int cnt);
int EVP_DigestSignFinal(EVP_MD_CTX *ctx, unsigned char *sig, size_t *siglen);
```

DESCRIPTION

The EVP signature routines are a high level interface to digital signatures.

EVP_DigestSignInit() sets up signing context **ctx** to use digest **type** from ENGINE **impl** and private key **pkey**. **ctx** must be initialized with EVP_MD_CTX_init() before calling this function. If **pctx** is not NULL the EVP_PKEY_CTX of the signing operation will be written to ***pctx**: this can be used to set alternative signing options.

EVP_DigestSignUpdate() hashes **cnt** bytes of data at **d** into the signature context **ctx**. This function can be called several times on the same **ctx** to include additional data. This function is currently implemented usig a macro.

EVP_DigestSignFinal() signs the data in **ctx** places the signature in **sig**. If **sig** is **NULL** then the maximum size of the output buffer is written to the **siglen** parameter. If **sig** is not **NULL** then before the call the **siglen** parameter should contain the length of the **sig** buffer, if the call is successful the signature is written to **sig** and the amount of data written to **siglen**.

RETURN VALUES

EVP_DigestSignInit() EVP_DigestSignUpdate() and EVP_DigestSignaFinal() return 1 for success and 0 or a negative value for failure. In particular a return value of -2 indicates the operation is not supported by the public key algorithm.

The error codes can be obtained from ERR_get_error(3).

NOTES

The **EVP** interface to digital signatures should almost always be used in preference to the low level interfaces. This is because the code then becomes transparent to the algorithm used and much more flexible.

In previous versions of OpenSSL there was a link between message digest types and public key algorithms. This meant that "clone" digests such as EVP_dss1() needed to be used to sign using SHA1 and DSA. This is no longer necessary and the use of clone digest is now discouraged.

For some key types and parameters the random number generator must be seeded or the operation will fail.

The call to EVP_DigestSignFinal() internally finalizes a copy of the digest context. This means that calls to EVP_DigestSignUpdate() and EVP_DigestSignFinal() can be called later to digest and sign additional data.

Since only a copy of the digest context is ever finalized the context must be cleaned up after use by calling EVP_MD_CTX_cleanup() or a memory leak will occur.

The use of EVP_PKEY_size() with these functions is discouraged because some signature operations may have a signature length which depends on the parameters set. As a result EVP_PKEY_size() would have to return a value which indicates the maximum possible signature for any set of parameters.

SEE ALSO

EVP_DigestVerifyInit(3), EVP_DigestInit(3), err(3), evp(3), hmac(3), md2(3), md5(3), mdc2(3), ripemd(3), sha(3), dgst(1)

HISTORY

EVP_DigestSignInit(), EVP_DigestSignUpdate() and EVP_DigestSignFinal() were first added to OpenSSL 1.0.0.

Name

EVP_DigestVerifyInit, EVP_DigestVerifyUpdate and EVP_DigestVerifyFinal — EVP signature verification functions

Synopsis

```
#include  <openssl/evp.h>
```

```
int EVP_DigestVerifyInit(EVP_MD_CTX *ctx, EVP_PKEY_CTX **pctx,
                       const EVP_MD *type, ENGINE *e, EVP_PKEY *pkey);
int EVP_DigestVerifyUpdate(EVP_MD_CTX *ctx, const void *d, unsigned int cnt);
int EVP_DigestVerifyFinal(EVP_MD_CTX *ctx, unsigned char *sig, size_t siglen);
```

DESCRIPTION

The EVP signature routines are a high level interface to digital signatures.

EVP_DigestVerifyInit() sets up verification context **ctx** to use digest **type** from ENGINE **impl** and public key **pkey**. **ctx** must be initialized with EVP_MD_CTX_init() before calling this function. If **pctx** is not NULL the EVP_PKEY_CTX of the verification operation will be written to ***pctx**: this can be used to set alternative verification options.

EVP_DigestVerifyUpdate() hashes **cnt** bytes of data at **d** into the verification context **ctx**. This function can be called several times on the same **ctx** to include additional data. This function is currently implemented using a macro.

EVP_DigestVerifyFinal() verifies the data in **ctx** against the signature in **sig** of length **siglen**.

RETURN VALUES

EVP_DigestVerifyInit() and EVP_DigestVerifyUpdate() return 1 for success and 0 or a negative value for failure. In particular a return value of -2 indicates the operation is not supported by the public key algorithm.

EVP_DigestVerifyFinal() returns 1 for success; any other value indicates failure. A return value of zero indicates that the signature did not verify successfully (that is, tbs did not match the original data or the signature had an invalid form), while other values indicate a more serious error (and sometimes also indicate an invalid signature form).

The error codes can be obtained from ERR_get_error(3).

NOTES

The **EVP** interface to digital signatures should almost always be used in preference to the low level interfaces. This is because the code then becomes transparent to the algorithm used and much more flexible.

In previous versions of OpenSSL there was a link between message digest types and public key algorithms. This meant that "clone" digests such as EVP_dss1() needed to be used to sign using SHA1 and DSA. This is no longer necessary and the use of clone digest is now discouraged.

For some key types and parameters the random number generator must be seeded or the operation will fail.

The call to EVP_DigestVerifyFinal() internally finalizes a copy of the digest context. This means that EVP_VerifyUpdate() and EVP_VerifyFinal() can be called later to digest and verify additional data.

Since only a copy of the digest context is ever finalized the context must be cleaned up after use by calling EVP_MD_CTX_cleanup() or a memory leak will occur.

SEE ALSO

EVP_DigestSignInit(3), EVP_DigestInit(3), err(3), evp(3), hmac(3), md2(3), md5(3), mdc2(3), ripemd(3), sha(3), dgst(1)

HISTORY

EVP_DigestVerifyInit(), EVP_DigestVerifyUpdate() and EVP_DigestVerifyFinal() were first added to OpenSSL 1.0.0.

Name

EVP_EncodeInit, EVP_EncodeUpdate, EVP_EncodeFinal, EVP_EncodeBlock, EVP_DecodeInit, EVP_DecodeUpdate, EVP_DecodeFinal and EVP_DecodeBlock — EVP base 64 encode/decode routines

Synopsis

```
#include  <openssl/evp.h>
```

```
void EVP_EncodeInit(EVP_ENCODE_CTX *ctx);
void EVP_EncodeUpdate(EVP_ENCODE_CTX *ctx, unsigned char *out, int *outl,
                      const unsigned char *in, int inl);
void EVP_EncodeFinal(EVP_ENCODE_CTX *ctx, unsigned char *out, int *outl);
int EVP_EncodeBlock(unsigned char *t, const unsigned char *f, int n);
```

```
void EVP_DecodeInit(EVP_ENCODE_CTX *ctx);
int EVP_DecodeUpdate(EVP_ENCODE_CTX *ctx, unsigned char *out, int *outl,
                     const unsigned char *in, int inl);
int EVP_DecodeFinal(EVP_ENCODE_CTX *ctx, unsigned
                    char *out, int *outl);
int EVP_DecodeBlock(unsigned char *t, const unsigned char *f, int n);
```

DESCRIPTION

The EVP encode routines provide a high level interface to base 64 encoding and decoding. Base 64 encoding converts binary data into a printable form that uses the characters A-Z, a-z, 0-9, "+" and "/" to represent the data. For every 3 bytes of binary data provided 4 bytes of base 64 encoded data will be produced plus some occasional newlines (see below). If the input data length is not a multiple of 3 then the output data will be padded at the end using the "=" character.

Encoding of binary data is performed in blocks of 48 input bytes (or less for the final block). For each 48 byte input block encoded 64 bytes of base 64 data is output plus an additional newline character (i.e. 65 bytes in total). The final block (which may be less than 48 bytes) will output 4 bytes for every 3 bytes of input. If the data length is not divisible by 3 then a full 4 bytes is still output for the final 1 or 2 bytes of input. Similarly a newline character will also be output.

EVP_EncodeInit() initialises **ctx** for the start of a new encoding operation.

EVP_EncodeUpdate() encode **inl** bytes of data found in the buffer pointed to by **in**. The output is stored in the buffer **out** and the number of bytes output is stored in ***outl**. It is the caller's responsibility to ensure that the buffer at **out** is sufficiently large to accommodate the output data. Only full blocks of data (48 bytes) will be immediately processed and output by this function. Any remainder is held in the **ctx** object and will be processed by a subsequent call to EVP_EncodeUpdate() or EVP_EncodeFinal(). To calculate the required size of the output buffer add together the value of **inl** with the amount of unprocessed data held in **ctx** and divide the result by 48 (ignore any remainder). This gives the number of blocks of data that will be processed. Ensure the output buffer contains 65 bytes of storage for each block, plus an additional byte for a NUL terminator. EVP_EncodeUpdate() may be called repeatedly to process large amounts of input data. In the event of an error EVP_EncodeUpdate() will set ***outl** to 0.

EVP_EncodeFinal() must be called at the end of an encoding operation. It will process any partial block of data remaining in the **ctx** object. The output data will be stored in **out** and the length of the data written will be stored in ***outl**. It is the caller's responsibility to ensure that **out** is sufficiently large to accommodate the output data which will never be more than 65 bytes plus an additional NUL terminator (i.e. 66 bytes in total).

EVP_EncodeBlock() encodes a full block of input data in **f** and of length **dlen** and stores it in **t**. For every 3 bytes of input provided 4 bytes of output data will be produced. If **dlen** is not divisible by 3 then the block is encoded as a final block of data and the output is padded such that it is always divisible by 4. Additionally a NUL terminator character will be added. For example if 16 bytes of input data is provided then 24 bytes of encoded data is created plus 1 byte for a NUL terminator (i.e. 25 bytes in total). The length of the data generated *without* the NUL terminator is returned from the function.

EVP_DecodeInit() initialises **ctx** for the start of a new decoding operation.

EVP_DecodeUpdate() decodes **inl** characters of data found in the buffer pointed to by **in**. The output is stored in the buffer **out** and the number of bytes output is stored in ***outl**. It is the caller's responsibility to ensure that the buffer at **out** is sufficiently large to accommodate the output data. This function will attempt to decode as much data as possible in 4 byte chunks. Any whitespace, newline or carriage return characters are ignored. Any partial chunk of unprocessed data (1, 2 or 3 bytes) that remains at the end will be held in the **ctx** object and processed by a subsequent call to EVP_DecodeUpdate(). If any illegal base 64 characters are encountered or if the base 64 padding character "=" is encountered in the middle of the data then the function returns -1 to indicate an error. A return value of 0 or 1 indicates successful processing of the data. A return value of 0 additionally indicates that the last input data characters processed included the base 64 padding character "=" and therefore no more non-padding character data is expected to be processed. For every 4 valid base 64 bytes processed (ignoring whitespace, carriage returns and line feeds), 3 bytes of binary output data will be produced (or less at the end of the data where the padding character "=" has been used).

EVP_DecodeFinal() must be called at the end of a decoding operation. If there is any unprocessed data still in **ctx** then the input data must not have been a multiple of 4 and therefore an error has occurred. The function will return -1 in this case. Otherwise the function returns 1 on success.

EVP_DecodeBlock() will decode the block of **n** characters of base 64 data contained in **f** and store the result in **t**. Any leading whitespace will be trimmed as will any trailing whitespace, newlines, carriage returns or EOF characters. After such trimming the length of the data in **f** must be divisbile by 4. For every 4 input bytes exactly 3 output bytes will be produced. The output will be padded with 0 bits if necessary to ensure that the output is always 3 bytes for every 4 input bytes. This function will return the length of the data decoded or -1 on error.

RETURN VALUES

EVP_EncodeBlock() returns the number of bytes encoded excluding the NUL terminator.

EVP_DecodeUpdate() returns -1 on error and 0 or 1 on success. If 0 is returned then no more non-padding base 64 characters are expected.

EVP_DecodeFinal() returns -1 on error or 1 on success.

EVP_DecodeBlock() returns the length of the data decoded or -1 on error.

SEE ALSO

evp(3)

Name

EVP_CIPHER_CTX_init, EVP_EncryptInit_ex, EVP_EncryptUpdate, EVP_EncryptFinal_ex, EVP_DecryptInit_ex, EVP_DecryptUpdate, EVP_DecryptFinal_ex, EVP_CipherInit_ex, EVP_CipherUpdate, EVP_CipherFinal_ex, EVP_CIPHER_CTX_set_key_length, EVP_CIPHER_CTX_ctrl, EVP_CIPHER_CTX_cleanup, EVP_EncryptInit, EVP_EncryptFinal, EVP_DecryptInit, EVP_DecryptFinal, EVP_CipherInit, EVP_CipherFinal, EVP_get_cipherbyname, EVP_get_cipherbynid, EVP_get_cipherbyobj, EVP_CIPHER_nid, EVP_CIPHER_block_size, EVP_CIPHER_key_length, EVP_CIPHER_iv_length, EVP_CIPHER_flags, EVP_CIPHER_mode, EVP_CIPHER_type, EVP_CIPHER_CTX_cipher, EVP_CIPHER_CTX_nid, EVP_CIPHER_CTX_block_size, EVP_CIPHER_CTX_key_length, EVP_CIPHER_CTX_iv_length, EVP_CIPHER_CTX_get_app_data, EVP_CIPHER_CTX_set_app_data, EVP_CIPHER_CTX_type, EVP_CIPHER_CTX_flags, EVP_CIPHER_CTX_mode, EVP_CIPHER_param_to_asn1, EVP_CIPHER_asn1_to_param and EVP_CIPHER_CTX_set_padding — EVP cipher routines

Synopsis

```
#include  <openssl/evp.h>
```

```
void EVP_CIPHER_CTX_init(EVP_CIPHER_CTX *a);
```

```
int EVP_EncryptInit_ex(EVP_CIPHER_CTX *ctx, const EVP_CIPHER *type,
        ENGINE *impl, unsigned char *key, unsigned char *iv);
int EVP_EncryptUpdate(EVP_CIPHER_CTX *ctx, unsigned char *out,
        int *outl, unsigned char *in, int inl);
int EVP_EncryptFinal_ex(EVP_CIPHER_CTX *ctx, unsigned char *out,
        int *outl);
```

```
int EVP_DecryptInit_ex(EVP_CIPHER_CTX *ctx, const EVP_CIPHER *type,
        ENGINE *impl, unsigned char *key, unsigned char *iv);
int EVP_DecryptUpdate(EVP_CIPHER_CTX *ctx, unsigned char *out,
        int *outl, unsigned char *in, int inl);
int EVP_DecryptFinal_ex(EVP_CIPHER_CTX *ctx, unsigned char *outm,
        int *outl);
```

```
int EVP_CipherInit_ex(EVP_CIPHER_CTX *ctx, const EVP_CIPHER *type,
        ENGINE *impl, unsigned char *key, unsigned char *iv, int enc);
int EVP_CipherUpdate(EVP_CIPHER_CTX *ctx, unsigned char *out,
        int *outl, unsigned char *in, int inl);
int EVP_CipherFinal_ex(EVP_CIPHER_CTX *ctx, unsigned char *outm,
        int *outl);
```

```
int EVP_EncryptInit(EVP_CIPHER_CTX *ctx, const EVP_CIPHER *type,
        unsigned char *key, unsigned char *iv);
int EVP_EncryptFinal(EVP_CIPHER_CTX *ctx, unsigned char *out,
        int *outl);
```

```
int EVP_DecryptInit(EVP_CIPHER_CTX *ctx, const EVP_CIPHER *type,
        unsigned char *key, unsigned char *iv);
int EVP_DecryptFinal(EVP_CIPHER_CTX *ctx, unsigned char *outm,
        int *outl);
```

```
int EVP_CipherInit(EVP_CIPHER_CTX *ctx, const EVP_CIPHER *type,
        unsigned char *key, unsigned char *iv, int enc);
int EVP_CipherFinal(EVP_CIPHER_CTX *ctx, unsigned char *outm,
        int *outl);
```

```
int EVP_CIPHER_CTX_set_padding(EVP_CIPHER_CTX *x, int padding);
int EVP_CIPHER_CTX_set_key_length(EVP_CIPHER_CTX *x, int keylen);
int EVP_CIPHER_CTX_ctrl(EVP_CIPHER_CTX *ctx, int type, int arg, void *ptr);
int EVP_CIPHER_CTX_cleanup(EVP_CIPHER_CTX *a);
```

```
const EVP_CIPHER *EVP_get_cipherbyname(const char *name);
#define EVP_get_cipherbynid(a) EVP_get_cipherbyname(OBJ_nid2sn(a))
#define EVP_get_cipherbyobj(a) EVP_get_cipherbynid(OBJ_obj2nid(a))
```

```
#define EVP_CIPHER_nid(e)              ((e)->nid)
```

```
#define EVP_CIPHER_block_size(e)       ((e)->block_size)
#define EVP_CIPHER_key_length(e)       ((e)->key_len)
#define EVP_CIPHER_iv_length(e)                ((e)->iv_len)
#define EVP_CIPHER_flags(e)            ((e)->flags)
#define EVP_CIPHER_mode(e)             ((e)->flags) & EVP_CIPH_MODE)
int EVP_CIPHER_type(const EVP_CIPHER *ctx);
```

```
#define EVP_CIPHER_CTX_cipher(e)       ((e)->cipher)
#define EVP_CIPHER_CTX_nid(e)          ((e)->cipher->nid)
#define EVP_CIPHER_CTX_block_size(e)   ((e)->cipher->block_size)
#define EVP_CIPHER_CTX_key_length(e)   ((e)->key_len)
#define EVP_CIPHER_CTX_iv_length(e)    ((e)->cipher->iv_len)
#define EVP_CIPHER_CTX_get_app_data(e) ((e)->app_data)
#define EVP_CIPHER_CTX_set_app_data(e,d) ((e)->app_data=(char *)(d))
#define EVP_CIPHER_CTX_type(c)         EVP_CIPHER_type(EVP_CIPHER_CTX_cipher(c))
#define EVP_CIPHER_CTX_flags(e)                ((e)->cipher->flags)
#define EVP_CIPHER_CTX_mode(e)         ((e)->cipher->flags & EVP_CIPH_MODE)
```

```
int EVP_CIPHER_param_to_asn1(EVP_CIPHER_CTX *c, ASN1_TYPE *type);
int EVP_CIPHER_asn1_to_param(EVP_CIPHER_CTX *c, ASN1_TYPE *type);
```

DESCRIPTION

The EVP cipher routines are a high level interface to certain symmetric ciphers.

EVP_CIPHER_CTX_init() initializes cipher contex **ctx**.

EVP_EncryptInit_ex() sets up cipher context **ctx** for encryption with cipher **type** from ENGINE **impl**. **ctx** must be initialized before calling this function. **type** is normally supplied by a function such as EVP_des_cbc(). If **impl** is NULL then the default implementation is used. **key** is the symmetric key to use and **iv** is the IV to use (if necessary), the actual number of bytes used for the key and IV depends on the cipher. It is possible to set all parameters to NULL except **type** in an initial call and supply the remaining parameters in subsequent calls, all of which have **type** set to NULL. This is done when the default cipher parameters are not appropriate.

EVP_EncryptUpdate() encrypts **inl** bytes from the buffer **in** and writes the encrypted version to **out**. This function can be called multiple times to encrypt successive blocks of data. The amount of data written depends on the block alignment of the encrypted data: as a result the amount of data written may be anything from zero bytes to (inl + cipher_block_size - 1) so **out** should contain sufficient room. The actual number of bytes written is placed in **outl**.

If padding is enabled (the default) then EVP_EncryptFinal_ex() encrypts the "final" data, that is any data that remains in a partial block. It uses “standard block padding” (aka PKCS padding). The encrypted final data is written to **out** which should have sufficient space for one cipher block. The number of bytes written is placed in **outl**. After this function is called the encryption operation is finished and no further calls to EVP_EncryptUpdate() should be made.

If padding is disabled then EVP_EncryptFinal_ex() will not encrypt any more data and it will return an error if any data remains in a partial block: that is if the total data length is not a multiple of the block size.

EVP_DecryptInit_ex(), EVP_DecryptUpdate() and EVP_DecryptFinal_ex() are the corresponding decryption operations. EVP_DecryptFinal() will return an error code if padding is enabled and the final block is not correctly formatted. The parameters and restrictions are identical to the encryption operations except that if padding is enabled the decrypted data buffer **out** passed to EVP_DecryptUpdate() should have sufficient room for (**inl** + cipher_block_size) bytes unless the cipher block size is 1 in which case **inl** bytes is sufficient.

EVP_CipherInit_ex(), EVP_CipherUpdate() and EVP_CipherFinal_ex() are functions that can be used for decryption or encryption. The operation performed depends on the value of the **enc** parameter. It should be set to 1 for encryption, 0 for decryption and -1 to leave the value unchanged (the actual value of 'enc' being supplied in a previous call).

EVP_CIPHER_CTX_cleanup() clears all information from a cipher context and free up any allocated memory associate with it. It should be called after all operations using a cipher are complete so sensitive information does not remain in memory.

EVP_EncryptInit(), EVP_DecryptInit() and EVP_CipherInit() behave in a similar way to EVP_EncryptInit_ex(), EVP_DecryptInit_ex and EVP_CipherInit_ex() except the **ctx** parameter does not need to be initialized and they always use the default cipher implementation.

EVP_EncryptFinal(), EVP_DecryptFinal() and EVP_CipherFinal() behave in a similar way to EVP_EncryptFinal_ex(), EVP_DecryptFinal_ex() and EVP_CipherFinal_ex() except **ctx** is automatically cleaned up after the call.

EVP_get_cipherbyname(), EVP_get_cipherbynid() and EVP_get_cipherbyobj() return an EVP_CIPHER structure when passed a cipher name, a NID or an ASN1_OBJECT structure.

EVP_CIPHER_nid() and EVP_CIPHER_CTX_nid() return the NID of a cipher when passed an **EVP_CIPHER** or **EVP_CIPHER_CTX** structure. The actual NID value is an internal value which may not have a corresponding OBJECT IDENTIFIER.

EVP_CIPHER_CTX_set_padding() enables or disables padding. By default encryption operations are padded using standard block padding and the padding is checked and removed when decrypting. If the **pad** parameter is zero then no padding is performed, the total amount of data encrypted or decrypted must then be a multiple of the block size or an error will occur.

EVP_CIPHER_key_length() and EVP_CIPHER_CTX_key_length() return the key length of a cipher when passed an **EVP_CIPHER** or **EVP_CIPHER_CTX** structure. The constant **EVP_MAX_KEY_LENGTH** is the maximum key length for all ciphers. Note: although EVP_CIPHER_key_length() is fixed for a given cipher, the value of EVP_CIPHER_CTX_key_length() may be different for variable key length ciphers.

EVP_CIPHER_CTX_set_key_length() sets the key length of the cipher ctx. If the cipher is a fixed length cipher then attempting to set the key length to any value other than the fixed value is an error.

EVP_CIPHER_iv_length() and EVP_CIPHER_CTX_iv_length() return the IV length of a cipher when passed an **EVP_CIPHER** or **EVP_CIPHER_CTX**. It will return zero if the cipher does not use an IV. The constant **EVP_MAX_IV_LENGTH** is the maximum IV length for all ciphers.

EVP_CIPHER_block_size() and EVP_CIPHER_CTX_block_size() return the block size of a cipher when passed an **EVP_CIPHER** or **EVP_CIPHER_CTX** structure. The constant **EVP_MAX_IV_LENGTH** is also the maximum block length for all ciphers.

EVP_CIPHER_type() and EVP_CIPHER_CTX_type() return the type of the passed cipher or context. This "type" is the actual NID of the cipher OBJECT IDENTIFIER as such it ignores the cipher parameters and 40 bit RC2 and 128 bit RC2 have the same NID. If the cipher does not have an object identifier or does not have ASN1 support this function will return **NID_undef**.

EVP_CIPHER_CTX_cipher() returns the **EVP_CIPHER** structure when passed an **EVP_CIPHER_CTX** structure.

EVP_CIPHER_mode() and EVP_CIPHER_CTX_mode() return the block cipher mode: EVP_CIPH_ECB_MODE, EVP_CIPH_CBC_MODE, EVP_CIPH_CFB_MODE or EVP_CIPH_OFB_MODE. If the cipher is a stream cipher then EVP_CIPH_STREAM_CIPHER is returned.

EVP_CIPHER_param_to_asn1() sets the AlgorithmIdentifier "parameter" based on the passed cipher. This will typically include any parameters and an IV. The cipher IV (if any) must be set when this call is made. This call should be made before the cipher is actually "used" (before any EVP_EncryptUpdate(), EVP_DecryptUpdate() calls for example). This function may fail if the cipher does not have any ASN1 support.

EVP_CIPHER_asn1_to_param() sets the cipher parameters based on an ASN1 AlgorithmIdentifier "parameter". The precise effect depends on the cipher In the case of RC2, for example, it will set the IV and effective key length. This function should be called after the base cipher type is set but before the key is set. For example EVP_CipherInit() will be called with the IV and key set to NULL, EVP_CIPHER_asn1_to_param() will be called and finally EVP_CipherInit() again with all parameters except the key set to NULL. It is possible for this function to fail if the cipher does not have any ASN1 support or the parameters cannot be set (for example the RC2 effective key length is not supported.

EVP_CIPHER_CTX_ctrl() allows various cipher specific parameters to be determined and set. Currently only the RC2 effective key length and the number of rounds of RC5 can be set.

RETURN VALUES

EVP_EncryptInit_ex(), EVP_EncryptUpdate() and EVP_EncryptFinal_ex() return 1 for success and 0 for failure.

EVP_DecryptInit_ex() and EVP_DecryptUpdate() return 1 for success and 0 for failure. EVP_DecryptFinal_ex() returns 0 if the decrypt failed or 1 for success.

EVP_CipherInit_ex() and EVP_CipherUpdate() return 1 for success and 0 for failure. EVP_CipherFinal_ex() returns 0 for a decryption failure or 1 for success.

EVP_CIPHER_CTX_cleanup() returns 1 for success and 0 for failure.

EVP_get_cipherbyname(), EVP_get_cipherbynid() and EVP_get_cipherbyobj() return an **EVP_CIPHER** structure or NULL on error.

EVP_CIPHER_nid() and EVP_CIPHER_CTX_nid() return a NID.

EVP_CIPHER_block_size() and EVP_CIPHER_CTX_block_size() return the block size.

EVP_CIPHER_key_length() and EVP_CIPHER_CTX_key_length() return the key length.

EVP_CIPHER_CTX_set_padding() always returns 1.

EVP_CIPHER_iv_length() and EVP_CIPHER_CTX_iv_length() return the IV length or zero if the cipher does not use an IV.

EVP_CIPHER_type() and EVP_CIPHER_CTX_type() return the NID of the cipher's OBJECT IDENTIFIER or NID_undef if it has no defined OBJECT IDENTIFIER.

EVP_CIPHER_CTX_cipher() returns an **EVP_CIPHER** structure.

EVP_CIPHER_param_to_asn1() and EVP_CIPHER_asn1_to_param() return 1 for success or zero for failure.

CIPHER LISTING

All algorithms have a fixed key length unless otherwise stated.

EVP_enc_null()

> Null cipher: does nothing.

EVP_des_cbc(void)
EVP_des_ecb(void)
EVP_des_cfb(void)
EVP_des_ofb(void)

> DES in CBC, ECB, CFB and OFB modes respectively.

EVP_des_ede_cbc(void)
EVP_des_ede()
EVP_des_ede_ofb(void)
EVP_des_ede_cfb(void)

> Two key triple DES in CBC, ECB, CFB and OFB modes respectively.

EVP_des_ede3_cbc(void)
EVP_des_ede3()
EVP_des_ede3_ofb(void)
EVP_des_ede3_cfb(void)

Three key triple DES in CBC, ECB, CFB and OFB modes respectively.

EVP_desx_cbc(void)

DESX algorithm in CBC mode.

EVP_rc4(void)

RC4 stream cipher. This is a variable key length cipher with default key length 128 bits.

EVP_rc4_40(void)

RC4 stream cipher with 40 bit key length. This is obsolete and new code should use EVP_rc4() and the EVP_CIPHER_CTX_set_key_length() function.

EVP_idea_cbc() EVP_idea_ecb(void)
EVP_idea_cfb(void)
EVP_idea_ofb(void)
EVP_idea_cbc(void)

IDEA encryption algorithm in CBC, ECB, CFB and OFB modes respectively.

EVP_rc2_cbc(void)
EVP_rc2_ecb(void)
EVP_rc2_cfb(void)
EVP_rc2_ofb(void)

RC2 encryption algorithm in CBC, ECB, CFB and OFB modes respectively. This is a variable key length cipher with an additional parameter called "effective key bits" or "effective key length". By default both are set to 128 bits.

EVP_rc2_40_cbc(void)
EVP_rc2_64_cbc(void)

RC2 algorithm in CBC mode with a default key length and effective key length of 40 and 64 bits. These are obsolete and new code should use EVP_rc2_cbc(), EVP_CIPHER_CTX_set_key_length() and EVP_CIPHER_CTX_ctrl() to set the key length and effective key length.

EVP_bf_cbc(void)
EVP_bf_ecb(void)
EVP_bf_cfb(void)
EVP_bf_ofb(void);

Blowfish encryption algorithm in CBC, ECB, CFB and OFB modes respectively. This is a variable key length cipher.

EVP_cast5_cbc(void)
EVP_cast5_ecb(void)
EVP_cast5_cfb(void)
EVP_cast5_ofb(void)

CAST encryption algorithm in CBC, ECB, CFB and OFB modes respectively. This is a variable key length cipher.

EVP_rc5_32_12_16_cbc(void)
EVP_rc5_32_12_16_ecb(void)
EVP_rc5_32_12_16_cfb(void)
EVP_rc5_32_12_16_ofb(void)

RC5 encryption algorithm in CBC, ECB, CFB and OFB modes respectively. This is a variable key length cipher with an additional "number of rounds" parameter. By default the key length is set to 128 bits and 12 rounds.

NOTES

Where possible the **EVP** interface to symmetric ciphers should be used in preference to the low level interfaces. This is because the code then becomes transparent to the cipher used and much more flexible. Additionally, the **EVP** interface will ensure the use of platform specific cryptographic acceleration such as AES-NI (the low level interfaces do not provide the guarantee).

PKCS padding works by adding **n** padding bytes of value **n** to make the total length of the encrypted data a multiple of the block size. Padding is always added so if the data is already a multiple of the block size **n** will equal the block size. For example if the block size is 8 and 11 bytes are to be encrypted then 5 padding bytes of value 5 will be added.

When decrypting the final block is checked to see if it has the correct form.

Although the decryption operation can produce an error if padding is enabled, it is not a strong test that the input data or key is correct. A random block has better than 1 in 256 chance of being of the correct format and problems with the input data earlier on will not produce a final decrypt error.

If padding is disabled then the decryption operation will always succeed if the total amount of data decrypted is a multiple of the block size.

The functions EVP_EncryptInit(), EVP_EncryptFinal(), EVP_DecryptInit(), EVP_CipherInit() and EVP_CipherFinal() are obsolete but are retained for compatibility with existing code. New code should use EVP_EncryptInit_ex(), EVP_EncryptFinal_ex(), EVP_DecryptInit_ex(), EVP_DecryptFinal_ex(), EVP_CipherInit_ex() and EVP_CipherFinal_ex() because they can reuse an existing context without allocating and freeing it up on each call.

BUGS

For RC5 the number of rounds can currently only be set to 8, 12 or 16. This is a limitation of the current RC5 code rather than the EVP interface.

EVP_MAX_KEY_LENGTH and EVP_MAX_IV_LENGTH only refer to the internal ciphers with default key lengths. If custom ciphers exceed these values the results are unpredictable. This is because it has become standard practice to define a generic key as a fixed unsigned char array containing EVP_MAX_KEY_LENGTH bytes.

The ASN1 code is incomplete (and sometimes inaccurate) it has only been tested for certain common S/MIME ciphers (RC2, DES, triple DES) in CBC mode.

EXAMPLES

Encrypt a string using IDEA:

```
int do_crypt(char *outfile)
       {
       unsigned char outbuf[1024];
       int outlen, tmplen;
       /* Bogus key and IV: we'd normally set these from
        * another source.
        */
       unsigned char key[] = {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
       unsigned char iv[] = {1,2,3,4,5,6,7,8};
       char intext[] = "Some Crypto Text";
       EVP_CIPHER_CTX ctx;
       FILE *out;

EVP_CIPHER_CTX_init(&ctx);
EVP_EncryptInit_ex(&ctx, EVP_idea_cbc(), NULL, key, iv);

if(!EVP_EncryptUpdate(&ctx, outbuf, &outlen, intext, strlen(intext)))
        {
        /* Error */
        return 0;
        }
/* Buffer passed to EVP_EncryptFinal() must be after data just
 * encrypted to avoid overwriting it.
 */
if(!EVP_EncryptFinal_ex(&ctx, outbuf + outlen, &tmplen))
        {
        /* Error */
        return 0;
        }
outlen += tmplen;
EVP_CIPHER_CTX_cleanup(&ctx);
/* Need binary mode for fopen because encrypted data is
 * binary data. Also cannot use strlen() on it because
 * it wont be null terminated and may contain embedded
 * nulls.
 */
out = fopen(outfile, "wb");
fwrite(outbuf, 1, outlen, out);
fclose(out);
return 1;
}
```

The ciphertext from the above example can be decrypted using the **openssl** utility with the command line (shown on two lines for clarity):

```
openssl idea -d <filename
         -K 000102030405060708090A0B0C0D0E0F -iv 0102030405060708
```

General encryption and decryption function example using FILE I/O and AES128 with a 128-bit key:

```
int do_crypt(FILE *in, FILE *out, int do_encrypt)
       {
       /* Allow enough space in output buffer for additional block */
       unsigned char inbuf[1024], outbuf[1024 + EVP_MAX_BLOCK_LENGTH];
       int inlen, outlen;
       EVP_CIPHER_CTX ctx;
       /* Bogus key and IV: we'd normally set these from
        * another source.
        */
       unsigned char key[] = "0123456789abcdeF";
       unsigned char iv[] = "1234567887654321";

/* Don't set key or IV right away; we want to check lengths */
EVP_CIPHER_CTX_init(&ctx);
EVP_CipherInit_ex(&ctx, EVP_aes_128_cbc(), NULL, NULL, NULL,
        do_encrypt);
OPENSSL_assert(EVP_CIPHER_CTX_key_length(&ctx) == 16);
OPENSSL_assert(EVP_CIPHER_CTX_iv_length(&ctx) == 16);

/* Now we can set key and IV */
EVP_CipherInit_ex(&ctx, NULL, NULL, key, iv, do_encrypt);

for(;;)
        {
        inlen = fread(inbuf, 1, 1024, in);
        if(inlen <= 0) break;
        if(!EVP_CipherUpdate(&ctx, outbuf, &outlen, inbuf, inlen))
                {
                /* Error */
                EVP_CIPHER_CTX_cleanup(&ctx);
                return 0;
                }
        fwrite(outbuf, 1, outlen, out);
        }
if(!EVP_CipherFinal_ex(&ctx, outbuf, &outlen))
        {
        /* Error */
        EVP_CIPHER_CTX_cleanup(&ctx);
        return 0;
        }
fwrite(outbuf, 1, outlen, out);

EVP_CIPHER_CTX_cleanup(&ctx);
return 1;
}
```

SEE ALSO

evp(3)

HISTORY

EVP_CIPHER_CTX_init(), EVP_EncryptInit_ex(), EVP_EncryptFinal_ex(), EVP_DecryptInit_ex(), EVP_DecryptFinal_ex(), EVP_CipherInit_ex(), EVP_CipherFinal_ex() and EVP_CIPHER_CTX_set_padding() appeared in OpenSSL 0.9.7.

IDEA appeared in OpenSSL 0.9.7 but was often disabled due to patent concerns; the last patents expired in 2012.

Name

EVP_OpenInit, EVP_OpenUpdate and EVP_OpenFinal — EVP envelope decryption

Synopsis

```
#include  <openssl/evp.h>
```

```
int EVP_OpenInit(EVP_CIPHER_CTX *ctx,EVP_CIPHER *type,unsigned char *ek,
               int ekl,unsigned char *iv,EVP_PKEY *priv);
int EVP_OpenUpdate(EVP_CIPHER_CTX *ctx, unsigned char *out,
        int *outl, unsigned char *in, int inl);
int EVP_OpenFinal(EVP_CIPHER_CTX *ctx, unsigned char *out,
        int *outl);
```

DESCRIPTION

The EVP envelope routines are a high level interface to envelope decryption. They decrypt a public key encrypted symmetric key and then decrypt data using it.

EVP_OpenInit() initializes a cipher context **ctx** for decryption with cipher **type**. It decrypts the encrypted symmetric key of length **ekl** bytes passed in the **ek** parameter using the private key **priv**. The IV is supplied in the **iv** parameter.

EVP_OpenUpdate() and EVP_OpenFinal() have exactly the same properties as the EVP_DecryptUpdate() and EVP_DecryptFinal() routines, as documented on the EVP_EncryptInit(3) manual page.

NOTES

It is possible to call EVP_OpenInit() twice in the same way as EVP_DecryptInit(). The first call should have **priv** set to NULL and (after setting any cipher parameters) it should be called again with **type** set to NULL.

If the cipher passed in the **type** parameter is a variable length cipher then the key length will be set to the value of the recovered key length. If the cipher is a fixed length cipher then the recovered key length must match the fixed cipher length.

RETURN VALUES

EVP_OpenInit() returns 0 on error or a non zero integer (actually the recovered secret key size) if successful.

EVP_OpenUpdate() returns 1 for success or 0 for failure.

EVP_OpenFinal() returns 0 if the decrypt failed or 1 for success.

SEE ALSO

evp(3), rand(3), EVP_EncryptInit(3), EVP_SealInit(3)

HISTORY

Name

EVP_PKEY_copy_parameters, EVP_PKEY_missing_parameters, EVP_PKEY_cmp_parameters and EVP_PKEY_cmp — public key parameter and comparison functions

Synopsis

```
#include  <openssl/evp.h>
```

```
int EVP_PKEY_missing_parameters(const EVP_PKEY *pkey);
int EVP_PKEY_copy_parameters(EVP_PKEY *to, const EVP_PKEY *from);
```

```
int EVP_PKEY_cmp_parameters(const EVP_PKEY *a, const EVP_PKEY *b);
int EVP_PKEY_cmp(const EVP_PKEY *a, const EVP_PKEY *b);
```

DESCRIPTION

The function EVP_PKEY_missing_parameters() returns 1 if the public key parameters of **pkey** are missing and 0 if they are present or the algorithm doesn't use parameters.

The function EVP_PKEY_copy_parameters() copies the parameters from key **from** to key **to**.

The funcion EVP_PKEY_cmp_parameters() compares the parameters of keys **a** and **b**.

The funcion EVP_PKEY_cmp() compares the public key components and paramters (if present) of keys **a** and **b**.

NOTES

The main purpose of the functions EVP_PKEY_missing_parameters() and EVP_PKEY_copy_parameters() is to handle public keys in certificates where the parameters are sometimes omitted from a public key if they are inherited from the CA that signed it.

Since OpenSSL private keys contain public key components too the function EVP_PKEY_cmp() can also be used to determine if a private key matches a public key.

RETURN VALUES

The function EVP_PKEY_missing_parameters() returns 1 if the public key parameters of **pkey** are missing and 0 if they are present or the algorithm doesn't use parameters.

These functions EVP_PKEY_copy_parameters() returns 1 for success and 0 for failure.

The function EVP_PKEY_cmp_parameters() and EVP_PKEY_cmp() return 1 if the keys match, 0 if they don't match, -1 if the key types are different and -2 if the operation is not supported.

SEE ALSO

EVP_PKEY_CTX_new(3), EVP_PKEY_keygen(3)

Name

EVP_PKEY_ctrl and EVP_PKEY_ctrl_str — algorithm specific control operations

Synopsis

```
#include  <openssl/evp.h>
```

```
int EVP_PKEY_CTX_ctrl(EVP_PKEY_CTX *ctx, int keytype, int optype,
                               int cmd, int p1, void *p2);
int EVP_PKEY_CTX_ctrl_str(EVP_PKEY_CTX *ctx, const char *type,
                                               const char *value);
```

```
int EVP_PKEY_get_default_digest_nid(EVP_PKEY *pkey, int *pnid);
```

```
#include  <openssl/rsa.h>
```

```
int EVP_PKEY_CTX_set_signature_md(EVP_PKEY_CTX *ctx, const EVP_MD *md);
```

```
int EVP_PKEY_CTX_set_rsa_padding(EVP_PKEY_CTX *ctx, int pad);
int EVP_PKEY_CTX_set_rsa_pss_saltlen(EVP_PKEY_CTX *ctx, int len);
int EVP_PKEY_CTX_set_rsa_rsa_keygen_bits(EVP_PKEY_CTX *ctx, int mbits);
int EVP_PKEY_CTX_set_rsa_keygen_pubexp(EVP_PKEY_CTX *ctx, BIGNUM *pubexp);
```

```
#include  <openssl/dsa.h>
int EVP_PKEY_CTX_set_dsa_paramgen_bits(EVP_PKEY_CTX *ctx, int nbits);
```

```
#include  <openssl/dh.h>
int EVP_PKEY_CTX_set_dh_paramgen_prime_len(EVP_PKEY_CTX *ctx, int len);
int EVP_PKEY_CTX_set_dh_paramgen_generator(EVP_PKEY_CTX *ctx, int gen);
```

```
#include  <openssl/ec.h>
int EVP_PKEY_CTX_set_ec_paramgen_curve_nid(EVP_PKEY_CTX *ctx, int nid);
```

DESCRIPTION

The function EVP_PKEY_CTX_ctrl() sends a control operation to the context **ctx**. The key type used must match **keytype** if it is not -1. The parameter **optype** is a mask indicating which operations the control can be applied to. The control command is indicated in **cmd** and any additional arguments in **p1** and **p2**.

Applications will not normally call EVP_PKEY_CTX_ctrl() directly but will instead call one of the algorithm specific macros below.

The function EVP_PKEY_ctrl_str() allows an application to send an algorithm specific control operation to a context **ctx** in string form. This is intended to be used for options specified on the command line or in text files. The commands supported are documented in the openssl utility command line pages for the option **-pkeyopt** which is supported by the **pkeyutl**, **genpkey** and **req** commands.

All the remaining "functions" are implemented as macros.

The EVP_PKEY_CTX_set_signature_md() macro sets the message digest type used in a signature. It can be used with any public key algorithm supporting signature operations.

The macro EVP_PKEY_CTX_set_rsa_padding() sets the RSA padding mode for **ctx**. The **pad** parameter can take the value RSA_PKCS1_PADDING for PKCS#1 padding, RSA_SSLV23_PADDING for SSLv23 padding, RSA_NO_PADDING for no padding, RSA_PKCS1_OAEP_PADDING for OAEP padding (encrypt and decrypt only), RSA_X931_PADDING for X9.31 padding (signature operations only) and RSA_PKCS1_PSS_PADDING (sign and verify only).

Two RSA padding modes behave differently if EVP_PKEY_CTX_set_signature_md() is used. If this macro is called for PKCS#1 padding the plaintext buffer is an actual digest value and is encapsulated in a DigestInfo structure according to PKCS#1 when signing and this structure is expected (and stripped off) when verifying. If this control is not used with RSA and PKCS#1 padding then the supplied data is used directly and not encapsulated. In the case of X9.31 padding for RSA the algorithm identifier byte is added or checked and removed if this control is called. If it is not called then the first byte of the plaintext buffer is expected to be the algorithm identifier byte.

The EVP_PKEY_CTX_set_rsa_pss_saltlen() macro sets the RSA PSS salt length to **len** as its name implies it is only supported for PSS padding. Two special values are supported: -1 sets the salt length to the digest length. When signing -2 sets the salt length to the maximum permissible value. When verifying -2 causes the salt length to be automatically determined based on the **PSS** block structure. If this macro is not called a salt length value of -2 is used by default.

The EVP_PKEY_CTX_set_rsa_rsa_keygen_bits() macro sets the RSA key length for RSA key genration to **bits**. If not specified 1024 bits is used.

The EVP_PKEY_CTX_set_rsa_keygen_pubexp() macro sets the public exponent value for RSA key generation to **pubexp** currently it should be an odd integer. The **pubexp** pointer is used internally by this function so it should not be modified or free after the call. If this macro is not called then 65537 is used.

The macro EVP_PKEY_CTX_set_dsa_paramgen_bits() sets the number of bits used for DSA parameter generation to **bits**. If not specified 1024 is used.

The macro EVP_PKEY_CTX_set_dh_paramgen_prime_len() sets the length of the DH prime parameter **p** for DH parameter generation. If this macro is not called then 1024 is used.

The EVP_PKEY_CTX_set_dh_paramgen_generator() macro sets DH generator to **gen** for DH parameter generation. If not specified 2 is used.

The EVP_PKEY_CTX_set_ec_paramgen_curve_nid() sets the EC curve for EC parameter generation to **nid**. For EC parameter generation this macro must be called or an error occurs because there is no default curve.

RETURN VALUES

EVP_PKEY_CTX_ctrl() and its macros return a positive value for success and 0 or a negative value for failure. In particular a return value of -2 indicates the operation is not supported by the public key algorithm.

SEE ALSO

EVP_PKEY_CTX_new(3), EVP_PKEY_encrypt(3), EVP_PKEY_decrypt(3), EVP_PKEY_sign(3), EVP_PKEY_verify(3), EVP_PKEY_verify_recover(3), EVP_PKEY_derive(3), EVP_PKEY_keygen(3)

HISTORY

These functions were first added to OpenSSL 1.0.0.

Name

EVP_PKEY_CTX_new, EVP_PKEY_CTX_new_id, EVP_PKEY_CTX_dup and EVP_PKEY_CTX_free — public key algorithm context functions.

Synopsis

```
#include  <openssl/evp.h>
```

```
EVP_PKEY_CTX *EVP_PKEY_CTX_new(EVP_PKEY *pkey, ENGINE *e);
EVP_PKEY_CTX *EVP_PKEY_CTX_new_id(int id, ENGINE *e);
EVP_PKEY_CTX *EVP_PKEY_CTX_dup(EVP_PKEY_CTX *ctx);
void EVP_PKEY_CTX_free(EVP_PKEY_CTX *ctx);
```

DESCRIPTION

The EVP_PKEY_CTX_new() function allocates public key algorithm context using the algorithm specified in **pkey** and ENGINE **e**.

The EVP_PKEY_CTX_new_id() function allocates public key algorithm context using the algorithm specified by **id** and ENGINE **e**. It is normally used when no **EVP_PKEY** structure is associated with the operations, for example during parameter generation of key genration for some algorithms.

EVP_PKEY_CTX_dup() duplicates the context **ctx**.

EVP_PKEY_CTX_free() frees up the context **ctx**.

NOTES

The **EVP_PKEY_CTX** structure is an opaque public key algorithm context used by the OpenSSL high level public key API. Contexts **MUST NOT** be shared between threads: that is it is not permissible to use the same context simultaneously in two threads.

RETURN VALUES

EVP_PKEY_CTX_new(), EVP_PKEY_CTX_new_id(), EVP_PKEY_CTX_dup() returns either the newly allocated **EVP_PKEY_CTX** structure of **NULL** if an error occurred.

EVP_PKEY_CTX_free() does not return a value.

SEE ALSO

EVP_PKEY_new(3)

HISTORY

These functions were first added to OpenSSL 1.0.0.

Name

EVP_PKEY_decrypt_init and EVP_PKEY_decrypt — decrypt using a public key algorithm

Synopsis

```
#include  <openssl/evp.h>
```

```
int EVP_PKEY_decrypt_init(EVP_PKEY_CTX *ctx);
int EVP_PKEY_decrypt(EVP_PKEY_CTX *ctx,
                       unsigned char *out, size_t *outlen,
                       const unsigned char *in, size_t inlen);
```

DESCRIPTION

The EVP_PKEY_decrypt_init() function initializes a public key algorithm context using key **pkey** for a decryption operation.

The EVP_PKEY_decrypt() function performs a public key decryption operation using **ctx**. The data to be decrypted is specified using the **in** and **inlen** parameters. If **out** is **NULL** then the maximum size of the output buffer is written to the **outlen** parameter. If **out** is not **NULL** then before the call the **outlen** parameter should contain the length of the **out** buffer, if the call is successful the decrypted data is written to **out** and the amount of data written to **outlen**.

NOTES

After the call to EVP_PKEY_decrypt_init() algorithm specific control operations can be performed to set any appropriate parameters for the operation.

The function EVP_PKEY_decrypt() can be called more than once on the same context if several operations are performed using the same parameters.

RETURN VALUES

EVP_PKEY_decrypt_init() and EVP_PKEY_decrypt() return 1 for success and 0 or a negative value for failure. In particular a return value of -2 indicates the operation is not supported by the public key algorithm.

EXAMPLE

Decrypt data using OAEP (for RSA keys):

```
#include <openssl/evp.h>
#include <openssl/rsa.h>

EVP_PKEY_CTX *ctx;
unsigned char *out, *in;
size_t outlen, inlen;
EVP_PKEY *key;
/* NB: assumes key in, inlen are already set up
 * and that key is an RSA private key
 */
ctx = EVP_PKEY_CTX_new(key);
if (!ctx)
       /* Error occurred */
if (EVP_PKEY_decrypt_init(ctx) <= 0)
       /* Error */
if (EVP_PKEY_CTX_set_rsa_padding(ctx, RSA_OAEP_PADDING) <= 0)
       /* Error */

/* Determine buffer length */
if (EVP_PKEY_decrypt(ctx, NULL, &outlen, in, inlen) <= 0)
       /* Error */
```

```
out = OPENSSL_malloc(outlen);

if (!out)
       /* malloc failure */

if (EVP_PKEY_decrypt(ctx, out, &outlen, in, inlen) <= 0)
       /* Error */

/* Decrypted data is outlen bytes written to buffer out */
```

SEE ALSO

EVP_PKEY_CTX_new(3), EVP_PKEY_encrypt(3), EVP_PKEY_sign(3), EVP_PKEY_verify(3), EVP_PKEY_verify_recover(3), EVP_PKEY_derive(3)

HISTORY

These functions were first added to OpenSSL 1.0.0.

Name

EVP_PKEY_derive_init, EVP_PKEY_derive_set_peer and EVP_PKEY_derive — derive public key algorithm shared secret.

Synopsis

```
#include  <openssl/evp.h>
```

```
int EVP_PKEY_derive_init(EVP_PKEY_CTX *ctx);
int EVP_PKEY_derive_set_peer(EVP_PKEY_CTX *ctx, EVP_PKEY *peer);
int EVP_PKEY_derive(EVP_PKEY_CTX *ctx, unsigned char *key, size_t *keylen);
```

DESCRIPTION

The EVP_PKEY_derive_init() function initializes a public key algorithm context using key **pkey** for shared secret derivation.

The EVP_PKEY_derive_set_peer() function sets the peer key: this will normally be a public key.

The EVP_PKEY_derive() derives a shared secret using **ctx**. If **key** is **NULL** then the maximum size of the output buffer is written to the **keylen** parameter. If **key** is not **NULL** then before the call the **keylen** parameter should contain the length of the **key** buffer, if the call is successful the shared secret is written to **key** and the amount of data written to **keylen**.

NOTES

After the call to EVP_PKEY_derive_init() algorithm specific control operations can be performed to set any appropriate parameters for the operation.

The function EVP_PKEY_derive() can be called more than once on the same context if several operations are performed using the same parameters.

RETURN VALUES

EVP_PKEY_derive_init() and EVP_PKEY_derive() return 1 for success and 0 or a negative value for failure. In particular a return value of -2 indicates the operation is not supported by the public key algorithm.

EXAMPLE

Derive shared secret (for example DH or EC keys):

```
#include <openssl/evp.h>
#include <openssl/rsa.h>

EVP_PKEY_CTX *ctx;
unsigned char *skey;
size_t skeylen;
EVP_PKEY *pkey, *peerkey;
/* NB: assumes pkey, peerkey have been already set up */

ctx = EVP_PKEY_CTX_new(pkey);
if (!ctx)
       /* Error occurred */
if (EVP_PKEY_derive_init(ctx) <= 0)
       /* Error */
if (EVP_PKEY_derive_set_peer(ctx, peerkey) <= 0)
       /* Error */

/* Determine buffer length */
if (EVP_PKEY_derive(ctx, NULL, &skeylen) <= 0)
       /* Error */
```

```
skey = OPENSSL_malloc(skeylen);

if (!skey)
       /* malloc failure */

if (EVP_PKEY_derive(ctx, skey, &skeylen) <= 0)
       /* Error */

/* Shared secret is skey bytes written to buffer skey */
```

SEE ALSO

EVP_PKEY_CTX_new(3), EVP_PKEY_encrypt(3), EVP_PKEY_decrypt(3), EVP_PKEY_sign(3), EVP_PKEY_verify(3), EVP_PKEY_verify_recover(3),

HISTORY

These functions were first added to OpenSSL 1.0.0.

Name

EVP_PKEY_encrypt_init and EVP_PKEY_encrypt — encrypt using a public key algorithm

Synopsis

```
#include  <openssl/evp.h>
```

```
int EVP_PKEY_encrypt_init(EVP_PKEY_CTX *ctx);
int EVP_PKEY_encrypt(EVP_PKEY_CTX *ctx,
                       unsigned char *out, size_t *outlen,
                       const unsigned char *in, size_t inlen);
```

DESCRIPTION

The EVP_PKEY_encrypt_init() function initializes a public key algorithm context using key **pkey** for an encryption operation.

The EVP_PKEY_encrypt() function performs a public key encryption operation using **ctx**. The data to be encrypted is specified using the **in** and **inlen** parameters. If **out** is **NULL** then the maximum size of the output buffer is written to the **outlen** parameter. If **out** is not **NULL** then before the call the **outlen** parameter should contain the length of the **out** buffer, if the call is successful the encrypted data is written to **out** and the amount of data written to **outlen**.

NOTES

After the call to EVP_PKEY_encrypt_init() algorithm specific control operations can be performed to set any appropriate parameters for the operation.

The function EVP_PKEY_encrypt() can be called more than once on the same context if several operations are performed using the same parameters.

RETURN VALUES

EVP_PKEY_encrypt_init() and EVP_PKEY_encrypt() return 1 for success and 0 or a negative value for failure. In particular a return value of -2 indicates the operation is not supported by the public key algorithm.

EXAMPLE

Encrypt data using OAEP (for RSA keys). See also PEM_read_PUBKEY(3) or d2i_X509(3) for means to load a public key. You may also simply set 'eng = NULL;' to start with the default OpenSSL RSA implementation:

```
#include <openssl/evp.h>
#include <openssl/rsa.h>
#include <openssl/engine.h>

EVP_PKEY_CTX *ctx;
ENGINE *eng;
unsigned char *out, *in;
size_t outlen, inlen;
EVP_PKEY *key;
/* NB: assumes eng, key, in, inlen are already set up,
 * and that key is an RSA public key
 */
ctx = EVP_PKEY_CTX_new(key,eng);
if (!ctx)
       /* Error occurred */
if (EVP_PKEY_encrypt_init(ctx) <= 0)
       /* Error */
if (EVP_PKEY_CTX_set_rsa_padding(ctx, RSA_OAEP_PADDING) <= 0)
       /* Error */
/* Determine buffer length */
if (EVP_PKEY_encrypt(ctx, NULL, &outlen, in, inlen) <= 0)
       /* Error */

out = OPENSSL_malloc(outlen);

if (!out)
       /* malloc failure */

if (EVP_PKEY_encrypt(ctx, out, &outlen, in, inlen) <= 0)
       /* Error */

/* Encrypted data is outlen bytes written to buffer out */
```

SEE ALSO

d2i_X509(3), engine(3), EVP_PKEY_CTX_new(3), EVP_PKEY_decrypt(3), EVP_PKEY_sign(3), EVP_PKEY_verify(3), EVP_PKEY_verify_recover(3), EVP_PKEY_derive(3)

HISTORY

These functions were first added to OpenSSL 1.0.0.

Name

EVP_PKEY_get_default_digest_nid — get default signature digest

Synopsis

```
#include  <openssl/evp.h>
int EVP_PKEY_get_default_digest_nid(EVP_PKEY *pkey, int *pnid);
```

DESCRIPTION

The EVP_PKEY_get_default_digest_nid() function sets **pnid** to the default message digest NID for the public key signature operations associated with key **pkey**.

NOTES

For all current standard OpenSSL public key algorithms SHA1 is returned.

RETURN VALUES

The EVP_PKEY_get_default_digest_nid() function returns 1 if the message digest is advisory (that is other digests can be used) and 2 if it is mandatory (other digests can not be used). It returns 0 or a negative value for failure. In particular a return value of -2 indicates the operation is not supported by the public key algorithm.

SEE ALSO

EVP_PKEY_CTX_new(3), EVP_PKEY_sign(3), EVP_PKEY_verify(3), EVP_PKEY_verify_recover(3),

HISTORY

This function was first added to OpenSSL 1.0.0.

Name

EVP_PKEY_keygen_init, EVP_PKEY_keygen, EVP_PKEY_paramgen_init, EVP_PKEY_paramgen, EVP_PKEY_CTX_set_cb, EVP_PKEY_CTX_get_cb, EVP_PKEY_CTX_get_keygen_info, EVP_PKEVP_PKEY_CTX_set_app_data and EVP_PKEY_CTX_get_app_data — key and parameter generation functions

Synopsis

```
#include  <openssl/evp.h>
```

```
int EVP_PKEY_keygen_init(EVP_PKEY_CTX *ctx);
int EVP_PKEY_keygen(EVP_PKEY_CTX *ctx, EVP_PKEY **ppkey);
int EVP_PKEY_paramgen_init(EVP_PKEY_CTX *ctx);
int EVP_PKEY_paramgen(EVP_PKEY_CTX *ctx, EVP_PKEY **ppkey);
```

```
typedef int EVP_PKEY_gen_cb(EVP_PKEY_CTX *ctx);
```

```
void EVP_PKEY_CTX_set_cb(EVP_PKEY_CTX *ctx, EVP_PKEY_gen_cb *cb);
EVP_PKEY_gen_cb *EVP_PKEY_CTX_get_cb(EVP_PKEY_CTX *ctx);
```

```
int EVP_PKEY_CTX_get_keygen_info(EVP_PKEY_CTX *ctx, int idx);
```

```
void EVP_PKEY_CTX_set_app_data(EVP_PKEY_CTX *ctx, void *data);
void *EVP_PKEY_CTX_get_app_data(EVP_PKEY_CTX *ctx);
```

DESCRIPTION

The EVP_PKEY_keygen_init() function initializes a public key algorithm context using key **pkey** for a key genration operation.

The EVP_PKEY_keygen() function performs a key generation operation, the generated key is written to **ppkey**.

The functions EVP_PKEY_paramgen_init() and EVP_PKEY_paramgen() are similar except parameters are generated.

The function EVP_PKEY_set_cb() sets the key or parameter generation callback to **cb**. The function EVP_PKEY_CTX_get_cb() returns the key or parameter generation callback.

The function EVP_PKEY_CTX_get_keygen_info() returns parameters associated with the generation operation. If **idx** is -1 the total number of parameters available is returned. Any non negative value returns the value of that parameter. EVP_PKEY_CTX_gen_keygen_info() with a non-negative value for **idx** should only be called within the generation callback.

If the callback returns 0 then the key genration operation is aborted and an error occurs. This might occur during a time consuming operation where a user clicks on a "cancel" button.

The functions EVP_PKEY_CTX_set_app_data() and EVP_PKEY_CTX_get_app_data() set and retrieve an opaque pointer. This can be used to set some application defined value which can be retrieved in the callback: for example a handle which is used to update a "progress dialog".

NOTES

After the call to EVP_PKEY_keygen_init() or EVP_PKEY_paramgen_init() algorithm specific control operations can be performed to set any appropriate parameters for the operation.

The functions EVP_PKEY_keygen() and EVP_PKEY_paramgen() can be called more than once on the same context if several operations are performed using the same parameters.

The meaning of the parameters passed to the callback will depend on the algorithm and the specifiic implementation of the algorithm. Some might not give any useful information at all during key or parameter generation. Others might not even call the callback.

The operation performed by key or parameter generation depends on the algorithm used. In some cases (e.g. EC with a supplied named curve) the "generation" option merely sets the appropriate fields in an EVP_PKEY structure.

In OpenSSL an EVP_PKEY structure containing a private key also contains the public key components and parameters (if any). An OpenSSL private key is equivalent to what some libraries call a "key pair". A private key can be used in functions which require the use of a public key or parameters.

RETURN VALUES

EVP_PKEY_keygen_init(), EVP_PKEY_paramgen_init(), EVP_PKEY_keygen() and EVP_PKEY_paramgen() return 1 for success and 0 or a negative value for failure. In particular a return value of -2 indicates the operation is not supported by the public key algorithm.

EXAMPLES

Generate a 2048 bit RSA key:

```
#include <openssl/evp.h>
#include <openssl/rsa.h>

EVP_PKEY_CTX *ctx;
EVP_PKEY *pkey = NULL;
ctx = EVP_PKEY_CTX_new_id(EVP_PKEY_RSA, NULL);
if (!ctx)
       /* Error occurred */
if (EVP_PKEY_keygen_init(ctx) <= 0)
       /* Error */
if (EVP_PKEY_CTX_set_rsa_keygen_bits(ctx, 2048) <= 0)
       /* Error */

/* Generate key */
if (EVP_PKEY_keygen(ctx, &pkey) <= 0)
       /* Error */
```

Generate a key from a set of parameters:

```
#include <openssl/evp.h>
#include <openssl/rsa.h>

EVP_PKEY_CTX *ctx;
EVP_PKEY *pkey = NULL, *param;
/* Assumed param is set up already */
ctx = EVP_PKEY_CTX_new(param);
if (!ctx)
       /* Error occurred */
if (EVP_PKEY_keygen_init(ctx) <= 0)
       /* Error */

/* Generate key */
if (EVP_PKEY_keygen(ctx, &pkey) <= 0)
       /* Error */
```

Example of generation callback for OpenSSL public key implementations:

```
/* Application data is a BIO to output status to */

EVP_PKEY_CTX_set_app_data(ctx, status_bio);

static int genpkey_cb(EVP_PKEY_CTX *ctx)
       {
       char c='*';
       BIO *b = EVP_PKEY_CTX_get_app_data(ctx);
       int p;
       p = EVP_PKEY_CTX_get_keygen_info(ctx, 0);
       if (p == 0) c='.';
       if (p == 1) c='+';
       if (p == 2) c='*';
       if (p == 3) c='\n';
       BIO_write(b,&c,1);
       (void)BIO_flush(b);
       return 1;
       }
```

SEE ALSO

EVP_PKEY_CTX_new(3), EVP_PKEY_encrypt(3), EVP_PKEY_decrypt(3), EVP_PKEY_sign(3), EVP_PKEY_verify(3), EVP_PKEY_verify_recover(3), EVP_PKEY_derive(3)

HISTORY

These functions were first added to OpenSSL 1.0.0.

Name

EVP_PKEY_new and EVP_PKEY_free — private key allocation functions.

Synopsis

```
#include  <openssl/evp.h>
```

```
EVP_PKEY *EVP_PKEY_new(void);
void EVP_PKEY_free(EVP_PKEY *key);
```

DESCRIPTION

The EVP_PKEY_new() function allocates an empty **EVP_PKEY** structure which is used by OpenSSL to store private keys.

EVP_PKEY_free() frees up the private key **key**.

NOTES

The **EVP_PKEY** structure is used by various OpenSSL functions which require a general private key without reference to any particular algorithm.

The structure returned by EVP_PKEY_new() is empty. To add a private key to this empty structure the functions described in EVP_PKEY_set1_RSA(3) should be used.

RETURN VALUES

EVP_PKEY_new() returns either the newly allocated **EVP_PKEY** structure of **NULL** if an error occurred.

EVP_PKEY_free() does not return a value.

SEE ALSO

EVP_PKEY_set1_RSA(3)

HISTORY

TBA

Name

EVP_PKEY_print_public, EVP_PKEY_print_private and EVP_PKEY_print_params — public key algorithm printing routines.

Synopsis

```
#include  <openssl/evp.h>
```

```
int EVP_PKEY_print_public(BIO *out, const EVP_PKEY *pkey,
                               int indent, ASN1_PCTX *pctx);
int EVP_PKEY_print_private(BIO *out, const EVP_PKEY *pkey,
                               int indent, ASN1_PCTX *pctx);
int EVP_PKEY_print_params(BIO *out, const EVP_PKEY *pkey,
                               int indent, ASN1_PCTX *pctx);
```

DESCRIPTION

The functions EVP_PKEY_print_public(), EVP_PKEY_print_private() and EVP_PKEY_print_params() print out the public, private or parameter components of key **pkey** respectively. The key is sent to BIO **out** in human readable form. The parameter **indent** indicated how far the printout should be indented.

The **pctx** parameter allows the print output to be finely tuned by using ASN1 printing options. If **pctx** is set to NULL then default values will be used.

NOTES

Currently no public key algorithms include any options in the **pctx** parameter parameter.

If the key does not include all the components indicated by the function then only those contained in the key will be printed. For example passing a public key to EVP_PKEY_print_private() will only print the public components.

RETURN VALUES

These functions all return 1 for success and 0 or a negative value for failure. In particular a return value of -2 indicates the operation is not supported by the public key algorithm.

SEE ALSO

EVP_PKEY_CTX_new(3), EVP_PKEY_keygen(3)

HISTORY

These functions were first added to OpenSSL 1.0.0.

Name

EVP_PKEY_set1_RSA, EVP_PKEY_set1_DSA, EVP_PKEY_set1_DH, EVP_PKEY_set1_EC_KEY, EVP_PKEY_get1_RSA, EVP_PKEY_get1_DSA, EVP_PKEY_get1_DH, EVP_PKEY_get1_EC_KEY, EVP_PKEY_assign_RSA, EVP_PKEY_assign_DSA, EVP_PKEY_assign_DH, EVP_PKEY_assign_EC_KEY and EVP_PKEY_type — EVP_PKEY assignment functions.

Synopsis

```
#include  <openssl/evp.h>
```

```
int EVP_PKEY_set1_RSA(EVP_PKEY *pkey,RSA *key);
int EVP_PKEY_set1_DSA(EVP_PKEY *pkey,DSA *key);
int EVP_PKEY_set1_DH(EVP_PKEY *pkey,DH *key);
int EVP_PKEY_set1_EC_KEY(EVP_PKEY *pkey,EC_KEY *key);
```

```
RSA *EVP_PKEY_get1_RSA(EVP_PKEY *pkey);
DSA *EVP_PKEY_get1_DSA(EVP_PKEY *pkey);
DH *EVP_PKEY_get1_DH(EVP_PKEY *pkey);
EC_KEY *EVP_PKEY_get1_EC_KEY(EVP_PKEY *pkey);
```

```
int EVP_PKEY_assign_RSA(EVP_PKEY *pkey,RSA *key);
int EVP_PKEY_assign_DSA(EVP_PKEY *pkey,DSA *key);
int EVP_PKEY_assign_DH(EVP_PKEY *pkey,DH *key);
int EVP_PKEY_assign_EC_KEY(EVP_PKEY *pkey,EC_KEY *key);
```

```
int EVP_PKEY_type(int type);
```

DESCRIPTION

EVP_PKEY_set1_RSA(), EVP_PKEY_set1_DSA(), EVP_PKEY_set1_DH() and EVP_PKEY_set1_EC_KEY() set the key referenced by **pkey** to **key**.

EVP_PKEY_get1_RSA(), EVP_PKEY_get1_DSA(), EVP_PKEY_get1_DH() and EVP_PKEY_get1_EC_KEY() return the referenced key in **pkey** or **NULL** if the key is not of the correct type.

EVP_PKEY_assign_RSA(), EVP_PKEY_assign_DSA(), EVP_PKEY_assign_DH() and EVP_PKEY_assign_EC_KEY() also set the referenced key to **key** however these use the supplied **key** internally and so **key** will be freed when the parent **pkey** is freed.

EVP_PKEY_type() returns the type of key corresponding to the value **type**. The type of a key can be obtained with EVP_PKEY_type(pkey->type). The return value will be EVP_PKEY_RSA, EVP_PKEY_DSA, EVP_PKEY_DH or EVP_PKEY_EC for the corresponding key types or NID_undef if the key type is unassigned.

NOTES

In accordance with the OpenSSL naming convention the key obtained from or assigned to the **pkey** using the **1** functions must be freed as well as **pkey**.

EVP_PKEY_assign_RSA(), EVP_PKEY_assign_DSA(), EVP_PKEY_assign_DH() and EVP_PKEY_assign_EC_KEY() are implemented as macros.

RETURN VALUES

EVP_PKEY_set1_RSA(), EVP_PKEY_set1_DSA(), EVP_PKEY_set1_DH() and EVP_PKEY_set1_EC_KEY() return 1 for success or 0 for failure.

EVP_PKEY_get1_RSA(), EVP_PKEY_get1_DSA(), EVP_PKEY_get1_DH() and EVP_PKEY_get1_EC_KEY() return the referenced key or **NULL** if an error occurred.

EVP_PKEY_assign_RSA(), EVP_PKEY_assign_DSA(), EVP_PKEY_assign_DH() and EVP_PKEY_assign_EC_KEY() return 1 for success and 0 for failure.

SEE ALSO

EVP_PKEY_new(3)

HISTORY

TBA

Name

EVP_PKEY_sign_init and EVP_PKEY_sign — sign using a public key algorithm

Synopsis

```
#include  <openssl/evp.h>
```

```
int EVP_PKEY_sign_init(EVP_PKEY_CTX *ctx);
int EVP_PKEY_sign(EVP_PKEY_CTX *ctx,
                       unsigned char *sig, size_t *siglen,
                       const unsigned char *tbs, size_t tbslen);
```

DESCRIPTION

The EVP_PKEY_sign_init() function initializes a public key algorithm context using key **pkey** for a signing operation.

The EVP_PKEY_sign() function performs a public key signing operation using **ctx**. The data to be signed is specified using the **tbs** and **tbslen** parameters. If **sig** is **NULL** then the maximum size of the output buffer is written to the **siglen** parameter. If **sig** is not **NULL** then before the call the **siglen** parameter should contain the length of the **sig** buffer, if the call is successful the signature is written to **sig** and the amount of data written to **siglen**.

NOTES

EVP_PKEY_sign() does not hash the data to be signed, and therefore is normally used to sign digests. For signing arbitrary messages, see the EVP_DigestSignInit(3) and EVP_SignInit(3) signing interfaces instead.

After the call to EVP_PKEY_sign_init() algorithm specific control operations can be performed to set any appropriate parameters for the operation (see EVP_PKEY_CTX_ctrl(3)).

The function EVP_PKEY_sign() can be called more than once on the same context if several operations are performed using the same parameters.

RETURN VALUES

EVP_PKEY_sign_init() and EVP_PKEY_sign() return 1 for success and 0 or a negative value for failure. In particular a return value of -2 indicates the operation is not supported by the public key algorithm.

EXAMPLE

Sign data using RSA with PKCS#1 padding and SHA256 digest:

```
#include <openssl/evp.h>
#include <openssl/rsa.h>

EVP_PKEY_CTX *ctx;
/* md is a SHA-256 digest in this example. */
unsigned char *md, *sig;
size_t mdlen = 32, siglen;
EVP_PKEY *signing_key;

/*
 * NB: assumes signing_key and md are set up before the next
 * step. signing_key must be an RSA private key and md must
 * point to the SHA-256 digest to be signed.
 */
ctx = EVP_PKEY_CTX_new(signing_key, NULL /* no engine */);
if (!ctx)
       /* Error occurred */
```

```
if (EVP_PKEY_sign_init(ctx) <= 0)
       /* Error */
if (EVP_PKEY_CTX_set_rsa_padding(ctx, RSA_PKCS1_PADDING) <= 0)
       /* Error */
if (EVP_PKEY_CTX_set_signature_md(ctx, EVP_sha256()) <= 0)
       /* Error */

/* Determine buffer length */
if (EVP_PKEY_sign(ctx, NULL, &siglen, md, mdlen) <= 0)
       /* Error */

sig = OPENSSL_malloc(siglen);

if (!sig)
       /* malloc failure */

if (EVP_PKEY_sign(ctx, sig, &siglen, md, mdlen) <= 0)
       /* Error */

/* Signature is siglen bytes written to buffer sig */
```

SEE ALSO

EVP_PKEY_CTX_new(3), EVP_PKEY_CTX_ctrl(3), EVP_PKEY_encrypt(3), EVP_PKEY_decrypt(3), EVP_PKEY_verify(3), EVP_PKEY_verify_recover(3), EVP_PKEY_derive(3)

HISTORY

These functions were first added to OpenSSL 1.0.0.

Name

EVP_PKEY_verify_init and EVP_PKEY_verify — signature verification using a public key algorithm

Synopsis

```
#include  <openssl/evp.h>
```

```
int EVP_PKEY_verify_init(EVP_PKEY_CTX *ctx);
int EVP_PKEY_verify(EVP_PKEY_CTX *ctx,
                        const unsigned char *sig, size_t siglen,
                        const unsigned char *tbs, size_t tbslen);
```

DESCRIPTION

The EVP_PKEY_verify_init() function initializes a public key algorithm context using key **pkey** for a signature verification operation.

The EVP_PKEY_verify() function performs a public key verification operation using **ctx**. The signature is specified using the **sig** and **siglen** parameters. The verified data (i.e. the data believed originally signed) is specified using the **tbs** and **tbslen** parameters.

NOTES

After the call to EVP_PKEY_verify_init() algorithm specific control operations can be performed to set any appropriate parameters for the operation.

The function EVP_PKEY_verify() can be called more than once on the same context if several operations are performed using the same parameters.

RETURN VALUES

EVP_PKEY_verify_init() and EVP_PKEY_verify() return 1 if the verification was successful and 0 if it failed. Unlike other functions the return value 0 from EVP_PKEY_verify() only indicates that the signature did not not verify successfully (that is tbs did not match the original data or the signature was of invalid form) it is not an indication of a more serious error.

A negative value indicates an error other that signature verification failure. In particular a return value of -2 indicates the operation is not supported by the public key algorithm.

EXAMPLE

Verify signature using PKCS#1 and SHA256 digest:

```
#include <openssl/evp.h>
#include <openssl/rsa.h>

EVP_PKEY_CTX *ctx;
unsigned char *md, *sig;
size_t mdlen, siglen;
EVP_PKEY *verify_key;
/* NB: assumes verify_key, sig, siglen md and mdlen are already set up
 * and that verify_key is an RSA public key
 */
ctx = EVP_PKEY_CTX_new(verify_key);
if (!ctx)
       /* Error occurred */
if (EVP_PKEY_verify_init(ctx) <= 0)
       /* Error */
if (EVP_PKEY_CTX_set_rsa_padding(ctx, RSA_PKCS1_PADDING) <= 0)
       /* Error */
```

```
if (EVP_PKEY_CTX_set_signature_md(ctx, EVP_sha256()) <= 0)
       /* Error */

/* Perform operation */
ret = EVP_PKEY_verify(ctx, sig, siglen, md, mdlen);

/* ret == 1 indicates success, 0 verify failure and < 0 for some
 * other error.
 */
```

SEE ALSO

EVP_PKEY_CTX_new(3), EVP_PKEY_encrypt(3), EVP_PKEY_decrypt(3), EVP_PKEY_sign(3), EVP_PKEY_verify_recover(3), EVP_PKEY_derive(3)

HISTORY

These functions were first added to OpenSSL 1.0.0.

Name

EVP_PKEY_verify_recover_init and EVP_PKEY_verify_recover — recover signature using a public key algorithm

Synopsis

```
#include  <openssl/evp.h>
```

```
int EVP_PKEY_verify_recover_init(EVP_PKEY_CTX *ctx);
int EVP_PKEY_verify_recover(EVP_PKEY_CTX *ctx,
                        unsigned char *rout, size_t *routlen,
                        const unsigned char *sig, size_t siglen);
```

DESCRIPTION

The EVP_PKEY_verify_recover_init() function initializes a public key algorithm context using key **pkey** for a verify recover operation.

The EVP_PKEY_verify_recover() function recovers signed data using **ctx**. The signature is specified using the **sig** and **siglen** parameters. If **rout** is **NULL** then the maximum size of the output buffer is written to the **routlen** parameter. If **rout** is not **NULL** then before the call the **routlen** parameter should contain the length of the **rout** buffer, if the call is successful recovered data is written to **rout** and the amount of data written to **routlen**.

NOTES

Normally an application is only interested in whether a signature verification operation is successful in those cases the EVP_verify() function should be used.

Sometimes however it is useful to obtain the data originally signed using a signing operation. Only certain public key algorithms can recover a signature in this way (for example RSA in PKCS padding mode).

After the call to EVP_PKEY_verify_recover_init() algorithm specific control operations can be performed to set any appropriate parameters for the operation.

The function EVP_PKEY_verify_recover() can be called more than once on the same context if several operations are performed using the same parameters.

RETURN VALUES

EVP_PKEY_verify_recover_init() and EVP_PKEY_verify_recover() return 1 for success and 0 or a negative value for failure. In particular a return value of -2 indicates the operation is not supported by the public key algorithm.

EXAMPLE

Recover digest originally signed using PKCS#1 and SHA256 digest:

```
#include <openssl/evp.h>
#include <openssl/rsa.h>

EVP_PKEY_CTX *ctx;
unsigned char *rout, *sig;
size_t routlen, siglen;
EVP_PKEY *verify_key;
/* NB: assumes verify_key, sig and siglen are already set up
 * and that verify_key is an RSA public key
 */
ctx = EVP_PKEY_CTX_new(verify_key);
if (!ctx)
```

```
       /* Error occurred */
if (EVP_PKEY_verify_recover_init(ctx) <= 0)
       /* Error */
if (EVP_PKEY_CTX_set_rsa_padding(ctx, RSA_PKCS1_PADDING) <= 0)
       /* Error */
if (EVP_PKEY_CTX_set_signature_md(ctx, EVP_sha256()) <= 0)
       /* Error */

/* Determine buffer length */
if (EVP_PKEY_verify_recover(ctx, NULL, &routlen, sig, siglen) <= 0)
       /* Error */

rout = OPENSSL_malloc(routlen);

if (!rout)
       /* malloc failure */

if (EVP_PKEY_verify_recover(ctx, rout, &routlen, sig, siglen) <= 0)
       /* Error */

/* Recovered data is routlen bytes written to buffer rout */
```

SEE ALSO

EVP_PKEY_CTX_new(3), EVP_PKEY_encrypt(3), EVP_PKEY_decrypt(3), EVP_PKEY_sign(3), EVP_PKEY_verify(3), EVP_PKEY_derive(3)

HISTORY

These functions were first added to OpenSSL 1.0.0.

Name

EVP_SealInit, EVP_SealUpdate and EVP_SealFinal — EVP envelope encryption

Synopsis

```
#include  <openssl/evp.h>
```

```
int EVP_SealInit(EVP_CIPHER_CTX *ctx, const EVP_CIPHER *type,
                 unsigned char **ek, int *ekl, unsigned char *iv,
                 EVP_PKEY **pubk, int npubk);
int EVP_SealUpdate(EVP_CIPHER_CTX *ctx, unsigned char *out,
        int *outl, unsigned char *in, int inl);
int EVP_SealFinal(EVP_CIPHER_CTX *ctx, unsigned char *out,
        int *outl);
```

DESCRIPTION

The EVP envelope routines are a high level interface to envelope encryption. They generate a random key and IV (if required) then "envelope" it by using public key encryption. Data can then be encrypted using this key.

EVP_SealInit() initializes a cipher context **ctx** for encryption with cipher **type** using a random secret key and IV. **type** is normally supplied by a function such as EVP_des_cbc(). The secret key is encrypted using one or more public keys, this allows the same encrypted data to be decrypted using any of the corresponding private keys. **ek** is an array of buffers where the public key encrypted secret key will be written, each buffer must contain enough room for the corresponding encrypted key: that is **ek[i]** must have room for **EVP_PKEY_size(pubk[i])** bytes. The actual size of each encrypted secret key is written to the array **ekl**. **pubk** is an array of **npubk** public keys.

The **iv** parameter is a buffer where the generated IV is written to. It must contain enough room for the corresponding cipher's IV, as determined by (for example) EVP_CIPHER_iv_length(type).

If the cipher does not require an IV then the **iv** parameter is ignored and can be **NULL**.

EVP_SealUpdate() and EVP_SealFinal() have exactly the same properties as the EVP_EncryptUpdate() and EVP_EncryptFinal() routines, as documented on the EVP_EncryptInit(3) manual page.

RETURN VALUES

EVP_SealInit() returns 0 on error or **npubk** if successful.

EVP_SealUpdate() and EVP_SealFinal() return 1 for success and 0 for failure.

NOTES

Because a random secret key is generated the random number generator must be seeded before calling EVP_SealInit().

The public key must be RSA because it is the only OpenSSL public key algorithm that supports key transport.

Envelope encryption is the usual method of using public key encryption on large amounts of data, this is because public key encryption is slow but symmetric encryption is fast. So symmetric encryption is used for bulk encryption and the small random symmetric key used is transferred using public key encryption.

It is possible to call EVP_SealInit() twice in the same way as EVP_EncryptInit(). The first call should have **npubk** set to 0 and (after setting any cipher parameters) it should be called again with **type** set to NULL.

SEE ALSO

evp(3), rand(3), EVP_EncryptInit(3), EVP_OpenInit(3)

HISTORY

EVP_SealFinal() did not return a value before OpenSSL 0.9.7.

Name

EVP_SignInit, EVP_SignInit_ex, EVP_SignUpdate and EVP_SignFinal — EVP signing functions

Synopsis

```
#include  <openssl/evp.h>
```

```
int EVP_SignInit_ex(EVP_MD_CTX *ctx, const EVP_MD *type, ENGINE *impl);
int EVP_SignUpdate(EVP_MD_CTX *ctx, const void *d, unsigned int cnt);
int EVP_SignFinal(EVP_MD_CTX *ctx,unsigned char *sig,unsigned int *s, EVP_PKEY *pkey);
```

```
void EVP_SignInit(EVP_MD_CTX *ctx, const EVP_MD *type);
```

```
int EVP_PKEY_size(EVP_PKEY *pkey);
```

DESCRIPTION

The EVP signature routines are a high level interface to digital signatures.

EVP_SignInit_ex() sets up signing context **ctx** to use digest **type** from ENGINE **impl**. **ctx** must be initialized with EVP_MD_CTX_init() before calling this function.

EVP_SignUpdate() hashes **cnt** bytes of data at **d** into the signature context **ctx**. This function can be called several times on the same **ctx** to include additional data.

EVP_SignFinal() signs the data in **ctx** using the private key **pkey** and places the signature in **sig**. **sig** must be at least EVP_PKEY_size(pkey) bytes in size. **s** is an OUT paramter, and not used as an IN parameter. The number of bytes of data written (i.e. the length of the signature) will be written to the integer at **s**, at most EVP_PKEY_size(pkey) bytes will be written.

EVP_SignInit() initializes a signing context **ctx** to use the default implementation of digest **type**.

EVP_PKEY_size() returns the maximum size of a signature in bytes. The actual signature returned by EVP_SignFinal() may be smaller.

RETURN VALUES

EVP_SignInit_ex(), EVP_SignUpdate() and EVP_SignFinal() return 1 for success and 0 for failure.

EVP_PKEY_size() returns the maximum size of a signature in bytes.

The error codes can be obtained by ERR_get_error(3).

NOTES

The **EVP** interface to digital signatures should almost always be used in preference to the low level interfaces. This is because the code then becomes transparent to the algorithm used and much more flexible.

Due to the link between message digests and public key algorithms the correct digest algorithm must be used with the correct public key type. A list of algorithms and associated public key algorithms appears in EVP_DigestInit(3).

When signing with DSA private keys the random number generator must be seeded or the operation will fail. The random number generator does not need to be seeded for RSA signatures.

The call to EVP_SignFinal() internally finalizes a copy of the digest context. This means that calls to EVP_SignUpdate() and EVP_SignFinal() can be called later to digest and sign additional data.

Since only a copy of the digest context is ever finalized the context must be cleaned up after use by calling EVP_MD_CTX_cleanup() or a memory leak will occur.

BUGS

Older versions of this documentation wrongly stated that calls to EVP_SignUpdate() could not be made after calling EVP_SignFinal().

Since the private key is passed in the call to EVP_SignFinal() any error relating to the private key (for example an unsuitable key and digest combination) will not be indicated until after potentially large amounts of data have been passed through EVP_SignUpdate().

It is not possible to change the signing parameters using these function.

The previous two bugs are fixed in the newer EVP_SignDigest*() function.

SEE ALSO

EVP_VerifyInit(3), EVP_DigestInit(3), err(3), evp(3), hmac(3), md2(3), md5(3), mdc2(3), ripemd(3), sha(3), dgst(1)

HISTORY

EVP_SignInit(), EVP_SignUpdate() and EVP_SignFinal() are available in all versions of SSLeay and OpenSSL.

EVP_SignInit_ex() was added in OpenSSL 0.9.7.

Name

EVP_VerifyInit, EVP_VerifyUpdate and EVP_VerifyFinal — EVP signature verification functions

Synopsis

```
#include  <openssl/evp.h>
```

```
int EVP_VerifyInit_ex(EVP_MD_CTX *ctx, const EVP_MD *type, ENGINE *impl);
int EVP_VerifyUpdate(EVP_MD_CTX *ctx, const void *d, unsigned int cnt);
int EVP_VerifyFinal(EVP_MD_CTX *ctx,unsigned char *sigbuf, unsigned int siglen,EVP_PKEY *pkey);
```

```
int EVP_VerifyInit(EVP_MD_CTX *ctx, const EVP_MD *type);
```

DESCRIPTION

The EVP signature verification routines are a high level interface to digital signatures.

EVP_VerifyInit_ex() sets up verification context **ctx** to use digest **type** from ENGINE **impl**. **ctx** must be initialized by calling EVP_MD_CTX_init() before calling this function.

EVP_VerifyUpdate() hashes **cnt** bytes of data at **d** into the verification context **ctx**. This function can be called several times on the same **ctx** to include additional data.

EVP_VerifyFinal() verifies the data in **ctx** using the public key **pkey** and against the **siglen** bytes at **sigbuf**.

EVP_VerifyInit() initializes verification context **ctx** to use the default implementation of digest **type**.

RETURN VALUES

EVP_VerifyInit_ex() and EVP_VerifyUpdate() return 1 for success and 0 for failure.

EVP_VerifyFinal() returns 1 for a correct signature, 0 for failure and -1 if some other error occurred.

The error codes can be obtained by ERR_get_error(3).

NOTES

The **EVP** interface to digital signatures should almost always be used in preference to the low level interfaces. This is because the code then becomes transparent to the algorithm used and much more flexible.

Due to the link between message digests and public key algorithms the correct digest algorithm must be used with the correct public key type. A list of algorithms and associated public key algorithms appears in EVP_DigestInit(3).

The call to EVP_VerifyFinal() internally finalizes a copy of the digest context. This means that calls to EVP_VerifyUpdate() and EVP_VerifyFinal() can be called later to digest and verify additional data.

Since only a copy of the digest context is ever finalized the context must be cleaned up after use by calling EVP_MD_CTX_cleanup() or a memory leak will occur.

BUGS

Older versions of this documentation wrongly stated that calls to EVP_VerifyUpdate() could not be made after calling EVP_VerifyFinal().

Since the public key is passed in the call to EVP_SignFinal() any error relating to the private key (for example an unsuitable key and digest combination) will not be indicated until after potentially large amounts of data have been passed through EVP_SignUpdate().

It is not possible to change the signing parameters using these function.

The previous two bugs are fixed in the newer EVP_VerifyDigest*() function.

SEE ALSO

evp(3), EVP_SignInit(3), EVP_DigestInit(3), err(3), evp(3), hmac(3), md2(3), md5(3), mdc2(3), ripemd(3), sha(3), dgst(1)

HISTORY

EVP_VerifyInit(), EVP_VerifyUpdate() and EVP_VerifyFinal() are available in all versions of SSLeay and OpenSSL.

EVP_VerifyInit_ex() was added in OpenSSL 0.9.7

Name

HMAC, HMAC_Init, HMAC_Update, HMAC_Final and HMAC_cleanup — HMAC message authentication code

Synopsis

```
#include  <openssl/hmac.h>
```

```
unsigned char *HMAC(const EVP_MD *evp_md, const void *key,
              int key_len, const unsigned char *d, int n,
              unsigned char *md, unsigned int *md_len);
```

```
void HMAC_CTX_init(HMAC_CTX *ctx);
```

```
int HMAC_Init(HMAC_CTX *ctx, const void *key, int key_len,
              const EVP_MD *md);
int HMAC_Init_ex(HMAC_CTX *ctx, const void *key, int key_len,
                  const EVP_MD *md, ENGINE *impl);
int HMAC_Update(HMAC_CTX *ctx, const unsigned char *data, int len);
int HMAC_Final(HMAC_CTX *ctx, unsigned char *md, unsigned int *len);
```

```
void HMAC_CTX_cleanup(HMAC_CTX *ctx);
void HMAC_cleanup(HMAC_CTX *ctx);
```

DESCRIPTION

HMAC is a MAC (message authentication code), i.e. a keyed hash function used for message authentication, which is based on a hash function.

HMAC() computes the message authentication code of the **n** bytes at **d** using the hash function **evp_md** and the key **key** which is **key_len** bytes long.

It places the result in **md** (which must have space for the output of the hash function, which is no more than **EVP_MAX_MD_SIZE** bytes). If **md** is NULL, the digest is placed in a static array. The size of the output is placed in **md_len**, unless it is **NULL**.

evp_md can be EVP_sha1(), EVP_ripemd160() etc.

HMAC_CTX_init() initialises a **HMAC_CTX** before first use. It must be called.

HMAC_CTX_cleanup() erases the key and other data from the **HMAC_CTX** and releases any associated resources. It must be called when an **HMAC_CTX** is no longer required.

HMAC_cleanup() is an alias for HMAC_CTX_cleanup() included for back compatibility with 0.9.6b, it is deprecated.

The following functions may be used if the message is not completely stored in memory:

HMAC_Init() initializes a **HMAC_CTX** structure to use the hash function **evp_md** and the key **key** which is **key_len** bytes long. It is deprecated and only included for backward compatibility with OpenSSL 0.9.6b.

HMAC_Init_ex() initializes or reuses a **HMAC_CTX** structure to use the function **evp_md** and key **key**. Either can be NULL, in which case the existing one will be reused. HMAC_CTX_init() must have been called before the first use of an **HMAC_CTX** in this function. **N.B. HMAC_Init() had this undocumented behaviour in previous versions of OpenSSL - failure to switch to HMAC_Init_ex() in programs that expect it will cause them to stop working**.

HMAC_Update() can be called repeatedly with chunks of the message to be authenticated (**len** bytes at **data**).

HMAC_Final() places the message authentication code in **md**, which must have space for the hash function output.

RETURN VALUES

HMAC() returns a pointer to the message authentication code or NULL if an error occurred.

HMAC_Init_ex(), HMAC_Update() and HMAC_Final() return 1 for success or 0 if an error occurred.

HMAC_CTX_init() and HMAC_CTX_cleanup() do not return values.

CONFORMING TO

RFC 2104

SEE ALSO

sha(3), evp(3)

HISTORY

HMAC(), HMAC_Init(), HMAC_Update(), HMAC_Final() and HMAC_cleanup() are available since SSLeay 0.9.0.

HMAC_CTX_init(), HMAC_Init_ex() and HMAC_CTX_cleanup() are available since OpenSSL 0.9.7.

HMAC_Init_ex(), HMAC_Update() and HMAC_Final() did not return values in versions of OpenSSL before 1.0.0.

Name

i2d_CMS_bio_stream — output CMS_ContentInfo structure in BER format.

Synopsis

```
#include  <openssl/cms.h>
```

```
int i2d_CMS_bio_stream(BIO *out, CMS_ContentInfo *cms, BIO *data, int flags);
```

DESCRIPTION

i2d_CMS_bio_stream() outputs a CMS_ContentInfo structure in BER format.

It is otherwise identical to the function SMIME_write_CMS().

NOTES

This function is effectively a version of the i2d_CMS_bio() supporting streaming.

BUGS

The prefix "i2d" is arguably wrong because the function outputs BER format.

RETURN VALUES

i2d_CMS_bio_stream() returns 1 for success or 0 for failure.

SEE ALSO

ERR_get_error(3), CMS_sign(3), CMS_verify(3), CMS_encrypt(3), CMS_decrypt(3), SMIME_write_CMS(3), PEM_write_bio_CMS_stream(3)

HISTORY

i2d_CMS_bio_stream() was added to OpenSSL 1.0.0

Name

i2d_PKCS7_bio_stream — output PKCS7 structure in BER format.

Synopsis

```
#include  <openssl/pkcs7.h>
```

```
int i2d_PKCS7_bio_stream(BIO *out, PKCS7 *p7, BIO *data, int flags);
```

DESCRIPTION

i2d_PKCS7_bio_stream() outputs a PKCS7 structure in BER format.

It is otherwise identical to the function SMIME_write_PKCS7().

NOTES

This function is effectively a version of the d2i_PKCS7_bio() supporting streaming.

BUGS

The prefix "d2i" is arguably wrong because the function outputs BER format.

RETURN VALUES

i2d_PKCS7_bio_stream() returns 1 for success or 0 for failure.

SEE ALSO

ERR_get_error(3), PKCS7_sign(3), PKCS7_verify(3), PKCS7_encrypt(3), PKCS7_decrypt(3), SMIME_write_PKCS7(3), PEM_write_bio_PKCS7_stream(3)

HISTORY

i2d_PKCS7_bio_stream() was added to OpenSSL 1.0.0

Name

lh_new, lh_free, lh_insert, lh_delete, lh_retrieve, lh_doall, lh_doall_arg and lh_error — dynamic hash table

Synopsis

```
#include  <openssl/lhash.h>
```

```
DECLARE_LHASH_OF(<type>);
```

```
LHASH *lh_<type>_new();
void lh_<type>_free(LHASH_OF(<type> *table);
```

```
<type> *lh_<type>_insert(LHASH_OF(<type> *table, <type> *data);
<type> *lh_<type>_delete(LHASH_OF(<type> *table, <type> *data);
<type> *lh_retrieve(LHASH_OF<type> *table, <type> *data);
```

```
void lh_<type>_doall(LHASH_OF(<type> *table, LHASH_DOALL_FN_TYPE func);
void lh_<type>_doall_arg(LHASH_OF(<type> *table, LHASH_DOALL_ARG_FN_TYPE func,
         <type>, <type> *arg);
```

```
int lh_<type>_error(LHASH_OF(<type> *table);
```

```
typedef int (*LHASH_COMP_FN_TYPE)(const void *, const void *);
typedef unsigned long (*LHASH_HASH_FN_TYPE)(const void *);
typedef void (*LHASH_DOALL_FN_TYPE)(const void *);
typedef void (*LHASH_DOALL_ARG_FN_TYPE)(const void *, const void *);
```

DESCRIPTION

This library implements type-checked dynamic hash tables. The hash table entries can be arbitrary structures. Usually they consist of key and value fields.

lh_<type>_new() creates a new **LHASH_OF(type)** structure to store arbitrary data entries, and provides the 'hash' and 'compare' callbacks to be used in organising the table's entries. The **hash** callback takes a pointer to a table entry as its argument and returns an unsigned long hash value for its key field. The hash value is normally truncated to a power of 2, so make sure that your hash function returns well mixed low order bits. The **compare** callback takes two arguments (pointers to two hash table entries), and returns 0 if their keys are equal, non-zero otherwise. If your hash table will contain items of some particular type and the **hash** and **compare** callbacks hash/compare these types, then the **DECLARE_LHASH_HASH_FN** and **IMPLEMENT_LHASH_COMP_FN** macros can be used to create callback wrappers of the prototypes required by lh_<type>_new(). These provide per-variable casts before calling the type-specific callbacks written by the application author. These macros, as well as those used for the "doall" callbacks, are defined as;

```
#define DECLARE_LHASH_HASH_FN(name, o_type) \
        unsigned long name##_LHASH_HASH(const void *);
#define IMPLEMENT_LHASH_HASH_FN(name, o_type) \
        unsigned long name##_LHASH_HASH(const void *arg) { \
                const o_type *a = arg; \
                return name##_hash(a); }
#define LHASH_HASH_FN(name) name##_LHASH_HASH

#define DECLARE_LHASH_COMP_FN(name, o_type) \
        int name##_LHASH_COMP(const void *, const void *);
#define IMPLEMENT_LHASH_COMP_FN(name, o_type) \
        int name##_LHASH_COMP(const void *arg1, const void *arg2) { \
                const o_type *a = arg1;                     \
                const o_type *b = arg2; \
                return name##_cmp(a,b); }
#define LHASH_COMP_FN(name) name##_LHASH_COMP

#define DECLARE_LHASH_DOALL_FN(name, o_type) \
        void name##_LHASH_DOALL(void *);
#define IMPLEMENT_LHASH_DOALL_FN(name, o_type) \
        void name##_LHASH_DOALL(void *arg) { \
                o_type *a = arg; \
                name##_doall(a); }
#define LHASH_DOALL_FN(name) name##_LHASH_DOALL

#define DECLARE_LHASH_DOALL_ARG_FN(name, o_type, a_type) \
        void name##_LHASH_DOALL_ARG(void *, void *);
#define IMPLEMENT_LHASH_DOALL_ARG_FN(name, o_type, a_type) \
        void name##_LHASH_DOALL_ARG(void *arg1, void *arg2) { \
                o_type *a = arg1; \
                a_type *b = arg2; \
                name##_doall_arg(a, b); }
#define LHASH_DOALL_ARG_FN(name) name##_LHASH_DOALL_ARG

An example of a hash table storing (pointers to) structures of type 'STUFF'
could be defined as follows;

/* Calculates the hash value of 'tohash' (implemented elsewhere) */
unsigned long STUFF_hash(const STUFF *tohash);
/* Orders 'arg1' and 'arg2' (implemented elsewhere) */
int stuff_cmp(const STUFF *arg1, const STUFF *arg2);
/* Create the type-safe wrapper functions for use in the LHASH internals */
static IMPLEMENT_LHASH_HASH_FN(stuff, STUFF);
static IMPLEMENT_LHASH_COMP_FN(stuff, STUFF);
/* … */
int main(int argc, char *argv[]) {
        /* Create the new hash table using the hash/compare wrappers */
        LHASH_OF(STUFF) *hashtable = lh_STUFF_new(LHASH_HASH_FN(STUFF_hash),
                                  LHASH_COMP_FN(STUFF_cmp));
        /* … */
}
```

lh_<type>_free() frees the **LHASH_OF(type)** structure **table**. Allocated hash table entries will not be freed; consider using lh_<type>_doall() to deallocate any remaining entries in the hash table (see below).

lh_<type>_insert() inserts the structure pointed to by **data** into **table**. If there already is an entry with the same key, the old value is replaced. Note that lh_<type>_insert() stores pointers, the data are not copied.

lh_<type>_delete() deletes an entry from **table**.

lh_<type>_retrieve() looks up an entry in **table**. Normally, **data** is a structure with the key field(s) set; the function will return a pointer to a fully populated structure.

lh_<type>_doall() will, for every entry in the hash table, call **func** with the data item as its parameter. For lh_<type>_doall() and lh_<type>_doall_arg(), function pointer casting should be avoided in the callbacks (see **NOTE**) - instead use the declare/ implement macros to create type-checked wrappers that cast variables prior to calling your type-specific callbacks. An example of this is illustrated here where the callback is used to cleanup resources for items in the hash table prior to the hashtable itself being deallocated:

```
/* Cleans up resources belonging to 'a' (this is implemented elsewhere) */
void STUFF_cleanup_doall(STUFF *a);
/* Implement a prototype-compatible wrapper for "STUFF_cleanup" */
IMPLEMENT_LHASH_DOALL_FN(STUFF_cleanup, STUFF)
        /* … then later in the code … */
/* So to run "STUFF_cleanup" against all items in a hash table … */
lh_STUFF_doall(hashtable, LHASH_DOALL_FN(STUFF_cleanup));
/* Then the hash table itself can be deallocated */
lh_STUFF_free(hashtable);
```

When doing this, be careful if you delete entries from the hash table in your callbacks: the table may decrease in size, moving the item that you are currently on down lower in the hash table - this could cause some entries to be skipped during the iteration. The second best solution to this problem is to set hash->down_load=0 before you start (which will stop the hash table ever decreasing in size). The best solution is probably to avoid deleting items from the hash table inside a "doall" callback!

lh_<type>_doall_arg() is the same as lh_<type>_doall() except that **func** will be called with **arg** as the second argument and **func** should be of type **LHASH_DOALL_ARG_FN_TYPE** (a callback prototype that is passed both the table entry and an extra argument). As with lh_doall(), you can instead choose to declare your callback with a prototype matching the types you are dealing with and use the declare/implement macros to create compatible wrappers that cast variables before calling your type-specific callbacks. An example of this is demonstrated here (printing all hash table entries to a BIO that is provided by the caller):

```
/* Prints item 'a' to 'output_bio' (this is implemented elsewhere) */
void STUFF_print_doall_arg(const STUFF *a, BIO *output_bio);
/* Implement a prototype-compatible wrapper for "STUFF_print" */
static IMPLEMENT_LHASH_DOALL_ARG_FN(STUFF, const STUFF, BIO)
        /* … then later in the code … */
/* Print out the entire hashtable to a particular BIO */
lh_STUFF_doall_arg(hashtable, LHASH_DOALL_ARG_FN(STUFF_print), BIO,
                   logging_bio);
```

lh_<type>_error() can be used to determine if an error occurred in the last operation. lh_<type>_error() is a macro.

RETURN VALUES

lh_<type>_new() returns **NULL** on error, otherwise a pointer to the new **LHASH** structure.

When a hash table entry is replaced, lh_<type>_insert() returns the value being replaced. **NULL** is returned on normal operation and on error.

lh_<type>_delete() returns the entry being deleted. **NULL** is returned if there is no such value in the hash table.

lh_<type>_retrieve() returns the hash table entry if it has been found, **NULL** otherwise.

lh_<type>_error() returns 1 if an error occurred in the last operation, 0 otherwise.

lh_<type>_free(), lh_<type>_doall() and lh_<type>_doall_arg() return no values.

NOTE

The various LHASH macros and callback types exist to make it possible to write type-checked code without resorting to function-prototype casting - an evil that makes application code much harder to audit/verify and also opens the window of opportunity for stack corruption and other hard-to-find bugs. It also, apparently, violates ANSI-C.

The LHASH code regards table entries as constant data. As such, it internally represents lh_insert()'d items with a "const void *" pointer type. This is why callbacks such as those used by lh_doall() and lh_doall_arg() declare their prototypes with "const", even for the parameters that pass back the table items' data pointers - for consistency, user-provided data is "const" at all times as far as the LHASH code is concerned. However, as callers are themselves providing these pointers, they can choose whether they too should be treating all such parameters as constant.

As an example, a hash table may be maintained by code that, for reasons of encapsulation, has only "const" access to the data being indexed in the hash table (ie. it is returned as "const" from elsewhere in their code) - in this case the LHASH prototypes are appropriate as-is. Conversely, if the caller is responsible for the life-time of the data in question, then they may well wish to make modifications to table item passed back in the lh_doall() or lh_doall_arg() callbacks (see the "STUFF_cleanup" example above). If so, the caller can either cast the "const" away (if they're providing the raw callbacks themselves) or use the macros to declare/implement the wrapper functions without "const" types.

Callers that only have "const" access to data they're indexing in a table, yet declare callbacks without constant types (or cast the "const" away themselves), are therefore creating their own risks/bugs without being encouraged to do so by the API. On a related note, those auditing code should pay special attention to any instances of DECLARE/IMPLEMENT_LHASH_DOALL_[ARG_]_FN macros that provide types without any "const" qualifiers.

BUGS

lh_<type>_insert() returns **NULL** both for success and error.

INTERNALS

The following description is based on the SSLeay documentation:

The **lhash** library implements a hash table described in the *Communications of the ACM* in 1991. What makes this hash table different is that as the table fills, the hash table is increased (or decreased) in size via OPENSSL_realloc(). When a 'resize' is done, instead of all hashes being redistributed over twice as many 'buckets', one bucket is split. So when an 'expand' is done, there is only a minimal cost to redistribute some values. Subsequent inserts will cause more single 'bucket' redistributions but there will never be a sudden large cost due to redistributing all the 'buckets'.

The state for a particular hash table is kept in the **LHASH** structure. The decision to increase or decrease the hash table size is made depending on the 'load' of the hash table. The load is the number of items in the hash table divided by the size of the hash table. The default values are as follows. If (hash->up_load < load) => expand. if (hash->down_load > load) => contract. The **up_load** has a default value of 1 and **down_load** has a default value of 2. These numbers can be modified by the application by just playing with the **up_load** and **down_load** variables. The 'load' is kept in a form which is multiplied by 256. So hash->up_load=8*256; will cause a load of 8 to be set.

If you are interested in performance the field to watch is num_comp_calls. The hash library keeps track of the 'hash' value for each item so when a lookup is done, the 'hashes' are compared, if there is a match, then a full compare is done, and hash->num_comp_calls is incremented. If num_comp_calls is not equal to num_delete plus num_retrieve it means that your hash function is generating hashes that are the same for different values. It is probably worth changing your hash function if this is the case because even if your hash table has 10 items in a 'bucket', it can be searched with 10 **unsigned long** compares and 10 linked list traverses. This will be much less expensive that 10 calls to your compare function.

lh_strhash() is a demo string hashing function:

```
unsigned long lh_strhash(const char *c);
```

Since the **LHASH** routines would normally be passed structures, this routine would not normally be passed to lh_<type>_new(), rather it would be used in the function passed to lh_<type>_new().

SEE ALSO

lh_stats(3)

HISTORY

The **lhash** library is available in all versions of SSLeay and OpenSSL. lh_error() was added in SSLeay 0.9.1b.

This manpage is derived from the SSLeay documentation.

In OpenSSL 0.9.7, all lhash functions that were passed function pointers were changed for better type safety, and the function types LHASH_COMP_FN_TYPE, LHASH_HASH_FN_TYPE, LHASH_DOALL_FN_TYPE and LHASH_DOALL_ARG_FN_TYPE became available.

In OpenSSL 1.0.0, the lhash interface was revamped for even better type checking.

Name

lh_stats, lh_node_stats, lh_node_usage_stats, lh_stats_bio, lh_node_stats_bio and lh_node_usage_stats_bio — LHASH statistics

Synopsis

```
#include  <openssl/lhash.h>
```

```
void lh_stats(LHASH *table, FILE *out);
void lh_node_stats(LHASH *table, FILE *out);
void lh_node_usage_stats(LHASH *table, FILE *out);
```

```
void lh_stats_bio(LHASH *table, BIO *out);
void lh_node_stats_bio(LHASH *table, BIO *out);
void lh_node_usage_stats_bio(LHASH *table, BIO *out);
```

DESCRIPTION

The **LHASH** structure records statistics about most aspects of accessing the hash table. This is mostly a legacy of Eric Young writing this library for the reasons of implementing what looked like a nice algorithm rather than for a particular software product.

lh_stats() prints out statistics on the size of the hash table, how many entries are in it, and the number and result of calls to the routines in this library.

lh_node_stats() prints the number of entries for each 'bucket' in the hash table.

lh_node_usage_stats() prints out a short summary of the state of the hash table. It prints the 'load' and the 'actual load'. The load is the average number of data items per 'bucket' in the hash table. The 'actual load' is the average number of items per 'bucket', but only for buckets which contain entries. So the 'actual load' is the average number of searches that will need to find an item in the hash table, while the 'load' is the average number that will be done to record a miss.

lh_stats_bio(), lh_node_stats_bio() and lh_node_usage_stats_bio() are the same as the above, except that the output goes to a **BIO**.

RETURN VALUES

These functions do not return values.

SEE ALSO

bio(3), lhash(3)

HISTORY

These functions are available in all versions of SSLeay and OpenSSL.

This manpage is derived from the SSLeay documentation.

Name

MD2, MD4, MD5, MD2_Init, MD2_Update, MD2_Final, MD4_Init, MD4_Update, MD4_Final, MD5_Init, MD5_Update and MD5_Final — MD2, MD4, and MD5 hash functions

Synopsis

```
#include  <openssl/md2.h>
```

```
unsigned char *MD2(const unsigned char *d, unsigned long n,
                 unsigned char *md);
```

```
int MD2_Init(MD2_CTX *c);
int MD2_Update(MD2_CTX *c, const unsigned char *data,
                 unsigned long len);
int MD2_Final(unsigned char *md, MD2_CTX *c);
```

```
#include  <openssl/md4.h>
```

```
unsigned char *MD4(const unsigned char *d, unsigned long n,
                 unsigned char *md);
```

```
int MD4_Init(MD4_CTX *c);
int MD4_Update(MD4_CTX *c, const void *data,
                 unsigned long len);
int MD4_Final(unsigned char *md, MD4_CTX *c);
```

```
#include  <openssl/md5.h>
```

```
unsigned char *MD5(const unsigned char *d, unsigned long n,
                 unsigned char *md);
```

```
int MD5_Init(MD5_CTX *c);
int MD5_Update(MD5_CTX *c, const void *data,
                 unsigned long len);
int MD5_Final(unsigned char *md, MD5_CTX *c);
```

DESCRIPTION

MD2, MD4, and MD5 are cryptographic hash functions with a 128 bit output.

MD2(), MD4(), and MD5() compute the MD2, MD4, and MD5 message digest of the **n** bytes at **d** and place it in **md** (which must have space for MD2_DIGEST_LENGTH == MD4_DIGEST_LENGTH == MD5_DIGEST_LENGTH == 16 bytes of output). If **md** is NULL, the digest is placed in a static array.

The following functions may be used if the message is not completely stored in memory:

MD2_Init() initializes a **MD2_CTX** structure.

MD2_Update() can be called repeatedly with chunks of the message to be hashed (**len** bytes at **data**).

MD2_Final() places the message digest in **md**, which must have space for MD2_DIGEST_LENGTH == 16 bytes of output, and erases the **MD2_CTX**.

MD4_Init(), MD4_Update(), MD4_Final(), MD5_Init(), MD5_Update(), and MD5_Final() are analogous using an **MD4_CTX** and **MD5_CTX** structure.

Applications should use the higher level functions EVP_DigestInit(3) etc. instead of calling the hash functions directly.

NOTE

MD2, MD4, and MD5 are recommended only for compatibility with existing applications. In new applications, SHA-1 or RIPEMD-160 should be preferred.

RETURN VALUES

MD2(), MD4(), and MD5() return pointers to the hash value.

MD2_Init(), MD2_Update(), MD2_Final(), MD4_Init(), MD4_Update(), MD4_Final(), MD5_Init(), MD5_Update(), and MD5_Final() return 1 for success, 0 otherwise.

CONFORMING TO

RFC 1319, RFC 1320, RFC 1321

SEE ALSO

sha(3), ripemd(3), EVP_DigestInit(3)

HISTORY

MD2(), MD2_Init(), MD2_Update() MD2_Final(), MD5(), MD5_Init(), MD5_Update() and MD5_Final() are available in all versions of SSLeay and OpenSSL.

MD4(), MD4_Init(), and MD4_Update() are available in OpenSSL 0.9.6 and above.

Name

MDC2, MDC2_Init, MDC2_Update and MDC2_Final — MDC2 hash function

Synopsis

```
#include  <openssl/mdc2.h>
```

```
unsigned char *MDC2(const unsigned char *d, unsigned long n,
                 unsigned char *md);
```

```
int MDC2_Init(MDC2_CTX *c);
int MDC2_Update(MDC2_CTX *c, const unsigned char *data,
                 unsigned long len);
int MDC2_Final(unsigned char *md, MDC2_CTX *c);
```

DESCRIPTION

MDC2 is a method to construct hash functions with 128 bit output from block ciphers. These functions are an implementation of MDC2 with DES.

MDC2() computes the MDC2 message digest of the **n** bytes at **d** and places it in **md** (which must have space for MDC2_DIGEST_LENGTH == 16 bytes of output). If **md** is NULL, the digest is placed in a static array.

The following functions may be used if the message is not completely stored in memory:

MDC2_Init() initializes a **MDC2_CTX** structure.

MDC2_Update() can be called repeatedly with chunks of the message to be hashed (**len** bytes at **data**).

MDC2_Final() places the message digest in **md**, which must have space for MDC2_DIGEST_LENGTH == 16 bytes of output, and erases the **MDC2_CTX**.

Applications should use the higher level functions EVP_DigestInit(3) etc. instead of calling the hash functions directly.

RETURN VALUES

MDC2() returns a pointer to the hash value.

MDC2_Init(), MDC2_Update() and MDC2_Final() return 1 for success, 0 otherwise.

CONFORMING TO

ISO/IEC 10118-2, with DES

SEE ALSO

sha(3), EVP_DigestInit(3)

HISTORY

MDC2(), MDC2_Init(), MDC2_Update() and MDC2_Final() are available since SSLeay 0.8.

Name

OBJ_nid2obj, OBJ_nid2ln, OBJ_nid2sn, OBJ_obj2nid, OBJ_txt2nid, OBJ_ln2nid, OBJ_sn2nid, OBJ_cmp, OBJ_dup, OBJ_txt2obj, OBJ_obj2txt, OBJ_create and OBJ_cleanup — ASN1 object utility functions

Synopsis

```
#include  <openssl/objects.h>
```

```
ASN1_OBJECT * OBJ_nid2obj(int n);
const char *  OBJ_nid2ln(int n);
const char *  OBJ_nid2sn(int n);
```

```
int OBJ_obj2nid(const ASN1_OBJECT *o);
int OBJ_ln2nid(const char *ln);
int OBJ_sn2nid(const char *sn);
```

```
int OBJ_txt2nid(const char *s);
```

```
ASN1_OBJECT * OBJ_txt2obj(const char *s, int no_name);
int OBJ_obj2txt(char *buf, int buf_len, const ASN1_OBJECT *a, int no_name);
```

```
int OBJ_cmp(const ASN1_OBJECT *a,const ASN1_OBJECT *b);
ASN1_OBJECT * OBJ_dup(const ASN1_OBJECT *o);
```

```
int OBJ_create(const char *oid,const char *sn,const char *ln);
void OBJ_cleanup(void);
```

DESCRIPTION

The ASN1 object utility functions process ASN1_OBJECT structures which are a representation of the ASN1 OBJECT IDENTIFIER (OID) type.

OBJ_nid2obj(), OBJ_nid2ln() and OBJ_nid2sn() convert the NID **n** to an ASN1_OBJECT structure, its long name and its short name respectively, or **NULL** is an error occurred.

OBJ_obj2nid(), OBJ_ln2nid(), OBJ_sn2nid() return the corresponding NID for the object **o**, the long name <ln> or the short name <sn> respectively or NID_undef if an error occurred.

OBJ_txt2nid() returns NID corresponding to text string <s>. **s** can be a long name, a short name or the numerical respresentation of an object.

OBJ_txt2obj() converts the text string **s** into an ASN1_OBJECT structure. If **no_name** is 0 then long names and short names will be interpreted as well as numerical forms. If **no_name** is 1 only the numerical form is acceptable.

OBJ_obj2txt() converts the **ASN1_OBJECT a** into a textual representation. The representation is written as a null terminated string to **buf** at most **buf_len** bytes are written, truncating the result if necessary. The total amount of space required is returned. If **no_name** is 0 then if the object has a long or short name then that will be used, otherwise the numerical form will be used. If **no_name** is 1 then the numerical form will always be used.

OBJ_cmp() compares **a** to **b**. If the two are identical 0 is returned.

OBJ_dup() returns a copy of **o**.

OBJ_create() adds a new object to the internal table. **oid** is the numerical form of the object, **sn** the short name and **ln** the long name. A new NID is returned for the created object.

OBJ_cleanup() cleans up OpenSSLs internal object table: this should be called before an application exits if any new objects were added using OBJ_create().

NOTES

Objects in OpenSSL can have a short name, a long name and a numerical identifier (NID) associated with them. A standard set of objects is represented in an internal table. The appropriate values are defined in the header file **objects.h**.

For example the OID for commonName has the following definitions:

```
#define SN_commonName                   "CN"
#define LN_commonName                   "commonName"
#define NID_commonName                  13
```

New objects can be added by calling OBJ_create().

Table objects have certain advantages over other objects: for example their NIDs can be used in a C language switch statement. They are also static constant structures which are shared: that is there is only a single constant structure for each table object.

Objects which are not in the table have the NID value NID_undef.

Objects do not need to be in the internal tables to be processed, the functions OBJ_txt2obj() and OBJ_obj2txt() can process the numerical form of an OID.

EXAMPLES

Create an object for **commonName**:

```
ASN1_OBJECT *o;
o = OBJ_nid2obj(NID_commonName);
```

Check if an object is **commonName**

```
if (OBJ_obj2nid(obj) == NID_commonName)
       /* Do something */
```

Create a new NID and initialize an object from it:

```
int new_nid;
ASN1_OBJECT *obj;
new_nid = OBJ_create("1.2.3.4", "NewOID", "New Object Identifier");

obj = OBJ_nid2obj(new_nid);
```

Create a new object directly:

```
obj = OBJ_txt2obj("1.2.3.4", 1);
```

BUGS

OBJ_obj2txt() is awkward and messy to use: it doesn't follow the convention of other OpenSSL functions where the buffer can be set to **NULL** to determine the amount of data that should be written. Instead **buf** must point to a valid buffer and **buf_len** should be set to a positive value. A buffer length of 80 should be more than enough to handle any OID encountered in practice.

RETURN VALUES

OBJ_nid2obj() returns an **ASN1_OBJECT** structure or **NULL** is an error occurred.

OBJ_nid2ln() and OBJ_nid2sn() returns a valid string or **NULL** on error.

OBJ_obj2nid(), OBJ_ln2nid(), OBJ_sn2nid() and OBJ_txt2nid() return a NID or **NID_undef** on error.

SEE ALSO

ERR_get_error(3)

HISTORY

TBA

Name

OpenSSL_add_all_algorithms, OpenSSL_add_all_ciphers and OpenSSL_add_all_digests — add algorithms to internal table

Synopsis

```
#include  <openssl/evp.h>
```

```
void OpenSSL_add_all_algorithms(void);
void OpenSSL_add_all_ciphers(void);
void OpenSSL_add_all_digests(void);
```

```
void EVP_cleanup(void);
```

DESCRIPTION

OpenSSL keeps an internal table of digest algorithms and ciphers. It uses this table to lookup ciphers via functions such as EVP_get_cipher_byname().

OpenSSL_add_all_digests() adds all digest algorithms to the table.

OpenSSL_add_all_algorithms() adds all algorithms to the table (digests and ciphers).

OpenSSL_add_all_ciphers() adds all encryption algorithms to the table including password based encryption algorithms.

EVP_cleanup() removes all ciphers and digests from the table.

RETURN VALUES

None of the functions return a value.

NOTES

A typical application will call OpenSSL_add_all_algorithms() initially and EVP_cleanup() before exiting.

An application does not need to add algorithms to use them explicitly, for example by EVP_sha1(). It just needs to add them if it (or any of the functions it calls) needs to lookup algorithms.

The cipher and digest lookup functions are used in many parts of the library. If the table is not initialized several functions will misbehave and complain they cannot find algorithms. This includes the PEM, PKCS#12, SSL and S/MIME libraries. This is a common query in the OpenSSL mailing lists.

Calling OpenSSL_add_all_algorithms() links in all algorithms: as a result a statically linked executable can be quite large. If this is important it is possible to just add the required ciphers and digests.

BUGS

Although the functions do not return error codes it is possible for them to fail. This will only happen as a result of a memory allocation failure so this is not too much of a problem in practice.

SEE ALSO

evp(3), EVP_DigestInit(3), EVP_EncryptInit(3)

Name

OPENSSL_Applink — glue between OpenSSL BIO and Win32 compiler run-time

Synopsis

```
__declspec(dllexport) void **OPENSSL_Applink();
```

DESCRIPTION

OPENSSL_Applink is application-side interface which provides a glue between OpenSSL BIO layer and Win32 compiler run-time environment. Even though it appears at application side, it's essentially OpenSSL private interface. For this reason application developers are not expected to implement it, but to compile provided module with compiler of their choice and link it into the target application. The referred module is available as <openssl>/ms/applink.c.

Name

OPENSSL_config and OPENSSL_no_config — simple OpenSSL configuration functions

Synopsis

```
#include  <openssl/conf.h>
```

```
void OPENSSL_config(const char *config_name);
void OPENSSL_no_config(void);
```

DESCRIPTION

OPENSSL_config() configures OpenSSL using the standard **openssl.cnf** configuration file name using **config_name**. If **config_name** is NULL then the file specified in the environment variable **OPENSSL_CONF** will be used, and if that is not set then a system default location is used. Errors are silently ignored. Multiple calls have no effect.

OPENSSL_no_config() disables configuration. If called before OPENSSL_config() no configuration takes place.

NOTES

The OPENSSL_config() function is designed to be a very simple "call it and forget it" function. It is however **much** better than nothing. Applications which need finer control over their configuration functionality should use the configuration functions such as CONF_modules_load() directly. This function is deprecated and its use should be avoided. Applications should instead call CONF_modules_load() during initialization (that is before starting any threads).

There are several reasons why calling the OpenSSL configuration routines is advisable. For example new ENGINE functionality was added to OpenSSL 0.9.7. In OpenSSL 0.9.7 control functions can be supported by ENGINEs, this can be used (among other things) to load dynamic ENGINEs from shared libraries (DSOs). However very few applications currently support the control interface and so very few can load and use dynamic ENGINEs. Equally in future more sophisticated ENGINEs will require certain control operations to customize them. If an application calls OPENSSL_config() it doesn't need to know or care about ENGINE control operations because they can be performed by editing a configuration file.

Applications should free up configuration at application closedown by calling CONF_modules_free().

RESTRICTIONS

The OPENSSL_config() function is designed to be a very simple "call it and forget it" function. As a result its behaviour is somewhat limited. It ignores all errors silently and it can only load from the standard configuration file location for example.

It is however **much** better than nothing. Applications which need finer control over their configuration functionality should use the configuration functions such as CONF_load_modules() directly.

RETURN VALUES

Neither OPENSSL_config() nor OPENSSL_no_config() return a value.

SEE ALSO

conf(5), CONF_load_modules_file(3), CONF_modules_free(3)

HISTORY

OPENSSL_config() and OPENSSL_no_config() first appeared in OpenSSL 0.9.7

Name

OPENSSL_ia32cap — finding the IA-32 processor capabilities

Synopsis

```
unsigned long *OPENSSL_ia32cap_loc(void);
#define OPENSSL_ia32cap (*(OPENSSL_ia32cap_loc()))
```

DESCRIPTION

Value returned by OPENSSL_ia32cap_loc() is address of a variable containing IA-32 processor capabilities bit vector as it appears in EDX register after executing CPUID instruction with EAX=1 input value (see Intel Application Note #241618). Naturally it's meaningful on IA-32[E] platforms only. The variable is normally set up automatically upon toolkit initialization, but can be manipulated afterwards to modify crypto library behaviour. For the moment of this writing six bits are significant, namely:

1. bit #28 denoting Hyperthreading, which is used to distiguish cores with shared cache; 2. bit #26 denoting SSE2 support; 3. bit #25 denoting SSE support; 4. bit #23 denoting MMX support; 5. bit #20, reserved by Intel, is used to choose between RC4 code pathes; 6. bit #4 denoting presence of Time-Stamp Counter.

For example, clearing bit #26 at run-time disables high-performance SSE2 code present in the crypto library. You might have to do this if target OpenSSL application is executed on SSE2 capable CPU, but under control of OS which does not support SSE2 extentions. Even though you can manipulate the value programmatically, you most likely will find it more appropriate to set up an environment variable with the same name prior starting target application, e.g. on Intel P4 processor 'env OPENSSL_ia32cap=0x12900010 apps/openssl', to achieve same effect without modifying the application source code. Alternatively you can reconfigure the toolkit with no-sse2 option and recompile.

Name

OPENSSL_load_builtin_modules — add standard configuration modules

Synopsis

```
#include  <openssl/conf.h>
```

```
void OPENSSL_load_builtin_modules(void);
void ASN1_add_oid_module(void);
ENGINE_add_conf_module();
```

DESCRIPTION

The function OPENSSL_load_builtin_modules() adds all the standard OpenSSL configuration modules to the internal list. They can then be used by the OpenSSL configuration code.

ASN1_add_oid_module() adds just the ASN1 OBJECT module.

ENGINE_add_conf_module() adds just the ENGINE configuration module.

NOTES

If the simple configuration function OPENSSL_config() is called then OPENSSL_load_builtin_modules() is called automatically.

Applications which use the configuration functions directly will need to call OPENSSL_load_builtin_modules() themselves *before* any other configuration code.

Applications should call OPENSSL_load_builtin_modules() to load all configuration modules instead of adding modules selectively: otherwise functionality may be missing from the application if an when new modules are added.

RETURN VALUE

None of the functions return a value.

SEE ALSO

conf(3), OPENSSL_config(3)

HISTORY

These functions first appeared in OpenSSL 0.9.7.

Name

OPENSSL_VERSION_NUMBER, SSLeay and SSLeay_version — get OpenSSL version number

Synopsis

```
#include  <openssl/opensslv.h>
#define OPENSSL_VERSION_NUMBER 0xnnnnnnnnnL
```

```
#include  <openssl/crypto.h>
long SSLeay(void);
const char *SSLeay_version(int t);
```

DESCRIPTION

OPENSSL_VERSION_NUMBER is a numeric release version identifier:

```
MMNNFFPPS: major minor fix patch status
```

The status nibble has one of the values 0 for development, 1 to e for betas 1 to 14, and f for release.

for example

```
0x000906000 == 0.9.6 dev
0x000906023 == 0.9.6b beta 3
0x00090605f == 0.9.6e release
```

Versions prior to 0.9.3 have identifiers < 0x0930. Versions between 0.9.3 and 0.9.5 had a version identifier with this interpretation:

```
MMNNFFRBB major minor fix final beta/patch
```

for example

```
0x000904100 == 0.9.4 release
0x000905000 == 0.9.5 dev
```

Version 0.9.5a had an interim interpretation that is like the current one, except the patch level got the highest bit set, to keep continuity. The number was therefore 0x0090581f.

For backward compatibility, SSLEAY_VERSION_NUMBER is also defined.

SSLeay() returns this number. The return value can be compared to the macro to make sure that the correct version of the library has been loaded, especially when using DLLs on Windows systems.

SSLeay_version() returns different strings depending on **t**:

SSLEAY_VERSION

The text variant of the version number and the release date. For example, "OpenSSL 0.9.5a 1 Apr 2000".

SSLEAY_CFLAGS

The compiler flags set for the compilation process in the form "compiler: …" if available or "compiler: information not available" otherwise.

SSLEAY_BUILT_ON

The date of the build process in the form "built on: …" if available or "built on: date not available" otherwise.

SSLEAY_PLATFORM

The "Configure" target of the library build in the form "platform: …" if available or "platform: information not available" otherwise.

SSLEAY_DIR

> The "OPENSSLDIR" setting of the library build in the form "OPENSSLDIR: "..."" if available or "OPENSSLDIR: N/A" otherwise.

For an unknown **t**, the text "not available" is returned.

RETURN VALUE

The version number.

SEE ALSO

crypto(3)

HISTORY

SSLeay() and SSLEAY_VERSION_NUMBER are available in all versions of SSLeay and OpenSSL. OPENSSL_VERSION_NUMBER is available in all versions of OpenSSL. **SSLEAY_DIR** was added in OpenSSL 0.9.7.

Name

PEM, PEM_read_bio_PrivateKey, PEM_read_PrivateKey, PEM_write_bio_PrivateKey, PEM_write_PrivateKey, PEM_write_bio_PKCS8PrivateKey, PEM_write_PKCS8PrivateKey, PEM_write_bio_PKCS8PrivateKey_nid, PEM_write_PKCS8PrivateKey_nid, PEM_read_bio_PUBKEY, PEM_read_PUBKEY, PEM_write_bio_PUBKEY, PEM_write_PUBKEY, PEM_read_bio_RSAPrivateKey, PEM_read_RSAPrivateKey, PEM_write_bio_RSAPrivateKey, PEM_write_RSAPrivateKey, PEM_read_bio_RSAPublicKey, PEM_read_RSAPublicKey, PEM_write_bio_RSAPublicKey, PEM_write_RSAPublicKey, PEM_read_bio_RSA_PUBKEY, PEM_read_RSA_PUBKEY, PEM_write_bio_RSA_PUBKEY, PEM_write_RSA_PUBKEY, PEM_read_bio_DSAPrivateKey, PEM_read_DSAPrivateKey, PEM_write_bio_DSAPrivateKey, PEM_write_DSAPrivateKey, PEM_read_bio_DSA_PUBKEY, PEM_read_DSA_PUBKEY, PEM_write_bio_DSA_PUBKEY, PEM_write_DSA_PUBKEY, PEM_read_bio_DSAparams, PEM_read_DSAparams, PEM_write_bio_DSAparams, PEM_write_DSAparams, PEM_read_bio_DHparams, PEM_read_DHparams, PEM_write_bio_DHparams, PEM_write_DHparams, PEM_read_bio_X509, PEM_read_X509, PEM_write_bio_X509, PEM_write_X509, PEM_read_bio_X509_AUX, PEM_read_X509_AUX, PEM_write_bio_X509_AUX, PEM_write_X509_AUX, PEM_read_bio_X509_REQ, PEM_read_X509_REQ, PEM_write_bio_X509_REQ, PEM_write_X509_REQ, PEM_write_bio_X509_REQ_NEW, PEM_write_X509_REQ_NEW, PEM_read_bio_X509_CRL, PEM_read_X509_CRL, PEM_write_bio_X509_CRL, PEM_write_X509_CRL, PEM_read_bio_PKCS7, PEM_read_PKCS7, PEM_write_bio_PKCS7, PEM_write_PKCS7, PEM_read_bio_NETSCAPE_CERT_SEQUENCE, PEM_read_NETSCAPE_CERT_SEQUENCE, PEM_write_bio_NETSCAPE_CERT_SEQUENCE and PEM_write_NETSCAPE_CERT_SEQUENCE — PEM routines

Synopsis

```
#include  <openssl/pem.h>

EVP_PKEY *PEM_read_bio_PrivateKey(BIO *bp, EVP_PKEY **x,
                                       pem_password_cb *cb, void *u);

EVP_PKEY *PEM_read_PrivateKey(FILE *fp, EVP_PKEY **x,
                                       pem_password_cb *cb, void *u);

int PEM_write_bio_PrivateKey(BIO *bp, EVP_PKEY *x, const EVP_CIPHER *enc,
                                       unsigned char *kstr, int klen,
                                       pem_password_cb *cb, void *u);

int PEM_write_PrivateKey(FILE *fp, EVP_PKEY *x, const EVP_CIPHER *enc,
                                       unsigned char *kstr, int klen,
                                       pem_password_cb *cb, void *u);

int PEM_write_bio_PKCS8PrivateKey(BIO *bp, EVP_PKEY *x, const EVP_CIPHER *enc,
                                       char *kstr, int klen,
                                       pem_password_cb *cb, void *u);

int PEM_write_PKCS8PrivateKey(FILE *fp, EVP_PKEY *x, const EVP_CIPHER *enc,
                                       char *kstr, int klen,
                                       pem_password_cb *cb, void *u);

int PEM_write_bio_PKCS8PrivateKey_nid(BIO *bp, EVP_PKEY *x, int nid,
                                       char *kstr, int klen,
                                       pem_password_cb *cb, void *u);

int PEM_write_PKCS8PrivateKey_nid(FILE *fp, EVP_PKEY *x, int nid,
                                       char *kstr, int klen,
                                       pem_password_cb *cb, void *u);

EVP_PKEY *PEM_read_bio_PUBKEY(BIO *bp, EVP_PKEY **x,
                                       pem_password_cb *cb, void *u);

EVP_PKEY *PEM_read_PUBKEY(FILE *fp, EVP_PKEY **x,
                                       pem_password_cb *cb, void *u);

int PEM_write_bio_PUBKEY(BIO *bp, EVP_PKEY *x);
int PEM_write_PUBKEY(FILE *fp, EVP_PKEY *x);

RSA *PEM_read_bio_RSAPrivateKey(BIO *bp, RSA **x,
                                       pem_password_cb *cb, void *u);

RSA *PEM_read_RSAPrivateKey(FILE *fp, RSA **x,
                                       pem_password_cb *cb, void *u);

int PEM_write_bio_RSAPrivateKey(BIO *bp, RSA *x, const EVP_CIPHER *enc,
                                       unsigned char *kstr, int klen,
                                       pem_password_cb *cb, void *u);

int PEM_write_RSAPrivateKey(FILE *fp, RSA *x, const EVP_CIPHER *enc,
                                       unsigned char *kstr, int klen,
                                       pem_password_cb *cb, void *u);

RSA *PEM_read_bio_RSAPublicKey(BIO *bp, RSA **x,
                                       pem_password_cb *cb, void *u);

RSA *PEM_read_RSAPublicKey(FILE *fp, RSA **x,
                                       pem_password_cb *cb, void *u);

int PEM_write_bio_RSAPublicKey(BIO *bp, RSA *x);

int PEM_write_RSAPublicKey(FILE *fp, RSA *x);

RSA *PEM_read_bio_RSA_PUBKEY(BIO *bp, RSA **x,
                                       pem_password_cb *cb, void *u);

RSA *PEM_read_RSA_PUBKEY(FILE *fp, RSA **x,
                                       pem_password_cb *cb, void *u);

int PEM_write_bio_RSA_PUBKEY(BIO *bp, RSA *x);

int PEM_write_RSA_PUBKEY(FILE *fp, RSA *x);

DSA *PEM_read_bio_DSAPrivateKey(BIO *bp, DSA **x,
                                       pem_password_cb *cb, void *u);

DSA *PEM_read_DSAPrivateKey(FILE *fp, DSA **x,
                                       pem_password_cb *cb, void *u);

int PEM_write_bio_DSAPrivateKey(BIO *bp, DSA *x, const EVP_CIPHER *enc,
                                       unsigned char *kstr, int klen,
                                       pem_password_cb *cb, void *u);

int PEM_write_DSAPrivateKey(FILE *fp, DSA *x, const EVP_CIPHER *enc,
                                       unsigned char *kstr, int klen,
                                       pem_password_cb *cb, void *u);

DSA *PEM_read_bio_DSA_PUBKEY(BIO *bp, DSA **x,
                                       pem_password_cb *cb, void *u);

DSA *PEM_read_DSA_PUBKEY(FILE *fp, DSA **x,
                                       pem_password_cb *cb, void *u);

int PEM_write_bio_DSA_PUBKEY(BIO *bp, DSA *x);

int PEM_write_DSA_PUBKEY(FILE *fp, DSA *x);

DSA *PEM_read_bio_DSAparams(BIO *bp, DSA **x, pem_password_cb *cb, void *u);

DSA *PEM_read_DSAparams(FILE *fp, DSA **x, pem_password_cb *cb, void *u);

int PEM_write_bio_DSAparams(BIO *bp, DSA *x);

int PEM_write_DSAparams(FILE *fp, DSA *x);

DH *PEM_read_bio_DHparams(BIO *bp, DH **x, pem_password_cb *cb, void *u);

DH *PEM_read_DHparams(FILE *fp, DH **x, pem_password_cb *cb, void *u);

int PEM_write_bio_DHparams(BIO *bp, DH *x);

int PEM_write_DHparams(FILE *fp, DH *x);
```

```
X509 *PEM_read_bio_X509(BIO *bp, X509 **x, pem_password_cb *cb, void *u);
```

```
X509 *PEM_read_X509(FILE *fp, X509 **x, pem_password_cb *cb, void *u);
```

```
int PEM_write_bio_X509(BIO *bp, X509 *x);
```

```
int PEM_write_X509(FILE *fp, X509 *x);
```

```
X509 *PEM_read_bio_X509_AUX(BIO *bp, X509 **x, pem_password_cb *cb, void *u);
```

```
X509 *PEM_read_X509_AUX(FILE *fp, X509 **x, pem_password_cb *cb, void *u);
```

```
int PEM_write_bio_X509_AUX(BIO *bp, X509 *x);
```

```
int PEM_write_X509_AUX(FILE *fp, X509 *x);
```

```
X509_REQ *PEM_read_bio_X509_REQ(BIO *bp, X509_REQ **x,
                                       pem_password_cb *cb, void *u);
```

```
X509_REQ *PEM_read_X509_REQ(FILE *fp, X509_REQ **x,
                                       pem_password_cb *cb, void *u);
```

```
int PEM_write_bio_X509_REQ(BIO *bp, X509_REQ *x);
```

```
int PEM_write_X509_REQ(FILE *fp, X509_REQ *x);
```

```
int PEM_write_bio_X509_REQ_NEW(BIO *bp, X509_REQ *x);
```

```
int PEM_write_X509_REQ_NEW(FILE *fp, X509_REQ *x);
```

```
X509_CRL *PEM_read_bio_X509_CRL(BIO *bp, X509_CRL **x,
                                       pem_password_cb *cb, void *u);
X509_CRL *PEM_read_X509_CRL(FILE *fp, X509_CRL **x,
                                       pem_password_cb *cb, void *u);
int PEM_write_bio_X509_CRL(BIO *bp, X509_CRL *x);
int PEM_write_X509_CRL(FILE *fp, X509_CRL *x);
```

```
PKCS7 *PEM_read_bio_PKCS7(BIO *bp, PKCS7 **x, pem_password_cb *cb, void *u);
```

```
PKCS7 *PEM_read_PKCS7(FILE *fp, PKCS7 **x, pem_password_cb *cb, void *u);
```

```
int PEM_write_bio_PKCS7(BIO *bp, PKCS7 *x);
```

```
int PEM_write_PKCS7(FILE *fp, PKCS7 *x);
```

```
NETSCAPE_CERT_SEQUENCE *PEM_read_bio_NETSCAPE_CERT_SEQUENCE(BIO *bp,
                                               NETSCAPE_CERT_SEQUENCE **x,
                                               pem_password_cb *cb, void *u);
```

```
NETSCAPE_CERT_SEQUENCE *PEM_read_NETSCAPE_CERT_SEQUENCE(FILE *fp,
                                               NETSCAPE_CERT_SEQUENCE **x,
                                               pem_password_cb *cb, void *u);
```

```
int PEM_write_bio_NETSCAPE_CERT_SEQUENCE(BIO *bp, NETSCAPE_CERT_SEQUENCE *x);
```

```
int PEM_write_NETSCAPE_CERT_SEQUENCE(FILE *fp, NETSCAPE_CERT_SEQUENCE *x);
```

DESCRIPTION

The PEM functions read or write structures in PEM format. In this sense PEM format is simply base64 encoded data surrounded by header lines.

For more details about the meaning of arguments see the **PEM FUNCTION ARGUMENTS** section.

Each operation has four functions associated with it. For clarity the term "**foobar** functions" will be used to collectively refer to the PEM_read_bio_foobar(), PEM_read_foobar(), PEM_write_bio_foobar() and PEM_write_foobar() functions.

The **PrivateKey** functions read or write a private key in PEM format using an EVP_PKEY structure. The write routines use "traditional" private key format and can handle both RSA and DSA private keys. The read functions can additionally transparently handle PKCS#8 format encrypted and unencrypted keys too.

PEM_write_bio_PKCS8PrivateKey() and PEM_write_PKCS8PrivateKey() write a private key in an EVP_PKEY structure in PKCS#8 EncryptedPrivateKeyInfo format using PKCS#5 v2.0 password based encryption algorithms. The **cipher** argument specifies the encryption algorithm to use: unlike all other PEM routines the encryption is applied at the PKCS#8 level and not in the PEM headers. If **cipher** is NULL then no encryption is used and a PKCS#8 PrivateKeyInfo structure is used instead.

PEM_write_bio_PKCS8PrivateKey_nid() and PEM_write_PKCS8PrivateKey_nid() also write out a private key as a PKCS#8 EncryptedPrivateKeyInfo however it uses PKCS#5 v1.5 or PKCS#12 encryption algorithms instead. The algorithm to use is specified in the **nid** parameter and should be the NID of the corresponding OBJECT IDENTIFIER (see NOTES section).

The **PUBKEY** functions process a public key using an EVP_PKEY structure. The public key is encoded as a SubjectPublicKeyInfo structure.

The **RSAPrivateKey** functions process an RSA private key using an RSA structure. It handles the same formats as the **PrivateKey** functions but an error occurs if the private key is not RSA.

The **RSAPublicKey** functions process an RSA public key using an RSA structure. The public key is encoded using a PKCS#1 RSAPublicKey structure.

The **RSA_PUBKEY** functions also process an RSA public key using an RSA structure. However the public key is encoded using a SubjectPublicKeyInfo structure and an error occurs if the public key is not RSA.

The **DSAPrivateKey** functions process a DSA private key using a DSA structure. It handles the same formats as the **PrivateKey** functions but an error occurs if the private key is not DSA.

The **DSA_PUBKEY** functions process a DSA public key using a DSA structure. The public key is encoded using a SubjectPublicKeyInfo structure and an error occurs if the public key is not DSA.

The **DSAparams** functions process DSA parameters using a DSA structure. The parameters are encoded using a Dss-Parms structure as defined in RFC2459.

The **DHparams** functions process DH parameters using a DH structure. The parameters are encoded using a PKCS#3 DHparameter structure.

The **X509** functions process an X509 certificate using an X509 structure. They will also process a trusted X509 certificate but any trust settings are discarded.

The **X509_AUX** functions process a trusted X509 certificate using an X509 structure.

The **X509_REQ** and **X509_REQ_NEW** functions process a PKCS#10 certificate request using an X509_REQ structure. The **X509_REQ** write functions use **CERTIFICATE REQUEST** in the header whereas the **X509_REQ_NEW** functions use **NEW CERTIFICATE REQUEST** (as required by some CAs). The **X509_REQ** read functions will handle either form so there are no **X509_REQ_NEW** read functions.

The **X509_CRL** functions process an X509 CRL using an X509_CRL structure.

The **PKCS7** functions process a PKCS#7 ContentInfo using a PKCS7 structure.

The **NETSCAPE_CERT_SEQUENCE** functions process a Netscape Certificate Sequence using a NETSCAPE_CERT_SEQUENCE structure.

PEM FUNCTION ARGUMENTS

The PEM functions have many common arguments.

The **bp** BIO parameter (if present) specifies the BIO to read from or write to.

The **fp** FILE parameter (if present) specifies the FILE pointer to read from or write to.

The PEM read functions all take an argument **TYPE **x** and return a **TYPE *** pointer. Where **TYPE** is whatever structure the function uses. If **x** is NULL then the parameter is ignored. If **x** is not NULL but ***x** is NULL then the structure returned will be written to ***x**. If neither **x** nor ***x** is NULL then an attempt is made to reuse the structure at ***x** (but see BUGS and EXAMPLES sections). Irrespective of the value of **x** a pointer to the structure is always returned (or NULL if an error occurred).

The PEM functions which write private keys take an **enc** parameter which specifies the encryption algorithm to use, encryption is done at the PEM level. If this parameter is set to NULL then the private key is written in unencrypted form.

The **cb** argument is the callback to use when querying for the pass phrase used for encrypted PEM structures (normally only private keys).

For the PEM write routines if the **kstr** parameter is not NULL then **klen** bytes at **kstr** are used as the passphrase and **cb** is ignored.

If the **cb** parameters is set to NULL and the **u** parameter is not NULL then the **u** parameter is interpreted as a null terminated string to use as the passphrase. If both **cb** and **u** are NULL then the default callback routine is used which will typically prompt for the passphrase on the current terminal with echoing turned off.

The default passphrase callback is sometimes inappropriate (for example in a GUI application) so an alternative can be supplied. The callback routine has the following form:

```
int cb(char *buf, int size, int rwflag, void *u);
```

buf is the buffer to write the passphrase to. **size** is the maximum length of the passphrase (i.e. the size of buf). **rwflag** is a flag which is set to 0 when reading and 1 when writing. A typical routine will ask the user to verify the passphrase (for example by prompting for it twice) if **rwflag** is 1. The **u** parameter has the same value as the **u** parameter passed to the PEM routine. It allows arbitrary data to be passed to the callback by the application (for example a window handle in a GUI application). The callback **must** return the number of characters in the passphrase or 0 if an error occurred.

EXAMPLES

Although the PEM routines take several arguments in almost all applications most of them are set to 0 or NULL.

Read a certificate in PEM format from a BIO:

```
X509 *x;
x = PEM_read_bio_X509(bp, NULL, 0, NULL);
if (x == NULL)
       {
       /* Error */
       }
```

Alternative method:

```
X509 *x = NULL;
if (!PEM_read_bio_X509(bp, &x, 0, NULL))
       {
       /* Error */
       }
```

Write a certificate to a BIO:

```
if (!PEM_write_bio_X509(bp, x))
       {
       /* Error */
       }
```

Write an unencrypted private key to a FILE pointer:

```
if (!PEM_write_PrivateKey(fp, key, NULL, NULL, 0, 0, NULL))
       {
       /* Error */
       }
```

Write a private key (using traditional format) to a BIO using triple DES encryption, the pass phrase is prompted for:

```
if (!PEM_write_bio_PrivateKey(bp, key, EVP_des_ede3_cbc(), NULL, 0, 0, NULL))
       {
       /* Error */
       }
```

Write a private key (using PKCS#8 format) to a BIO using triple DES encryption, using the pass phrase "hello":

```
if (!PEM_write_bio_PKCS8PrivateKey(bp, key, EVP_des_ede3_cbc(), NULL, 0, 0, "hello"))
       {
       /* Error */
       }
```

Read a private key from a BIO using the pass phrase "hello":

```
key = PEM_read_bio_PrivateKey(bp, NULL, 0, "hello");
if (key == NULL)
       {
       /* Error */
       }
```

Read a private key from a BIO using a pass phrase callback:

```
key = PEM_read_bio_PrivateKey(bp, NULL, pass_cb, "My Private Key");
if (key == NULL)
       {
       /* Error */
       }
```

Skeleton pass phrase callback:

```
int pass_cb(char *buf, int size, int rwflag, void *u);
       {
       int len;
       char *tmp;
       /* We'd probably do something else if 'rwflag' is 1 */
       printf("Enter pass phrase for \"%s\"\n", u);

/* get pass phrase, length 'len' into 'tmp' */
tmp = "hello";
len = strlen(tmp);

if (len <= 0) return 0;
/* if too long, truncate */
if (len > size) len = size;
memcpy(buf, tmp, len);
return len;
}
```

NOTES

The old **PrivateKey** write routines are retained for compatibility. New applications should write private keys using the PEM_write_bio_PKCS8PrivateKey() or PEM_write_PKCS8PrivateKey() routines because they are more secure (they use an iteration count of 2048 whereas the traditional routines use a count of 1) unless compatibility with older versions of OpenSSL is important.

The **PrivateKey** read routines can be used in all applications because they handle all formats transparently.

A frequent cause of problems is attempting to use the PEM routines like this:

```
X509 *x;
PEM_read_bio_X509(bp, &x, 0, NULL);
```

this is a bug because an attempt will be made to reuse the data at **x** which is an uninitialised pointer.

PEM ENCRYPTION FORMAT

This old **PrivateKey** routines use a non standard technique for encryption.

The private key (or other data) takes the following form:

```
-----BEGIN RSA PRIVATE KEY-----
Proc-Type: 4,ENCRYPTED
DEK-Info: DES-EDE3-CBC,3F17F5316E2BAC89

…base64 encoded data…
-----END RSA PRIVATE KEY-----
```

The line beginning DEK-Info contains two comma separated pieces of information: the encryption algorithm name as used by EVP_get_cipherbyname() and an 8 byte **salt** encoded as a set of hexadecimal digits.

After this is the base64 encoded encrypted data.

The encryption key is determined using EVP_BytesToKey(), using **salt** and an iteration count of 1. The IV used is the value of **salt** and *not* the IV returned by EVP_BytesToKey().

BUGS

The PEM read routines in some versions of OpenSSL will not correctly reuse an existing structure. Therefore the following:

```
PEM_read_bio_X509(bp, &x, 0, NULL);
```

where **x** already contains a valid certificate, may not work, whereas:

```
X509_free(x);
x = PEM_read_bio_X509(bp, NULL, 0, NULL);
```

is guaranteed to work.

RETURN CODES

The read routines return either a pointer to the structure read or NULL if an error occurred.

The write routines return 1 for success or 0 for failure.

SEE ALSO

EVP_get_cipherbyname(3), EVP_BytesToKey(3)

Name

PEM_write_bio_CMS_stream — output CMS_ContentInfo structure in PEM format.

Synopsis

```
#include  <openssl/cms.h>
#include  <openssl/pem.h>
```

```
int PEM_write_bio_CMS_stream(BIO *out, CMS_ContentInfo *cms, BIO *data, int flags);
```

DESCRIPTION

PEM_write_bio_CMS_stream() outputs a CMS_ContentInfo structure in PEM format.

It is otherwise identical to the function SMIME_write_CMS().

NOTES

This function is effectively a version of the PEM_write_bio_CMS() supporting streaming.

RETURN VALUES

PEM_write_bio_CMS_stream() returns 1 for success or 0 for failure.

SEE ALSO

ERR_get_error(3), CMS_sign(3), CMS_verify(3), CMS_encrypt(3), CMS_decrypt(3), SMIME_write_CMS(3), i2d_CMS_bio_stream(3)

HISTORY

PEM_write_bio_CMS_stream() was added to OpenSSL 1.0.0

Name

PEM_write_bio_PKCS7_stream — output PKCS7 structure in PEM format.

Synopsis

```
#include  <openssl/pkcs7.h>
#include  <openssl/pem.h>
```

```
int PEM_write_bio_PKCS7_stream(BIO *out, PKCS7 *p7, BIO *data, int flags);
```

DESCRIPTION

PEM_write_bio_PKCS7_stream() outputs a PKCS7 structure in PEM format.

It is otherwise identical to the function SMIME_write_PKCS7().

NOTES

This function is effectively a version of the PEM_write_bio_PKCS7() supporting streaming.

RETURN VALUES

PEM_write_bio_PKCS7_stream() returns 1 for success or 0 for failure.

SEE ALSO

ERR_get_error(3), PKCS7_sign(3), PKCS7_verify(3), PKCS7_encrypt(3), PKCS7_decrypt(3), SMIME_write_PKCS7(3), i2d_PKCS7_bio_stream(3)

HISTORY

PEM_write_bio_PKCS7_stream() was added to OpenSSL 1.0.0

Name

PKCS12_create — create a PKCS#12 structure

Synopsis

```
#include  <openssl/pkcs12.h>
```

```
PKCS12 *PKCS12_create(char *pass, char *name, EVP_PKEY *pkey, X509 *cert, STACK_OF(X509) *ca,
                               int nid_key, int nid_cert, int iter, int mac_iter, int keytype);
```

DESCRIPTION

PKCS12_create() creates a PKCS#12 structure.

pass is the passphrase to use. **name** is the **friendlyName** to use for the supplied certifictate and key. **pkey** is the private key to include in the structure and **cert** its corresponding certificates. **ca**, if not **NULL** is an optional set of certificates to also include in the structure.

nid_key and **nid_cert** are the encryption algorithms that should be used for the key and certificate respectively. **iter** is the encryption algorithm iteration count to use and **mac_iter** is the MAC iteration count to use. **keytype** is the type of key.

NOTES

The parameters **nid_key**, **nid_cert**, **iter**, **mac_iter** and **keytype** can all be set to zero and sensible defaults will be used.

These defaults are: 40 bit RC2 encryption for certificates, triple DES encryption for private keys, a key iteration count of PKCS12_DEFAULT_ITER (currently 2048) and a MAC iteration count of 1.

The default MAC iteration count is 1 in order to retain compatibility with old software which did not interpret MAC iteration counts. If such compatibility is not required then **mac_iter** should be set to PKCS12_DEFAULT_ITER.

keytype adds a flag to the store private key. This is a non standard extension that is only currently interpreted by MSIE. If set to zero the flag is omitted, if set to **KEY_SIG** the key can be used for signing only, if set to **KEY_EX** it can be used for signing and encryption. This option was useful for old export grade software which could use signing only keys of arbitrary size but had restrictions on the permissible sizes of keys which could be used for encryption.

NEW FUNCTIONALITY IN OPENSSL 0.9.8

Some additional functionality was added to PKCS12_create() in OpenSSL 0.9.8. These extensions are detailed below.

If a certificate contains an **alias** or **keyid** then this will be used for the corresponding **friendlyName** or **localKeyID** in the PKCS12 structure.

Either **pkey**, **cert** or both can be **NULL** to indicate that no key or certficate is required. In previous versions both had to be present or a fatal error is returned.

nid_key or **nid_cert** can be set to -1 indicating that no encryption should be used.

mac_iter can be set to -1 and the MAC will then be omitted entirely.

SEE ALSO

d2i_PKCS12(3)

HISTORY

PKCS12_create was added in OpenSSL 0.9.3

Name

PKCS12_parse — parse a PKCS#12 structure

Synopsis

```
#include  <openssl/pkcs12.h>
```

```
int PKCS12_parse(PKCS12 *p12, const char *pass, EVP_PKEY **pkey, X509 **cert, STACK_OF(X509) **ca);
```

DESCRIPTION

PKCS12_parse() parses a PKCS12 structure.

p12 is the **PKCS12** structure to parse. **pass** is the passphrase to use. If successful the private key will be written to ***pkey**, the corresponding certificate to ***cert** and any additional certificates to ***ca**.

NOTES

The parameters **pkey** and **cert** cannot be **NULL**. **ca** can be <NULL> in which case additional certificates will be discarded. ***ca** can also be a valid STACK in which case additional certificates are appended to ***ca**. If ***ca** is **NULL** a new STACK will be allocated.

The **friendlyName** and **localKeyID** attributes (if present) on each certificate will be stored in the **alias** and **keyid** attributes of the **X509** structure.

RETURN VALUES

PKCS12_parse() returns 1 for success and zero if an error occurred.

The error can be obtained from ERR_get_error(3)

BUGS

Only a single private key and corresponding certificate is returned by this function. More complex PKCS#12 files with multiple private keys will only return the first match.

Only **friendlyName** and **localKeyID** attributes are currently stored in certificates. Other attributes are discarded.

Attributes currently cannot be stored in the private key **EVP_PKEY** structure.

SEE ALSO

d2i_PKCS12(3)

HISTORY

PKCS12_parse was added in OpenSSL 0.9.3

Name

PKCS7_decrypt — decrypt content from a PKCS#7 envelopedData structure

Synopsis

```
#include  <openssl/pkcs7.h>
```

```
int PKCS7_decrypt(PKCS7 *p7, EVP_PKEY *pkey, X509 *cert, BIO *data, int flags);
```

DESCRIPTION

PKCS7_decrypt() extracts and decrypts the content from a PKCS#7 envelopedData structure. **pkey** is the private key of the recipient, **cert** is the recipients certificate, **data** is a BIO to write the content to and **flags** is an optional set of flags.

NOTES

OpenSSL_add_all_algorithms() (or equivalent) should be called before using this function or errors about unknown algorithms will occur.

Although the recipients certificate is not needed to decrypt the data it is needed to locate the appropriate (of possible several) recipients in the PKCS#7 structure.

The following flags can be passed in the **flags** parameter.

If the **PKCS7_TEXT** flag is set MIME headers for type **text/plain** are deleted from the content. If the content is not of type **text/plain** then an error is returned.

RETURN VALUES

PKCS7_decrypt() returns either 1 for success or 0 for failure. The error can be obtained from ERR_get_error(3)

BUGS

PKCS7_decrypt() must be passed the correct recipient key and certificate. It would be better if it could look up the correct key and certificate from a database.

The lack of single pass processing and need to hold all data in memory as mentioned in PKCS7_sign() also applies to PKCS7_verify().

SEE ALSO

ERR_get_error(3), PKCS7_encrypt(3)

HISTORY

PKCS7_decrypt() was added to OpenSSL 0.9.5

Name

PKCS7_encrypt — create a PKCS#7 envelopedData structure

Synopsis

```
#include  <openssl/pkcs7.h>
```

```
PKCS7 *PKCS7_encrypt(STACK_OF(X509) *certs, BIO *in, const EVP_CIPHER *cipher, int flags);
```

DESCRIPTION

PKCS7_encrypt() creates and returns a PKCS#7 envelopedData structure. **certs** is a list of recipient certificates. **in** is the content to be encrypted. **cipher** is the symmetric cipher to use. **flags** is an optional set of flags.

NOTES

Only RSA keys are supported in PKCS#7 and envelopedData so the recipient certificates supplied to this function must all contain RSA public keys, though they do not have to be signed using the RSA algorithm.

EVP_des_ede3_cbc() (triple DES) is the algorithm of choice for S/MIME use because most clients will support it.

Some old "export grade" clients may only support weak encryption using 40 or 64 bit RC2. These can be used by passing EVP_rc2_40_cbc() and EVP_rc2_64_cbc() respectively.

The algorithm passed in the **cipher** parameter must support ASN1 encoding of its parameters.

Many browsers implement a "sign and encrypt" option which is simply an S/MIME envelopedData containing an S/MIME signed message. This can be readily produced by storing the S/MIME signed message in a memory BIO and passing it to PKCS7_encrypt().

The following flags can be passed in the **flags** parameter.

If the **PKCS7_TEXT** flag is set MIME headers for type **text/plain** are prepended to the data.

Normally the supplied content is translated into MIME canonical format (as required by the S/MIME specifications) if **PKCS7_BINARY** is set no translation occurs. This option should be used if the supplied data is in binary format otherwise the translation will corrupt it. If **PKCS7_BINARY** is set then **PKCS7_TEXT** is ignored.

If the **PKCS7_STREAM** flag is set a partial **PKCS7** structure is output suitable for streaming I/O: no data is read from the BIO **in**.

NOTES

If the flag **PKCS7_STREAM** is set the returned **PKCS7** structure is **not** complete and outputting its contents via a function that does not properly finalize the **PKCS7** structure will give unpredictable results.

Several functions including SMIME_write_PKCS7(), i2d_PKCS7_bio_stream(), PEM_write_bio_PKCS7_stream() finalize the structure. Alternatively finalization can be performed by obtaining the streaming ASN1 **BIO** directly using BIO_new_PKCS7().

RETURN VALUES

PKCS7_encrypt() returns either a PKCS7 structure or NULL if an error occurred. The error can be obtained from ERR_get_error(3).

SEE ALSO

ERR_get_error(3), PKCS7_decrypt(3)

HISTORY

PKCS7_decrypt() was added to OpenSSL 0.9.5 The **PKCS7_STREAM** flag was first supported in OpenSSL 1.0.0.

Name

PKCS7_sign_add_signer — add a signer PKCS7 signed data structure.

Synopsis

```
#include  <openssl/pkcs7.h>
```

```
PKCS7_SIGNER_INFO *PKCS7_sign_add_signer(PKCS7 *p7, X509 *signcert, EVP_PKEY *pkey,
    const EVP_MD *md, int flags);
```

DESCRIPTION

PKCS7_sign_add_signer() adds a signer with certificate **signcert** and private key **pkey** using message digest **md** to a PKCS7 signed data structure **p7**.

The PKCS7 structure should be obtained from an initial call to PKCS7_sign() with the flag **PKCS7_PARTIAL** set or in the case or re-signing a valid PKCS7 signed data structure.

If the **md** parameter is **NULL** then the default digest for the public key algorithm will be used.

Unless the **PKCS7_REUSE_DIGEST** flag is set the returned PKCS7 structure is not complete and must be finalized either by streaming (if applicable) or a call to PKCS7_final().

NOTES

The main purpose of this function is to provide finer control over a PKCS#7 signed data structure where the simpler PKCS7_sign() function defaults are not appropriate. For example if multiple signers or non default digest algorithms are needed.

Any of the following flags (ored together) can be passed in the **flags** parameter.

If **PKCS7_REUSE_DIGEST** is set then an attempt is made to copy the content digest value from the PKCS7 struture: to add a signer to an existing structure. An error occurs if a matching digest value cannot be found to copy. The returned PKCS7 structure will be valid and finalized when this flag is set.

If **PKCS7_PARTIAL** is set in addition to **PKCS7_REUSE_DIGEST** then the **PKCS7_SIGNER_INO** structure will not be finalized so additional attributes can be added. In this case an explicit call to PKCS7_SIGNER_INFO_sign() is needed to finalize it.

If **PKCS7_NOCERTS** is set the signer's certificate will not be included in the PKCS7 structure, the signer's certificate must still be supplied in the **signcert** parameter though. This can reduce the size of the signature if the signers certificate can be obtained by other means: for example a previously signed message.

The signedData structure includes several PKCS#7 autenticatedAttributes including the signing time, the PKCS#7 content type and the supported list of ciphers in an SMIMECapabilities attribute. If **PKCS7_NOATTR** is set then no authenticatedAttributes will be used. If **PKCS7_NOSMIMECAP** is set then just the SMIMECapabilities are omitted.

If present the SMIMECapabilities attribute indicates support for the following algorithms: triple DES, 128 bit RC2, 64 bit RC2, DES and 40 bit RC2. If any of these algorithms is disabled then it will not be included.

PKCS7_sign_add_signers() returns an internal pointer to the PKCS7_SIGNER_INFO structure just added, this can be used to set additional attributes before it is finalized.

RETURN VALUES

PKCS7_sign_add_signers() returns an internal pointer to the PKCS7_SIGNER_INFO structure just added or NULL if an error occurs.

SEE ALSO

ERR_get_error(3), PKCS7_sign(3), PKCS7_final(3),

HISTORY

PPKCS7_sign_add_signer() was added to OpenSSL 1.0.0

Name

PKCS7_sign — create a PKCS#7 signedData structure

Synopsis

```
#include  <openssl/pkcs7.h>
```

```
PKCS7 *PKCS7_sign(X509 *signcert, EVP_PKEY *pkey, STACK_OF(X509) *certs, BIO *data, int flags);
```

DESCRIPTION

PKCS7_sign() creates and returns a PKCS#7 signedData structure. **signcert** is the certificate to sign with, **pkey** is the corresponsding private key. **certs** is an optional additional set of certificates to include in the PKCS#7 structure (for example any intermediate CAs in the chain).

The data to be signed is read from BIO **data**.

flags is an optional set of flags.

NOTES

Any of the following flags (ored together) can be passed in the **flags** parameter.

Many S/MIME clients expect the signed content to include valid MIME headers. If the **PKCS7_TEXT** flag is set MIME headers for type **text/plain** are prepended to the data.

If **PKCS7_NOCERTS** is set the signer's certificate will not be included in the PKCS7 structure, the signer's certificate must still be supplied in the **signcert** parameter though. This can reduce the size of the signature if the signers certificate can be obtained by other means: for example a previously signed message.

The data being signed is included in the PKCS7 structure, unless **PKCS7_DETACHED** is set in which case it is omitted. This is used for PKCS7 detached signatures which are used in S/MIME plaintext signed messages for example.

Normally the supplied content is translated into MIME canonical format (as required by the S/MIME specifications) if **PKCS7_BINARY** is set no translation occurs. This option should be used if the supplied data is in binary format otherwise the translation will corrupt it.

The signedData structure includes several PKCS#7 autenticatedAttributes including the signing time, the PKCS#7 content type and the supported list of ciphers in an SMIMECapabilities attribute. If **PKCS7_NOATTR** is set then no authenticatedAttributes will be used. If **PKCS7_NOSMIMECAP** is set then just the SMIMECapabilities are omitted.

If present the SMIMECapabilities attribute indicates support for the following algorithms: triple DES, 128 bit RC2, 64 bit RC2, DES and 40 bit RC2. If any of these algorithms is disabled then it will not be included.

If the flags **PKCS7_STREAM** is set then the returned **PKCS7** structure is just initialized ready to perform the signing operation. The signing is however **not** performed and the data to be signed is not read from the **data** parameter. Signing is deferred until after the data has been written. In this way data can be signed in a single pass.

If the **PKCS7_PARTIAL** flag is set a partial **PKCS7** structure is output to which additional signers and capabilities can be added before finalization.

NOTES

If the flag **PKCS7_STREAM** is set the returned **PKCS7** structure is **not** complete and outputting its contents via a function that does not properly finalize the **PKCS7** structure will give unpredictable results.

Several functions including SMIME_write_PKCS7(), i2d_PKCS7_bio_stream(), PEM_write_bio_PKCS7_stream() finalize the structure. Alternatively finalization can be performed by obtaining the streaming ASN1 **BIO** directly using BIO_new_PKCS7().

If a signer is specified it will use the default digest for the signing algorithm. This is **SHA1** for both RSA and DSA keys.

In OpenSSL 1.0.0 the **certs**, **signcert** and **pkey** parameters can all be **NULL** if the **PKCS7_PARTIAL** flag is set. One or more signers can be added using the function **PKCS7_sign_add_signer()**. **PKCS7_final()** must also be called to finalize the structure if streaming is not enabled. Alternative signing digests can also be specified using this method.

In OpenSSL 1.0.0 if **signcert** and **pkey** are NULL then a certificates only PKCS#7 structure is output.

In versions of OpenSSL before 1.0.0 the **signcert** and **pkey** parameters must **NOT** be NULL.

BUGS

Some advanced attributes such as counter signatures are not supported.

RETURN VALUES

PKCS7_sign() returns either a valid PKCS7 structure or NULL if an error occurred. The error can be obtained from ERR_get_error(3).

SEE ALSO

ERR_get_error(3), PKCS7_verify(3)

HISTORY

PKCS7_sign() was added to OpenSSL 0.9.5

The **PKCS7_PARTIAL** flag was added in OpenSSL 1.0.0

The **PKCS7_STREAM** flag was added in OpenSSL 1.0.0

Name

PKCS7_verify — verify a PKCS#7 signedData structure

Synopsis

```
#include  <openssl/pkcs7.h>
```

```
int PKCS7_verify(PKCS7 *p7, STACK_OF(X509) *certs, X509_STORE *store, BIO *indata, BIO *out, int flags);
```

```
STACK_OF(X509) *PKCS7_get0_signers(PKCS7 *p7, STACK_OF(X509) *certs, int flags);
```

DESCRIPTION

PKCS7_verify() verifies a PKCS#7 signedData structure. **p7** is the PKCS7 structure to verify. **certs** is a set of certificates in which to search for the signer's certificate. **store** is a trusted certficate store (used for chain verification). **indata** is the signed data if the content is not present in **p7** (that is it is detached). The content is written to **out** if it is not NULL.

flags is an optional set of flags, which can be used to modify the verify operation.

PKCS7_get0_signers() retrieves the signer's certificates from **p7**, it does **not** check their validity or whether any signatures are valid. The **certs** and **flags** parameters have the same meanings as in PKCS7_verify().

VERIFY PROCESS

Normally the verify process proceeds as follows.

Initially some sanity checks are performed on **p7**. The type of **p7** must be signedData. There must be at least one signature on the data and if the content is detached **indata** cannot be **NULL**.

An attempt is made to locate all the signer's certificates, first looking in the **certs** parameter (if it is not **NULL**) and then looking in any certificates contained in the **p7** structure itself. If any signer's certificates cannot be located the operation fails.

Each signer's certificate is chain verified using the **smimesign** purpose and the supplied trusted certificate store. Any internal certificates in the message are used as untrusted CAs. If any chain verify fails an error code is returned.

Finally the signed content is read (and written to **out** is it is not NULL) and the signature's checked.

If all signature's verify correctly then the function is successful.

Any of the following flags (ored together) can be passed in the **flags** parameter to change the default verify behaviour. Only the flag **PKCS7_NOINTERN** is meaningful to PKCS7_get0_signers().

If **PKCS7_NOINTERN** is set the certificates in the message itself are not searched when locating the signer's certificate. This means that all the signers certificates must be in the **certs** parameter.

If the **PKCS7_TEXT** flag is set MIME headers for type **text/plain** are deleted from the content. If the content is not of type **text/plain** then an error is returned.

If **PKCS7_NOVERIFY** is set the signer's certificates are not chain verified.

If **PKCS7_NOCHAIN** is set then the certificates contained in the message are not used as untrusted CAs. This means that the whole verify chain (apart from the signer's certificate) must be contained in the trusted store.

If **PKCS7_NOSIGS** is set then the signatures on the data are not checked.

NOTES

One application of **PKCS7_NOINTERN** is to only accept messages signed by a small number of certificates. The acceptable certificates would be passed in the **certs** parameter. In this case if the signer is not one of the certificates supplied in **certs** then the verify will fail because the signer cannot be found.

Care should be taken when modifying the default verify behaviour, for example setting **PKCS7_NOVERIFY|PKCS7_NOSIGS** will totally disable all verification and any signed message will be considered valid. This combination is however useful if one merely wishes to write the content to **out** and its validity is not considered important.

Chain verification should arguably be performed using the signing time rather than the current time. However since the signing time is supplied by the signer it cannot be trusted without additional evidence (such as a trusted timestamp).

RETURN VALUES

PKCS7_verify() returns 1 for a successful verification and zero or a negative value if an error occurs.

PKCS7_get0_signers() returns all signers or **NULL** if an error occurred.

The error can be obtained from ERR_get_error(3)

BUGS

The trusted certificate store is not searched for the signers certificate, this is primarily due to the inadequacies of the current **X509_STORE** functionality.

The lack of single pass processing and need to hold all data in memory as mentioned in PKCS7_sign() also applies to PKCS7_verify().

SEE ALSO

ERR_get_error(3), PKCS7_sign(3)

HISTORY

PKCS7_verify() was added to OpenSSL 0.9.5

Name

rand — pseudo-random number generator

Synopsis

```
#include  <openssl/rand.h>

int  RAND_set_rand_engine(ENGINE *engine);

int  RAND_bytes(unsigned char *buf, int num);
int  RAND_pseudo_bytes(unsigned char *buf, int num);

void RAND_seed(const void *buf, int num);
void RAND_add(const void *buf, int num, int entropy);
int  RAND_status(void);

int  RAND_load_file(const char *file, long max_bytes);
int  RAND_write_file(const char *file);
const char *RAND_file_name(char *file, size_t num);

int  RAND_egd(const char *path);

void RAND_set_rand_method(const RAND_METHOD *meth);
const RAND_METHOD *RAND_get_rand_method(void);
RAND_METHOD *RAND_SSLeay(void);

void RAND_cleanup(void);

/* For Win32 only */
void RAND_screen(void);
int RAND_event(UINT, WPARAM, LPARAM);
```

DESCRIPTION

Since the introduction of the ENGINE API, the recommended way of controlling default implementations is by using the ENGINE API functions. The default **RAND_METHOD**, as set by RAND_set_rand_method() and returned by RAND_get_rand_method(), is only used if no ENGINE has been set as the default "rand" implementation. Hence, these two functions are no longer the recommened way to control defaults.

If an alternative **RAND_METHOD** implementation is being used (either set directly or as provided by an ENGINE module), then it is entirely responsible for the generation and management of a cryptographically secure PRNG stream. The mechanisms described below relate solely to the software PRNG implementation built in to OpenSSL and used by default.

These functions implement a cryptographically secure pseudo-random number generator (PRNG). It is used by other library functions for example to generate random keys, and applications can use it when they need randomness.

A cryptographic PRNG must be seeded with unpredictable data such as mouse movements or keys pressed at random by the user. This is described in RAND_add(3). Its state can be saved in a seed file (see RAND_load_file(3)) to avoid having to go through the seeding process whenever the application is started.

RAND_bytes(3) describes how to obtain random data from the PRNG.

INTERNALS

The RAND_SSLeay() method implements a PRNG based on a cryptographic hash function.

The following description of its design is based on the SSLeay documentation:

First up I will state the things I believe I need for a good RNG.

1. A good hashing algorithm to mix things up and to convert the RNG 'state' to random numbers.

2. An initial source of random 'state'.

3. The state should be very large. If the RNG is being used to generate 4096 bit RSA keys, 2 2048 bit random strings are required (at a minimum). If your RNG state only has 128 bits, you are obviously limiting the search space to 128 bits, not 2048. I'm probably getting a little carried away on this last point but it does indicate that it may not be a bad idea to keep quite a lot of RNG state. It should be easier to break a cipher than guess the RNG seed data.

4. Any RNG seed data should influence all subsequent random numbers generated. This implies that any random seed data entered will have an influence on all subsequent random numbers generated.

5. When using data to seed the RNG state, the data used should not be extractable from the RNG state. I believe this should be a requirement because one possible source of 'secret' semi random data would be a private key or a password. This data must not be disclosed by either subsequent random numbers or a 'core' dump left by a program crash.

6. Given the same initial 'state', 2 systems should deviate in their RNG state (and hence the random numbers generated) over time if at all possible.

7. Given the random number output stream, it should not be possible to determine the RNG state or the next random number.

The algorithm is as follows.

There is global state made up of a 1023 byte buffer (the 'state'), a working hash value ('md'), and a counter ('count').

Whenever seed data is added, it is inserted into the 'state' as follows.

The input is chopped up into units of 20 bytes (or less for the last block). Each of these blocks is run through the hash function as follows: The data passed to the hash function is the current 'md', the same number of bytes from the 'state' (the location determined by in incremented looping index) as the current 'block', the new key data 'block', and 'count' (which is incremented after each use). The result of this is kept in 'md' and also xored into the 'state' at the same locations that were used as input into the hash function. I believe this system addresses points 1 (hash function; currently SHA-1), 3 (the 'state'), 4 (via the 'md'), 5 (by the use of a hash function and xor).

When bytes are extracted from the RNG, the following process is used. For each group of 10 bytes (or less), we do the following:

Input into the hash function the local 'md' (which is initialized from the global 'md' before any bytes are generated), the bytes that are to be overwritten by the random bytes, and bytes from the 'state' (incrementing looping index). From this digest output (which is kept in 'md'), the top (up to) 10 bytes are returned to the caller and the bottom 10 bytes are xored into the 'state'.

Finally, after we have finished 'num' random bytes for the caller, 'count' (which is incremented) and the local and global 'md' are fed into the hash function and the results are kept in the global 'md'.

I believe the above addressed points 1 (use of SHA-1), 6 (by hashing into the 'state' the 'old' data from the caller that is about to be overwritten) and 7 (by not using the 10 bytes given to the caller to update the 'state', but they are used to update 'md').

So of the points raised, only 2 is not addressed (but see RAND_add(3)).

SEE ALSO

BN_rand(3), RAND_add(3), RAND_load_file(3), RAND_egd(3), RAND_bytes(3), RAND_set_rand_method(3), RAND_cleanup(3)

Name

RAND_add, RAND_seed, RAND_status, RAND_event and RAND_screen — add entropy to the PRNG

Synopsis

```
#include  <openssl/rand.h>

void RAND_seed(const void *buf, int num);

void RAND_add(const void *buf, int num, double entropy);

int  RAND_status(void);

int  RAND_event(UINT iMsg, WPARAM wParam, LPARAM lParam);
void RAND_screen(void);
```

DESCRIPTION

RAND_add() mixes the **num** bytes at **buf** into the PRNG state. Thus, if the data at **buf** are unpredictable to an adversary, this increases the uncertainty about the state and makes the PRNG output less predictable. Suitable input comes from user interaction (random key presses, mouse movements) and certain hardware events. The **entropy** argument is (the lower bound of) an estimate of how much randomness is contained in **buf**, measured in bytes. Details about sources of randomness and how to estimate their entropy can be found in the literature, e.g. RFC 1750.

RAND_add() may be called with sensitive data such as user entered passwords. The seed values cannot be recovered from the PRNG output.

OpenSSL makes sure that the PRNG state is unique for each thread. On systems that provide `/dev/urandom`, the randomness device is used to seed the PRNG transparently. However, on all other systems, the application is responsible for seeding the PRNG by calling RAND_add(), RAND_egd(3) or RAND_load_file(3).

RAND_seed() is equivalent to RAND_add() when **num == entropy**.

RAND_event() collects the entropy from Windows events such as mouse movements and other user interaction. It should be called with the **iMsg**, **wParam** and **lParam** arguments of *all* messages sent to the window procedure. It will estimate the entropy contained in the event message (if any), and add it to the PRNG. The program can then process the messages as usual.

The RAND_screen() function is available for the convenience of Windows programmers. It adds the current contents of the screen to the PRNG. For applications that can catch Windows events, seeding the PRNG by calling RAND_event() is a significantly better source of randomness. It should be noted that both methods cannot be used on servers that run without user interaction.

RETURN VALUES

RAND_status() and RAND_event() return 1 if the PRNG has been seeded with enough data, 0 otherwise.

The other functions do not return values.

SEE ALSO

rand(3), RAND_egd(3), RAND_load_file(3), RAND_cleanup(3)

HISTORY

RAND_seed() and RAND_screen() are available in all versions of SSLeay and OpenSSL. RAND_add() and RAND_status() have been added in OpenSSL 0.9.5, RAND_event() in OpenSSL 0.9.5a.

Name

RAND_bytes and RAND_pseudo_bytes — generate random data

Synopsis

```
#include  <openssl/rand.h>
```

```
int RAND_bytes(unsigned char *buf, int num);
```

```
int RAND_pseudo_bytes(unsigned char *buf, int num);
```

DESCRIPTION

RAND_bytes() puts **num** cryptographically strong pseudo-random bytes into **buf**. An error occurs if the PRNG has not been seeded with enough randomness to ensure an unpredictable byte sequence.

RAND_pseudo_bytes() puts **num** pseudo-random bytes into **buf**. Pseudo-random byte sequences generated by RAND_pseudo_bytes() will be unique if they are of sufficient length, but are not necessarily unpredictable. They can be used for non-cryptographic purposes and for certain purposes in cryptographic protocols, but usually not for key generation etc.

The contents of **buf** is mixed into the entropy pool before retrieving the new pseudo-random bytes unless disabled at compile time (see FAQ).

RETURN VALUES

RAND_bytes() returns 1 on success, 0 otherwise. The error code can be obtained by ERR_get_error(3). RAND_pseudo_bytes() returns 1 if the bytes generated are cryptographically strong, 0 otherwise. Both functions return -1 if they are not supported by the current RAND method.

SEE ALSO

rand(3), ERR_get_error(3), RAND_add(3)

HISTORY

RAND_bytes() is available in all versions of SSLeay and OpenSSL. It has a return value since OpenSSL 0.9.5. RAND_pseudo_bytes() was added in OpenSSL 0.9.5.

Name

RAND_cleanup — erase the PRNG state

Synopsis

```
#include  <openssl/rand.h>
```

```
void RAND_cleanup(void);
```

DESCRIPTION

RAND_cleanup() erases the memory used by the PRNG.

RETURN VALUE

RAND_cleanup() returns no value.

SEE ALSO

rand(3)

HISTORY

RAND_cleanup() is available in all versions of SSLeay and OpenSSL.

Name

RAND_egd — query entropy gathering daemon

Synopsis

```
#include  <openssl/rand.h>

int RAND_egd(const char *path);
int RAND_egd_bytes(const char *path, int bytes);

int RAND_query_egd_bytes(const char *path, unsigned char *buf, int bytes);
```

DESCRIPTION

RAND_egd() queries the entropy gathering daemon EGD on socket **path**. It queries 255 bytes and uses RAND_add(3) to seed the OpenSSL built-in PRNG. RAND_egd(path) is a wrapper for RAND_egd_bytes(path, 255);

RAND_egd_bytes() queries the entropy gathering daemon EGD on socket **path**. It queries **bytes** bytes and uses RAND_add(3) to seed the OpenSSL built-in PRNG. This function is more flexible than RAND_egd(). When only one secret key must be generated, it is not necessary to request the full amount 255 bytes from the EGD socket. This can be advantageous, since the amount of entropy that can be retrieved from EGD over time is limited.

RAND_query_egd_bytes() performs the actual query of the EGD daemon on socket **path**. If **buf** is given, **bytes** bytes are queried and written into **buf**. If **buf** is NULL, **bytes** bytes are queried and used to seed the OpenSSL built-in PRNG using RAND_add(3).

NOTES

On systems without /dev/*random devices providing entropy from the kernel, the EGD entropy gathering daemon can be used to collect entropy. It provides a socket interface through which entropy can be gathered in chunks up to 255 bytes. Several chunks can be queried during one connection.

EGD is available from http://www.lothar.com/tech/crypto/ (`perl Makefile.PL; make; make install` to install). It is run as **egd** *path*, where *path* is an absolute path designating a socket. When RAND_egd() is called with that path as an argument, it tries to read random bytes that EGD has collected. RAND_egd() retrieves entropy from the daemon using the daemon's "non-blocking read" command which shall be answered immediately by the daemon without waiting for additional entropy to be collected. The write and read socket operations in the communication are blocking.

Alternatively, the EGD-interface compatible daemon PRNGD can be used. It is available from http://prngd.sourceforge.net/ . PRNGD does employ an internal PRNG itself and can therefore never run out of entropy.

OpenSSL automatically queries EGD when entropy is requested via RAND_bytes() or the status is checked via RAND_status() for the first time, if the socket is located at /var/run/egd-pool, /dev/egd-pool or /etc/egd-pool.

RETURN VALUE

RAND_egd() and RAND_egd_bytes() return the number of bytes read from the daemon on success, and -1 if the connection failed or the daemon did not return enough data to fully seed the PRNG.

RAND_query_egd_bytes() returns the number of bytes read from the daemon on success, and -1 if the connection failed. The PRNG state is not considered.

SEE ALSO

rand(3), RAND_add(3), RAND_cleanup(3)

HISTORY

RAND_egd() is available since OpenSSL 0.9.5.

RAND_egd_bytes() is available since OpenSSL 0.9.6.

RAND_query_egd_bytes() is available since OpenSSL 0.9.7.

The automatic query of /var/run/egd-pool et al was added in OpenSSL 0.9.7.

Name

RAND_load_file, RAND_write_file and RAND_file_name — PRNG seed file

Synopsis

```
#include  <openssl/rand.h>
```

```
const char *RAND_file_name(char *buf, size_t num);
```

```
int RAND_load_file(const char *filename, long max_bytes);
```

```
int RAND_write_file(const char *filename);
```

DESCRIPTION

RAND_file_name() generates a default path for the random seed file. **buf** points to a buffer of size **num** in which to store the filename. The seed file is $RANDFILE if that environment variable is set, $HOME/.rnd otherwise. If $HOME is not set either, or **num** is too small for the path name, an error occurs.

RAND_load_file() reads a number of bytes from file **filename** and adds them to the PRNG. If **max_bytes** is non-negative, up to to **max_bytes** are read; starting with OpenSSL 0.9.5, if **max_bytes** is -1, the complete file is read.

RAND_write_file() writes a number of random bytes (currently 1024) to file **filename** which can be used to initialize the PRNG by calling RAND_load_file() in a later session.

RETURN VALUES

RAND_load_file() returns the number of bytes read.

RAND_write_file() returns the number of bytes written, and -1 if the bytes written were generated without appropriate seed.

RAND_file_name() returns a pointer to **buf** on success, and NULL on error.

SEE ALSO

rand(3), RAND_add(3), RAND_cleanup(3)

HISTORY

RAND_load_file(), RAND_write_file() and RAND_file_name() are available in all versions of SSLeay and OpenSSL.

Name

RAND_set_rand_method, RAND_get_rand_method and RAND_SSLeay — select RAND method

Synopsis

```
#include  <openssl/rand.h>
```

```
void RAND_set_rand_method(const RAND_METHOD *meth);
```

```
const RAND_METHOD *RAND_get_rand_method(void);
```

```
RAND_METHOD *RAND_SSLeay(void);
```

DESCRIPTION

A **RAND_METHOD** specifies the functions that OpenSSL uses for random number generation. By modifying the method, alternative implementations such as hardware RNGs may be used. IMPORTANT: See the NOTES section for important information about how these RAND API functions are affected by the use of **ENGINE** API calls.

Initially, the default RAND_METHOD is the OpenSSL internal implementation, as returned by RAND_SSLeay().

RAND_set_default_method() makes **meth** the method for PRNG use. **NB**: This is true only whilst no ENGINE has been set as a default for RAND, so this function is no longer recommended.

RAND_get_default_method() returns a pointer to the current RAND_METHOD. However, the meaningfulness of this result is dependent on whether the ENGINE API is being used, so this function is no longer recommended.

THE RAND_METHOD STRUCTURE

```
typedef struct rand_meth_st
{
       void (*seed)(const void *buf, int num);
       int (*bytes)(unsigned char *buf, int num);
       void (*cleanup)(void);
       void (*add)(const void *buf, int num, int entropy);
       int (*pseudorand)(unsigned char *buf, int num);
       int (*status)(void);
} RAND_METHOD;
```

The components point to the implementation of RAND_seed(), RAND_bytes(), RAND_cleanup(), RAND_add(), RAND_pseudo_rand() and RAND_status(). Each component may be NULL if the function is not implemented.

RETURN VALUES

RAND_set_rand_method() returns no value. RAND_get_rand_method() and RAND_SSLeay() return pointers to the respective methods.

NOTES

As of version 0.9.7, RAND_METHOD implementations are grouped together with other algorithmic APIs (eg. RSA_METHOD, EVP_CIPHER, etc) in **ENGINE** modules. If a default ENGINE is specified for RAND functionality using an ENGINE API function, that will override any RAND defaults set using the RAND API (ie. RAND_set_rand_method()). For this reason, the ENGINE API is the recommended way to control default implementations for use in RAND and other cryptographic algorithms.

SEE ALSO

rand(3), engine(3)

HISTORY

RAND_set_rand_method(), RAND_get_rand_method() and RAND_SSLeay() are available in all versions of OpenSSL.

In the engine version of version 0.9.6, RAND_set_rand_method() was altered to take an ENGINE pointer as its argument. As of version 0.9.7, that has been reverted as the ENGINE API transparently overrides RAND defaults if used, otherwise RAND API functions work as before. RAND_set_rand_engine() was also introduced in version 0.9.7.

Name

RC4_set_key and RC4 — RC4 encryption

Synopsis

```
#include  <openssl/rc4.h>
```

```
void RC4_set_key(RC4_KEY *key, int len, const unsigned char *data);
```

```
void RC4(RC4_KEY *key, unsigned long len, const unsigned char *indata,
         unsigned char *outdata);
```

DESCRIPTION

This library implements the Alleged RC4 cipher, which is described for example in *Applied Cryptography*. It is believed to be compatible with RC4[TM], a proprietary cipher of RSA Security Inc.

RC4 is a stream cipher with variable key length. Typically, 128 bit (16 byte) keys are used for strong encryption, but shorter insecure key sizes have been widely used due to export restrictions.

RC4 consists of a key setup phase and the actual encryption or decryption phase.

RC4_set_key() sets up the **RC4_KEY key** using the **len** bytes long key at **data**.

RC4() encrypts or decrypts the **len** bytes of data at **indata** using **key** and places the result at **outdata**. Repeated RC4() calls with the same **key** yield a continuous key stream.

Since RC4 is a stream cipher (the input is XORed with a pseudo-random key stream to produce the output), decryption uses the same function calls as encryption.

Applications should use the higher level functions EVP_EncryptInit(3) etc. instead of calling the RC4 functions directly.

RETURN VALUES

RC4_set_key() and RC4() do not return values.

NOTE

Certain conditions have to be observed to securely use stream ciphers. It is not permissible to perform multiple encryptions using the same key stream.

SEE ALSO

blowfish(3), des(3), rc2(3)

HISTORY

RC4_set_key() and RC4() are available in all versions of SSLeay and OpenSSL.

Name

RIPEMD160, RIPEMD160_Init, RIPEMD160_Update and RIPEMD160_Final — RIPEMD-160 hash function

Synopsis

```
#include  <openssl/ripemd.h>
```

```
unsigned char *RIPEMD160(const unsigned char *d, unsigned long n,
                 unsigned char *md);
```

```
int RIPEMD160_Init(RIPEMD160_CTX *c);
int RIPEMD160_Update(RIPEMD_CTX *c, const void *data,
                 unsigned long len);
int RIPEMD160_Final(unsigned char *md, RIPEMD160_CTX *c);
```

DESCRIPTION

RIPEMD-160 is a cryptographic hash function with a 160 bit output.

RIPEMD160() computes the RIPEMD-160 message digest of the **n** bytes at **d** and places it in **md** (which must have space for RIPEMD160_DIGEST_LENGTH == 20 bytes of output). If **md** is NULL, the digest is placed in a static array.

The following functions may be used if the message is not completely stored in memory:

RIPEMD160_Init() initializes a **RIPEMD160_CTX** structure.

RIPEMD160_Update() can be called repeatedly with chunks of the message to be hashed (**len** bytes at **data**).

RIPEMD160_Final() places the message digest in **md**, which must have space for RIPEMD160_DIGEST_LENGTH == 20 bytes of output, and erases the **RIPEMD160_CTX**.

Applications should use the higher level functions EVP_DigestInit(3) etc. instead of calling the hash functions directly.

RETURN VALUES

RIPEMD160() returns a pointer to the hash value.

RIPEMD160_Init(), RIPEMD160_Update() and RIPEMD160_Final() return 1 for success, 0 otherwise.

CONFORMING TO

ISO/IEC 10118-3 (draft) (??)

SEE ALSO

sha(3), hmac(3), EVP_DigestInit(3)

HISTORY

RIPEMD160(), RIPEMD160_Init(), RIPEMD160_Update() and RIPEMD160_Final() are available since SSLeay 0.9.0.

Name

rsa — RSA public key cryptosystem

Synopsis

```
#include  <openssl/rsa.h>
#include  <openssl/engine.h>
```

```
RSA * RSA_new(void);
void RSA_free(RSA *rsa);
```

```
int RSA_public_encrypt(int flen, unsigned char *from,
   unsigned char *to, RSA *rsa, int padding);
int RSA_private_decrypt(int flen, unsigned char *from,
   unsigned char *to, RSA *rsa, int padding);
int RSA_private_encrypt(int flen, unsigned char *from,
   unsigned char *to, RSA *rsa,int padding);
int RSA_public_decrypt(int flen, unsigned char *from,
   unsigned char *to, RSA *rsa,int padding);
```

```
int RSA_sign(int type, unsigned char *m, unsigned int m_len,
   unsigned char *sigret, unsigned int *siglen, RSA *rsa);
int RSA_verify(int type, unsigned char *m, unsigned int m_len,
   unsigned char *sigbuf, unsigned int siglen, RSA *rsa);
```

```
int RSA_size(const RSA *rsa);
```

```
RSA *RSA_generate_key(int num, unsigned long e,
   void (*callback)(int,int,void *), void *cb_arg);
```

```
int RSA_check_key(RSA *rsa);
```

```
int RSA_blinding_on(RSA *rsa, BN_CTX *ctx);
void RSA_blinding_off(RSA *rsa);
```

```
void RSA_set_default_method(const RSA_METHOD *meth);
const RSA_METHOD *RSA_get_default_method(void);
int RSA_set_method(RSA *rsa, const RSA_METHOD *meth);
const RSA_METHOD *RSA_get_method(const RSA *rsa);
RSA_METHOD *RSA_PKCS1_SSLeay(void);
RSA_METHOD *RSA_null_method(void);
int RSA_flags(const RSA *rsa);
RSA *RSA_new_method(ENGINE *engine);
```

```
int RSA_print(BIO *bp, RSA *x, int offset);
int RSA_print_fp(FILE *fp, RSA *x, int offset);
```

```
int RSA_get_ex_new_index(long argl, char *argp, int (*new_func)(),
   int (*dup_func)(), void (*free_func)());
int RSA_set_ex_data(RSA *r,int idx,char *arg);
char *RSA_get_ex_data(RSA *r, int idx);
```

```
int RSA_sign_ASN1_OCTET_STRING(int dummy, unsigned char *m,
   unsigned int m_len, unsigned char *sigret, unsigned int *siglen,
   RSA *rsa);
int RSA_verify_ASN1_OCTET_STRING(int dummy, unsigned char *m,
   unsigned int m_len, unsigned char *sigbuf, unsigned int siglen,
   RSA *rsa);
```

DESCRIPTION

These functions implement RSA public key encryption and signatures as defined in PKCS #1 v2.0 [RFC 2437].

The **RSA** structure consists of several BIGNUM components. It can contain public as well as private RSA keys:

```
struct
```

```
       {
       BIGNUM *n;              // public modulus
       BIGNUM *e;              // public exponent
       BIGNUM *d;              // private exponent
       BIGNUM *p;              // secret prime factor
       BIGNUM *q;              // secret prime factor
       BIGNUM *dmp1;           // d mod (p-1)
       BIGNUM *dmq1;           // d mod (q-1)
       BIGNUM *iqmp;           // q^-1 mod p
       // ...
       };
RSA
```

In public keys, the private exponent and the related secret values are **NULL**.

p, **q**, **dmp1**, **dmq1** and **iqmp** may be **NULL** in private keys, but the RSA operations are much faster when these values are available.

Note that RSA keys may use non-standard **RSA_METHOD** implementations, either directly or by the use of **ENGINE** modules. In some cases (eg. an ENGINE providing support for hardware-embedded keys), these BIGNUM values will not be used by the implementation or may be used for alternative data storage. For this reason, applications should generally avoid using RSA structure elements directly and instead use API functions to query or modify keys.

CONFORMING TO

SSL, PKCS #1 v2.0

PATENTS

RSA was covered by a US patent which expired in September 2000.

SEE ALSO

rsa(1), bn(3), dsa(3), dh(3), rand(3), engine(3), RSA_new(3), RSA_public_encrypt(3), RSA_sign(3), RSA_size(3), RSA_generate_key(3), RSA_check_key(3), RSA_blinding_on(3), RSA_set_method(3), RSA_print(3), RSA_get_ex_new_index(3), RSA_private_encrypt(3), RSA_sign_ASN1_OCTET_STRING(3), RSA_padding_add_PKCS1_type_1(3)

Name

RSA_blinding_on and RSA_blinding_off — protect the RSA operation from timing attacks

Synopsis

```
#include  <openssl/rsa.h>
```

```
int RSA_blinding_on(RSA *rsa, BN_CTX *ctx);
```

```
void RSA_blinding_off(RSA *rsa);
```

DESCRIPTION

RSA is vulnerable to timing attacks. In a setup where attackers can measure the time of RSA decryption or signature operations, blinding must be used to protect the RSA operation from that attack.

RSA_blinding_on() turns blinding on for key **rsa** and generates a random blinding factor. **ctx** is **NULL** or a pre-allocated and initialized **BN_CTX**. The random number generator must be seeded prior to calling RSA_blinding_on().

RSA_blinding_off() turns blinding off and frees the memory used for the blinding factor.

RETURN VALUES

RSA_blinding_on() returns 1 on success, and 0 if an error occurred.

RSA_blinding_off() returns no value.

SEE ALSO

rsa(3), rand(3)

HISTORY

RSA_blinding_on() and RSA_blinding_off() appeared in SSLeay 0.9.0.

Name

RSA_check_key — validate private RSA keys

Synopsis

```
#include  <openssl/rsa.h>
```

```
int RSA_check_key(RSA *rsa);
```

DESCRIPTION

This function validates RSA keys. It checks that **p** and **q** are in fact prime, and that **n = p*q**.

It also checks that **d*e = 1 mod (p-1*q-1)**, and that **dmp1**, **dmq1** and **iqmp** are set correctly or are **NULL**.

As such, this function can not be used with any arbitrary RSA key object, even if it is otherwise fit for regular RSA operation. See **NOTES** for more information.

RETURN VALUE

RSA_check_key() returns 1 if **rsa** is a valid RSA key, and 0 otherwise. -1 is returned if an error occurs while checking the key.

If the key is invalid or an error occurred, the reason code can be obtained using ERR_get_error(3).

NOTES

This function does not work on RSA public keys that have only the modulus and public exponent elements populated. It performs integrity checks on all the RSA key material, so the RSA key structure must contain all the private key data too.

Unlike most other RSA functions, this function does **not** work transparently with any underlying ENGINE implementation because it uses the key data in the RSA structure directly. An ENGINE implementation can override the way key data is stored and handled, and can even provide support for HSM keys - in which case the RSA structure may contain **no** key data at all! If the ENGINE in question is only being used for acceleration or analysis purposes, then in all likelihood the RSA key data is complete and untouched, but this can't be assumed in the general case.

BUGS

A method of verifying the RSA key using opaque RSA API functions might need to be considered. Right now RSA_check_key() simply uses the RSA structure elements directly, bypassing the RSA_METHOD table altogether (and completely violating encapsulation and object-orientation in the process). The best fix will probably be to introduce a "check_key()" handler to the RSA_METHOD function table so that alternative implementations can also provide their own verifiers.

SEE ALSO

rsa(3), ERR_get_error(3)

HISTORY

RSA_check_key() appeared in OpenSSL 0.9.4.

Name

RSA_generate_key — generate RSA key pair

Synopsis

```
#include  <openssl/rsa.h>
```

```
RSA *RSA_generate_key(int num, unsigned long e,
   void (*callback)(int,int,void *), void *cb_arg);
```

DESCRIPTION

RSA_generate_key() generates a key pair and returns it in a newly allocated **RSA** structure. The pseudo-random number generator must be seeded prior to calling RSA_generate_key().

The modulus size will be **num** bits, and the public exponent will be **e**. Key sizes with **num** < 1024 should be considered insecure. The exponent is an odd number, typically 3, 17 or 65537.

A callback function may be used to provide feedback about the progress of the key generation. If **callback** is not **NULL**, it will be called as follows:

- While a random prime number is generated, it is called as described in BN_generate_prime(3).
- When the n-th randomly generated prime is rejected as not suitable for the key, **callback(2, n, cb_arg)** is called.
- When a random p has been found with p-1 relatively prime to **e**, it is called as **callback(3, 0, cb_arg)**.

The process is then repeated for prime q with **callback(3, 1, cb_arg)**.

RETURN VALUE

If key generation fails, RSA_generate_key() returns **NULL**; the error codes can be obtained by ERR_get_error(3).

BUGS

callback(2, x, cb_arg) is used with two different meanings.

RSA_generate_key() goes into an infinite loop for illegal input values.

SEE ALSO

ERR_get_error(3), rand(3), rsa(3), RSA_free(3)

HISTORY

The **cb_arg** argument was added in SSLeay 0.9.0.

Name

RSA_get_ex_new_index, RSA_set_ex_data and RSA_get_ex_data — add application specific data to RSA structures

Synopsis

```
#include  <openssl/rsa.h>
```

```
int RSA_get_ex_new_index(long argl, void *argp,
               CRYPTO_EX_new *new_func,
               CRYPTO_EX_dup *dup_func,
               CRYPTO_EX_free *free_func);
```

```
int RSA_set_ex_data(RSA *r, int idx, void *arg);
```

```
void *RSA_get_ex_data(RSA *r, int idx);
```

```
typedef int CRYPTO_EX_new(void *parent, void *ptr, CRYPTO_EX_DATA *ad,
                          int idx, long argl, void *argp);
typedef void CRYPTO_EX_free(void *parent, void *ptr, CRYPTO_EX_DATA *ad,
                            int idx, long argl, void *argp);
typedef int CRYPTO_EX_dup(CRYPTO_EX_DATA *to, CRYPTO_EX_DATA *from, void *from_d,
                          int idx, long argl, void *argp);
```

DESCRIPTION

Several OpenSSL structures can have application specific data attached to them. This has several potential uses, it can be used to cache data associated with a structure (for example the hash of some part of the structure) or some additional data (for example a handle to the data in an external library).

Since the application data can be anything at all it is passed and retrieved as a **void *** type.

The **RSA_get_ex_new_index()** function is initially called to "register" some new application specific data. It takes three optional function pointers which are called when the parent structure (in this case an RSA structure) is initially created, when it is copied and when it is freed up. If any or all of these function pointer arguments are not used they should be set to NULL. The precise manner in which these function pointers are called is described in more detail below. **RSA_get_ex_new_index()** also takes additional long and pointer parameters which will be passed to the supplied functions but which otherwise have no special meaning. It returns an **index** which should be stored (typically in a static variable) and passed used in the **idx** parameter in the remaining functions. Each successful call to **RSA_get_ex_new_index()** will return an index greater than any previously returned, this is important because the optional functions are called in order of increasing index value.

RSA_set_ex_data() is used to set application specific data, the data is supplied in the **arg** parameter and its precise meaning is up to the application.

RSA_get_ex_data() is used to retrieve application specific data. The data is returned to the application, this will be the same value as supplied to a previous **RSA_set_ex_data()** call.

new_func() is called when a structure is initially allocated (for example with **RSA_new()**. The parent structure members will not have any meaningful values at this point. This function will typically be used to allocate any application specific structure.

free_func() is called when a structure is being freed up. The dynamic parent structure members should not be accessed because they will be freed up when this function is called.

new_func() and **free_func()** take the same parameters. **parent** is a pointer to the parent RSA structure. **ptr** is a the application specific data (this wont be of much use in **new_func()**. **ad** is a pointer to the **CRYPTO_EX_DATA** structure from the parent RSA structure: the functions **CRYPTO_get_ex_data()** and **CRYPTO_set_ex_data()** can be called to manipulate it. The **idx** parameter is the index: this will be the same value returned by **RSA_get_ex_new_index()** when the functions were initially registered. Finally the **argl** and **argp** parameters are the values originally passed to the same corresponding parameters when **RSA_get_ex_new_index()** was called.

dup_func() is called when a structure is being copied. Pointers to the destination and source **CRYPTO_EX_DATA** structures are passed in the **to** and **from** parameters respectively. The **from_d** parameter is passed a pointer to the source application data when the function is called, when the function returns the value is copied to the destination: the application can thus modify the data pointed to by **from_d** and have different values in the source and destination. The **idx**, **argl** and **argp** parameters are the same as those in **new_func()** and **free_func()**.

RETURN VALUES

RSA_get_ex_new_index() returns a new index or -1 on failure (note 0 is a valid index value).

RSA_set_ex_data() returns 1 on success or 0 on failure.

RSA_get_ex_data() returns the application data or 0 on failure. 0 may also be valid application data but currently it can only fail if given an invalid **idx** parameter.

new_func() and **dup_func()** should return 0 for failure and 1 for success.

On failure an error code can be obtained from ERR_get_error(3).

BUGS

dup_func() is currently never called.

The return value of **new_func()** is ignored.

The **new_func()** function isn't very useful because no meaningful values are present in the parent RSA structure when it is called.

SEE ALSO

rsa(3), CRYPTO_set_ex_data(3)

HISTORY

RSA_get_ex_new_index(), RSA_set_ex_data() and RSA_get_ex_data() are available since SSLeay 0.9.0.

Name

RSA_new and RSA_free — allocate and free RSA objects

Synopsis

```
#include  <openssl/rsa.h>
```

```
RSA * RSA_new(void);
```

```
void RSA_free(RSA *rsa);
```

DESCRIPTION

RSA_new() allocates and initializes an **RSA** structure. It is equivalent to calling RSA_new_method(NULL).

RSA_free() frees the **RSA** structure and its components. The key is erased before the memory is returned to the system.

RETURN VALUES

If the allocation fails, RSA_new() returns **NULL** and sets an error code that can be obtained by ERR_get_error(3). Otherwise it returns a pointer to the newly allocated structure.

RSA_free() returns no value.

SEE ALSO

ERR_get_error(3), rsa(3), RSA_generate_key(3), RSA_new_method(3)

HISTORY

RSA_new() and RSA_free() are available in all versions of SSLeay and OpenSSL.

Name

RSA_padding_add_PKCS1_type_1, RSA_padding_check_PKCS1_type_1, RSA_padding_add_PKCS1_type_2, RSA_padding_check_PKCS1_type_2, RSA_padding_add_PKCS1_OAEP, RSA_padding_check_PKCS1_OAEP, RSA_padding_add_SSLv23, RSA_padding_check_SSLv23, RSA_padding_add_none and RSA_padding_check_none — asymmetric encryption padding

Synopsis

```
#include  <openssl/rsa.h>

int RSA_padding_add_PKCS1_type_1(unsigned char *to, int tlen,
   unsigned char *f, int fl);

int RSA_padding_check_PKCS1_type_1(unsigned char *to, int tlen,
   unsigned char *f, int fl, int rsa_len);

int RSA_padding_add_PKCS1_type_2(unsigned char *to, int tlen,
   unsigned char *f, int fl);

int RSA_padding_check_PKCS1_type_2(unsigned char *to, int tlen,
   unsigned char *f, int fl, int rsa_len);

int RSA_padding_add_PKCS1_OAEP(unsigned char *to, int tlen,
   unsigned char *f, int fl, unsigned char *p, int pl);

int RSA_padding_check_PKCS1_OAEP(unsigned char *to, int tlen,
   unsigned char *f, int fl, int rsa_len, unsigned char *p, int pl);

int RSA_padding_add_SSLv23(unsigned char *to, int tlen,
   unsigned char *f, int fl);

int RSA_padding_check_SSLv23(unsigned char *to, int tlen,
   unsigned char *f, int fl, int rsa_len);

int RSA_padding_add_none(unsigned char *to, int tlen,
   unsigned char *f, int fl);

int RSA_padding_check_none(unsigned char *to, int tlen,
   unsigned char *f, int fl, int rsa_len);
```

DESCRIPTION

The RSA_padding_xxx_xxx() functions are called from the RSA encrypt, decrypt, sign and verify functions. Normally they should not be called from application programs.

However, they can also be called directly to implement padding for other asymmetric ciphers. RSA_padding_add_PKCS1_OAEP() and RSA_padding_check_PKCS1_OAEP() may be used in an application combined with **RSA_NO_PADDING** in order to implement OAEP with an encoding parameter.

RSA_padding_add_xxx() encodes **fl** bytes from **f** so as to fit into **tlen** bytes and stores the result at **to**. An error occurs if **fl** does not meet the size requirements of the encoding method.

The following encoding methods are implemented:

PKCS1_type_1	PKCS #1 v2.0 EMSA-PKCS1-v1_5 (PKCS #1 v1.5 block type 1); used for signatures
PKCS1_type_2	PKCS #1 v2.0 EME-PKCS1-v1_5 (PKCS #1 v1.5 block type 2)
PKCS1_OAEP	PKCS #1 v2.0 EME-OAEP
SSLv23	PKCS #1 EME-PKCS1-v1_5 with SSL-specific modification

none simply copy the data

The random number generator must be seeded prior to calling RSA_padding_add_xxx().

RSA_padding_check_xxx() verifies that the **fl** bytes at **f** contain a valid encoding for a **rsa_len** byte RSA key in the respective encoding method and stores the recovered data of at most **tlen** bytes (for **RSA_NO_PADDING**: of size **tlen**) at **to**.

For RSA_padding_xxx_OAEP(), **p** points to the encoding parameter of length **pl**. **p** may be **NULL** if **pl** is 0.

RETURN VALUES

The RSA_padding_add_xxx() functions return 1 on success, 0 on error. The RSA_padding_check_xxx() functions return the length of the recovered data, -1 on error. Error codes can be obtained by calling ERR_get_error(3).

SEE ALSO

RSA_public_encrypt(3), RSA_private_decrypt(3), RSA_sign(3), RSA_verify(3)

HISTORY

RSA_padding_add_PKCS1_type_1(), RSA_padding_check_PKCS1_type_1(), RSA_padding_add_PKCS1_type_2(), RSA_padding_check_PKCS1_type_2(), RSA_padding_add_SSLv23(), RSA_padding_check_SSLv23(), RSA_padding_add_none() and RSA_padding_check_none() appeared in SSLeay 0.9.0.

RSA_padding_add_PKCS1_OAEP() and RSA_padding_check_PKCS1_OAEP() were added in OpenSSL 0.9.2b.

Name

RSA_print, RSA_print_fp, DSAparams_print, DSAparams_print_fp, DSA_print, DSA_print_fp, DHparams_print and DHparams_print_fp — print cryptographic parameters

Synopsis

```
#include  <openssl/rsa.h>
```

```
int RSA_print(BIO *bp, RSA *x, int offset);
int RSA_print_fp(FILE *fp, RSA *x, int offset);
```

```
#include  <openssl/dsa.h>
```

```
int DSAparams_print(BIO *bp, DSA *x);
int DSAparams_print_fp(FILE *fp, DSA *x);
int DSA_print(BIO *bp, DSA *x, int offset);
int DSA_print_fp(FILE *fp, DSA *x, int offset);
```

```
#include  <openssl/dh.h>
```

```
int DHparams_print(BIO *bp, DH *x);
int DHparams_print_fp(FILE *fp, DH *x);
```

DESCRIPTION

A human-readable hexadecimal output of the components of the RSA key, DSA parameters or key or DH parameters is printed to **bp** or **fp**.

The output lines are indented by **offset** spaces.

RETURN VALUES

These functions return 1 on success, 0 on error.

SEE ALSO

dh(3), dsa(3), rsa(3), BN_bn2bin(3)

HISTORY

RSA_print(), RSA_print_fp(), DSA_print(), DSA_print_fp(), DH_print(), DH_print_fp() are available in all versions of SSLeay and OpenSSL. DSAparams_print() and DSAparams_print_fp() were added in SSLeay 0.8.

Name

RSA_private_encrypt and RSA_public_decrypt — low level signature operations

Synopsis

```
#include  <openssl/rsa.h>
```

```
int RSA_private_encrypt(int flen, unsigned char *from,
   unsigned char *to, RSA *rsa, int padding);
```

```
int RSA_public_decrypt(int flen, unsigned char *from,
   unsigned char *to, RSA *rsa, int padding);
```

DESCRIPTION

These functions handle RSA signatures at a low level.

RSA_private_encrypt() signs the **flen** bytes at **from** (usually a message digest with an algorithm identifier) using the private key **rsa** and stores the signature in **to**. **to** must point to **RSA_size(rsa)** bytes of memory.

padding denotes one of the following modes:

RSA_PKCS1_PADDING

> PKCS #1 v1.5 padding. This function does not handle the **algorithmIdentifier** specified in PKCS #1. When generating or verifying PKCS #1 signatures, RSA_sign(3) and RSA_verify(3) should be used.

RSA_NO_PADDING

> Raw RSA signature. This mode should *only* be used to implement cryptographically sound padding modes in the application code. Signing user data directly with RSA is insecure.

RSA_public_decrypt() recovers the message digest from the **flen** bytes long signature at **from** using the signer's public key **rsa**. **to** must point to a memory section large enough to hold the message digest (which is smaller than **RSA_size(rsa) - 11**). **padding** is the padding mode that was used to sign the data.

RETURN VALUES

RSA_private_encrypt() returns the size of the signature (i.e., RSA_size(rsa)). RSA_public_decrypt() returns the size of the recovered message digest.

On error, -1 is returned; the error codes can be obtained by ERR_get_error(3).

SEE ALSO

ERR_get_error(3), rsa(3), RSA_sign(3), RSA_verify(3)

HISTORY

The **padding** argument was added in SSLeay 0.8. RSA_NO_PADDING is available since SSLeay 0.9.0.

Name

RSA_public_encrypt and RSA_private_decrypt — RSA public key cryptography

Synopsis

```
#include  <openssl/rsa.h>
```

```
int RSA_public_encrypt(int flen, unsigned char *from,
   unsigned char *to, RSA *rsa, int padding);
```

```
int RSA_private_decrypt(int flen, unsigned char *from,
    unsigned char *to, RSA *rsa, int padding);
```

DESCRIPTION

RSA_public_encrypt() encrypts the **flen** bytes at **from** (usually a session key) using the public key **rsa** and stores the ciphertext in **to**. **to** must point to RSA_size(**rsa**) bytes of memory.

padding denotes one of the following modes:

RSA_PKCS1_PADDING

> PKCS #1 v1.5 padding. This currently is the most widely used mode.

RSA_PKCS1_OAEP_PADDING

> EME-OAEP as defined in PKCS #1 v2.0 with SHA-1, MGF1 and an empty encoding parameter. This mode is recommended for all new applications.

RSA_SSLV23_PADDING

> PKCS #1 v1.5 padding with an SSL-specific modification that denotes that the server is SSL3 capable.

RSA_NO_PADDING

> Raw RSA encryption. This mode should *only* be used to implement cryptographically sound padding modes in the application code. Encrypting user data directly with RSA is insecure.

flen must be less than RSA_size(**rsa**) - 11 for the PKCS #1 v1.5 based padding modes, less than RSA_size(**rsa**) - 41 for RSA_PKCS1_OAEP_PADDING and exactly RSA_size(**rsa**) for RSA_NO_PADDING. The random number generator must be seeded prior to calling RSA_public_encrypt().

RSA_private_decrypt() decrypts the **flen** bytes at **from** using the private key **rsa** and stores the plaintext in **to**. **to** must point to a memory section large enough to hold the decrypted data (which is smaller than RSA_size(**rsa**)). **padding** is the padding mode that was used to encrypt the data.

RETURN VALUES

RSA_public_encrypt() returns the size of the encrypted data (i.e., RSA_size(**rsa**)). RSA_private_decrypt() returns the size of the recovered plaintext.

On error, -1 is returned; the error codes can be obtained by ERR_get_error(3).

CONFORMING TO

SSL, PKCS #1 v2.0

SEE ALSO

ERR_get_error(3), rand(3), rsa(3), RSA_size(3)

HISTORY

The **padding** argument was added in SSLeay 0.8. RSA_NO_PADDING is available since SSLeay 0.9.0, OAEP was added in OpenSSL 0.9.2b.

Name

RSA_set_default_method, RSA_get_default_method, RSA_set_method, RSA_get_method, RSA_PKCS1_SSLeay, RSA_null_method, RSA_flags and RSA_new_method — select RSA method

Synopsis

```
#include  <openssl/rsa.h>
```

```
void RSA_set_default_method(const RSA_METHOD *meth);
```

```
RSA_METHOD *RSA_get_default_method(void);
```

```
int RSA_set_method(RSA *rsa, const RSA_METHOD *meth);
```

```
RSA_METHOD *RSA_get_method(const RSA *rsa);
```

```
RSA_METHOD *RSA_PKCS1_SSLeay(void);
```

```
RSA_METHOD *RSA_null_method(void);
```

```
int RSA_flags(const RSA *rsa);
```

```
RSA *RSA_new_method(RSA_METHOD *method);
```

DESCRIPTION

An **RSA_METHOD** specifies the functions that OpenSSL uses for RSA operations. By modifying the method, alternative implementations such as hardware accelerators may be used. IMPORTANT: See the NOTES section for important information about how these RSA API functions are affected by the use of **ENGINE** API calls.

Initially, the default RSA_METHOD is the OpenSSL internal implementation, as returned by RSA_PKCS1_SSLeay().

RSA_set_default_method() makes **meth** the default method for all RSA structures created later. **NB**: This is true only whilst no ENGINE has been set as a default for RSA, so this function is no longer recommended.

RSA_get_default_method() returns a pointer to the current default RSA_METHOD. However, the meaningfulness of this result is dependent on whether the ENGINE API is being used, so this function is no longer recommended.

RSA_set_method() selects **meth** to perform all operations using the key **rsa**. This will replace the RSA_METHOD used by the RSA key and if the previous method was supplied by an ENGINE, the handle to that ENGINE will be released during the change. It is possible to have RSA keys that only work with certain RSA_METHOD implementations (eg. from an ENGINE module that supports embedded hardware-protected keys), and in such cases attempting to change the RSA_METHOD for the key can have unexpected results.

RSA_get_method() returns a pointer to the RSA_METHOD being used by **rsa**. This method may or may not be supplied by an ENGINE implementation, but if it is, the return value can only be guaranteed to be valid as long as the RSA key itself is valid and does not have its implementation changed by RSA_set_method().

RSA_flags() returns the **flags** that are set for **rsa**'s current RSA_METHOD. See the BUGS section.

RSA_new_method() allocates and initializes an RSA structure so that **engine** will be used for the RSA operations. If **engine** is NULL, the default ENGINE for RSA operations is used, and if no default ENGINE is set, the RSA_METHOD controlled by RSA_set_default_method() is used.

RSA_flags() returns the **flags** that are set for **rsa**'s current method.

RSA_new_method() allocates and initializes an **RSA** structure so that **method** will be used for the RSA operations. If **method** is **NULL**, the default method is used.

THE RSA_METHOD STRUCTURE

```
typedef struct rsa_meth_st
```

```
{
    /* name of the implementation */
       const char *name;

/* encrypt */
   int (*rsa_pub_enc)(int flen, unsigned char *from,
     unsigned char *to, RSA *rsa, int padding);

/* verify arbitrary data */
   int (*rsa_pub_dec)(int flen, unsigned char *from,
     unsigned char *to, RSA *rsa, int padding);

/* sign arbitrary data */
   int (*rsa_priv_enc)(int flen, unsigned char *from,
     unsigned char *to, RSA *rsa, int padding);

/* decrypt */
   int (*rsa_priv_dec)(int flen, unsigned char *from,
     unsigned char *to, RSA *rsa, int padding);

/* compute r0 = r0 ^ I mod rsa->n (May be NULL for some
                                   implementations) */
   int (*rsa_mod_exp)(BIGNUM *r0, BIGNUM *I, RSA *rsa);

/* compute r = a ^ p mod m (May be NULL for some implementations) */
   int (*bn_mod_exp)(BIGNUM *r, BIGNUM *a, const BIGNUM *p,
     const BIGNUM *m, BN_CTX *ctx, BN_MONT_CTX *m_ctx);

/* called at RSA_new */
   int (*init)(RSA *rsa);

/* called at RSA_free */
   int (*finish)(RSA *rsa);

/* RSA_FLAG_EXT_PKEY        - rsa_mod_exp is called for private key
 *                            operations, even if p,q,dmp1,dmq1,iqmp
 *                            are NULL
 * RSA_FLAG_SIGN_VER        - enable rsa_sign and rsa_verify
 * RSA_METHOD_FLAG_NO_CHECK - don't check pub/private match
 */
   int flags;

char *app_data; /* ?? */

/* sign. For backward compatibility, this is used only
 * if (flags & RSA_FLAG_SIGN_VER)
 */
   int (*rsa_sign)(int type,
           const unsigned char *m, unsigned int m_length,
           unsigned char *sigret, unsigned int *siglen, const RSA *rsa);
/* verify. For backward compatibility, this is used only
 * if (flags & RSA_FLAG_SIGN_VER)
 */
   int (*rsa_verify)(int dtype,
           const unsigned char *m, unsigned int m_length,
           const unsigned char *sigbuf, unsigned int siglen,
                                                           const RSA *rsa);
/* keygen. If NULL builtin RSA key generation will be used */
   int (*rsa_keygen)(RSA *rsa, int bits, BIGNUM *e, BN_GENCB *cb);

} RSA_METHOD;
```

RETURN VALUES

RSA_PKCS1_SSLeay(), RSA_PKCS1_null_method(), RSA_get_default_method() and RSA_get_method() return pointers to the respective RSA_METHODs.

RSA_set_default_method() returns no value.

RSA_set_method() returns a pointer to the old RSA_METHOD implementation that was replaced. However, this return value should probably be ignored because if it was supplied by an ENGINE, the pointer could be invalidated at any time if the ENGINE is unloaded (in fact it could be unloaded as a result of the RSA_set_method() function releasing its handle to the ENGINE). For this reason, the return type may be replaced with a **void** declaration in a future release.

RSA_new_method() returns NULL and sets an error code that can be obtained by ERR_get_error(3) if the allocation fails. Otherwise it returns a pointer to the newly allocated structure.

NOTES

As of version 0.9.7, RSA_METHOD implementations are grouped together with other algorithmic APIs (eg. DSA_METHOD, EVP_CIPHER, etc) into **ENGINE** modules. If a default ENGINE is specified for RSA functionality using an ENGINE API function, that will override any RSA defaults set using the RSA API (ie. RSA_set_default_method()). For this reason, the ENGINE API is the recommended way to control default implementations for use in RSA and other cryptographic algorithms.

BUGS

The behaviour of RSA_flags() is a mis-feature that is left as-is for now to avoid creating compatibility problems. RSA functionality, such as the encryption functions, are controlled by the **flags** value in the RSA key itself, not by the **flags** value in the RSA_METHOD attached to the RSA key (which is what this function returns). If the flags element of an RSA key is changed, the changes will be honoured by RSA functionality but will not be reflected in the return value of the RSA_flags() function - in effect RSA_flags() behaves more like an RSA_default_flags() function (which does not currently exist).

SEE ALSO

rsa(3), RSA_new(3)

HISTORY

RSA_new_method() and RSA_set_default_method() appeared in SSLeay 0.8. RSA_get_default_method(), RSA_set_method() and RSA_get_method() as well as the rsa_sign and rsa_verify components of RSA_METHOD were added in OpenSSL 0.9.4.

RSA_set_default_openssl_method() and RSA_get_default_openssl_method() replaced RSA_set_default_method() and RSA_get_default_method() respectively, and RSA_set_method() and RSA_new_method() were altered to use **ENGINE**s rather than **RSA_METHOD**s during development of the engine version of OpenSSL 0.9.6. For 0.9.7, the handling of defaults in the ENGINE API was restructured so that this change was reversed, and behaviour of the other functions resembled more closely the previous behaviour. The behaviour of defaults in the ENGINE API now transparently overrides the behaviour of defaults in the RSA API without requiring changing these function prototypes.

Name

RSA_sign_ASN1_OCTET_STRING and RSA_verify_ASN1_OCTET_STRING — RSA signatures

Synopsis

```
#include  <openssl/rsa.h>
```

```
int RSA_sign_ASN1_OCTET_STRING(int dummy, unsigned char *m,
   unsigned int m_len, unsigned char *sigret, unsigned int *siglen,
   RSA *rsa);
```

```
int RSA_verify_ASN1_OCTET_STRING(int dummy, unsigned char *m,
   unsigned int m_len, unsigned char *sigbuf, unsigned int siglen,
   RSA *rsa);
```

DESCRIPTION

RSA_sign_ASN1_OCTET_STRING() signs the octet string **m** of size **m_len** using the private key **rsa** represented in DER using PKCS #1 padding. It stores the signature in **sigret** and the signature size in **siglen**. **sigret** must point to **RSA_size(rsa)** bytes of memory.

dummy is ignored.

The random number generator must be seeded prior to calling RSA_sign_ASN1_OCTET_STRING().

RSA_verify_ASN1_OCTET_STRING() verifies that the signature **sigbuf** of size **siglen** is the DER representation of a given octet string **m** of size **m_len**. **dummy** is ignored. **rsa** is the signer's public key.

RETURN VALUES

RSA_sign_ASN1_OCTET_STRING() returns 1 on success, 0 otherwise. RSA_verify_ASN1_OCTET_STRING() returns 1 on successful verification, 0 otherwise.

The error codes can be obtained by ERR_get_error(3).

BUGS

These functions serve no recognizable purpose.

SEE ALSO

ERR_get_error(3), objects(3), rand(3), rsa(3), RSA_sign(3), RSA_verify(3)

HISTORY

RSA_sign_ASN1_OCTET_STRING() and RSA_verify_ASN1_OCTET_STRING() were added in SSLeay 0.8.

Name

RSA_sign and RSA_verify — RSA signatures

Synopsis

```
#include  <openssl/rsa.h>
```

```
int RSA_sign(int type, const unsigned char *m, unsigned int m_len,
   unsigned char *sigret, unsigned int *siglen, RSA *rsa);
```

```
int RSA_verify(int type, const unsigned char *m, unsigned int m_len,
   unsigned char *sigbuf, unsigned int siglen, RSA *rsa);
```

DESCRIPTION

RSA_sign() signs the message digest **m** of size **m_len** using the private key **rsa** as specified in PKCS #1 v2.0. It stores the signature in **sigret** and the signature size in **siglen**. **sigret** must point to RSA_size(**rsa**) bytes of memory. Note that PKCS #1 adds meta-data, placing limits on the size of the key that can be used. See RSA_private_encrypt(3) for lower-level operations.

type denotes the message digest algorithm that was used to generate **m**. It usually is one of **NID_sha1**, **NID_ripemd160** and **NID_md5**; see objects(3) for details. If **type** is **NID_md5_sha1**, an SSL signature (MD5 and SHA1 message digests with PKCS #1 padding and no algorithm identifier) is created.

RSA_verify() verifies that the signature **sigbuf** of size **siglen** matches a given message digest **m** of size **m_len**. **type** denotes the message digest algorithm that was used to generate the signature. **rsa** is the signer's public key.

RETURN VALUES

RSA_sign() returns 1 on success, 0 otherwise. RSA_verify() returns 1 on successful verification, 0 otherwise.

The error codes can be obtained by ERR_get_error(3).

BUGS

Certain signatures with an improper algorithm identifier are accepted for compatibility with SSLeay 0.4.5 :-)

CONFORMING TO

SSL, PKCS #1 v2.0

SEE ALSO

ERR_get_error(3), objects(3), rsa(3), RSA_private_encrypt(3), RSA_public_decrypt(3)

HISTORY

RSA_sign() and RSA_verify() are available in all versions of SSLeay and OpenSSL.

Name

RSA_size — get RSA modulus size

Synopsis

```
#include  <openssl/rsa.h>
```

```
int RSA_size(const RSA *rsa);
```

DESCRIPTION

This function returns the RSA modulus size in bytes. It can be used to determine how much memory must be allocated for an RSA encrypted value.

rsa->n must not be **NULL**.

RETURN VALUE

The size in bytes.

SEE ALSO

rsa(3)

HISTORY

RSA_size() is available in all versions of SSLeay and OpenSSL.

Name

SHA1, SHA1_Init, SHA1_Update and SHA1_Final — Secure Hash Algorithm

Synopsis

```
#include  <openssl/sha.h>
```

```
unsigned char *SHA1(const unsigned char *d, unsigned long n,
                 unsigned char *md);
```

```
int SHA1_Init(SHA_CTX *c);
int SHA1_Update(SHA_CTX *c, const void *data,
                 unsigned long len);
int SHA1_Final(unsigned char *md, SHA_CTX *c);
```

DESCRIPTION

SHA-1 (Secure Hash Algorithm) is a cryptographic hash function with a 160 bit output.

SHA1() computes the SHA-1 message digest of the **n** bytes at **d** and places it in **md** (which must have space for SHA_DIGEST_LENGTH == 20 bytes of output). If **md** is NULL, the digest is placed in a static array.

The following functions may be used if the message is not completely stored in memory:

SHA1_Init() initializes a **SHA_CTX** structure.

SHA1_Update() can be called repeatedly with chunks of the message to be hashed (**len** bytes at **data**).

SHA1_Final() places the message digest in **md**, which must have space for SHA_DIGEST_LENGTH == 20 bytes of output, and erases the **SHA_CTX**.

Applications should use the higher level functions EVP_DigestInit(3) etc. instead of calling the hash functions directly.

The predecessor of SHA-1, SHA, is also implemented, but it should be used only when backward compatibility is required.

RETURN VALUES

SHA1() returns a pointer to the hash value.

SHA1_Init(), SHA1_Update() and SHA1_Final() return 1 for success, 0 otherwise.

CONFORMING TO

SHA: US Federal Information Processing Standard FIPS PUB 180 (Secure Hash Standard), SHA-1: US Federal Information Processing Standard FIPS PUB 180-1 (Secure Hash Standard), ANSI X9.30

SEE ALSO

ripemd(3), hmac(3), EVP_DigestInit(3)

HISTORY

SHA1(), SHA1_Init(), SHA1_Update() and SHA1_Final() are available in all versions of SSLeay and OpenSSL.

Name

SMIME_read_CMS — parse S/MIME message.

Synopsis

```
#include  <openssl/cms.h>
```

```
CMS_ContentInfo *SMIME_read_CMS(BIO *in, BIO **bcont);
```

DESCRIPTION

SMIME_read_CMS() parses a message in S/MIME format.

in is a BIO to read the message from.

If cleartext signing is used then the content is saved in a memory bio which is written to ***bcont**, otherwise ***bcont** is set to NULL.

The parsed CMS_ContentInfo structure is returned or NULL if an error occurred.

NOTES

If ***bcont** is not NULL then the message is clear text signed. ***bcont** can then be passed to CMS_verify() with the **CMS_DETACHED** flag set.

Otherwise the type of the returned structure can be determined using CMS_get0_type().

To support future functionality if **bcont** is not NULL ***bcont** should be initialized to NULL. For example:

```
BIO *cont = NULL;
CMS_ContentInfo *cms;

cms = SMIME_read_CMS(in, &cont);
```

BUGS

The MIME parser used by SMIME_read_CMS() is somewhat primitive. While it will handle most S/MIME messages more complex compound formats may not work.

The parser assumes that the CMS_ContentInfo structure is always base64 encoded and will not handle the case where it is in binary format or uses quoted printable format.

The use of a memory BIO to hold the signed content limits the size of message which can be processed due to memory restraints: a streaming single pass option should be available.

RETURN VALUES

SMIME_read_CMS() returns a valid **CMS_ContentInfo** structure or **NULL** if an error occurred. The error can be obtained from ERR_get_error(3).

SEE ALSO

ERR_get_error(3), CMS_type(3), SMIME_read_CMS(3), CMS_sign(3), CMS_verify(3), CMS_encrypt(3), CMS_decrypt(3)

HISTORY

SMIME_read_CMS() was added to OpenSSL 0.9.8

Name

SMIME_read_PKCS7 — parse S/MIME message.

Synopsis

```
#include  <openssl/pkcs7.h>
```

```
PKCS7 *SMIME_read_PKCS7(BIO *in, BIO **bcont);
```

DESCRIPTION

SMIME_read_PKCS7() parses a message in S/MIME format.

in is a BIO to read the message from.

If cleartext signing is used then the content is saved in a memory bio which is written to ***bcont**, otherwise ***bcont** is set to **NULL**.

The parsed PKCS#7 structure is returned or **NULL** if an error occurred.

NOTES

If ***bcont** is not **NULL** then the message is clear text signed. ***bcont** can then be passed to PKCS7_verify() with the **PKCS7_DETACHED** flag set.

Otherwise the type of the returned structure can be determined using PKCS7_type().

To support future functionality if **bcont** is not **NULL** ***bcont** should be initialized to **NULL**. For example:

```
BIO *cont = NULL;
PKCS7 *p7;

p7 = SMIME_read_PKCS7(in, &cont);
```

BUGS

The MIME parser used by SMIME_read_PKCS7() is somewhat primitive. While it will handle most S/MIME messages more complex compound formats may not work.

The parser assumes that the PKCS7 structure is always base64 encoded and will not handle the case where it is in binary format or uses quoted printable format.

The use of a memory BIO to hold the signed content limits the size of message which can be processed due to memory restraints: a streaming single pass option should be available.

RETURN VALUES

SMIME_read_PKCS7() returns a valid **PKCS7** structure or **NULL** is an error occurred. The error can be obtained from ERR_get_error(3).

SEE ALSO

ERR_get_error(3), PKCS7_type(3), SMIME_read_PKCS7(3), PKCS7_sign(3), PKCS7_verify(3), PKCS7_encrypt(3), PKCS7_decrypt(3)

HISTORY

SMIME_read_PKCS7() was added to OpenSSL 0.9.5

Name

SMIME_write_CMS — convert CMS structure to S/MIME format.

Synopsis

```
#include  <openssl/cms.h>
```

```
int SMIME_write_CMS(BIO *out, CMS_ContentInfo *cms, BIO *data, int flags);
```

DESCRIPTION

SMIME_write_CMS() adds the appropriate MIME headers to a CMS structure to produce an S/MIME message.

out is the BIO to write the data to. **cms** is the appropriate **CMS_ContentInfo** structure. If streaming is enabled then the content must be supplied in the **data** argument. **flags** is an optional set of flags.

NOTES

The following flags can be passed in the **flags** parameter.

If **CMS_DETACHED** is set then cleartext signing will be used, this option only makes sense for SignedData where **CMS_DETACHED** is also set when CMS_sign() is called.

If the **CMS_TEXT** flag is set MIME headers for type **text/plain** are added to the content, this only makes sense if **CMS_DETACHED** is also set.

If the **CMS_STREAM** flag is set streaming is performed. This flag should only be set if **CMS_STREAM** was also set in the previous call to a CMS_ContentInfo creation function.

If cleartext signing is being used and **CMS_STREAM** not set then the data must be read twice: once to compute the signature in CMS_sign() and once to output the S/MIME message.

If streaming is performed the content is output in BER format using indefinite length constructed encoding except in the case of signed data with detached content where the content is absent and DER format is used.

BUGS

SMIME_write_CMS() always base64 encodes CMS structures, there should be an option to disable this.

RETURN VALUES

SMIME_write_CMS() returns 1 for success or 0 for failure.

SEE ALSO

ERR_get_error(3), CMS_sign(3), CMS_verify(3), CMS_encrypt(3), CMS_decrypt(3)

HISTORY

SMIME_write_CMS() was added to OpenSSL 0.9.8

Name

SMIME_write_PKCS7 — convert PKCS#7 structure to S/MIME format.

Synopsis

```
#include  <openssl/pkcs7.h>
```

```
int SMIME_write_PKCS7(BIO *out, PKCS7 *p7, BIO *data, int flags);
```

DESCRIPTION

SMIME_write_PKCS7() adds the appropriate MIME headers to a PKCS#7 structure to produce an S/MIME message.

out is the BIO to write the data to. **p7** is the appropriate **PKCS7** structure. If streaming is enabled then the content must be supplied in the **data** argument. **flags** is an optional set of flags.

NOTES

The following flags can be passed in the **flags** parameter.

If **PKCS7_DETACHED** is set then cleartext signing will be used, this option only makes sense for signedData where **PKCS7_DETACHED** is also set when PKCS7_sign() is also called.

If the **PKCS7_TEXT** flag is set MIME headers for type **text/plain** are added to the content, this only makes sense if **PKCS7_DETACHED** is also set.

If the **PKCS7_STREAM** flag is set streaming is performed. This flag should only be set if **PKCS7_STREAM** was also set in the previous call to PKCS7_sign() or **PKCS7_encrypt()**.

If cleartext signing is being used and **PKCS7_STREAM** not set then the data must be read twice: once to compute the signature in PKCS7_sign() and once to output the S/MIME message.

If streaming is performed the content is output in BER format using indefinite length constructuted encoding except in the case of signed data with detached content where the content is absent and DER format is used.

BUGS

SMIME_write_PKCS7() always base64 encodes PKCS#7 structures, there should be an option to disable this.

RETURN VALUES

SMIME_write_PKCS7() returns 1 for success or 0 for failure.

SEE ALSO

ERR_get_error(3), PKCS7_sign(3), PKCS7_verify(3), PKCS7_encrypt(3), PKCS7_decrypt(3)

HISTORY

SMIME_write_PKCS7() was added to OpenSSL 0.9.5

Name

CRYPTO_THREADID_set_callback, CRYPTO_THREADID_get_callback, CRYPTO_THREADID_current, CRYPTO_THREADID_cmp, CRYPTO_THREADID_cpy, CRYPTO_THREADID_hash, CRYPTO_set_locking_callback, CRYPTO_num_locks, CRYPTO_set_dynlock_create_callback, CRYPTO_set_dynlock_lock_callback, CRYPTO_set_dynlock_destroy_callback, CRYPTO_get_new_dynlockid, CRYPTO_destroy_dynlockid and CRYPTO_lock — OpenSSL thread support

Synopsis

```
#include  <openssl/crypto.h>
```

```
/* Don't use this structure directly. */
typedef struct crypto_threadid_st
        {
        void *ptr;
        unsigned long val;
        } CRYPTO_THREADID;
/* Only use CRYPTO_THREADID_set_[numeric|pointer]() within callbacks */
void CRYPTO_THREADID_set_numeric(CRYPTO_THREADID *id, unsigned long val);
void CRYPTO_THREADID_set_pointer(CRYPTO_THREADID *id, void *ptr);
int CRYPTO_THREADID_set_callback(void (*threadid_func)(CRYPTO_THREADID *));
void (*CRYPTO_THREADID_get_callback(void))(CRYPTO_THREADID *);
void CRYPTO_THREADID_current(CRYPTO_THREADID *id);
int CRYPTO_THREADID_cmp(const CRYPTO_THREADID *a,
                        const CRYPTO_THREADID *b);
void CRYPTO_THREADID_cpy(CRYPTO_THREADID *dest,
                         const CRYPTO_THREADID *src);
unsigned long CRYPTO_THREADID_hash(const CRYPTO_THREADID *id);
```

```
int CRYPTO_num_locks(void);
```

```
/* struct CRYPTO_dynlock_value needs to be defined by the user */
struct CRYPTO_dynlock_value;
```

```
void CRYPTO_set_dynlock_create_callback(struct CRYPTO_dynlock_value *
       (*dyn_create_function)(char *file, int line));
void CRYPTO_set_dynlock_lock_callback(void (*dyn_lock_function)
       (int mode, struct CRYPTO_dynlock_value *l,
       const char *file, int line));
void CRYPTO_set_dynlock_destroy_callback(void (*dyn_destroy_function)
       (struct CRYPTO_dynlock_value *l, const char *file, int line));
```

```
int CRYPTO_get_new_dynlockid(void);
```

```
void CRYPTO_destroy_dynlockid(int i);
```

```
void CRYPTO_lock(int mode, int n, const char *file, int line);
```

```
#define CRYPTO_w_lock(type)    \
       CRYPTO_lock(CRYPTO_LOCK|CRYPTO_WRITE,type,__FILE__,__LINE__)
#define CRYPTO_w_unlock(type)  \
       CRYPTO_lock(CRYPTO_UNLOCK|CRYPTO_WRITE,type,__FILE__,__LINE__)
#define CRYPTO_r_lock(type)    \
       CRYPTO_lock(CRYPTO_LOCK|CRYPTO_READ,type,__FILE__,__LINE__)
#define CRYPTO_r_unlock(type)  \
       CRYPTO_lock(CRYPTO_UNLOCK|CRYPTO_READ,type,__FILE__,__LINE__)
#define CRYPTO_add(addr,amount,type)   \
       CRYPTO_add_lock(addr,amount,type,__FILE__,__LINE__)
```

DESCRIPTION

OpenSSL can safely be used in multi-threaded applications provided that at least two callback functions are set, locking_function and threadid_func.

locking_function(int mode, int n, const char *file, int line) is needed to perform locking on shared data structures. (Note that OpenSSL uses a number of global data structures that will be implicitly shared whenever multiple threads use OpenSSL.) Multi-threaded applications will crash at random if it is not set.

locking_function() must be able to handle up to CRYPTO_num_locks() different mutex locks. It sets the **n**-th lock if **mode** & **CRYPTO_LOCK**, and releases it otherwise.

file and **line** are the file number of the function setting the lock. They can be useful for debugging.

threadid_func(CRYPTO_THREADID *id) is needed to record the currently-executing thread's identifier into **id**. The implementation of this callback should not fill in **id** directly, but should use CRYPTO_THREADID_set_numeric() if thread IDs are numeric, or CRYPTO_THREADID_set_pointer() if they are pointer-based. If the application does not register such a callback using CRYPTO_THREADID_set_callback(), then a default implementation is used - on Windows and BeOS this uses the system's default thread identifying APIs, and on all other platforms it uses the address of **errno**. The latter is satisfactory for thread-safety if and only if the platform has a thread-local error number facility.

Once threadid_func() is registered, or if the built-in default implementation is to be used;

- CRYPTO_THREADID_current() records the currently-executing thread ID into the given **id** object.
- CRYPTO_THREADID_cmp() compares two thread IDs (returning zero for equality, ie. the same semantics as memcmp()).
- CRYPTO_THREADID_cpy() duplicates a thread ID value,
- CRYPTO_THREADID_hash() returns a numeric value usable as a hash-table key. This is usually the exact numeric or pointer-based thread ID used internally, however this also handles the unusual case where pointers are larger than 'long' variables and the platform's thread IDs are pointer-based - in this case, mixing is done to attempt to produce a unique numeric value even though it is not as wide as the platform's true thread IDs.

Additionally, OpenSSL supports dynamic locks, and sometimes, some parts of OpenSSL need it for better performance. To enable this, the following is required:

- Three additional callback function, dyn_create_function, dyn_lock_function and dyn_destroy_function.
- A structure defined with the data that each lock needs to handle.

struct CRYPTO_dynlock_value has to be defined to contain whatever structure is needed to handle locks.

dyn_create_function(const char *file, int line) is needed to create a lock. Multi-threaded applications might crash at random if it is not set.

dyn_lock_function(int mode, CRYPTO_dynlock *l, const char *file, int line) is needed to perform locking off dynamic lock numbered n. Multi-threaded applications might crash at random if it is not set.

dyn_destroy_function(CRYPTO_dynlock *l, const char *file, int line) is needed to destroy the lock l. Multi-threaded applications might crash at random if it is not set.

CRYPTO_get_new_dynlockid() is used to create locks. It will call dyn_create_function for the actual creation.

CRYPTO_destroy_dynlockid() is used to destroy locks. It will call dyn_destroy_function for the actual destruction.

CRYPTO_lock() is used to lock and unlock the locks. mode is a bitfield describing what should be done with the lock. n is the number of the lock as returned from CRYPTO_get_new_dynlockid(). mode can be combined from the following values. These values are pairwise exclusive, with undefined behaviour if misused (for example, CRYPTO_READ and CRYPTO_WRITE should not be used together):

```
CRYPTO_LOCK     0x01
CRYPTO_UNLOCK   0x02
```

```
CRYPTO_READ     0x04
CRYPTO_WRITE    0x08
```

RETURN VALUES

CRYPTO_num_locks() returns the required number of locks.

CRYPTO_get_new_dynlockid() returns the index to the newly created lock.

The other functions return no values.

NOTES

You can find out if OpenSSL was configured with thread support:

```
#define OPENSSL_THREAD_DEFINES
#include <openssl/opensslconf.h>
#if defined(OPENSSL_THREADS)
  // thread support enabled
#else
  // no thread support
#endif
```

Also, dynamic locks are currently not used internally by OpenSSL, but may do so in the future.

EXAMPLES

crypto/threads/mttest.c shows examples of the callback functions on Solaris, Irix and Win32.

HISTORY

CRYPTO_set_locking_callback() is available in all versions of SSLeay and OpenSSL. CRYPTO_num_locks() was added in OpenSSL 0.9.4. All functions dealing with dynamic locks were added in OpenSSL 0.9.5b-dev. **CRYPTO_THREADID** and associated functions were introduced in OpenSSL 1.0.0 to replace (actually, deprecate) the previous CRYPTO_set_id_callback(), CRYPTO_get_id_callback(), and CRYPTO_thread_id() functions which assumed thread IDs to always be represented by 'unsigned long'.

SEE ALSO

crypto(3)

Name

des_read_password, des_read_2passwords, des_read_pw_string and des_read_pw — Compatibility user interface functions

Synopsis

```
#include  <openssl/des_old.h>
```

```
int des_read_password(DES_cblock *key,const char *prompt,int verify);
int des_read_2passwords(DES_cblock *key1,DES_cblock *key2,
       const char *prompt,int verify);
```

```
int des_read_pw_string(char *buf,int length,const char *prompt,int verify);
int des_read_pw(char *buf,char *buff,int size,const char *prompt,int verify);
```

DESCRIPTION

The DES library contained a few routines to prompt for passwords. These aren't necessarely dependent on DES, and have therefore become part of the UI compatibility library.

des_read_pw() writes the string specified by *prompt* to standard output turns echo off and reads an input string from the terminal. The string is returned in *buf*, which must have spac for at least *size* bytes. If *verify* is set, the user is asked for the password twice and unless the two copies match, an error is returned. The second password is stored in *buff*, which must therefore also be at least *size* bytes. A return code of -1 indicates a system error, 1 failure due to use interaction, and 0 is success. All other functions described here use des_read_pw() to do the work.

des_read_pw_string() is a variant of des_read_pw() that provides a buffer for you if *verify* is set.

des_read_password() calls des_read_pw() and converts the password to a DES key by calling DES_string_to_key(); des_read_2password() operates in the same way as des_read_password() except that it generates two keys by using the DES_string_to_2key() function.

NOTES

des_read_pw_string() is available in the MIT Kerberos library as well, and is also available under the name EVP_read_pw_string().

SEE ALSO

ui(3), ui_create(3)

AUTHOR

Richard Levitte (richard@levitte.org) for the OpenSSL project (http://www.openssl.org).

Name

UI_new, UI_new_method, UI_free, UI_add_input_string, UI_dup_input_string, UI_add_verify_string, UI_dup_verify_string, UI_add_input_boolean, UI_dup_input_boolean, UI_add_info_string, UI_dup_info_string, UI_add_error_string, UI_dup_error_string, UI_construct_prompt, UI_add_user_data, UI_get0_user_data, UI_get0_result, UI_process, UI_ctrl, UI_set_default_method, UI_get_default_method, UI_get_method, UI_set_method, UI_OpenSSL and ERR_load_UI_strings — New User Interface

Synopsis

```
#include  <openssl/ui.h>

typedef struct ui_st UI;
typedef struct ui_method_st UI_METHOD;

UI *UI_new(void);
UI *UI_new_method(const UI_METHOD *method);
void UI_free(UI *ui);

int UI_add_input_string(UI *ui, const char *prompt, int flags,
       char *result_buf, int minsize, int maxsize);
int UI_dup_input_string(UI *ui, const char *prompt, int flags,
       char *result_buf, int minsize, int maxsize);
int UI_add_verify_string(UI *ui, const char *prompt, int flags,
       char *result_buf, int minsize, int maxsize, const char *test_buf);
int UI_dup_verify_string(UI *ui, const char *prompt, int flags,
       char *result_buf, int minsize, int maxsize, const char *test_buf);
int UI_add_input_boolean(UI *ui, const char *prompt, const char *action_desc,
       const char *ok_chars, const char *cancel_chars,
       int flags, char *result_buf);
int UI_dup_input_boolean(UI *ui, const char *prompt, const char *action_desc,
       const char *ok_chars, const char *cancel_chars,
       int flags, char *result_buf);
int UI_add_info_string(UI *ui, const char *text);
int UI_dup_info_string(UI *ui, const char *text);
int UI_add_error_string(UI *ui, const char *text);
int UI_dup_error_string(UI *ui, const char *text);

/* These are the possible flags.  They can be or'ed together. */
#define UI_INPUT_FLAG_ECHO             0x01
#define UI_INPUT_FLAG_DEFAULT_PWD      0x02

char *UI_construct_prompt(UI *ui_method,
       const char *object_desc, const char *object_name);

void *UI_add_user_data(UI *ui, void *user_data);
void *UI_get0_user_data(UI *ui);

const char *UI_get0_result(UI *ui, int i);

int UI_process(UI *ui);

int UI_ctrl(UI *ui, int cmd, long i, void *p, void (*f)());
#define UI_CTRL_PRINT_ERRORS           1
#define UI_CTRL_IS_REDOABLE            2

void UI_set_default_method(const UI_METHOD *meth);
const UI_METHOD *UI_get_default_method(void);
const UI_METHOD *UI_get_method(UI *ui);
const UI_METHOD *UI_set_method(UI *ui, const UI_METHOD *meth);

UI_METHOD *UI_OpenSSL(void);
```

DESCRIPTION

UI stands for User Interface, and is general purpose set of routines to prompt the user for text-based information. Through user-written methods (see ui_create(3)), prompting can be done in any way imaginable, be it plain text prompting, through dialog boxes or from a cell phone.

All the functions work through a context of the type UI. This context contains all the information needed to prompt correctly as well as a reference to a UI_METHOD, which is an ordered vector of functions that carry out the actual prompting.

The first thing to do is to create a UI with UI_new() or UI_new_method(), then add information to it with the UI_add or UI_dup functions. Also, user-defined random data can be passed down to the underlying method through calls to UI_add_user_data. The default UI method doesn't care about these data, but other methods might. Finally, use UI_process() to actually perform the prompting and UI_get0_result() to find the result to the prompt.

A UI can contain more than one prompt, which are performed in the given sequence. Each prompt gets an index number which is returned by the UI_add and UI_dup functions, and has to be used to get the corresponding result with UI_get0_result().

The functions are as follows:

UI_new() creates a new UI using the default UI method. When done with this UI, it should be freed using UI_free().

UI_new_method() creates a new UI using the given UI method. When done with this UI, it should be freed using UI_free().

UI_OpenSSL() returns the built-in UI method (note: not the default one, since the default can be changed. See further on). This method is the most machine/OS dependent part of OpenSSL and normally generates the most problems when porting.

UI_free() removes a UI from memory, along with all other pieces of memory that's connected to it, like duplicated input strings, results and others.

UI_add_input_string() and UI_add_verify_string() add a prompt to the UI, as well as flags and a result buffer and the desired minimum and maximum sizes of the result. The given information is used to prompt for information, for example a password, and to verify a password (i.e. having the user enter it twice and check that the same string was entered twice). UI_add_verify_string() takes and extra argument that should be a pointer to the result buffer of the input string that it's supposed to verify, or verification will fail.

UI_add_input_boolean() adds a prompt to the UI that's supposed to be answered in a boolean way, with a single character for yes and a different character for no. A set of characters that can be used to cancel the prompt is given as well. The prompt itself is divided in two, one part being the descriptive text (given through the *prompt* argument) and one describing the possible answers (given through the *action_desc* argument).

UI_add_info_string() and UI_add_error_string() add strings that are shown at the same time as the prompt for extra information or to show an error string. The difference between the two is only conceptual. With the builtin method, there's no technical difference between them. Other methods may make a difference between them, however.

The flags currently supported are UI_INPUT_FLAG_ECHO, which is relevant for UI_add_input_string() and will have the users response be echoed (when prompting for a password, this flag should obviously not be used, and UI_INPUT_FLAG_DEFAULT_PWD, which means that a default password of some sort will be used (completely depending on the application and the UI method).

UI_dup_input_string(), UI_dup_verify_string(), UI_dup_input_boolean(), UI_dup_info_string() and UI_dup_error_string() are basically the same as their UI_add counterparts, except that they make their own copies of all strings.

UI_construct_prompt() is a helper function that can be used to create a prompt from two pieces of information: an description and a name. The default constructor (if there is none provided by the method used) creates a string "Enter *description* for *name*:". With the description "pass phrase" and the file name "foo.key", that becomes "Enter pass phrase for foo.key:". Other methods may create whatever string and may include encodings that will be processed by the other method functions.

UI_add_user_data() adds a piece of memory for the method to use at any time. The builtin UI method doesn't care about this info. Note that several calls to this function doesn't add data, it replaces the previous blob with the one given as argument.

UI_get0_user_data() retrieves the data that has last been given to the UI with UI_add_user_data().

UI_get0_result() returns a pointer to the result buffer associated with the information indexed by *i*.

UI_process() goes through the information given so far, does all the printing and prompting and returns.

UI_ctrl() adds extra control for the application author. For now, it understands two commands: UI_CTRL_PRINT_ERRORS, which makes UI_process() print the OpenSSL error stack as part of processing the UI, and UI_CTRL_IS_REDOABLE, which returns a flag saying if the used UI can be used again or not.

UI_set_default_method() changes the default UI method to the one given.

UI_get_default_method() returns a pointer to the current default UI method.

UI_get_method() returns the UI method associated with a given UI.

UI_set_method() changes the UI method associated with a given UI.

SEE ALSO

ui_create(3), ui_compat(3)

HISTORY

The UI section was first introduced in OpenSSL 0.9.7.

AUTHOR

Richard Levitte (richard@levitte.org) for the OpenSSL project (http://www.openssl.org).

Name

x509 — X.509 certificate handling

Synopsis

```
#include  <openssl/x509.h>
```

DESCRIPTION

A X.509 certificate is a structured grouping of information about an individual, a device, or anything one can imagine. A X.509 CRL (certificate revocation list) is a tool to help determine if a certificate is still valid. The exact definition of those can be found in the X.509 document from ITU-T, or in RFC3280 from PKIX. In OpenSSL, the type X509 is used to express such a certificate, and the type X509_CRL is used to express a CRL.

A related structure is a certificate request, defined in PKCS#10 from RSA Security, Inc, also reflected in RFC2896. In OpenSSL, the type X509_REQ is used to express such a certificate request.

To handle some complex parts of a certificate, there are the types X509_NAME (to express a certificate name), X509_ATTRIBUTE (to express a certificate attributes), X509_EXTENSION (to express a certificate extension) and a few more.

Finally, there's the supertype X509_INFO, which can contain a CRL, a certificate and a corresponding private key.

X509_..., **d2i_X509_**... and **i2d_X509_**... handle X.509 certificates, with some exceptions, shown below.

X509_CRL_..., **d2i_X509_CRL_**... and **i2d_X509_CRL_**... handle X.509 CRLs.

X509_REQ_..., **d2i_X509_REQ_**... and **i2d_X509_REQ_**... handle PKCS#10 certificate requests.

X509_NAME_... handle certificate names.

X509_ATTRIBUTE_... handle certificate attributes.

X509_EXTENSION_... handle certificate extensions.

SEE ALSO

X509_NAME_ENTRY_get_object(3), X509_NAME_add_entry_by_txt(3), X509_NAME_add_entry_by_NID(3), X509_NAME_print_ex(3), X509_NAME_new(3), d2i_X509(3), d2i_X509_ALGOR(3), d2i_X509_CRL(3), d2i_X509_NAME(3), d2i_X509_REQ(3), d2i_X509_SIG(3), crypto(3), x509v3(3)

Name

X509_NAME_add_entry_by_txt, X509_NAME_add_entry_by_OBJ, X509_NAME_add_entry_by_NID, X509_NAME_add_entry and X509_NAME_delete_entry — X509_NAME modification functions

Synopsis

```
#include  <openssl/x509.h>
```

```
int X509_NAME_add_entry_by_txt(X509_NAME *name, const char *field, int type, const unsigned char *bytes,
    int len, int loc, int set);
```

```
int X509_NAME_add_entry_by_OBJ(X509_NAME *name, ASN1_OBJECT *obj, int type, unsigned char *bytes,
    int len, int loc, int set);
```

```
int X509_NAME_add_entry_by_NID(X509_NAME *name, int nid, int type, unsigned char *bytes, int len,
    int loc, int set);
```

```
int X509_NAME_add_entry(X509_NAME *name,X509_NAME_ENTRY *ne, int loc, int set);
```

```
X509_NAME_ENTRY *X509_NAME_delete_entry(X509_NAME *name, int loc);
```

DESCRIPTION

X509_NAME_add_entry_by_txt(), X509_NAME_add_entry_by_OBJ() and X509_NAME_add_entry_by_NID() add a field whose name is defined by a string **field**, an object **obj** or a NID **nid** respectively. The field value to be added is in **bytes** of length **len**. If **len** is -1 then the field length is calculated internally using strlen(bytes).

The type of field is determined by **type** which can either be a definition of the type of **bytes** (such as **MBSTRING_ASC**) or a standard ASN1 type (such as **V_ASN1_IA5STRING**). The new entry is added to a position determined by **loc** and **set**.

X509_NAME_add_entry() adds a copy of **X509_NAME_ENTRY** structure **ne** to **name**. The new entry is added to a position determined by **loc** and **set**. Since a copy of **ne** is added **ne** must be freed up after the call.

X509_NAME_delete_entry() deletes an entry from **name** at position **loc**. The deleted entry is returned and must be freed up.

NOTES

The use of string types such as **MBSTRING_ASC** or **MBSTRING_UTF8** is strongly recommened for the **type** parameter. This allows the internal code to correctly determine the type of the field and to apply length checks according to the relevant standards. This is done using ASN1_STRING_set_by_NID().

If instead an ASN1 type is used no checks are performed and the supplied data in **bytes** is used directly.

In X509_NAME_add_entry_by_txt() the **field** string represents the field name using OBJ_txt2obj(field, 0).

The **loc** and **set** parameters determine where a new entry should be added. For almost all applications **loc** can be set to -1 and **set** to 0. This adds a new entry to the end of **name** as a single valued RelativeDistinguishedName (RDN).

loc actually determines the index where the new entry is inserted: if it is -1 it is appended.

set determines how the new type is added. If it is zero a new RDN is created.

If **set** is -1 or 1 it is added to the previous or next RDN structure respectively. This will then be a multivalued RDN: since multivalues RDNs are very seldom used **set** is almost always set to zero.

EXAMPLES

Create an **X509_NAME** structure:

"C=UK, O=Disorganized Organization, CN=Joe Bloggs"

```
X509_NAME *nm;
nm = X509_NAME_new();
if (nm == NULL)
       /* Some error */
if (!X509_NAME_add_entry_by_txt(nm, "C", MBSTRING_ASC,
                       "UK", -1, -1, 0))
       /* Error */
if (!X509_NAME_add_entry_by_txt(nm, "O", MBSTRING_ASC,
                       "Disorganized Organization", -1, -1, 0))
       /* Error */
if (!X509_NAME_add_entry_by_txt(nm, "CN", MBSTRING_ASC,
                       "Joe Bloggs", -1, -1, 0))
       /* Error */
```

RETURN VALUES

X509_NAME_add_entry_by_txt(), X509_NAME_add_entry_by_OBJ(), X509_NAME_add_entry_by_NID() and X509_NAME_add_entry() return 1 for success of 0 if an error occurred.

X509_NAME_delete_entry() returns either the deleted **X509_NAME_ENTRY** structure of **NULL** if an error occurred.

BUGS

type can still be set to **V_ASN1_APP_CHOOSE** to use a different algorithm to determine field types. Since this form does not understand multicharacter types, performs no length checks and can result in invalid field types its use is strongly discouraged.

SEE ALSO

ERR_get_error(3), d2i_X509_NAME(3)

HISTORY

Name

X509_NAME_ENTRY_get_object, X509_NAME_ENTRY_get_data, X509_NAME_ENTRY_set_object, X509_NAME_ENTRY_set_data, X509_NAME_ENTRY_create_by_txt, X509_NAME_ENTRY_create_by_NID and X509_NAME_ENTRY_create_by_OBJ — X509_NAME_ENTRY utility functions

Synopsis

```
#include  <openssl/x509.h>
```

```
ASN1_OBJECT * X509_NAME_ENTRY_get_object(X509_NAME_ENTRY *ne);
ASN1_STRING * X509_NAME_ENTRY_get_data(X509_NAME_ENTRY *ne);
```

```
int X509_NAME_ENTRY_set_object(X509_NAME_ENTRY *ne, ASN1_OBJECT *obj);
int X509_NAME_ENTRY_set_data(X509_NAME_ENTRY *ne, int type, const unsigned char *bytes, int len);
```

```
X509_NAME_ENTRY *X509_NAME_ENTRY_create_by_txt(X509_NAME_ENTRY **ne, const char *field, int type,
    const unsigned char *bytes, int len);
X509_NAME_ENTRY *X509_NAME_ENTRY_create_by_NID(X509_NAME_ENTRY **ne, int nid, int type,
    unsigned char *bytes, int len);
X509_NAME_ENTRY *X509_NAME_ENTRY_create_by_OBJ(X509_NAME_ENTRY **ne, ASN1_OBJECT *obj,
    int type, const unsigned char *bytes, int len);
```

DESCRIPTION

X509_NAME_ENTRY_get_object() retrieves the field name of **ne** in and **ASN1_OBJECT** structure.

X509_NAME_ENTRY_get_data() retrieves the field value of **ne** in and **ASN1_STRING** structure.

X509_NAME_ENTRY_set_object() sets the field name of **ne** to **obj**.

X509_NAME_ENTRY_set_data() sets the field value of **ne** to string type **type** and value determined by **bytes** and **len**.

X509_NAME_ENTRY_create_by_txt(), X509_NAME_ENTRY_create_by_NID() and X509_NAME_ENTRY_create_by_OBJ() create and return an **X509_NAME_ENTRY** structure.

NOTES

X509_NAME_ENTRY_get_object() and X509_NAME_ENTRY_get_data() can be used to examine an **X509_NAME_ENTRY** function as returned by X509_NAME_get_entry() for example.

X509_NAME_ENTRY_create_by_txt(), X509_NAME_ENTRY_create_by_NID(), and X509_NAME_ENTRY_create_by_OBJ() create and return an

X509_NAME_ENTRY_create_by_txt(), X509_NAME_ENTRY_create_by_OBJ(), X509_NAME_ENTRY_create_by_NID() and X509_NAME_ENTRY_set_data() are seldom used in practice because **X509_NAME_ENTRY** structures are almost always part of **X509_NAME** structures and the corresponding **X509_NAME** functions are typically used to create and add new entries in a single operation.

The arguments of these functions support similar options to the similarly named ones of the corresponding **X509_NAME** functions such as X509_NAME_add_entry_by_txt(). So for example **type** can be set to **MBSTRING_ASC** but in the case of X509_set_data() the field name must be set first so the relevant field information can be looked up internally.

RETURN VALUES

SEE ALSO

ERR_get_error(3), d2i_X509_NAME(3), OBJ_nid2obj(3)

HISTORY

TBA

Name

X509_NAME_get_index_by_NID, X509_NAME_get_index_by_OBJ, X509_NAME_get_entry, X509_NAME_entry_count, X509_NAME_get_text_by_NID and X509_NAME_get_text_by_OBJ — X509_NAME lookup and enumeration functions

Synopsis

```
#include  <openssl/x509.h>
```

```
int X509_NAME_get_index_by_NID(X509_NAME *name,int nid,int lastpos);
int X509_NAME_get_index_by_OBJ(X509_NAME *name,ASN1_OBJECT *obj, int lastpos);
```

```
int X509_NAME_entry_count(X509_NAME *name);
X509_NAME_ENTRY *X509_NAME_get_entry(X509_NAME *name, int loc);
```

```
int X509_NAME_get_text_by_NID(X509_NAME *name, int nid, char *buf,int len);
int X509_NAME_get_text_by_OBJ(X509_NAME *name, ASN1_OBJECT *obj, char *buf,int len);
```

DESCRIPTION

These functions allow an **X509_NAME** structure to be examined. The **X509_NAME** structure is the same as the **Name** type defined in RFC2459 (and elsewhere) and used for example in certificate subject and issuer names.

X509_NAME_get_index_by_NID() and X509_NAME_get_index_by_OBJ() retrieve the next index matching **nid** or **obj** after **lastpos**. **lastpos** should initially be set to -1. If there are no more entries -1 is returned. If **nid** is invalid (doesn't correspond to a valid OID) then -2 is returned.

X509_NAME_entry_count() returns the total number of entries in **name**.

X509_NAME_get_entry() retrieves the **X509_NAME_ENTRY** from **name** corresponding to index **loc**. Acceptable values for **loc** run from 0 to (X509_NAME_entry_count(name) - 1). The value returned is an internal pointer which must not be freed.

X509_NAME_get_text_by_NID(), X509_NAME_get_text_by_OBJ() retrieve the "text" from the first entry in **name** which matches **nid** or **obj**, if no such entry exists -1 is returned. At most **len** bytes will be written and the text written to **buf** will be null terminated. The length of the output string written is returned excluding the terminating null. If **buf** is <NULL> then the amount of space needed in **buf** (excluding the final null) is returned.

NOTES

X509_NAME_get_text_by_NID() and X509_NAME_get_text_by_OBJ() are legacy functions which have various limitations which make them of minimal use in practice. They can only find the first matching entry and will copy the contents of the field verbatim: this can be highly confusing if the target is a muticharacter string type like a BMPString or a UTF8String.

For a more general solution X509_NAME_get_index_by_NID() or X509_NAME_get_index_by_OBJ() should be used followed by X509_NAME_get_entry() on any matching indices and then the various **X509_NAME_ENTRY** utility functions on the result.

The list of all relevant **NID_*** and **OBJ_* codes** can be found in the source code header files <openssl/obj_mac.h> and/or <openssl/objects.h>.

Applications which could pass invalid NIDs to X509_NAME_get_index_by_NID() should check for the return value of -2. Alternatively the NID validity can be determined first by checking OBJ_nid2obj(nid) is not NULL.

EXAMPLES

Process all entries:

```
int i;
X509_NAME_ENTRY *e;
```

```
for (i = 0; i < X509_NAME_entry_count(nm); i++)
       {
       e = X509_NAME_get_entry(nm, i);
       /* Do something with e */
       }
```

Process all commonName entries:

```
int loc;
X509_NAME_ENTRY *e;

loc = -1;
for (;;)
       {
       lastpos = X509_NAME_get_index_by_NID(nm, NID_commonName, lastpos);
       if (lastpos == -1)
               break;
       e = X509_NAME_get_entry(nm, lastpos);
       /* Do something with e */
       }
```

RETURN VALUES

X509_NAME_get_index_by_NID() and X509_NAME_get_index_by_OBJ() return the index of the next matching entry or -1 if not found. X509_NAME_get_index_by_NID() can also return -2 if the supplied NID is invalid.

X509_NAME_entry_count() returns the total number of entries.

X509_NAME_get_entry() returns an **X509_NAME** pointer to the requested entry or **NULL** if the index is invalid.

SEE ALSO

ERR_get_error(3), d2i_X509_NAME(3)

HISTORY

TBA

Name

X509_NAME_print_ex, X509_NAME_print_ex_fp, X509_NAME_print and X509_NAME_oneline — X509_NAME printing routines.

Synopsis

```
#include  <openssl/x509.h>
```

```
int X509_NAME_print_ex(BIO *out, X509_NAME *nm, int indent, unsigned long flags);
int X509_NAME_print_ex_fp(FILE *fp, X509_NAME *nm, int indent, unsigned long flags);
char * X509_NAME_oneline(X509_NAME *a,char *buf,int size);
int X509_NAME_print(BIO *bp, X509_NAME *name, int obase);
```

DESCRIPTION

X509_NAME_print_ex() prints a human readable version of **nm** to BIO **out**. Each line (for multiline formats) is indented by **indent** spaces. The output format can be extensively customised by use of the **flags** parameter.

X509_NAME_print_ex_fp() is identical to X509_NAME_print_ex() except the output is written to FILE pointer **fp**.

X509_NAME_oneline() prints an ASCII version of **a** to **buf**. At most **size** bytes will be written. If **buf** is **NULL** then a buffer is dynamically allocated and returned, otherwise **buf** is returned.

X509_NAME_print() prints out **name** to **bp** indenting each line by **obase** characters. Multiple lines are used if the output (including indent) exceeds 80 characters.

NOTES

The functions X509_NAME_oneline() and X509_NAME_print() are legacy functions which produce a non standard output form, they don't handle multi character fields and have various quirks and inconsistencies. Their use is strongly discouraged in new applications.

Although there are a large number of possible flags for most purposes **XN_FLAG_ONELINE**, **XN_FLAG_MULTILINE** or **XN_FLAG_RFC2253** will suffice. As noted on the ASN1_STRING_print_ex(3) manual page for UTF8 terminals the **ASN1_STRFLGS_ESC_MSB** should be unset: so for example **XN_FLAG_ONELINE & ~ASN1_STRFLGS_ESC_MSB** would be used.

The complete set of the flags supported by X509_NAME_print_ex() is listed below.

Several options can be ored together.

The options **XN_FLAG_SEP_COMMA_PLUS**, **XN_FLAG_SEP_CPLUS_SPC**, **XN_FLAG_SEP_SPLUS_SPC** and **XN_FLAG_SEP_MULTILINE** determine the field separators to use. Two distinct separators are used between distinct RelativeDistinguishedName components and separate values in the same RDN for a multi-valued RDN. Multi-valued RDNs are currently very rare so the second separator will hardly ever be used.

XN_FLAG_SEP_COMMA_PLUS uses comma and plus as separators. **XN_FLAG_SEP_CPLUS_SPC** uses comma and plus with spaces: this is more readable that plain comma and plus. **XN_FLAG_SEP_SPLUS_SPC** uses spaced semicolon and plus. **XN_FLAG_SEP_MULTILINE** uses spaced newline and plus respectively.

If **XN_FLAG_DN_REV** is set the whole DN is printed in reversed order.

The fields **XN_FLAG_FN_SN**, **XN_FLAG_FN_LN**, **XN_FLAG_FN_OID**, **XN_FLAG_FN_NONE** determine how a field name is displayed. It will use the short name (e.g. CN) the long name (e.g. commonName) always use OID numerical form (normally OIDs are only used if the field name is not recognised) and no field name respectively.

If **XN_FLAG_SPC_EQ** is set then spaces will be placed around the '=' character separating field names and values.

If **XN_FLAG_DUMP_UNKNOWN_FIELDS** is set then the encoding of unknown fields is printed instead of the values.

If **XN_FLAG_FN_ALIGN** is set then field names are padded to 20 characters: this is only of use for multiline format.

Additionally all the options supported by ASN1_STRING_print_ex() can be used to control how each field value is displayed.

In addition a number options can be set for commonly used formats.

XN_FLAG_RFC2253 sets options which produce an output compatible with RFC2253 it is equivalent to: **ASN1_STRFLGS_RFC2253 | XN_FLAG_SEP_COMMA_PLUS | XN_FLAG_DN_REV | XN_FLAG_FN_SN | XN_FLAG_DUMP_UNKNOWN_FIELDS**

XN_FLAG_ONELINE is a more readable one line format which is the same as: **ASN1_STRFLGS_RFC2253 | ASN1_STRFLGS_ESC_QUOTE | XN_FLAG_SEP_CPLUS_SPC | XN_FLAG_SPC_EQ | XN_FLAG_FN_SN**

XN_FLAG_MULTILINE is a multiline format which is the same as: **ASN1_STRFLGS_ESC_CTRL | ASN1_STRFLGS_ESC_MSB | XN_FLAG_SEP_MULTILINE | XN_FLAG_SPC_EQ | XN_FLAG_FN_LN | XN_FLAG_FN_ALIGN**

XN_FLAG_COMPAT uses a format identical to X509_NAME_print(): in fact it calls X509_NAME_print() internally.

SEE ALSO

ASN1_STRING_print_ex(3)

HISTORY

TBA

Name

X509_new and X509_free — X509 certificate ASN1 allocation functions

Synopsis

```
#include  <openssl/x509.h>
```

```
X509 *X509_new(void);
void X509_free(X509 *a);
```

DESCRIPTION

The X509 ASN1 allocation routines, allocate and free an X509 structure, which represents an X509 certificate.

X509_new() allocates and initializes a X509 structure.

X509_free() frees up the **X509** structure **a**.

RETURN VALUES

If the allocation fails, X509_new() returns **NULL** and sets an error code that can be obtained by ERR_get_error(3). Otherwise it returns a pointer to the newly allocated structure.

X509_free() returns no value.

SEE ALSO

ERR_get_error(3), d2i_X509(3)

HISTORY

X509_new() and X509_free() are available in all versions of SSLeay and OpenSSL.

Name

X509_STORE_CTX_get_error, X509_STORE_CTX_set_error, X509_STORE_CTX_get_error_depth, X509_STORE_CTX_get_current_cert, X509_STORE_CTX_get1_chain and X509_verify_cert_error_string — get or set certificate verification status information

Synopsis

```
#include  <openssl/x509.h>
#include  <openssl/x509_vfy.h>
```

```
int    X509_STORE_CTX_get_error(X509_STORE_CTX *ctx);
void   X509_STORE_CTX_set_error(X509_STORE_CTX *ctx,int s);
int    X509_STORE_CTX_get_error_depth(X509_STORE_CTX *ctx);
X509 * X509_STORE_CTX_get_current_cert(X509_STORE_CTX *ctx);
```

```
STACK_OF(X509) *X509_STORE_CTX_get1_chain(X509_STORE_CTX *ctx);
```

```
const char *X509_verify_cert_error_string(long n);
```

DESCRIPTION

These functions are typically called after X509_verify_cert() has indicated an error or in a verification callback to determine the nature of an error.

X509_STORE_CTX_get_error() returns the error code of **ctx**, see the **ERROR CODES** section for a full description of all error codes.

X509_STORE_CTX_set_error() sets the error code of **ctx** to **s**. For example it might be used in a verification callback to set an error based on additional checks.

X509_STORE_CTX_get_error_depth() returns the **depth** of the error. This is a non-negative integer representing where in the certificate chain the error occurred. If it is zero it occured in the end entity certificate, one if it is the certificate which signed the end entity certificate and so on.

X509_STORE_CTX_get_current_cert() returns the certificate in **ctx** which caused the error or **NULL** if no certificate is relevant.

X509_STORE_CTX_get1_chain() returns a complete validate chain if a previous call to X509_verify_cert() is successful. If the call to X509_verify_cert() is **not** successful the returned chain may be incomplete or invalid. The returned chain persists after the **ctx** structure is freed, when it is no longer needed it should be free up using:

```
sk_X509_pop_free(chain, X509_free);
```

X509_verify_cert_error_string() returns a human readable error string for verification error **n**.

RETURN VALUES

X509_STORE_CTX_get_error() returns **X509_V_OK** or an error code.

X509_STORE_CTX_get_error_depth() returns a non-negative error depth.

X509_STORE_CTX_get_current_cert() returns the cerificate which caused the error or **NULL** if no certificate is relevant to the error.

X509_verify_cert_error_string() returns a human readable error string for verification error **n**.

ERROR CODES

A list of error codes and messages is shown below. Some of the error codes are defined but currently never returned: these are described as "unused".

X509_V_OK: ok

the operation was successful.

X509_V_ERR_UNABLE_TO_GET_ISSUER_CERT: unable to get issuer certificate

the issuer certificate could not be found: this occurs if the issuer certificate of an untrusted certificate cannot be found.

X509_V_ERR_UNABLE_TO_GET_CRL: unable to get certificate CRL

the CRL of a certificate could not be found.

X509_V_ERR_UNABLE_TO_DECRYPT_CERT_SIGNATURE: unable to decrypt certificate's signature

the certificate signature could not be decrypted. This means that the actual signature value could not be determined rather than it not matching the expected value, this is only meaningful for RSA keys.

X509_V_ERR_UNABLE_TO_DECRYPT_CRL_SIGNATURE: unable to decrypt CRL's signature

the CRL signature could not be decrypted: this means that the actual signature value could not be determined rather than it not matching the expected value. Unused.

X509_V_ERR_UNABLE_TO_DECODE_ISSUER_PUBLIC_KEY: unable to decode issuer public key

the public key in the certificate SubjectPublicKeyInfo could not be read.

X509_V_ERR_CERT_SIGNATURE_FAILURE: certificate signature failure

the signature of the certificate is invalid.

X509_V_ERR_CRL_SIGNATURE_FAILURE: CRL signature failure

the signature of the certificate is invalid.

X509_V_ERR_CERT_NOT_YET_VALID: certificate is not yet valid

the certificate is not yet valid: the notBefore date is after the current time.

X509_V_ERR_CERT_HAS_EXPIRED: certificate has expired

the certificate has expired: that is the notAfter date is before the current time.

X509_V_ERR_CRL_NOT_YET_VALID: CRL is not yet valid

the CRL is not yet valid.

X509_V_ERR_CRL_HAS_EXPIRED: CRL has expired

the CRL has expired.

X509_V_ERR_ERROR_IN_CERT_NOT_BEFORE_FIELD: format error in certificate's notBefore field

the certificate notBefore field contains an invalid time.

X509_V_ERR_ERROR_IN_CERT_NOT_AFTER_FIELD: format error in certificate's notAfter field

the certificate notAfter field contains an invalid time.

X509_V_ERR_ERROR_IN_CRL_LAST_UPDATE_FIELD: format error in CRL's lastUpdate field

the CRL lastUpdate field contains an invalid time.

X509_V_ERR_ERROR_IN_CRL_NEXT_UPDATE_FIELD: format error in CRL's nextUpdate field

the CRL nextUpdate field contains an invalid time.

X509_V_ERR_OUT_OF_MEM: out of memory

an error occurred trying to allocate memory. This should never happen.

X509_V_ERR_DEPTH_ZERO_SELF_SIGNED_CERT: self signed certificate

the passed certificate is self signed and the same certificate cannot be found in the list of trusted certificates.

X509_V_ERR_SELF_SIGNED_CERT_IN_CHAIN: self signed certificate in certificate chain

the certificate chain could be built up using the untrusted certificates but the root could not be found locally.

X509_V_ERR_UNABLE_TO_GET_ISSUER_CERT_LOCALLY: unable to get local issuer certificate

the issuer certificate of a locally looked up certificate could not be found. This normally means the list of trusted certificates is not complete.

X509_V_ERR_UNABLE_TO_VERIFY_LEAF_SIGNATURE: unable to verify the first certificate

no signatures could be verified because the chain contains only one certificate and it is not self signed.

X509_V_ERR_CERT_CHAIN_TOO_LONG: certificate chain too long

the certificate chain length is greater than the supplied maximum depth. Unused.

X509_V_ERR_CERT_REVOKED: certificate revoked

the certificate has been revoked.

X509_V_ERR_INVALID_CA: invalid CA certificate

a CA certificate is invalid. Either it is not a CA or its extensions are not consistent with the supplied purpose.

X509_V_ERR_PATH_LENGTH_EXCEEDED: path length constraint exceeded

the basicConstraints pathlength parameter has been exceeded.

X509_V_ERR_INVALID_PURPOSE: unsupported certificate purpose

the supplied certificate cannot be used for the specified purpose.

X509_V_ERR_CERT_UNTRUSTED: certificate not trusted

the root CA is not marked as trusted for the specified purpose.

X509_V_ERR_CERT_REJECTED: certificate rejected

the root CA is marked to reject the specified purpose.

X509_V_ERR_SUBJECT_ISSUER_MISMATCH: subject issuer mismatch

the current candidate issuer certificate was rejected because its subject name did not match the issuer name of the current certificate. This is only set if issuer check debugging is enabled it is used for status notification and is **not** in itself an error.

X509_V_ERR_AKID_SKID_MISMATCH: authority and subject key identifier mismatch

the current candidate issuer certificate was rejected because its subject key identifier was present and did not match the authority key identifier current certificate. This is only set if issuer check debugging is enabled it is used for status notification and is **not** in itself an error.

X509_V_ERR_AKID_ISSUER_SERIAL_MISMATCH: authority and issuer serial number mismatch

the current candidate issuer certificate was rejected because its issuer name and serial number was present and did not match the authority key identifier of the current certificate. This is only set if issuer check debugging is enabled it is used for status notification and is **not** in itself an error.

X509_V_ERR_KEYUSAGE_NO_CERTSIGN:key usage does not include certificate signing

the current candidate issuer certificate was rejected because its keyUsage extension does not permit certificate signing. This is only set if issuer check debugging is enabled it is used for status notification and is **not** in itself an error.

X509_V_ERR_INVALID_EXTENSION: invalid or inconsistent certificate extension

A certificate extension had an invalid value (for example an incorrect encoding) or some value inconsistent with other extensions.

X509_V_ERR_INVALID_POLICY_EXTENSION: invalid or inconsistent certificate policy extension

A certificate policies extension had an invalid value (for example an incorrect encoding) or some value inconsistent with other extensions. This error only occurs if policy processing is enabled.

X509_V_ERR_NO_EXPLICIT_POLICY: no explicit policy

The verification flags were set to require and explicit policy but none was present.

X509_V_ERR_DIFFERENT_CRL_SCOPE: Different CRL scope

The only CRLs that could be found did not match the scope of the certificate.

X509_V_ERR_UNSUPPORTED_EXTENSION_FEATURE: Unsupported extension feature

Some feature of a certificate extension is not supported. Unused.

X509_V_ERR_PERMITTED_VIOLATION: permitted subtree violation

A name constraint violation occured in the permitted subtrees.

X509_V_ERR_EXCLUDED_VIOLATION: excluded subtree violation

A name constraint violation occured in the excluded subtrees.

X509_V_ERR_SUBTREE_MINMAX: name constraints minimum and maximum not supported

A certificate name constraints extension included a minimum or maximum field: this is not supported.

X509_V_ERR_UNSUPPORTED_CONSTRAINT_TYPE: unsupported name constraint type

An unsupported name constraint type was encountered. OpenSSL currently only supports directory name, DNS name, email and URI types.

X509_V_ERR_UNSUPPORTED_CONSTRAINT_SYNTAX: unsupported or invalid name constraint syntax

The format of the name constraint is not recognised: for example an email address format of a form not mentioned in RFC3280. This could be caused by a garbage extension or some new feature not currently supported.

X509_V_ERR_CRL_PATH_VALIDATION_ERROR: CRL path validation error

An error occured when attempting to verify the CRL path. This error can only happen if extended CRL checking is enabled.

X509_V_ERR_APPLICATION_VERIFICATION: application verification failure

an application specific error. This will never be returned unless explicitly set by an application.

NOTES

The above functions should be used instead of directly referencing the fields in the **X509_VERIFY_CTX** structure.

In versions of OpenSSL before 1.0 the current certificate returned by X509_STORE_CTX_get_current_cert() was never **NULL**. Applications should check the return value before printing out any debugging information relating to the current certificate.

If an unrecognised error code is passed to X509_verify_cert_error_string() the numerical value of the unknown code is returned in a static buffer. This is not thread safe but will never happen unless an invalid code is passed.

SEE ALSO

X509_verify_cert(3)

HISTORY

TBA

Name

X509_STORE_CTX_get_ex_new_index, X509_STORE_CTX_set_ex_data and X509_STORE_CTX_get_ex_data — add application specific data to X509_STORE_CTX structures

Synopsis

```
#include  <openssl/x509_vfy.h>
```

```
int X509_STORE_CTX_get_ex_new_index(long argl, void *argp,
               CRYPTO_EX_new *new_func,
               CRYPTO_EX_dup *dup_func,
               CRYPTO_EX_free *free_func);
```

```
int X509_STORE_CTX_set_ex_data(X509_STORE_CTX *d, int idx, void *arg);
```

```
void *X509_STORE_CTX_get_ex_data(X509_STORE_CTX *d, int idx);
```

DESCRIPTION

These functions handle application specific data in X509_STORE_CTX structures. Their usage is identical to that of RSA_get_ex_new_index(), RSA_set_ex_data() and RSA_get_ex_data() as described in RSA_get_ex_new_index(3).

NOTES

This mechanism is used internally by the **ssl** library to store the **SSL** structure associated with a verification operation in an **X509_STORE_CTX** structure.

SEE ALSO

RSA_get_ex_new_index(3)

HISTORY

X509_STORE_CTX_get_ex_new_index(), X509_STORE_CTX_set_ex_data() and X509_STORE_CTX_get_ex_data() are available since OpenSSL 0.9.5.

Name

X509_STORE_CTX_new, X509_STORE_CTX_cleanup, X509_STORE_CTX_free, X509_STORE_CTX_init, X509_STORE_CTX_trusted_stack, X509_STORE_CTX_set_cert, X509_STORE_CTX_set_chain, X509_STORE_CTX_set0_crls, X509_STORE_CTX_get0_param, X509_STORE_CTX_set0_param and X509_STORE_CTX_set_default — X509_STORE_CTX initialisation

Synopsis

```
#include  <openssl/x509_vfy.h>
```

```
X509_STORE_CTX *X509_STORE_CTX_new(void);
void X509_STORE_CTX_cleanup(X509_STORE_CTX *ctx);
void X509_STORE_CTX_free(X509_STORE_CTX *ctx);
```

```
int X509_STORE_CTX_init(X509_STORE_CTX *ctx, X509_STORE *store,
                        X509 *x509, STACK_OF(X509) *chain);
```

```
void X509_STORE_CTX_trusted_stack(X509_STORE_CTX *ctx, STACK_OF(X509) *sk);
```

```
void   X509_STORE_CTX_set_cert(X509_STORE_CTX *ctx,X509 *x);
void   X509_STORE_CTX_set_chain(X509_STORE_CTX *ctx,STACK_OF(X509) *sk);
void   X509_STORE_CTX_set0_crls(X509_STORE_CTX *ctx, STACK_OF(X509_CRL) *sk);
```

```
X509_VERIFY_PARAM *X509_STORE_CTX_get0_param(X509_STORE_CTX *ctx);
void X509_STORE_CTX_set0_param(X509_STORE_CTX *ctx, X509_VERIFY_PARAM *param);
int X509_STORE_CTX_set_default(X509_STORE_CTX *ctx, const char *name);
```

DESCRIPTION

These functions initialise an **X509_STORE_CTX** structure for subsequent use by X509_verify_cert().

X509_STORE_CTX_new() returns a newly initialised **X509_STORE_CTX** structure.

X509_STORE_CTX_cleanup() internally cleans up an **X509_STORE_CTX** structure. The context can then be reused with an new call to X509_STORE_CTX_init().

X509_STORE_CTX_free() completely frees up **ctx**. After this call **ctx** is no longer valid.

X509_STORE_CTX_init() sets up **ctx** for a subsequent verification operation. It must be called before each call to X509_verify_cert(), i.e. a **ctx** is only good for one call to X509_verify_cert(); if you want to verify a second certificate with the same **ctx** then you must call X509_XTORE_CTX_cleanup() and then X509_STORE_CTX_init() again before the second call to X509_verify_cert(). The trusted certificate store is set to **store**, the end entity certificate to be verified is set to **x509** and a set of additional certificates (which will be untrusted but may be used to build the chain) in **chain**. Any or all of the **store**, **x509** and **chain** parameters can be **NULL**.

X509_STORE_CTX_trusted_stack() sets the set of trusted certificates of **ctx** to **sk**. This is an alternative way of specifying trusted certificates instead of using an **X509_STORE**.

X509_STORE_CTX_set_cert() sets the certificate to be vertified in **ctx** to **x**.

X509_STORE_CTX_set_chain() sets the additional certificate chain used by **ctx** to **sk**.

X509_STORE_CTX_set0_crls() sets a set of CRLs to use to aid certificate verification to **sk**. These CRLs will only be used if CRL verification is enabled in the associated **X509_VERIFY_PARAM** structure. This might be used where additional "useful" CRLs are supplied as part of a protocol, for example in a PKCS#7 structure.

X509_VERIFY_PARAM *X509_STORE_CTX_get0_param() retrieves an intenal pointer to the verification parameters associated with **ctx**.

X509_STORE_CTX_set0_param() sets the intenal verification parameter pointer to **param**. After this call **param** should not be used.

X509_STORE_CTX_set_default() looks up and sets the default verification method to **name**. This uses the function X509_VERIFY_PARAM_lookup() to find an appropriate set of parameters from **name**.

NOTES

The certificates and CRLs in a store are used internally and should **not** be freed up until after the associated **X509_STORE_CTX** is freed. Legacy applications might implicitly use an **X509_STORE_CTX** like this:

```
X509_STORE_CTX ctx;
X509_STORE_CTX_init(&ctx, store, cert, chain);
```

this is **not** recommended in new applications they should instead do:

```
X509_STORE_CTX *ctx;
ctx = X509_STORE_CTX_new();
if (ctx == NULL)
      /* Bad error */
X509_STORE_CTX_init(ctx, store, cert, chain);
```

BUGS

The certificates and CRLs in a context are used internally and should **not** be freed up until after the associated **X509_STORE_CTX** is freed. Copies should be made or reference counts increased instead.

RETURN VALUES

X509_STORE_CTX_new() returns an newly allocates context or **NULL** is an error occurred.

X509_STORE_CTX_init() returns 1 for success or 0 if an error occurred.

X509_STORE_CTX_get0_param() returns a pointer to an **X509_VERIFY_PARAM** structure or **NULL** if an error occurred.

X509_STORE_CTX_cleanup(), X509_STORE_CTX_free(), X509_STORE_CTX_trusted_stack(), X509_STORE_CTX_set_cert(), X509_STORE_CTX_set_chain(), X509_STORE_CTX_set0_crls() and X509_STORE_CTX_set0_param() do not return values.

X509_STORE_CTX_set_default() returns 1 for success or 0 if an error occurred.

SEE ALSO

X509_verify_cert(3) X509_VERIFY_PARAM_set_flags(3)

HISTORY

X509_STORE_CTX_set0_crls() was first added to OpenSSL 1.0.0

Name

X509_STORE_CTX_set_verify_cb — set verification callback

Synopsis

```
#include  <openssl/x509_vfy.h>
```

```
void X509_STORE_CTX_set_verify_cb(X509_STORE_CTX *ctx,
                               int (*verify_cb)(int ok, X509_STORE_CTX *ctx));
```

DESCRIPTION

X509_STORE_CTX_set_verify_cb() sets the verification callback of **ctx** to **verify_cb** overwriting any existing callback.

The verification callback can be used to customise the operation of certificate verification, either by overriding error conditions or logging errors for debugging purposes.

However a verification callback is **not** essential and the default operation is often sufficient.

The **ok** parameter to the callback indicates the value the callback should return to retain the default behaviour. If it is zero then and error condition is indicated. If it is 1 then no error occurred. If the flag **X509_V_FLAG_NOTIFY_POLICY** is set then **ok** is set to 2 to indicate the policy checking is complete.

The **ctx** parameter to the callback is the **X509_STORE_CTX** structure that is performing the verification operation. A callback can examine this structure and receive additional information about the error, for example by calling X509_STORE_CTX_get_current_cert(). Additional application data can be passed to the callback via the **ex_data** mechanism.

WARNING

In general a verification callback should **NOT** unconditionally return 1 in all circumstances because this will allow verification to succeed no matter what the error. This effectively removes all security from the application because **any** certificate (including untrusted generated ones) will be accepted.

NOTES

The verification callback can be set and inherited from the parent structure performing the operation. In some cases (such as S/MIME verification) the **X509_STORE_CTX** structure is created and destroyed internally and the only way to set a custom verification callback is by inheriting it from the associated **X509_STORE**.

RETURN VALUES

X509_STORE_CTX_set_verify_cb() does not return a value.

EXAMPLES

Default callback operation:

```
int verify_callback(int ok, X509_STORE_CTX *ctx)
{
    return ok;
}
```

Simple example, suppose a certificate in the chain is expired and we wish to continue after this error:

```
int verify_callback(int ok, X509_STORE_CTX *ctx)
{
    /* Tolerate certificate expiration */
    if (X509_STORE_CTX_get_error(ctx) == X509_V_ERR_CERT_HAS_EXPIRED)
                    return 1;
    /* Otherwise don't override */
    return ok;
}
```

More complex example, we don't wish to continue after **any** certificate has expired just one specific case:

```
int verify_callback(int ok, X509_STORE_CTX *ctx)
{
    int err = X509_STORE_CTX_get_error(ctx);
    X509 *err_cert = X509_STORE_CTX_get_current_cert(ctx);
    if (err == X509_V_ERR_CERT_HAS_EXPIRED)
    {
        if (check_is_acceptable_expired_cert(err_cert)
               return 1;
    }
    return ok;
}
```

Full featured logging callback. In this case the **bio_err** is assumed to be a global logging **BIO**, an alternative would to store a BIO in **ctx** using **ex_data**.

```
int verify_callback(int ok, X509_STORE_CTX *ctx)
{
    X509 *err_cert;
    int err,depth;

    err_cert = X509_STORE_CTX_get_current_cert(ctx);
    err =   X509_STORE_CTX_get_error(ctx);
    depth = X509_STORE_CTX_get_error_depth(ctx);

    BIO_printf(bio_err,"depth=%d ",depth);
    if (err_cert)
    {
        X509_NAME_print_ex(bio_err, X509_get_subject_name(err_cert),
                                0, XN_FLAG_ONELINE);
        BIO_puts(bio_err, "\n");
    }
    else
        BIO_puts(bio_err, "<no cert>\n");
    if (!ok)
        BIO_printf(bio_err,"verify error:num=%d:%s\n",err,
                X509_verify_cert_error_string(err));
    switch (err)
            {
    case X509_V_ERR_UNABLE_TO_GET_ISSUER_CERT:
            BIO_puts(bio_err,"issuer= ");
            X509_NAME_print_ex(bio_err, X509_get_issuer_name(err_cert),
                                    0, XN_FLAG_ONELINE);
            BIO_puts(bio_err, "\n");
            break;
    case X509_V_ERR_CERT_NOT_YET_VALID:
    case X509_V_ERR_ERROR_IN_CERT_NOT_BEFORE_FIELD:
            BIO_printf(bio_err,"notBefore=");
            ASN1_TIME_print(bio_err,X509_get_notBefore(err_cert));
            BIO_printf(bio_err,"\n");
            break;
    case X509_V_ERR_CERT_HAS_EXPIRED:
    case X509_V_ERR_ERROR_IN_CERT_NOT_AFTER_FIELD:
            BIO_printf(bio_err,"notAfter=");
            ASN1_TIME_print(bio_err,X509_get_notAfter(err_cert));
            BIO_printf(bio_err,"\n");
            break;
    case X509_V_ERR_NO_EXPLICIT_POLICY:
            policies_print(bio_err, ctx);
            break;
            }
    if (err == X509_V_OK && ok == 2)
        /* print out policies */

    BIO_printf(bio_err,"verify return:%d\n",ok);
    return(ok);
}
```

SEE ALSO

X509_STORE_CTX_get_error(3) X509_STORE_set_verify_cb_func(3) X509_STORE_CTX_get_ex_new_index(3)

HISTORY

X509_STORE_CTX_set_verify_cb() is available in all versions of SSLeay and OpenSSL.

Name

X509_STORE_set_verify_cb_func and X509_STORE_set_verify_cb — set verification callback

Synopsis

```
#include  <openssl/x509_vfy.h>
```

```
void X509_STORE_set_verify_cb(X509_STORE *st,
                               int (*verify_cb)(int ok, X509_STORE_CTX *ctx));
```

```
void X509_STORE_set_verify_cb_func(X509_STORE *st,
                               int (*verify_cb)(int ok, X509_STORE_CTX *ctx));
```

DESCRIPTION

X509_STORE_set_verify_cb() sets the verification callback of **ctx** to **verify_cb** overwriting any existing callback.

X509_STORE_set_verify_cb_func() also sets the verification callback but it is implemented as a macro.

NOTES

The verification callback from an **X509_STORE** is inherited by the corresponding **X509_STORE_CTX** structure when it is initialized. This can be used to set the verification callback when the **X509_STORE_CTX** is otherwise inaccessible (for example during S/MIME verification).

BUGS

The macro version of this function was the only one available before OpenSSL 1.0.0.

RETURN VALUES

X509_STORE_set_verify_cb() and X509_STORE_set_verify_cb_func() do not return a value.

SEE ALSO

X509_STORE_CTX_set_verify_cb(3) CMS_verify(3)

HISTORY

X509_STORE_set_verify_cb_func() is available in all versions of SSLeay and OpenSSL.

X509_STORE_set_verify_cb() was added to OpenSSL 1.0.0.

Name

X509_verify_cert — discover and verify X509 certificte chain

Synopsis

```
#include  <openssl/x509.h>
```

```
int X509_verify_cert(X509_STORE_CTX *ctx);
```

DESCRIPTION

The X509_verify_cert() function attempts to discover and validate a certificate chain based on parameters in **ctx**. A complete description of the process is contained in the verify(1) manual page.

RETURN VALUES

If a complete chain can be built and validated this function returns 1, otherwise it return zero, in exceptional circumstances it can also return a negative code.

If the function fails additional error information can be obtained by examining **ctx** using, for example X509_STORE_CTX_get_error().

NOTES

Applications rarely call this function directly but it is used by OpenSSL internally for certificate validation, in both the S/MIME and SSL/TLS code.

A negative return value from X509_verify_cert() can occur if it is invoked incorrectly, such as with no certificate set in **ctx**, or when it is called twice in succession without reinitialising **ctx** for the second call. A negative return value can also happen due to internal resource problems or if a retry operation is requested during internal lookups (which never happens with standard lookup methods). Applications must check for <= 0 return value on error.

BUGS

This function uses the header **x509.h** as opposed to most chain verification functiosn which use **x509_vfy.h**.

SEE ALSO

X509_STORE_CTX_get_error(3)

HISTORY

X509_verify_cert() is available in all versions of SSLeay and OpenSSL.

Name

X509_VERIFY_PARAM_set_flags, X509_VERIFY_PARAM_clear_flags, X509_VERIFY_PARAM_get_flags, X509_VERIFY_PARAM_set_purpose, X509_VERIFY_PARAM_set_trust, X509_VERIFY_PARAM_set_depth, X509_VERIFY_PARAM_get_depth, X509_VERIFY_PARAM_set_time, X509_VERIFY_PARAM_add0_policy and X509_VERIFY_PARAM_set1_policies — X509 verification parameters

Synopsis

```
#include  <openssl/x509_vfy.h>
```

```
int X509_VERIFY_PARAM_set_flags(X509_VERIFY_PARAM *param, unsigned long flags);
int X509_VERIFY_PARAM_clear_flags(X509_VERIFY_PARAM *param,
                                                       unsigned long flags);
unsigned long X509_VERIFY_PARAM_get_flags(X509_VERIFY_PARAM *param);
```

```
int X509_VERIFY_PARAM_set_purpose(X509_VERIFY_PARAM *param, int purpose);
int X509_VERIFY_PARAM_set_trust(X509_VERIFY_PARAM *param, int trust);
```

```
void X509_VERIFY_PARAM_set_time(X509_VERIFY_PARAM *param, time_t t);
```

```
int X509_VERIFY_PARAM_add0_policy(X509_VERIFY_PARAM *param,
                                               ASN1_OBJECT *policy);
int X509_VERIFY_PARAM_set1_policies(X509_VERIFY_PARAM *param,
                                       STACK_OF(ASN1_OBJECT) *policies);
```

```
void X509_VERIFY_PARAM_set_depth(X509_VERIFY_PARAM *param, int depth);
int X509_VERIFY_PARAM_get_depth(const X509_VERIFY_PARAM *param);
```

DESCRIPTION

These functions manipulate the **X509_VERIFY_PARAM** structure associated with a certificate verification operation.

The X509_VERIFY_PARAM_set_flags() function sets the flags in **param** by oring it with **flags**. See the **VERIFICATION FLAGS** section for a complete description of values the **flags** parameter can take.

X509_VERIFY_PARAM_get_flags() returns the flags in **param**.

X509_VERIFY_PARAM_clear_flags() clears the flags **flags** in **param**.

X509_VERIFY_PARAM_set_purpose() sets the verification purpose in **param** to **purpose**. This determines the acceptable purpose of the certificate chain, for example SSL client or SSL server.

X509_VERIFY_PARAM_set_trust() sets the trust setting in **param** to **trust**.

X509_VERIFY_PARAM_set_time() sets the verification time in **param** to **t**. Normally the current time is used.

X509_VERIFY_PARAM_add0_policy() enables policy checking (it is disabled by default) and adds **policy** to the acceptable policy set.

X509_VERIFY_PARAM_set1_policies() enables policy checking (it is disabled by default) and sets the acceptable policy set to **policies**. Any existing policy set is cleared. The **policies** parameter can be **NULL** to clear an existing policy set.

X509_VERIFY_PARAM_set_depth() sets the maximum verification depth to **depth**. That is the maximum number of untrusted CA certificates that can appear in a chain.

RETURN VALUES

X509_VERIFY_PARAM_set_flags(), X509_VERIFY_PARAM_clear_flags(), X509_VERIFY_PARAM_set_purpose(), X509_VERIFY_PARAM_set_trust(), X509_VERIFY_PARAM_add0_policy() and X509_VERIFY_PARAM_set1_policies() return 1 for success and 0 for failure.

X509_VERIFY_PARAM_get_flags() returns the current verification flags.

X509_VERIFY_PARAM_set_time() and X509_VERIFY_PARAM_set_depth() do not return values.

X509_VERIFY_PARAM_get_depth() returns the current verification depth.

VERIFICATION FLAGS

The verification flags consists of zero or more of the following flags ored together.

X509_V_FLAG_CRL_CHECK enables CRL checking for the certificate chain leaf certificate. An error occurs if a suitable CRL cannot be found.

X509_V_FLAG_CRL_CHECK_ALL enables CRL checking for the entire certificate chain.

X509_V_FLAG_IGNORE_CRITICAL disabled critical extension checking. By default any unhandled critical extensions in certificates or (if checked) CRLs results in a fatal error. If this flag is set unhandled critical extensions are ignored. **WARNING** setting this option for anything other than debugging purposes can be a security risk. Finer control over which extensions are supported can be performed in the verification callback.

THe **X509_V_FLAG_X509_STRICT** flag disables workarounds for some broken certificates and makes the verification strictly apply **X509** rules.

X509_V_FLAG_ALLOW_PROXY_CERTS enables proxy certificate verification.

X509_V_FLAG_POLICY_CHECK enables certificate policy checking, by default no policy checking is peformed. Additional information is sent to the verification callback relating to policy checking.

X509_V_FLAG_EXPLICIT_POLICY, **X509_V_FLAG_INHIBIT_ANY** and **X509_V_FLAG_INHIBIT_MAP** set the **require explicit policy**, **inhibit any policy** and **inhibit policy mapping** flags respectively as defined in **RFC3280**. Policy checking is automatically enabled if any of these flags are set.

If **X509_V_FLAG_NOTIFY_POLICY** is set and the policy checking is successful a special status code is set to the verification callback. This permits it to examine the valid policy tree and perform additional checks or simply log it for debugging purposes.

By default some additional features such as indirect CRLs and CRLs signed by different keys are disabled. If **X509_V_FLAG_EXTENDED_CRL_SUPPORT** is set they are enabled.

If **X509_V_FLAG_USE_DELTAS** ise set delta CRLs (if present) are used to determine certificate status. If not set deltas are ignored.

X509_V_FLAG_CHECK_SS_SIGNATURE enables checking of the root CA self signed cerificate signature. By default this check is disabled because it doesn't add any additional security but in some cases applications might want to check the signature anyway. A side effect of not checking the root CA signature is that disabled or unsupported message digests on the root CA are not treated as fatal errors.

The **X509_V_FLAG_CB_ISSUER_CHECK** flag enables debugging of certificate issuer checks. It is **not** needed unless you are logging certificate verification. If this flag is set then additional status codes will be sent to the verification callback and it **must** be prepared to handle such cases without assuming they are hard errors.

The **X509_V_FLAG_NO_ALT_CHAINS** flag suppresses checking for alternative chains. By default, when building a certificate chain, if the first certificate chain found is not trusted, then OpenSSL will continue to check to see if an alternative chain can be found that is trusted. With this flag set the behaviour will match that of OpenSSL versions prior to 1.0.1n and 1.0.2b.

NOTES

The above functions should be used to manipulate verification parameters instead of legacy functions which work in specific structures such as X509_STORE_CTX_set_flags().

BUGS

Delta CRL checking is currently primitive. Only a single delta can be used and (partly due to limitations of **X509_STORE**) constructed CRLs are not maintained.

If CRLs checking is enable CRLs are expected to be available in the corresponding **X509_STORE** structure. No attempt is made to download CRLs from the CRL distribution points extension.

EXAMPLE

Enable CRL checking when performing certificate verification during SSL connections associated with an **SSL_CTX** structure **ctx**:

```
X509_VERIFY_PARAM *param;
param = X509_VERIFY_PARAM_new();
X509_VERIFY_PARAM_set_flags(param, X509_V_FLAG_CRL_CHECK);
SSL_CTX_set1_param(ctx, param);
X509_VERIFY_PARAM_free(param);
```

SEE ALSO

X509_verify_cert(3)

HISTORY

The **X509_V_FLAG_NO_ALT_CHAINS** flag was added in OpenSSL 1.0.1n and 1.0.2b

SSL Functions

Name

SSL — OpenSSL SSL/TLS library

Synopsis

DESCRIPTION

The OpenSSL **ssl** library implements the Secure Sockets Layer (SSL v2/v3) and Transport Layer Security (TLS v1) protocols. It provides a rich API which is documented here.

At first the library must be initialized; see SSL_library_init(3).

Then an **SSL_CTX** object is created as a framework to establish TLS/SSL enabled connections (see SSL_CTX_new(3)). Various options regarding certificates, algorithms etc. can be set in this object.

When a network connection has been created, it can be assigned to an **SSL** object. After the **SSL** object has been created using SSL_new(3), SSL_set_fd(3) or SSL_set_bio(3) can be used to associate the network connection with the object.

Then the TLS/SSL handshake is performed using SSL_accept(3) or SSL_connect(3) respectively. SSL_read(3) and SSL_write(3) are used to read and write data on the TLS/SSL connection. SSL_shutdown(3) can be used to shut down the TLS/SSL connection.

DATA STRUCTURES

Currently the OpenSSL **ssl** library functions deals with the following data structures:

SSL_METHOD (SSL Method)

> That's a dispatch structure describing the internal **ssl** library methods/functions which implement the various protocol versions (SSLv1, SSLv2 and TLSv1). It's needed to create an **SSL_CTX**.

SSL_CIPHER (SSL Cipher)

> This structure holds the algorithm information for a particular cipher which are a core part of the SSL/TLS protocol. The available ciphers are configured on a **SSL_CTX** basis and the actually used ones are then part of the **SSL_SESSION**.

SSL_CTX (SSL Context)

> That's the global context structure which is created by a server or client once per program life-time and which holds mainly default values for the **SSL** structures which are later created for the connections.

SSL_SESSION (SSL Session)

> This is a structure containing the current TLS/SSL session details for a connection: **SSL_CIPHER**s, client and server certificates, keys, etc.

SSL (SSL Connection)

> That's the main SSL/TLS structure which is created by a server or client per established connection. This actually is the core structure in the SSL API. Under run-time the application usually deals with this structure which has links to mostly all other structures.

HEADER FILES

Currently the OpenSSL **ssl** library provides the following C header files containing the prototypes for the data structures and and functions:

ssl.h

That's the common header file for the SSL/TLS API. Include it into your program to make the API of the **ssl** library available. It internally includes both more private SSL headers and headers from the **crypto** library. Whenever you need hard-core details on the internals of the SSL API, look inside this header file.

ssl2.h

That's the sub header file dealing with the SSLv2 protocol only. *Usually you don't have to include it explicitly because it's already included by ssl.h*.

ssl3.h

That's the sub header file dealing with the SSLv3 protocol only. *Usually you don't have to include it explicitly because it's already included by ssl.h*.

ssl23.h

That's the sub header file dealing with the combined use of the SSLv2 and SSLv3 protocols. *Usually you don't have to include it explicitly because it's already included by ssl.h*.

tls1.h

That's the sub header file dealing with the TLSv1 protocol only. *Usually you don't have to include it explicitly because it's already included by ssl.h*.

API FUNCTIONS

Currently the OpenSSL **ssl** library exports 214 API functions. They are documented in the following:

DEALING WITH PROTOCOL METHODS

Here we document the various API functions which deal with the SSL/TLS protocol methods defined in **SSL_METHOD** structures.

const SSL_METHOD ***SSLv23_method**(void);

Constructor for the *version-flexible* SSL_METHOD structure for clients, servers or both. See SSL_CTX_new(3) for details.

const SSL_METHOD ***SSLv23_client_method**(void);

Constructor for the *version-flexible* SSL_METHOD structure for clients.

const SSL_METHOD ***SSLv23_client_method**(void);

Constructor for the *version-flexible* SSL_METHOD structure for servers.

const SSL_METHOD ***TLSv1_2_method**(void);

Constructor for the TLSv1.2 SSL_METHOD structure for clients, servers or both.

const SSL_METHOD ***TLSv1_2_client_method**(void);

Constructor for the TLSv1.2 SSL_METHOD structure for clients.

const SSL_METHOD ***TLSv1_2_server_method**(void);

Constructor for the TLSv1.2 SSL_METHOD structure for servers.

const SSL_METHOD ***TLSv1_1_method**(void);

Constructor for the TLSv1.1 SSL_METHOD structure for clients, servers or both.

const SSL_METHOD ***TLSv1_1_client_method**(void);

Constructor for the TLSv1.1 SSL_METHOD structure for clients.

const SSL_METHOD ***TLSv1_1_server_method**(void);

Constructor for the TLSv1.1 SSL_METHOD structure for servers.

const SSL_METHOD ***TLSv1_method**(void);

Constructor for the TLSv1 SSL_METHOD structure for clients, servers or both.

const SSL_METHOD ***TLSv1_client_method**(void);

Constructor for the TLSv1 SSL_METHOD structure for clients.

const SSL_METHOD ***TLSv1_server_method**(void);

Constructor for the TLSv1 SSL_METHOD structure for servers.

const SSL_METHOD ***SSLv3_method**(void);

Constructor for the SSLv3 SSL_METHOD structure for clients, servers or both.

const SSL_METHOD ***SSLv3_client_method**(void);

Constructor for the SSLv3 SSL_METHOD structure for clients.

const SSL_METHOD ***SSLv3_server_method**(void);

Constructor for the SSLv3 SSL_METHOD structure for servers.

const SSL_METHOD ***SSLv2_method**(void);

Constructor for the SSLv2 SSL_METHOD structure for clients, servers or both.

const SSL_METHOD ***SSLv2_client_method**(void);

Constructor for the SSLv2 SSL_METHOD structure for clients.

const SSL_METHOD ***SSLv2_server_method**(void);

Constructor for the SSLv2 SSL_METHOD structure for servers.

DEALING WITH CIPHERS

Here we document the various API functions which deal with the SSL/TLS ciphers defined in **SSL_CIPHER** structures.

char ***SSL_CIPHER_description**(SSL_CIPHER *cipher, char *buf, int len);

Write a string to *buf* (with a maximum size of *len*) containing a human readable description of *cipher*. Returns *buf*.

int **SSL_CIPHER_get_bits**(SSL_CIPHER *cipher, int *alg_bits);

Determine the number of bits in *cipher*. Because of export crippled ciphers there are two bits: The bits the algorithm supports in general (stored to *alg_bits*) and the bits which are actually used (the return value).

const char ***SSL_CIPHER_get_name**(SSL_CIPHER *cipher);

> Return the internal name of *cipher* as a string. These are the various strings defined by the *SSL2_TXT_xxx*, *SSL3_TXT_xxx* and *TLS1_TXT_xxx* definitions in the header files.

char ***SSL_CIPHER_get_version**(SSL_CIPHER *cipher);

> Returns a string like "`TLSv1/SSLv3`" or "`SSLv2`" which indicates the SSL/TLS protocol version to which *cipher* belongs (i.e. where it was defined in the specification the first time).

DEALING WITH PROTOCOL CONTEXTS

Here we document the various API functions which deal with the SSL/TLS protocol context defined in the **SSL_CTX** structure.

int **SSL_CTX_add_client_CA**(SSL_CTX *ctx, X509 *x);
long **SSL_CTX_add_extra_chain_cert**(SSL_CTX *ctx, X509 *x509);
int **SSL_CTX_add_session**(SSL_CTX *ctx, SSL_SESSION *c);
int **SSL_CTX_check_private_key**(const SSL_CTX *ctx);
long **SSL_CTX_ctrl**(SSL_CTX *ctx, int cmd, long larg, char *parg);
void **SSL_CTX_flush_sessions**(SSL_CTX *s, long t);
void **SSL_CTX_free**(SSL_CTX *a);
char ***SSL_CTX_get_app_data**(SSL_CTX *ctx);
X509_STORE ***SSL_CTX_get_cert_store**(SSL_CTX *ctx);
STACK ***SSL_CTX_get_client_CA_list**(const SSL_CTX *ctx);
int (***SSL_CTX_get_client_cert_cb**(SSL_CTX *ctx))(SSL *ssl, X509 **x509, EVP_PKEY **pkey);
void **SSL_CTX_get_default_read_ahead**(SSL_CTX *ctx);
char ***SSL_CTX_get_ex_data**(const SSL_CTX *s, int idx);
int **SSL_CTX_get_ex_new_index**(long argl, char *argp, int (*new_func);(void), int (*dup_func)(void), void (*free_func)(void))

void (***SSL_CTX_get_info_callback**(SSL_CTX *ctx))(SSL *ssl, int cb, int ret);
int **SSL_CTX_get_quiet_shutdown**(const SSL_CTX *ctx);
void **SSL_CTX_get_read_ahead**(SSL_CTX *ctx);
int **SSL_CTX_get_session_cache_mode**(SSL_CTX *ctx);
long **SSL_CTX_get_timeout**(const SSL_CTX *ctx);
int (***SSL_CTX_get_verify_callback**(const SSL_CTX *ctx))(int ok, X509_STORE_CTX *ctx);
int **SSL_CTX_get_verify_mode**(SSL_CTX *ctx);
int **SSL_CTX_load_verify_locations**(SSL_CTX *ctx, char *CAfile, char *CApath);
long **SSL_CTX_need_tmp_RSA**(SSL_CTX *ctx);
SSL_CTX ***SSL_CTX_new**(const SSL_METHOD *meth);
int **SSL_CTX_remove_session**(SSL_CTX *ctx, SSL_SESSION *c);
int **SSL_CTX_sess_accept**(SSL_CTX *ctx);
int **SSL_CTX_sess_accept_good**(SSL_CTX *ctx);
int **SSL_CTX_sess_accept_renegotiate**(SSL_CTX *ctx);
int **SSL_CTX_sess_cache_full**(SSL_CTX *ctx);
int **SSL_CTX_sess_cb_hits**(SSL_CTX *ctx);
int **SSL_CTX_sess_connect**(SSL_CTX *ctx);
int **SSL_CTX_sess_connect_good**(SSL_CTX *ctx);
int **SSL_CTX_sess_connect_renegotiate**(SSL_CTX *ctx);
int **SSL_CTX_sess_get_cache_size**(SSL_CTX *ctx);
SSL_SESSION *(***SSL_CTX_sess_get_get_cb**(SSL_CTX *ctx))(SSL *ssl, unsigned char *data, int len, int *copy);

int (***SSL_CTX_sess_get_new_cb**(SSL_CTX *ctx)(SSL *ssl, SSL_SESSION *sess);
void (***SSL_CTX_sess_get_remove_cb**(SSL_CTX *ctx)(SSL_CTX *ctx, SSL_SESSION *sess);
int **SSL_CTX_sess_hits**(SSL_CTX *ctx);
int **SSL_CTX_sess_misses**(SSL_CTX *ctx);
int **SSL_CTX_sess_number**(SSL_CTX *ctx);
void **SSL_CTX_sess_set_cache_size**(SSL_CTX *ctx,t);
void **SSL_CTX_sess_set_get_cb**(SSL_CTX *ctx, SSL_SESSION *(*cb)(SSL *ssl, unsigned char *data, int len, int *copy));
void **SSL_CTX_sess_set_new_cb**(SSL_CTX *ctx, int (*cb)(SSL *ssl, SSL_SESSION *sess));
void **SSL_CTX_sess_set_remove_cb**(SSL_CTX *ctx, void (*cb)(SSL_CTX *ctx, SSL_SESSION *sess));
int **SSL_CTX_sess_timeouts**(SSL_CTX *ctx);
LHASH ***SSL_CTX_sessions**(SSL_CTX *ctx);

void **SSL_CTX_set_app_data**(SSL_CTX *ctx, void *arg);
void **SSL_CTX_set_cert_store**(SSL_CTX *ctx, X509_STORE *cs);
void **SSL_CTX_set_cert_verify_cb**(SSL_CTX *ctx, int (*cb)(), char *arg)
int **SSL_CTX_set_cipher_list**(SSL_CTX *ctx, char *str);
void **SSL_CTX_set_client_CA_list**(SSL_CTX *ctx, STACK *list);
void **SSL_CTX_set_client_cert_cb**(SSL_CTX *ctx, int (*cb)(SSL *ssl, X509 **x509, EVP_PKEY **pkey));
void **SSL_CTX_set_default_passwd_cb**(SSL_CTX *ctx, int (*cb);(void))
void **SSL_CTX_set_default_read_ahead**(SSL_CTX *ctx, int m);
int **SSL_CTX_set_default_verify_paths**(SSL_CTX *ctx);
int **SSL_CTX_set_ex_data**(SSL_CTX *s, int idx, char *arg);
void **SSL_CTX_set_info_callback**(SSL_CTX *ctx, void (*cb)(SSL *ssl, int cb, int ret));
void **SSL_CTX_set_msg_callback**(SSL_CTX *ctx, void (*cb)(int write_p, int version, int content_type, const void *buf, size_t len, SSL *ssl, void *arg));

void **SSL_CTX_set_msg_callback_arg**(SSL_CTX *ctx, void *arg);
void **SSL_CTX_set_options**(SSL_CTX *ctx, unsigned long op);
void **SSL_CTX_set_quiet_shutdown**(SSL_CTX *ctx, int mode);
void **SSL_CTX_set_read_ahead**(SSL_CTX *ctx, int m);
void **SSL_CTX_set_session_cache_mode**(SSL_CTX *ctx, int mode);
int **SSL_CTX_set_ssl_version**(SSL_CTX *ctx, const SSL_METHOD *meth);
void **SSL_CTX_set_timeout**(SSL_CTX *ctx, long t);
long **SSL_CTX_set_tmp_dh**(SSL_CTX* ctx, DH *dh);
long **SSL_CTX_set_tmp_dh_callback**(SSL_CTX *ctx, DH *(*cb)(void));
long **SSL_CTX_set_tmp_rsa**(SSL_CTX *ctx, RSA *rsa);
SSL_CTX_set_tmp_rsa_callback

long **SSL_CTX_set_tmp_rsa_callback**(SSL_CTX ***ctx**, RSA *(***cb**)(SSL ***ssl**, int **export**, int **keylength**));

Sets the callback which will be called when a temporary private key is required. The `export` flag will be set if the reason for needing a temp key is that an export ciphersuite is in use, in which case, `keylength` will contain the required keylength in bits. Generate a key of appropriate size (using ???) and return it.

SSL_set_tmp_rsa_callback

long **SSL_set_tmp_rsa_callback**(SSL *ssl, RSA *(*cb)(SSL *ssl, int export, int keylength));

The same as **SSL_CTX_set_tmp_rsa_callback**, except it operates on an SSL session instead of a context.

void **SSL_CTX_set_verify**(SSL_CTX *ctx, int mode, int (*cb);(void))
int **SSL_CTX_use_PrivateKey**(SSL_CTX *ctx, EVP_PKEY *pkey);
int **SSL_CTX_use_PrivateKey_ASN1**(int type, SSL_CTX *ctx, unsigned char *d, long len);
int **SSL_CTX_use_PrivateKey_file**(SSL_CTX *ctx, char *file, int type);
int **SSL_CTX_use_RSAPrivateKey**(SSL_CTX *ctx, RSA *rsa);
int **SSL_CTX_use_RSAPrivateKey_ASN1**(SSL_CTX *ctx, unsigned char *d, long len);
int **SSL_CTX_use_RSAPrivateKey_file**(SSL_CTX *ctx, char *file, int type);
int **SSL_CTX_use_certificate**(SSL_CTX *ctx, X509 *x);
int **SSL_CTX_use_certificate_ASN1**(SSL_CTX *ctx, int len, unsigned char *d);
int **SSL_CTX_use_certificate_file**(SSL_CTX *ctx, char *file, int type);
void **SSL_CTX_set_psk_client_callback**(SSL_CTX *ctx, unsigned int (*callback)(SSL *ssl, const char *hint, char *identity, unsigned int max_identity_len, unsigned char *psk, unsigned int max_psk_len));

int **SSL_CTX_use_psk_identity_hint**(SSL_CTX *ctx, const char *hint);
void **SSL_CTX_set_psk_server_callback**(SSL_CTX *ctx, unsigned int (*callback)(SSL *ssl, const char *identity, unsigned char *psk, int max_psk_len));

DEALING WITH SESSIONS

Here we document the various API functions which deal with the SSL/TLS sessions defined in the **SSL_SESSION** structures.

int **SSL_SESSION_cmp**(const SSL_SESSION *a, const SSL_SESSION *b);
void **SSL_SESSION_free**(SSL_SESSION *ss);
char ***SSL_SESSION_get_app_data**(SSL_SESSION *s);
char ***SSL_SESSION_get_ex_data**(const SSL_SESSION *s, int idx);
int **SSL_SESSION_get_ex_new_index**(long argl, char *argp, int (*new_func);(void), int (*dup_func)(void), void (*free_func)(void))

long **SSL_SESSION_get_time**(const SSL_SESSION *s);
long **SSL_SESSION_get_timeout**(const SSL_SESSION *s);
unsigned long **SSL_SESSION_hash**(const SSL_SESSION *a);
SSL_SESSION ***SSL_SESSION_new**(void);
int **SSL_SESSION_print**(BIO *bp, const SSL_SESSION *x);
int **SSL_SESSION_print_fp**(FILE *fp, const SSL_SESSION *x);
void **SSL_SESSION_set_app_data**(SSL_SESSION *s, char *a);
int **SSL_SESSION_set_ex_data**(SSL_SESSION *s, int idx, char *arg);
long **SSL_SESSION_set_time**(SSL_SESSION *s, long t);
long **SSL_SESSION_set_timeout**(SSL_SESSION *s, long t);

DEALING WITH CONNECTIONS

Here we document the various API functions which deal with the SSL/TLS connection defined in the **SSL** structure.

int **SSL_accept**(SSL *ssl);
int **SSL_add_dir_cert_subjects_to_stack**(STACK *stack, const char *dir);
int **SSL_add_file_cert_subjects_to_stack**(STACK *stack, const char *file);
int **SSL_add_client_CA**(SSL *ssl, X509 *x);
char ***SSL_alert_desc_string**(int value);
char ***SSL_alert_desc_string_long**(int value);
char ***SSL_alert_type_string**(int value);
char ***SSL_alert_type_string_long**(int value);
int **SSL_check_private_key**(const SSL *ssl);
void **SSL_clear**(SSL *ssl);
long **SSL_clear_num_renegotiations**(SSL *ssl);
int **SSL_connect**(SSL *ssl);
void **SSL_copy_session_id**(SSL *t, const SSL *f);
long **SSL_ctrl**(SSL *ssl, int cmd, long larg, char *parg);
int **SSL_do_handshake**(SSL *ssl);
SSL ***SSL_dup**(SSL *ssl);
STACK ***SSL_dup_CA_list**(STACK *sk);
void **SSL_free**(SSL *ssl);
SSL_CTX ***SSL_get_SSL_CTX**(const SSL *ssl);
char ***SSL_get_app_data**(SSL *ssl);

X509 ***SSL_get_certificate**(const SSL *ssl);
const char ***SSL_get_cipher**(const SSL *ssl);
int **SSL_get_cipher_bits**(const SSL *ssl, int *alg_bits);
char ***SSL_get_cipher_list**(const SSL *ssl, int n);
char ***SSL_get_cipher_name**(const SSL *ssl);
char ***SSL_get_cipher_version**(const SSL *ssl);
STACK ***SSL_get_ciphers**(const SSL *ssl);
STACK ***SSL_get_client_CA_list**(const SSL *ssl);
SSL_CIPHER ***SSL_get_current_cipher**(SSL *ssl);
long **SSL_get_default_timeout**(const SSL *ssl);
int **SSL_get_error**(const SSL *ssl, int i);
char ***SSL_get_ex_data**(const SSL *ssl, int idx);
int **SSL_get_ex_data_X509_STORE_CTX_idx**(void);
int **SSL_get_ex_new_index**(long argl, char *argp, int (*new_func);(void), int (*dup_func)(void), void (*free_func)(void))

int **SSL_get_fd**(const SSL *ssl);
void (***SSL_get_info_callback**(const SSL *ssl);)()
STACK ***SSL_get_peer_cert_chain**(const SSL *ssl);
X509 ***SSL_get_peer_certificate**(const SSL *ssl);
EVP_PKEY ***SSL_get_privatekey**(SSL *ssl);
int **SSL_get_quiet_shutdown**(const SSL *ssl);
BIO ***SSL_get_rbio**(const SSL *ssl);
int **SSL_get_read_ahead**(const SSL *ssl);
SSL_SESSION ***SSL_get_session**(const SSL *ssl);
char ***SSL_get_shared_ciphers**(const SSL *ssl, char *buf, int len);
int **SSL_get_shutdown**(const SSL *ssl);
const SSL_METHOD ***SSL_get_ssl_method**(SSL *ssl);
int **SSL_get_state**(const SSL *ssl);
long **SSL_get_time**(const SSL *ssl);
long **SSL_get_timeout**(const SSL *ssl);
int (***SSL_get_verify_callback**(const SSL *ssl))(int,X509_STORE_CTX *)
int **SSL_get_verify_mode**(const SSL *ssl);
long **SSL_get_verify_result**(const SSL *ssl);
char ***SSL_get_version**(const SSL *ssl);
BIO ***SSL_get_wbio**(const SSL *ssl);
int **SSL_in_accept_init**(SSL *ssl);
int **SSL_in_before**(SSL *ssl);
int **SSL_in_connect_init**(SSL *ssl);
int **SSL_in_init**(SSL *ssl);
int **SSL_is_init_finished**(SSL *ssl);
STACK ***SSL_load_client_CA_file**(char *file);
void **SSL_load_error_strings**(void);
SSL ***SSL_new**(SSL_CTX *ctx);
long **SSL_num_renegotiations**(SSL *ssl);
int **SSL_peek**(SSL *ssl, void *buf, int num);
int **SSL_pending**(const SSL *ssl);
int **SSL_read**(SSL *ssl, void *buf, int num);
int **SSL_renegotiate**(SSL *ssl);

char ***SSL_rstate_string**(SSL *ssl);
char ***SSL_rstate_string_long**(SSL *ssl);
long **SSL_session_reused**(SSL *ssl);
void **SSL_set_accept_state**(SSL *ssl);
void **SSL_set_app_data**(SSL *ssl, char *arg);
void **SSL_set_bio**(SSL *ssl, BIO *rbio, BIO *wbio);
int **SSL_set_cipher_list**(SSL *ssl, char *str);
void **SSL_set_client_CA_list**(SSL *ssl, STACK *list);
void **SSL_set_connect_state**(SSL *ssl);
int **SSL_set_ex_data**(SSL *ssl, int idx, char *arg);
int **SSL_set_fd**(SSL *ssl, int fd);
void **SSL_set_info_callback**(SSL *ssl, void (*cb);(void))
void **SSL_set_msg_callback**(SSL *ctx, void (*cb)(int write_p, int version, int content_type, const void *buf, size_t len, SSL *ssl, void *arg));

void **SSL_set_msg_callback_arg**(SSL *ctx, void *arg);
void **SSL_set_options**(SSL *ssl, unsigned long op);
void **SSL_set_quiet_shutdown**(SSL *ssl, int mode);
void **SSL_set_read_ahead**(SSL *ssl, int yes);
int **SSL_set_rfd**(SSL *ssl, int fd);
int **SSL_set_session**(SSL *ssl, SSL_SESSION *session);
void **SSL_set_shutdown**(SSL *ssl, int mode);
int **SSL_set_ssl_method**(SSL *ssl, const SSL_METHOD *meth);
void **SSL_set_time**(SSL *ssl, long t);
void **SSL_set_timeout**(SSL *ssl, long t);
void **SSL_set_verify**(SSL *ssl, int mode, int (*callback);(void))
void **SSL_set_verify_result**(SSL *ssl, long arg);
int **SSL_set_wfd**(SSL *ssl, int fd);
int **SSL_shutdown**(SSL *ssl);
int **SSL_state**(const SSL *ssl);
char ***SSL_state_string**(const SSL *ssl);
char ***SSL_state_string_long**(const SSL *ssl);
long **SSL_total_renegotiations**(SSL *ssl);
int **SSL_use_PrivateKey**(SSL *ssl, EVP_PKEY *pkey);
int **SSL_use_PrivateKey_ASN1**(int type, SSL *ssl, unsigned char *d, long len);
int **SSL_use_PrivateKey_file**(SSL *ssl, char *file, int type);
int **SSL_use_RSAPrivateKey**(SSL *ssl, RSA *rsa);
int **SSL_use_RSAPrivateKey_ASN1**(SSL *ssl, unsigned char *d, long len);
int **SSL_use_RSAPrivateKey_file**(SSL *ssl, char *file, int type);
int **SSL_use_certificate**(SSL *ssl, X509 *x);
int **SSL_use_certificate_ASN1**(SSL *ssl, int len, unsigned char *d);
int **SSL_use_certificate_file**(SSL *ssl, char *file, int type);
int **SSL_version**(const SSL *ssl);
int **SSL_want**(const SSL *ssl);
int **SSL_want_nothing**(const SSL *ssl);
int **SSL_want_read**(const SSL *ssl);
int **SSL_want_write**(const SSL *ssl);
int **SSL_want_x509_lookup**(const SSL *ssl);
int **SSL_write**(SSL *ssl, const void *buf, int num);
void **SSL_set_psk_client_callback**(SSL *ssl, unsigned int (*callback)(SSL *ssl, const char *hint, char *identity, unsigned int max_identity_len, unsigned char *psk, unsigned int max_psk_len));

int **SSL_use_psk_identity_hint**(SSL *ssl, const char *hint);
void **SSL_set_psk_server_callback**(SSL *ssl, unsigned int (*callback)(SSL *ssl, const char *identity, unsigned char *psk, int max_psk_len));

const char ***SSL_get_psk_identity_hint**(SSL *ssl);
const char ***SSL_get_psk_identity**(SSL *ssl);

SEE ALSO

openssl(1), crypto(3), SSL_accept(3), SSL_clear(3), SSL_connect(3), SSL_CIPHER_get_name(3), SSL_COMP_add_compression_method(3), SSL_CTX_add_extra_chain_cert(3), SSL_CTX_add_session(3), SSL_CTX_ctrl(3), SSL_CTX_flush_sessions(3), SSL_CTX_get_ex_new_index(3), SSL_CTX_get_verify_mode(3), SSL_CTX_load_verify_locations(3)SSL_CTX_new(3), SSL_CTX_sess_number(3), SSL_CTX_sess_set_cache_size(3), SSL_CTX_sess_set_get_cb(3), SSL_CTX_sessions(3), SSL_CTX_set_cert_store(3), SSL_CTX_set_cert_verify_callback(3), SSL_CTX_set_cipher_list(3), SSL_CTX_set_client_CA_list(3), SSL_CTX_set_client_cert_cb(3), SSL_CTX_set_default_passwd_cb(3), SSL_CTX_set_generate_session_id(3), SSL_CTX_set_info_callback(3), SSL_CTX_set_max_cert_list(3), SSL_CTX_set_mode(3), SSL_CTX_set_msg_callback(3), SSL_CTX_set_options(3), SSL_CTX_set_quiet_shutdown(3), SSL_CTX_set_read_ahead(3), SSL_CTX_set_session_cache_mode(3), SSL_CTX_set_session_id_context(3), SSL_CTX_set_ssl_version(3), SSL_CTX_set_timeout(3), SSL_CTX_set_tmp_rsa_callback(3), SSL_CTX_set_tmp_dh_callback(3), SSL_CTX_set_verify(3), SSL_CTX_use_certificate(3), SSL_alert_type_string(3), SSL_do_handshake(3), SSL_get_SSL_CTX(3), SSL_get_ciphers(3), SSL_get_client_CA_list(3), SSL_get_default_timeout(3), SSL_get_error(3), SSL_get_ex_data_X509_STORE_CTX_idx(3), SSL_get_ex_new_index(3), SSL_get_fd(3), SSL_get_peer_cert_chain(3), SSL_get_rbio(3), SSL_get_session(3), SSL_get_verify_result(3), SSL_get_version(3), SSL_library_init(3), SSL_load_client_CA_file(3), SSL_new(3), SSL_pending(3), SSL_read(3), SSL_rstate_string(3), SSL_session_reused(3), SSL_set_bio(3), SSL_set_connect_state(3), SSL_set_fd(3), SSL_set_session(3), SSL_set_shutdown(3), SSL_shutdown(3), SSL_state_string(3), SSL_want(3), SSL_write(3), SSL_SESSION_free(3), SSL_SESSION_get_ex_new_index(3), SSL_SESSION_get_time(3), d2i_SSL_SESSION(3), SSL_CTX_set_psk_client_callback(3), SSL_CTX_use_psk_identity_hint(3), SSL_get_psk_identity(3)

HISTORY

The ssl(3) document appeared in OpenSSL 0.9.2

Name

d2i_SSL_SESSION and i2d_SSL_SESSION — convert SSL_SESSION object from/to ASN1 representation

Synopsis

```
#include  <openssl/ssl.h>
```

```
SSL_SESSION *d2i_SSL_SESSION(SSL_SESSION **a, const unsigned char **pp, long length);
int i2d_SSL_SESSION(SSL_SESSION *in, unsigned char **pp);
```

DESCRIPTION

d2i_SSL_SESSION() transforms the external ASN1 representation of an SSL/TLS session, stored as binary data at location **pp** with length **length**, into an SSL_SESSION object.

i2d_SSL_SESSION() transforms the SSL_SESSION object **in** into the ASN1 representation and stores it into the memory location pointed to by **pp**. The length of the resulting ASN1 representation is returned. If **pp** is the NULL pointer, only the length is calculated and returned.

NOTES

The SSL_SESSION object is built from several malloc()ed parts, it can therefore not be moved, copied or stored directly. In order to store session data on disk or into a database, it must be transformed into a binary ASN1 representation.

When using d2i_SSL_SESSION(), the SSL_SESSION object is automatically allocated. The reference count is 1, so that the session must be explicitly removed using SSL_SESSION_free(3), unless the SSL_SESSION object is completely taken over, when being called inside the get_session_cb() (see SSL_CTX_sess_set_get_cb(3)).

SSL_SESSION objects keep internal link information about the session cache list, when being inserted into one SSL_CTX object's session cache. One SSL_SESSION object, regardless of its reference count, must therefore only be used with one SSL_CTX object (and the SSL objects created from this SSL_CTX object).

When using i2d_SSL_SESSION(), the memory location pointed to by **pp** must be large enough to hold the binary representation of the session. There is no known limit on the size of the created ASN1 representation, so the necessary amount of space should be obtained by first calling i2d_SSL_SESSION() with **pp=NULL**, and obtain the size needed, then allocate the memory and call i2d_SSL_SESSION() again. Note that this will advance the value contained in ***pp** so it is necessary to save a copy of the original allocation. For example: int i,j; char *p, *temp; i = i2d_SSL_SESSION(sess, NULL); p = temp = malloc(i); j = i2d_SSL_SESSION(sess, &temp); assert(i == j); assert(p+i == temp);

RETURN VALUES

d2i_SSL_SESSION() returns a pointer to the newly allocated SSL_SESSION object. In case of failure the NULL-pointer is returned and the error message can be retrieved from the error stack.

i2d_SSL_SESSION() returns the size of the ASN1 representation in bytes. When the session is not valid, **0** is returned and no operation is performed.

SEE ALSO

ssl(3), SSL_SESSION_free(3), SSL_CTX_sess_set_get_cb(3)

Name

SSL_accept — wait for a TLS/SSL client to initiate a TLS/SSL handshake

Synopsis

```
#include  <openssl/ssl.h>
```

```
int SSL_accept(SSL *ssl);
```

DESCRIPTION

SSL_accept() waits for a TLS/SSL client to initiate the TLS/SSL handshake. The communication channel must already have been set and assigned to the **ssl** by setting an underlying **BIO**.

NOTES

The behaviour of SSL_accept() depends on the underlying BIO.

If the underlying BIO is **blocking**, SSL_accept() will only return once the handshake has been finished or an error occurred, except for SGC (Server Gated Cryptography). For SGC, SSL_accept() may return with -1, but SSL_get_error() will yield **SSL_ERROR_WANT_READ/WRITE** and SSL_accept() should be called again.

If the underlying BIO is **non-blocking**, SSL_accept() will also return when the underlying BIO could not satisfy the needs of SSL_accept() to continue the handshake, indicating the problem by the return value -1. In this case a call to SSL_get_error() with the return value of SSL_accept() will yield **SSL_ERROR_WANT_READ** or **SSL_ERROR_WANT_WRITE**. The calling process then must repeat the call after taking appropriate action to satisfy the needs of SSL_accept(). The action depends on the underlying BIO. When using a non-blocking socket, nothing is to be done, but select() can be used to check for the required condition. When using a buffering BIO, like a BIO pair, data must be written into or retrieved out of the BIO before being able to continue.

RETURN VALUES

The following return values can occur:

0 The TLS/SSL handshake was not successful but was shut down controlled and by the specifications of the TLS/SSL protocol. Call SSL_get_error() with the return value **ret** to find out the reason.

1 The TLS/SSL handshake was successfully completed, a TLS/SSL connection has been established.

<0 The TLS/SSL handshake was not successful because a fatal error occurred either at the protocol level or a connection failure occurred. The shutdown was not clean. It can also occur of action is need to continue the operation for non-blocking BIOs. Call SSL_get_error() with the return value **ret** to find out the reason.

SEE ALSO

SSL_get_error(3), SSL_connect(3), SSL_shutdown(3), ssl(3), bio(3), SSL_set_connect_state(3), SSL_do_handshake(3), SSL_CTX_new(3)

Name

SSL_alert_type_string, SSL_alert_type_string_long, SSL_alert_desc_string and SSL_alert_desc_string_long — get textual description of alert information

Synopsis

```
#include  <openssl/ssl.h>
```

```
const char *SSL_alert_type_string(int value);
const char *SSL_alert_type_string_long(int value);
```

```
const char *SSL_alert_desc_string(int value);
const char *SSL_alert_desc_string_long(int value);
```

DESCRIPTION

SSL_alert_type_string() returns a one letter string indicating the type of the alert specified by **value**.

SSL_alert_type_string_long() returns a string indicating the type of the alert specified by **value**.

SSL_alert_desc_string() returns a two letter string as a short form describing the reason of the alert specified by **value**.

SSL_alert_desc_string_long() returns a string describing the reason of the alert specified by **value**.

NOTES

When one side of an SSL/TLS communication wants to inform the peer about a special situation, it sends an alert. The alert is sent as a special message and does not influence the normal data stream (unless its contents results in the communication being canceled).

A warning alert is sent, when a non-fatal error condition occurs. The "close notify" alert is sent as a warning alert. Other examples for non-fatal errors are certificate errors ("certificate expired", "unsupported certificate"), for which a warning alert may be sent. (The sending party may however decide to send a fatal error.) The receiving side may cancel the connection on reception of a warning alert on it discretion.

Several alert messages must be sent as fatal alert messages as specified by the TLS RFC. A fatal alert always leads to a connection abort.

RETURN VALUES

The following strings can occur for SSL_alert_type_string() or SSL_alert_type_string_long():

"W"/"warning"

"F"/"fatal"

"U"/"unknown" This indicates that no support is available for this alert type. Probably **value** does not contain a correct alert message.

The following strings can occur for SSL_alert_desc_string() or SSL_alert_desc_string_long():

"CN"/"close notify" The connection shall be closed. This is a warning alert.

"UM"/"unexpected message" An inappropriate message was received. This alert is always fatal and should never be observed in communication between proper implementations.

"BM"/"bad record mac" This alert is returned if a record is received with an incorrect MAC. This message is always fatal.

"DF"/"decompression failure"

The decompression function received improper input (e.g. data that would expand to excessive length). This message is always fatal.

"HF"/"handshake failure"

Reception of a handshake_failure alert message indicates that the sender was unable to negotiate an acceptable set of security parameters given the options available. This is a fatal error.

"NC"/"no certificate"

A client, that was asked to send a certificate, does not send a certificate (SSLv3 only).

"BC"/"bad certificate"

A certificate was corrupt, contained signatures that did not verify correctly, etc

"UC"/"unsupported certificate"

A certificate was of an unsupported type.

"CR"/"certificate revoked"

A certificate was revoked by its signer.

"CE"/"certificate expired"

A certificate has expired or is not currently valid.

"CU"/"certificate unknown"

Some other (unspecified) issue arose in processing the certificate, rendering it unacceptable.

"IP"/"illegal parameter"

A field in the handshake was out of range or inconsistent with other fields. This is always fatal.

"DC"/"decryption failed"

A TLSCiphertext decrypted in an invalid way: either it wasn't an even multiple of the block length or its padding values, when checked, weren't correct. This message is always fatal.

"RO"/"record overflow"

A TLSCiphertext record was received which had a length more than 2^14+2048 bytes, or a record decrypted to a TLSCompressed record with more than 2^14+1024 bytes. This message is always fatal.

"CA"/"unknown CA"

A valid certificate chain or partial chain was received, but the certificate was not accepted because the CA certificate could not be located or couldn't be matched with a known, trusted CA. This message is always fatal.

"AD"/"access denied"

A valid certificate was received, but when access control was applied, the sender decided not to proceed with negotiation. This message is always fatal.

"DE"/"decode error"

A message could not be decoded because some field was out of the specified range or the length of the message was incorrect. This message is always fatal.

"CY"/"decrypt error"

A handshake cryptographic operation failed, including being unable to correctly verify a signature, decrypt a key exchange, or validate a finished message.

"ER"/"export restriction"

A negotiation not in compliance with export restrictions was detected; for example, attempting to transfer a 1024 bit ephemeral RSA key for the RSA_EXPORT handshake method. This message is always fatal.

"PV"/"protocol version"

The protocol version the client has attempted to negotiate is recognized, but not supported. (For example, old protocol versions might be avoided for security reasons). This message is always fatal.

"IS"/"insufficient security"

Returned instead of handshake_failure when a negotiation has failed specifically because the server requires ciphers more secure than those supported by the client. This message is always fatal.

"IE"/"internal error"

An internal error unrelated to the peer or the correctness of the protocol makes it impossible to continue (such as a memory allocation failure). This message is always fatal.

"US"/"user canceled"

This handshake is being canceled for some reason unrelated to a protocol failure. If the user cancels an operation after the handshake is complete, just closing the connection by sending a close_notify is more appropriate. This alert should be followed by a close_notify. This message is generally a warning.

"NR"/"no renegotiation"
Sent by the client in response to a hello request or by the server in response to a client hello after initial handshaking. Either of these would normally lead to renegotiation; when that is not appropriate, the recipient should respond with this alert; at that point, the original requester can decide whether to proceed with the connection. One case where this would be appropriate would be where a server has spawned a process to satisfy a request; the process might receive security parameters (key length, authentication, etc.) at startup and it might be difficult to communicate changes to these parameters after that point. This message is always a warning.

"UP"/"unknown PSK identity"
Sent by the server to indicate that it does not recognize a PSK identity or an SRP identity.

"UK"/"unknown"
This indicates that no description is available for this alert type. Probably **value** does not contain a correct alert message.

SEE ALSO

ssl(3), SSL_CTX_set_info_callback(3)

Name

SSL_CIPHER_get_name, SSL_CIPHER_get_bits, SSL_CIPHER_get_version and SSL_CIPHER_description — get SSL_CIPHER properties

Synopsis

```
#include  <openssl/ssl.h>
```

```
const char *SSL_CIPHER_get_name(const SSL_CIPHER *cipher);
int SSL_CIPHER_get_bits(const SSL_CIPHER *cipher, int *alg_bits);
char *SSL_CIPHER_get_version(const SSL_CIPHER *cipher);
char *SSL_CIPHER_description(const SSL_CIPHER *cipher, char *buf, int size);
```

DESCRIPTION

SSL_CIPHER_get_name() returns a pointer to the name of **cipher**. If the argument is the NULL pointer, a pointer to the constant value "NONE" is returned.

SSL_CIPHER_get_bits() returns the number of secret bits used for **cipher**. If **alg_bits** is not NULL, it contains the number of bits processed by the chosen algorithm. If **cipher** is NULL, 0 is returned.

SSL_CIPHER_get_version() returns string which indicates the SSL/TLS protocol version that first defined the cipher. This is currently **SSLv2** or **TLSv1/SSLv3**. In some cases it should possibly return "TLSv1.2" but does not; use SSL_CIPHER_description() instead. If **cipher** is NULL, "(NONE)" is returned.

SSL_CIPHER_description() returns a textual description of the cipher used into the buffer **buf** of length **len** provided. **len** must be at least 128 bytes, otherwise a pointer to the string "Buffer too small" is returned. If **buf** is NULL, a buffer of 128 bytes is allocated using OPENSSL_malloc(). If the allocation fails, a pointer to the string "OPENSSL_malloc Error" is returned.

NOTES

The number of bits processed can be different from the secret bits. An export cipher like e.g. EXP-RC4-MD5 has only 40 secret bits. The algorithm does use the full 128 bits (which would be returned for **alg_bits**), of which however 88bits are fixed. The search space is hence only 40 bits.

The string returned by SSL_CIPHER_description() in case of success consists of cleartext information separated by one or more blanks in the following sequence:

<ciphername>

> Textual representation of the cipher name.

<protocol version>

> Protocol version: **SSLv2**, **SSLv3**, **TLSv1.2**. The TLSv1.0 ciphers are flagged with SSLv3. No new ciphers were added by TLSv1.1.

Kx=<key exchange>

> Key exchange method: **RSA** (for export ciphers as **RSA(512)** or **RSA(1024)**), **DH** (for export ciphers as **DH(512)** or **DH(1024)**), **DH/RSA**, **DH/DSS**, **Fortezza**.

Au=<authentication>

> Authentication method: **RSA**, **DSS**, **DH**, **None**. None is the representation of anonymous ciphers.

Enc=<symmetric encryption method>

Encryption method with number of secret bits: **DES(40)**, **DES(56)**, **3DES(168)**, **RC4(40)**, **RC4(56)**, **RC4(64)**, **RC4(128)**, **RC2(40)**, **RC2(56)**, **RC2(128)**, **IDEA(128)**, **Fortezza**, **None**.

Mac=<message authentication code>

Message digest: **MD5**, **SHA1**.

<export flag>

If the cipher is flagged exportable with respect to old US crypto regulations, the word "**export**" is printed.

EXAMPLES

Some examples for the output of SSL_CIPHER_description():

```
EDH-RSA-DES-CBC3-SHA    SSLv3 Kx=DH       Au=RSA  Enc=3DES(168) Mac=SHA1
EDH-DSS-DES-CBC3-SHA    SSLv3 Kx=DH       Au=DSS  Enc=3DES(168) Mac=SHA1
RC4-MD5                 SSLv3 Kx=RSA      Au=RSA  Enc=RC4(128)  Mac=MD5
EXP-RC4-MD5             SSLv3 Kx=RSA(512) Au=RSA  Enc=RC4(40)   Mac=MD5  export
```

A comp[lete list can be retrieved by invoking the following command:

```
openssl ciphers -v ALL
```

BUGS

If SSL_CIPHER_description() is called with **cipher** being NULL, the library crashes.

If SSL_CIPHER_description() cannot handle a built-in cipher, the according description of the cipher property is **unknown**. This case should not occur.

RETURN VALUES

See DESCRIPTION

SEE ALSO

ssl(3), SSL_get_current_cipher(3), SSL_get_ciphers(3), ciphers(1)

Name

SSL_clear — reset SSL object to allow another connection

Synopsis

```
#include  <openssl/ssl.h>
```

```
int SSL_clear(SSL *ssl);
```

DESCRIPTION

Reset **ssl** to allow another connection. All settings (method, ciphers, BIOs) are kept.

NOTES

SSL_clear is used to prepare an SSL object for a new connection. While all settings are kept, a side effect is the handling of the current SSL session. If a session is still **open**, it is considered bad and will be removed from the session cache, as required by RFC2246. A session is considered open, if SSL_shutdown(3) was not called for the connection or at least SSL_set_shutdown(3) was used to set the SSL_SENT_SHUTDOWN state.

If a session was closed cleanly, the session object will be kept and all settings corresponding. This explicitly means, that e.g. the special method used during the session will be kept for the next handshake. So if the session was a TLSv1 session, a SSL client object will use a TLSv1 client method for the next handshake and a SSL server object will use a TLSv1 server method, even if SSLv23_*_methods were chosen on startup. This will might lead to connection failures (see SSL_new(3)) for a description of the method's properties.

WARNINGS

SSL_clear() resets the SSL object to allow for another connection. The reset operation however keeps several settings of the last sessions (some of these settings were made automatically during the last handshake). It only makes sense for a new connection with the exact same peer that shares these settings, and may fail if that peer changes its settings between connections. Use the sequence SSL_get_session(3); SSL_new(3); SSL_set_session(3); SSL_free(3) instead to avoid such failures (or simply SSL_free(3); SSL_new(3) if session reuse is not desired).

RETURN VALUES

The following return values can occur:

0 The SSL_clear() operation could not be performed. Check the error stack to find out the reason.

1 The SSL_clear() operation was successful.

SSL_new(3), SSL_free(3), SSL_shutdown(3), SSL_set_shutdown(3), SSL_CTX_set_options(3), ssl(3), SSL_CTX_set_client_cert_cb(3)

Name

SSL_COMP_add_compression_method — handle SSL/TLS integrated compression methods

Synopsis

```
#include  <openssl/ssl.h>
```

```
int SSL_COMP_add_compression_method(int id, COMP_METHOD *cm);
```

DESCRIPTION

SSL_COMP_add_compression_method() adds the compression method **cm** with the identifier **id** to the list of available compression methods. This list is globally maintained for all SSL operations within this application. It cannot be set for specific SSL_CTX or SSL objects.

NOTES

The TLS standard (or SSLv3) allows the integration of compression methods into the communication. The TLS RFC does however not specify compression methods or their corresponding identifiers, so there is currently no compatible way to integrate compression with unknown peers. It is therefore currently not recommended to integrate compression into applications. Applications for non-public use may agree on certain compression methods. Using different compression methods with the same identifier will lead to connection failure.

An OpenSSL client speaking a protocol that allows compression (SSLv3, TLSv1) will unconditionally send the list of all compression methods enabled with SSL_COMP_add_compression_method() to the server during the handshake. Unlike the mechanisms to set a cipher list, there is no method available to restrict the list of compression method on a per connection basis.

An OpenSSL server will match the identifiers listed by a client against its own compression methods and will unconditionally activate compression when a matching identifier is found. There is no way to restrict the list of compression methods supported on a per connection basis.

The OpenSSL library has the compression methods **COMP_rle()** and (when especially enabled during compilation) **COMP_zlib()** available.

WARNINGS

Once the identities of the compression methods for the TLS protocol have been standardized, the compression API will most likely be changed. Using it in the current state is not recommended.

RETURN VALUES

SSL_COMP_add_compression_method() may return the following values:

0 The operation succeeded.

1 The operation failed. Check the error queue to find out the reason.

SEE ALSO

ssl(3)

Name

SSL_connect — initiate the TLS/SSL handshake with an TLS/SSL server

Synopsis

```
#include  <openssl/ssl.h>
```

```
int SSL_connect(SSL *ssl);
```

DESCRIPTION

SSL_connect() initiates the TLS/SSL handshake with a server. The communication channel must already have been set and assigned to the **ssl** by setting an underlying **BIO**.

NOTES

The behaviour of SSL_connect() depends on the underlying BIO.

If the underlying BIO is **blocking**, SSL_connect() will only return once the handshake has been finished or an error occurred.

If the underlying BIO is **non-blocking**, SSL_connect() will also return when the underlying BIO could not satisfy the needs of SSL_connect() to continue the handshake, indicating the problem by the return value -1. In this case a call to SSL_get_error() with the return value of SSL_connect() will yield **SSL_ERROR_WANT_READ** or **SSL_ERROR_WANT_WRITE**. The calling process then must repeat the call after taking appropriate action to satisfy the needs of SSL_connect(). The action depends on the underlying BIO. When using a non-blocking socket, nothing is to be done, but select() can be used to check for the required condition. When using a buffering BIO, like a BIO pair, data must be written into or retrieved out of the BIO before being able to continue.

RETURN VALUES

The following return values can occur:

0 The TLS/SSL handshake was not successful but was shut down controlled and by the specifications of the TLS/SSL protocol. Call SSL_get_error() with the return value **ret** to find out the reason.

1 The TLS/SSL handshake was successfully completed, a TLS/SSL connection has been established.

<0 The TLS/SSL handshake was not successful, because a fatal error occurred either at the protocol level or a connection failure occurred. The shutdown was not clean. It can also occur of action is need to continue the operation for non-blocking BIOs. Call SSL_get_error() with the return value **ret** to find out the reason.

SEE ALSO

SSL_get_error(3), SSL_accept(3), SSL_shutdown(3), ssl(3), bio(3), SSL_set_connect_state(3), SSL_do_handshake(3), SSL_CTX_new(3)

Name

SSL_CTX_add_extra_chain_cert and SSL_CTX_clear_extra_chain_certs — add or clear extra chain certificates

Synopsis

```
#include  <openssl/ssl.h>
```

```
long SSL_CTX_add_extra_chain_cert(SSL_CTX *ctx, X509 *x509);
long SSL_CTX_clear_extra_chain_certs(SSL_CTX *ctx);
```

DESCRIPTION

SSL_CTX_add_extra_chain_cert() adds the certificate **x509** to the extra chain certificates associated with **ctx**. Several certificates can be added one after another.

SSL_CTX_clear_extra_chain_certs() clears all extra chain certificates associated with **ctx**.

These functions are implemented as macros.

NOTES

When sending a certificate chain, extra chain certificates are sent in order following the end entity certificate.

If no chain is specified, the library will try to complete the chain from the available CA certificates in the trusted CA storage, see SSL_CTX_load_verify_locations(3).

The **x509** certificate provided to SSL_CTX_add_extra_chain_cert() will be freed by the library when the **SSL_CTX** is destroyed. An application **should not** free the **x509** object.

RESTRICTIONS

Only one set of extra chain certificates can be specified per SSL_CTX structure. Different chains for different certificates (for example if both RSA and DSA certificates are specified by the same server) or different SSL structures with the same parent SSL_CTX cannot be specified using this function.

RETURN VALUES

SSL_CTX_add_extra_chain_cert() and SSL_CTX_clear_extra_chain_certs() return 1 on success and 0 for failure. Check out the error stack to find out the reason for failure.

SEE ALSO

ssl(3), SSL_CTX_use_certificate(3), SSL_CTX_set_client_cert_cb(3), SSL_CTX_load_verify_locations(3)

Name

SSL_CTX_add_session, SSL_add_session, SSL_CTX_remove_session and SSL_remove_session — manipulate session cache

Synopsis

```
#include  <openssl/ssl.h>
```

```
int SSL_CTX_add_session(SSL_CTX *ctx, SSL_SESSION *c);
int SSL_add_session(SSL_CTX *ctx, SSL_SESSION *c);
```

```
int SSL_CTX_remove_session(SSL_CTX *ctx, SSL_SESSION *c);
int SSL_remove_session(SSL_CTX *ctx, SSL_SESSION *c);
```

DESCRIPTION

SSL_CTX_add_session() adds the session **c** to the context **ctx**. The reference count for session **c** is incremented by 1. If a session with the same session id already exists, the old session is removed by calling SSL_SESSION_free(3).

SSL_CTX_remove_session() removes the session **c** from the context **ctx**. SSL_SESSION_free(3) is called once for **c**.

SSL_add_session() and SSL_remove_session() are synonyms for their SSL_CTX_*() counterparts.

NOTES

When adding a new session to the internal session cache, it is examined whether a session with the same session id already exists. In this case it is assumed that both sessions are identical. If the same session is stored in a different SSL_SESSION object, The old session is removed and replaced by the new session. If the session is actually identical (the SSL_SESSION object is identical), SSL_CTX_add_session() is a no-op, and the return value is 0.

If a server SSL_CTX is configured with the SSL_SESS_CACHE_NO_INTERNAL_STORE flag then the internal cache will not be populated automatically by new sessions negotiated by the SSL/TLS implementation, even though the internal cache will be searched automatically for session-resume requests (the latter can be suppressed by SSL_SESS_CACHE_NO_INTERNAL_LOOKUP). So the application can use SSL_CTX_add_session() directly to have full control over the sessions that can be resumed if desired.

RETURN VALUES

The following values are returned by all functions:

0
```
The operation failed. In case of the add operation, it was tried to add
the same (identical) session twice. In case of the remove operation, the
session was not found in the cache.
```

1
```
The operation succeeded.
```

SEE ALSO

ssl(3), SSL_CTX_set_session_cache_mode(3), SSL_SESSION_free(3)

Name

SSL_CTX_ctrl, SSL_CTX_callback_ctrl, SSL_ctrl and SSL_callback_ctrl — internal handling functions for SSL_CTX and SSL objects

Synopsis

```
#include  <openssl/ssl.h>
```

```
long SSL_CTX_ctrl(SSL_CTX *ctx, int cmd, long larg, void *parg);
long SSL_CTX_callback_ctrl(SSL_CTX *, int cmd, void (*fp)());
```

```
long SSL_ctrl(SSL *ssl, int cmd, long larg, void *parg);
long SSL_callback_ctrl(SSL *, int cmd, void (*fp)());
```

DESCRIPTION

The SSL_*_ctrl() family of functions is used to manipulate settings of the SSL_CTX and SSL objects. Depending on the command **cmd** the arguments **larg**, **parg**, or **fp** are evaluated. These functions should never be called directly. All functionalities needed are made available via other functions or macros.

RETURN VALUES

The return values of the SSL*_ctrl() functions depend on the command supplied via the **cmd** parameter.

SEE ALSO

ssl(3)

Name

SSL_CTX_flush_sessions and SSL_flush_sessions — remove expired sessions

Synopsis

```
#include  <openssl/ssl.h>
```

```
void SSL_CTX_flush_sessions(SSL_CTX *ctx, long tm);
void SSL_flush_sessions(SSL_CTX *ctx, long tm);
```

DESCRIPTION

SSL_CTX_flush_sessions() causes a run through the session cache of **ctx** to remove sessions expired at time **tm**.

SSL_flush_sessions() is a synonym for SSL_CTX_flush_sessions().

NOTES

If enabled, the internal session cache will collect all sessions established up to the specified maximum number (see SSL_CTX_sess_set_cache_size()). As sessions will not be reused ones they are expired, they should be removed from the cache to save resources. This can either be done automatically whenever 255 new sessions were established (see SSL_CTX_set_session_cache_mode(3)) or manually by calling SSL_CTX_flush_sessions().

The parameter **tm** specifies the time which should be used for the expiration test, in most cases the actual time given by time(0) will be used.

SSL_CTX_flush_sessions() will only check sessions stored in the internal cache. When a session is found and removed, the remove_session_cb is however called to synchronize with the external cache (see SSL_CTX_sess_set_get_cb(3)).

RETURN VALUES

SEE ALSO

ssl(3), SSL_CTX_set_session_cache_mode(3), SSL_CTX_set_timeout(3), SSL_CTX_sess_set_get_cb(3)

Name

SSL_CTX_free — free an allocated SSL_CTX object

Synopsis

```
#include  <openssl/ssl.h>
```

```
void SSL_CTX_free(SSL_CTX *ctx);
```

DESCRIPTION

SSL_CTX_free() decrements the reference count of **ctx**, and removes the SSL_CTX object pointed to by **ctx** and frees up the allocated memory if the the reference count has reached 0.

It also calls the free()ing procedures for indirectly affected items, if applicable: the session cache, the list of ciphers, the list of Client CAs, the certificates and keys.

WARNINGS

If a session-remove callback is set (SSL_CTX_sess_set_remove_cb()), this callback will be called for each session being freed from **ctx**'s session cache. This implies, that all corresponding sessions from an external session cache are removed as well. If this is not desired, the user should explicitly unset the callback by calling SSL_CTX_sess_set_remove_cb(**ctx**, NULL) prior to calling SSL_CTX_free().

RETURN VALUES

SSL_CTX_free() does not provide diagnostic information.

SEE ALSO

SSL_CTX_new(3), ssl(3), SSL_CTX_sess_set_get_cb(3)

Name

SSL_CTX_get_ex_new_index, SSL_CTX_set_ex_data and SSL_CTX_get_ex_data — internal application specific data functions

Synopsis

```
#include  <openssl/ssl.h>
```

```
int SSL_CTX_get_ex_new_index(long argl, void *argp,
               CRYPTO_EX_new *new_func,
               CRYPTO_EX_dup *dup_func,
               CRYPTO_EX_free *free_func);
```

```
int SSL_CTX_set_ex_data(SSL_CTX *ctx, int idx, void *arg);
```

```
void *SSL_CTX_get_ex_data(const SSL_CTX *ctx, int idx);
```

```
typedef int new_func(void *parent, void *ptr, CRYPTO_EX_DATA *ad,
               int idx, long argl, void *argp);
typedef void free_func(void *parent, void *ptr, CRYPTO_EX_DATA *ad,
               int idx, long argl, void *argp);
typedef int dup_func(CRYPTO_EX_DATA *to, CRYPTO_EX_DATA *from, void *from_d,
               int idx, long argl, void *argp);
```

DESCRIPTION

Several OpenSSL structures can have application specific data attached to them. These functions are used internally by OpenSSL to manipulate application specific data attached to a specific structure.

SSL_CTX_get_ex_new_index() is used to register a new index for application specific data.

SSL_CTX_set_ex_data() is used to store application data at **arg** for **idx** into the **ctx** object.

SSL_CTX_get_ex_data() is used to retrieve the information for **idx** from **ctx**.

A detailed description for the ***_get_ex_new_index()** functionality can be found in RSA_get_ex_new_index(3). The ***_get_ex_data()** and ***_set_ex_data()** functionality is described in CRYPTO_set_ex_data(3).

SEE ALSO

ssl(3), RSA_get_ex_new_index(3), CRYPTO_set_ex_data(3)

Name

SSL_CTX_get_verify_mode, SSL_get_verify_mode, SSL_CTX_get_verify_depth, SSL_get_verify_depth, SSL_get_verify_callback and SSL_CTX_get_verify_callback — get currently set verification parameters

Synopsis

```
#include  <openssl/ssl.h>
```

```
int SSL_CTX_get_verify_mode(const SSL_CTX *ctx);
int SSL_get_verify_mode(const SSL *ssl);
int SSL_CTX_get_verify_depth(const SSL_CTX *ctx);
int SSL_get_verify_depth(const SSL *ssl);
int (*SSL_CTX_get_verify_callback(const SSL_CTX *ctx))(int, X509_STORE_CTX *);
int (*SSL_get_verify_callback(const SSL *ssl))(int, X509_STORE_CTX *);
```

DESCRIPTION

SSL_CTX_get_verify_mode() returns the verification mode currently set in **ctx**.

SSL_get_verify_mode() returns the verification mode currently set in **ssl**.

SSL_CTX_get_verify_depth() returns the verification depth limit currently set in **ctx**. If no limit has been explicitly set, -1 is returned and the default value will be used.

SSL_get_verify_depth() returns the verification depth limit currently set in **ssl**. If no limit has been explicitly set, -1 is returned and the default value will be used.

SSL_CTX_get_verify_callback() returns a function pointer to the verification callback currently set in **ctx**. If no callback was explicitly set, the NULL pointer is returned and the default callback will be used.

SSL_get_verify_callback() returns a function pointer to the verification callback currently set in **ssl**. If no callback was explicitly set, the NULL pointer is returned and the default callback will be used.

RETURN VALUES

See DESCRIPTION

SEE ALSO

ssl(3), SSL_CTX_set_verify(3)

Name

SSL_CTX_load_verify_locations — set default locations for trusted CA certificates

Synopsis

```
#include  <openssl/ssl.h>
```

```
int SSL_CTX_load_verify_locations(SSL_CTX *ctx, const char *CAfile,
                                  const char *CApath);
```

DESCRIPTION

SSL_CTX_load_verify_locations() specifies the locations for **ctx**, at which CA certificates for verification purposes are located. The certificates available via **CAfile** and **CApath** are trusted.

NOTES

If **CAfile** is not NULL, it points to a file of CA certificates in PEM format. The file can contain several CA certificates identified by

```
-----BEGIN CERTIFICATE-----
… (CA certificate in base64 encoding) …
-----END CERTIFICATE-----
```

sequences. Before, between, and after the certificates text is allowed which can be used e.g. for descriptions of the certificates.

The **CAfile** is processed on execution of the SSL_CTX_load_verify_locations() function.

If **CApath** is not NULL, it points to a directory containing CA certificates in PEM format. The files each contain one CA certificate. The files are looked up by the CA subject name hash value, which must hence be available. If more than one CA certificate with the same name hash value exist, the extension must be different (e.g. 9d66eef0.0, 9d66eef0.1 etc). The search is performed in the ordering of the extension number, regardless of other properties of the certificates. Use the **c_rehash** utility to create the necessary links.

The certificates in **CApath** are only looked up when required, e.g. when building the certificate chain or when actually performing the verification of a peer certificate.

When looking up CA certificates, the OpenSSL library will first search the certificates in **CAfile**, then those in **CApath**. Certificate matching is done based on the subject name, the key identifier (if present), and the serial number as taken from the certificate to be verified. If these data do not match, the next certificate will be tried. If a first certificate matching the parameters is found, the verification process will be performed; no other certificates for the same parameters will be searched in case of failure.

In server mode, when requesting a client certificate, the server must send the list of CAs of which it will accept client certificates. This list is not influenced by the contents of **CAfile** or **CApath** and must explicitly be set using the SSL_CTX_set_client_CA_list(3) family of functions.

When building its own certificate chain, an OpenSSL client/server will try to fill in missing certificates from **CAfile**/**CApath**, if the certificate chain was not explicitly specified (see SSL_CTX_add_extra_chain_cert(3), SSL_CTX_use_certificate(3).

WARNINGS

If several CA certificates matching the name, key identifier, and serial number condition are available, only the first one will be examined. This may lead to unexpected results if the same CA certificate is available with different expiration dates. If a "certificate expired" verification error occurs, no other certificate will be searched. Make sure to not have expired certificates mixed with valid ones.

EXAMPLES

Generate a CA certificate file with descriptive text from the CA certificates ca1.pem ca2.pem ca3.pem:

```
#!/bin/sh
rm CAfile.pem
for i in ca1.pem ca2.pem ca3.pem ; do
  openssl x509 -in $i -text >> CAfile.pem
done
```

Prepare the directory /some/where/certs containing several CA certificates for use as **CApath**:

```
cd /some/where/certs
c_rehash .
```

RETURN VALUES

The following return values can occur:

0 The operation failed because **CAfile** and **CApath** are NULL or the processing at one of the locations specified failed. Check the error stack to find out the reason.

1 The operation succeeded.

SEE ALSO

ssl(3), SSL_CTX_set_client_CA_list(3), SSL_get_client_CA_list(3), SSL_CTX_use_certificate(3), SSL_CTX_add_extra_chain_cert(3), SSL_CTX_set_cert_store(3)

Name

SSL_CTX_new, SSLv23_method, SSLv23_server_method, SSLv23_client_method, TLSv1_2_method, TLSv1_2_server_method, TLSv1_2_client_method, TLSv1_1_method, TLSv1_1_server_method, TLSv1_1_client_method, TLSv1_method, TLSv1_server_method, TLSv1_client_method, SSLv3_method, SSLv3_server_method, SSLv3_client_method, SSLv2_method, SSLv2_server_method, SSLv2_client_method, DTLSv1_method, DTLSv1_server_method and DTLSv1_client_method — create a new SSL_CTX object as framework for TLS/SSL enabled functions

Synopsis

```
#include  <openssl/ssl.h>
```

```
SSL_CTX *SSL_CTX_new(const SSL_METHOD *method);
const SSL_METHOD *SSLv23_method(void);
const SSL_METHOD *SSLv23_server_method(void);
const SSL_METHOD *SSLv23_client_method(void);
const SSL_METHOD *TLSv1_2_method(void);
const SSL_METHOD *TLSv1_2_server_method(void);
const SSL_METHOD *TLSv1_2_client_method(void);
const SSL_METHOD *TLSv1_1_method(void);
const SSL_METHOD *TLSv1_1_server_method(void);
const SSL_METHOD *TLSv1_1_client_method(void);
const SSL_METHOD *TLSv1_method(void);
const SSL_METHOD *TLSv1_server_method(void);
const SSL_METHOD *TLSv1_client_method(void);
#ifndef OPENSSL_NO_SSL3_METHOD
const SSL_METHOD *SSLv3_method(void);
const SSL_METHOD *SSLv3_server_method(void);
const SSL_METHOD *SSLv3_client_method(void);
#endif
#ifndef OPENSSL_NO_SSL2
const SSL_METHOD *SSLv2_method(void);
const SSL_METHOD *SSLv2_server_method(void);
const SSL_METHOD *SSLv2_client_method(void);
#endif
```

```
const SSL_METHOD *DTLSv1_method(void);
const SSL_METHOD *DTLSv1_server_method(void);
const SSL_METHOD *DTLSv1_client_method(void);
```

DESCRIPTION

SSL_CTX_new() creates a new **SSL_CTX** object as framework to establish TLS/SSL enabled connections.

NOTES

The SSL_CTX object uses **method** as connection method. The methods exist in a generic type (for client and server use), a server only type, and a client only type. **method** can be of the following types:

SSLv23_method(), SSLv23_server_method(), SSLv23_client_method()

> These are the general-purpose *version-flexible* SSL/TLS methods. The actual protocol version used will be negotiated to the highest version mutually supported by the client and the server. The supported protocols are SSLv2, SSLv3, TLSv1, TLSv1.1 and TLSv1.2. Most applications should use these method, and avoid the version specific methods described below.
>
> The list of protocols available can be further limited using the **SSL_OP_NO_SSLv2**, **SSL_OP_NO_SSLv3**, **SSL_OP_NO_TLSv1**, **SSL_OP_NO_TLSv1_1** and **SSL_OP_NO_TLSv1_2** options of the SSL_CTX_set_options(3) or SSL_set_options(3) functions. Clients should avoid creating "holes" in the set of protocols they support, when disabling a protocol, make sure that you also disable either all previous or all subsequent protocol versions. In clients, when a protocol version is disabled without disabling *all* previous protocol versions, the effect is to also disable all subsequent protocol versions.

The SSLv2 and SSLv3 protocols are deprecated and should generally not be used. Applications should typically use SSL_CTX_set_options(3) in combination with the **SSL_OP_NO_SSLv3** flag to disable negotiation of SSLv3 via the above *version-flexible* SSL/TLS methods. The **SSL_OP_NO_SSLv2** option is set by default, and would need to be cleared via SSL_CTX_clear_options(3) in order to enable negotiation of SSLv2.

TLSv1_2_method(), TLSv1_2_server_method(), TLSv1_2_client_method()

A TLS/SSL connection established with these methods will only understand the TLSv1.2 protocol. A client will send out TLSv1.2 client hello messages and will also indicate that it only understand TLSv1.2. A server will only understand TLSv1.2 client hello messages.

TLSv1_1_method(), TLSv1_1_server_method(), TLSv1_1_client_method()

A TLS/SSL connection established with these methods will only understand the TLSv1.1 protocol. A client will send out TLSv1.1 client hello messages and will also indicate that it only understand TLSv1.1. A server will only understand TLSv1.1 client hello messages.

TLSv1_method(), TLSv1_server_method(), TLSv1_client_method()

A TLS/SSL connection established with these methods will only understand the TLSv1 protocol. A client will send out TLSv1 client hello messages and will indicate that it only understands TLSv1. A server will only understand TLSv1 client hello messages.

SSLv3_method(), SSLv3_server_method(), SSLv3_client_method()

A TLS/SSL connection established with these methods will only understand the SSLv3 protocol. A client will send out SSLv3 client hello messages and will indicate that it only understands SSLv3. A server will only understand SSLv3 client hello messages. The SSLv3 protocol is deprecated and should not be used.

SSLv2_method(), SSLv2_server_method(), SSLv2_client_method()

A TLS/SSL connection established with these methods will only understand the SSLv2 protocol. A client will send out SSLv2 client hello messages and will also indicate that it only understand SSLv2. A server will only understand SSLv2 client hello messages. The SSLv2 protocol offers little to no security and should not be used. As of OpenSSL 1.0.1s, EXPORT ciphers and 56-bit DES are no longer available with SSLv2.

DTLSv1_method(), DTLSv1_server_method(), DTLSv1_client_method()

These are the version-specific methods for DTLSv1.

SSL_CTX_new() initializes the list of ciphers, the session cache setting, the callbacks, the keys and certificates and the options to its default values.

RETURN VALUES

The following return values can occur:

NULL
The creation of a new SSL_CTX object failed. Check the error stack to find out the reason.

Pointer to an SSL_CTX object
The return value points to an allocated SSL_CTX object.

SEE ALSO

SSL_CTX_set_options(3), SSL_CTX_clear_options(3), SSL_set_options(3), SSL_CTX_free(3), SSL_accept(3), ssl(3), SSL_set_connect_state(3)

Name

SSL_CTX_sessions — access internal session cache

Synopsis

```
#include  <openssl/ssl.h>
```

```
struct lhash_st *SSL_CTX_sessions(SSL_CTX *ctx);
```

DESCRIPTION

SSL_CTX_sessions() returns a pointer to the lhash databases containing the internal session cache for **ctx**.

NOTES

The sessions in the internal session cache are kept in an lhash(3) type database. It is possible to directly access this database e.g. for searching. In parallel, the sessions form a linked list which is maintained separately from the lhash(3) operations, so that the database must not be modified directly but by using the SSL_CTX_add_session(3) family of functions.

SEE ALSO

ssl(3), lhash(3), SSL_CTX_add_session(3), SSL_CTX_set_session_cache_mode(3)

Name

SSL_CTX_sess_number, SSL_CTX_sess_connect, SSL_CTX_sess_connect_good, SSL_CTX_sess_connect_renegotiate, SSL_CTX_sess_accept, SSL_CTX_sess_accept_good, SSL_CTX_sess_accept_renegotiate, SSL_CTX_sess_hits, SSL_CTX_sess_cb_hits, SSL_CTX_sess_misses, SSL_CTX_sess_timeouts and SSL_CTX_sess_cache_full — obtain session cache statistics

Synopsis

```
#include  <openssl/ssl.h>
```

```
long SSL_CTX_sess_number(SSL_CTX *ctx);
long SSL_CTX_sess_connect(SSL_CTX *ctx);
long SSL_CTX_sess_connect_good(SSL_CTX *ctx);
long SSL_CTX_sess_connect_renegotiate(SSL_CTX *ctx);
long SSL_CTX_sess_accept(SSL_CTX *ctx);
long SSL_CTX_sess_accept_good(SSL_CTX *ctx);
long SSL_CTX_sess_accept_renegotiate(SSL_CTX *ctx);
long SSL_CTX_sess_hits(SSL_CTX *ctx);
long SSL_CTX_sess_cb_hits(SSL_CTX *ctx);
long SSL_CTX_sess_misses(SSL_CTX *ctx);
long SSL_CTX_sess_timeouts(SSL_CTX *ctx);
long SSL_CTX_sess_cache_full(SSL_CTX *ctx);
```

DESCRIPTION

SSL_CTX_sess_number() returns the current number of sessions in the internal session cache.

SSL_CTX_sess_connect() returns the number of started SSL/TLS handshakes in client mode.

SSL_CTX_sess_connect_good() returns the number of successfully established SSL/TLS sessions in client mode.

SSL_CTX_sess_connect_renegotiate() returns the number of start renegotiations in client mode.

SSL_CTX_sess_accept() returns the number of started SSL/TLS handshakes in server mode.

SSL_CTX_sess_accept_good() returns the number of successfully established SSL/TLS sessions in server mode.

SSL_CTX_sess_accept_renegotiate() returns the number of start renegotiations in server mode.

SSL_CTX_sess_hits() returns the number of successfully reused sessions. In client mode a session set with SSL_set_session(3) successfully reused is counted as a hit. In server mode a session successfully retrieved from internal or external cache is counted as a hit.

SSL_CTX_sess_cb_hits() returns the number of successfully retrieved sessions from the external session cache in server mode.

SSL_CTX_sess_misses() returns the number of sessions proposed by clients that were not found in the internal session cache in server mode.

SSL_CTX_sess_timeouts() returns the number of sessions proposed by clients and either found in the internal or external session cache in server mode, but that were invalid due to timeout. These sessions are not included in the SSL_CTX_sess_hits() count.

SSL_CTX_sess_cache_full() returns the number of sessions that were removed because the maximum session cache size was exceeded.

RETURN VALUES

The functions return the values indicated in the DESCRIPTION section.

SEE ALSO

ssl(3), SSL_set_session(3), SSL_CTX_set_session_cache_mode(3)SSL_CTX_sess_set_cache_size(3)

Name

SSL_CTX_sess_set_cache_size and SSL_CTX_sess_get_cache_size — manipulate session cache size

Synopsis

```
#include  <openssl/ssl.h>
```

```
long SSL_CTX_sess_set_cache_size(SSL_CTX *ctx, long t);
long SSL_CTX_sess_get_cache_size(SSL_CTX *ctx);
```

DESCRIPTION

SSL_CTX_sess_set_cache_size() sets the size of the internal session cache of context **ctx** to **t**.

SSL_CTX_sess_get_cache_size() returns the currently valid session cache size.

NOTES

The internal session cache size is SSL_SESSION_CACHE_MAX_SIZE_DEFAULT, currently 1024*20, so that up to 20000 sessions can be held. This size can be modified using the SSL_CTX_sess_set_cache_size() call. A special case is the size 0, which is used for unlimited size.

When the maximum number of sessions is reached, no more new sessions are added to the cache. New space may be added by calling SSL_CTX_flush_sessions(3) to remove expired sessions.

If the size of the session cache is reduced and more sessions are already in the session cache, old session will be removed at the next time a session shall be added. This removal is not synchronized with the expiration of sessions.

RETURN VALUES

SSL_CTX_sess_set_cache_size() returns the previously valid size.

SSL_CTX_sess_get_cache_size() returns the currently valid size.

SEE ALSO

ssl(3), SSL_CTX_set_session_cache_mode(3), SSL_CTX_sess_number(3), SSL_CTX_flush_sessions(3)

Name

SSL_CTX_sess_set_new_cb, SSL_CTX_sess_set_remove_cb, SSL_CTX_sess_set_get_cb, SSL_CTX_sess_get_new_cb, SSL_CTX_sess_get_remove_cb and SSL_CTX_sess_get_get_cb — provide callback functions for server side external session caching

Synopsis

```
#include  <openssl/ssl.h>
```

```
void SSL_CTX_sess_set_new_cb(SSL_CTX *ctx,
                             int (*new_session_cb)(SSL *, SSL_SESSION *));
void SSL_CTX_sess_set_remove_cb(SSL_CTX *ctx,
          void (*remove_session_cb)(SSL_CTX *ctx, SSL_SESSION *));
void SSL_CTX_sess_set_get_cb(SSL_CTX *ctx,
          SSL_SESSION (*get_session_cb)(SSL *, unsigned char *, int, int *));
```

```
int (*SSL_CTX_sess_get_new_cb(SSL_CTX *ctx))(struct ssl_st *ssl, SSL_SESSION *sess);
void (*SSL_CTX_sess_get_remove_cb(SSL_CTX *ctx))(struct ssl_ctx_st *ctx, SSL_SESSION *sess);
SSL_SESSION *(*SSL_CTX_sess_get_get_cb(SSL_CTX *ctx))(struct ssl_st *ssl, unsigned char *data,
              int len, int *copy);
```

```
int (*new_session_cb)(struct ssl_st *ssl, SSL_SESSION *sess);
void (*remove_session_cb)(struct ssl_ctx_st *ctx, SSL_SESSION *sess);
SSL_SESSION *(*get_session_cb)(struct ssl_st *ssl, unsigned char *data,
              int len, int *copy);
```

DESCRIPTION

SSL_CTX_sess_set_new_cb() sets the callback function, which is automatically called whenever a new session was negotiated.

SSL_CTX_sess_set_remove_cb() sets the callback function, which is automatically called whenever a session is removed by the SSL engine, because it is considered faulty or the session has become obsolete because of exceeding the timeout value.

SSL_CTX_sess_set_get_cb() sets the callback function which is called, whenever a SSL/TLS client proposed to resume a session but the session could not be found in the internal session cache (see SSL_CTX_set_session_cache_mode(3)). (SSL/TLS server only.)

SSL_CTX_sess_get_new_cb(), SSL_CTX_sess_get_remove_cb(), and SSL_CTX_sess_get_get_cb() allow to retrieve the function pointers of the provided callback functions. If a callback function has not been set, the NULL pointer is returned.

NOTES

In order to allow external session caching, synchronization with the internal session cache is realized via callback functions. Inside these callback functions, session can be saved to disk or put into a database using the d2i_SSL_SESSION(3) interface.

The new_session_cb() is called, whenever a new session has been negotiated and session caching is enabled (see SSL_CTX_set_session_cache_mode(3)). The new_session_cb() is passed the **ssl** connection and the ssl session **sess**. If the callback returns **0**, the session will be immediately removed again.

The remove_session_cb() is called, whenever the SSL engine removes a session from the internal cache. This happens when the session is removed because it is expired or when a connection was not shutdown cleanly. It also happens for all sessions in the internal session cache when SSL_CTX_free(3) is called. The remove_session_cb() is passed the **ctx** and the ssl session **sess**. It does not provide any feedback.

The get_session_cb() is only called on SSL/TLS servers with the session id proposed by the client. The get_session_cb() is always called, also when session caching was disabled. The get_session_cb() is passed the **ssl** connection, the session id of length **length** at the memory location **data**. With the parameter **copy** the callback can require the SSL engine to increment the reference count of the SSL_SESSION object, Normally the reference count is not incremented and therefore the session must not be explicitly freed with SSL_SESSION_free(3).

SEE ALSO

ssl(3), d2i_SSL_SESSION(3), SSL_CTX_set_session_cache_mode(3), SSL_CTX_flush_sessions(3), SSL_SESSION_free(3), SSL_CTX_free(3)

Name

SSL_CTX_set_cert_store and SSL_CTX_get_cert_store — manipulate X509 certificate verification storage

Synopsis

```
#include  <openssl/ssl.h>
```

```
void SSL_CTX_set_cert_store(SSL_CTX *ctx, X509_STORE *store);
X509_STORE *SSL_CTX_get_cert_store(const SSL_CTX *ctx);
```

DESCRIPTION

SSL_CTX_set_cert_store() sets/replaces the certificate verification storage of **ctx** to/with **store**. If another X509_STORE object is currently set in **ctx**, it will be X509_STORE_free()ed.

SSL_CTX_get_cert_store() returns a pointer to the current certificate verification storage.

NOTES

In order to verify the certificates presented by the peer, trusted CA certificates must be accessed. These CA certificates are made available via lookup methods, handled inside the X509_STORE. From the X509_STORE the X509_STORE_CTX used when verifying certificates is created.

Typically the trusted certificate store is handled indirectly via using SSL_CTX_load_verify_locations(3). Using the SSL_CTX_set_cert_store() and SSL_CTX_get_cert_store() functions it is possible to manipulate the X509_STORE object beyond the SSL_CTX_load_verify_locations(3) call.

Currently no detailed documentation on how to use the X509_STORE object is available. Not all members of the X509_STORE are used when the verification takes place. So will e.g. the verify_callback() be overridden with the verify_callback() set via the SSL_CTX_set_verify(3) family of functions. This document must therefore be updated when documentation about the X509_STORE object and its handling becomes available.

RETURN VALUES

SSL_CTX_set_cert_store() does not return diagnostic output.

SSL_CTX_get_cert_store() returns the current setting.

SEE ALSO

ssl(3), SSL_CTX_load_verify_locations(3), SSL_CTX_set_verify(3)

Name

SSL_CTX_set_cert_verify_callback — set peer certificate verification procedure

Synopsis

```
#include  <openssl/ssl.h>
```

```
void SSL_CTX_set_cert_verify_callback(SSL_CTX *ctx, int (*callback)(X509_STORE_CTX *,void *), void *arg);
```

DESCRIPTION

SSL_CTX_set_cert_verify_callback() sets the verification callback function for *ctx*. SSL objects that are created from *ctx* inherit the setting valid at the time when SSL_new(3) is called.

NOTES

Whenever a certificate is verified during a SSL/TLS handshake, a verification function is called. If the application does not explicitly specify a verification callback function, the built-in verification function is used. If a verification callback *callback* is specified via SSL_CTX_set_cert_verify_callback(), the supplied callback function is called instead. By setting *callback* to NULL, the default behaviour is restored.

When the verification must be performed, *callback* will be called with the arguments callback(X509_STORE_CTX *x509_store_ctx, void *arg). The argument *arg* is specified by the application when setting *callback*.

callback should return 1 to indicate verification success and 0 to indicate verification failure. If SSL_VERIFY_PEER is set and *callback* returns 0, the handshake will fail. As the verification procedure may allow to continue the connection in case of failure (by always returning 1) the verification result must be set in any case using the **error** member of *x509_store_ctx* so that the calling application will be informed about the detailed result of the verification procedure!

Within *x509_store_ctx*, *callback* has access to the *verify_callback* function set using SSL_CTX_set_verify(3).

WARNINGS

Do not mix the verification callback described in this function with the **verify_callback** function called during the verification process. The latter is set using the SSL_CTX_set_verify(3) family of functions.

Providing a complete verification procedure including certificate purpose settings etc is a complex task. The built-in procedure is quite powerful and in most cases it should be sufficient to modify its behaviour using the **verify_callback** function.

BUGS

RETURN VALUES

SSL_CTX_set_cert_verify_callback() does not provide diagnostic information.

SEE ALSO

ssl(3), SSL_CTX_set_verify(3), SSL_get_verify_result(3), SSL_CTX_load_verify_locations(3)

HISTORY

Previous to OpenSSL 0.9.7, the *arg* argument to **SSL_CTX_set_cert_verify_callback** was ignored, and *callback* was called simply as int (*callback)(X509_STORE_CTX *) To compile software written for previous versions of OpenSSL, a dummy argument will have to be added to *callback*.

Name

SSL_CTX_set_cipher_list and SSL_set_cipher_list — choose list of available SSL_CIPHERs

Synopsis

```
#include  <openssl/ssl.h>
```

```
int SSL_CTX_set_cipher_list(SSL_CTX *ctx, const char *str);
int SSL_set_cipher_list(SSL *ssl, const char *str);
```

DESCRIPTION

SSL_CTX_set_cipher_list() sets the list of available ciphers for **ctx** using the control string **str**. The format of the string is described in ciphers(1). The list of ciphers is inherited by all **ssl** objects created from **ctx**.

SSL_set_cipher_list() sets the list of ciphers only for **ssl**.

NOTES

The control string **str** should be universally usable and not depend on details of the library configuration (ciphers compiled in). Thus no syntax checking takes place. Items that are not recognized, because the corresponding ciphers are not compiled in or because they are mistyped, are simply ignored. Failure is only flagged if no ciphers could be collected at all.

It should be noted, that inclusion of a cipher to be used into the list is a necessary condition. On the client side, the inclusion into the list is also sufficient. On the server side, additional restrictions apply. All ciphers have additional requirements. ADH ciphers don't need a certificate, but DH-parameters must have been set. All other ciphers need a corresponding certificate and key.

A RSA cipher can only be chosen, when a RSA certificate is available. RSA export ciphers with a keylength of 512 bits for the RSA key require a temporary 512 bit RSA key, as typically the supplied key has a length of 1024 bit (see SSL_CTX_set_tmp_rsa_callback(3)). RSA ciphers using EDH need a certificate and key and additional DH-parameters (see SSL_CTX_set_tmp_dh_callback(3)).

A DSA cipher can only be chosen, when a DSA certificate is available. DSA ciphers always use DH key exchange and therefore need DH-parameters (see SSL_CTX_set_tmp_dh_callback(3)).

When these conditions are not met for any cipher in the list (e.g. a client only supports export RSA ciphers with a asymmetric key length of 512 bits and the server is not configured to use temporary RSA keys), the "no shared cipher" (SSL_R_NO_SHARED_CIPHER) error is generated and the handshake will fail.

If the cipher list does not contain any SSLv2 cipher suites (this is the default) then SSLv2 is effectively disabled and neither clients nor servers will attempt to use SSLv2.

RETURN VALUES

SSL_CTX_set_cipher_list() and SSL_set_cipher_list() return 1 if any cipher could be selected and 0 on complete failure.

SEE ALSO

ssl(3), SSL_get_ciphers(3), SSL_CTX_use_certificate(3), SSL_CTX_set_tmp_rsa_callback(3), SSL_CTX_set_tmp_dh_callback(3), ciphers(1)

Name

SSL_CTX_set_client_CA_list, SSL_set_client_CA_list, SSL_CTX_add_client_CA and SSL_add_client_CA — set list of CAs sent to the client when requesting a client certificate

Synopsis

```
#include  <openssl/ssl.h>
```

```
void SSL_CTX_set_client_CA_list(SSL_CTX *ctx, STACK_OF(X509_NAME) *list);
void SSL_set_client_CA_list(SSL *s, STACK_OF(X509_NAME) *list);
int SSL_CTX_add_client_CA(SSL_CTX *ctx, X509 *cacert);
int SSL_add_client_CA(SSL *ssl, X509 *cacert);
```

DESCRIPTION

SSL_CTX_set_client_CA_list() sets the **list** of CAs sent to the client when requesting a client certificate for **ctx**.

SSL_set_client_CA_list() sets the **list** of CAs sent to the client when requesting a client certificate for the chosen **ssl**, overriding the setting valid for **ssl**'s SSL_CTX object.

SSL_CTX_add_client_CA() adds the CA name extracted from **cacert** to the list of CAs sent to the client when requesting a client certificate for **ctx**.

SSL_add_client_CA() adds the CA name extracted from **cacert** to the list of CAs sent to the client when requesting a client certificate for the chosen **ssl**, overriding the setting valid for **ssl**'s SSL_CTX object.

NOTES

When a TLS/SSL server requests a client certificate (see **SSL_CTX_set_verify(3)**), it sends a list of CAs, for which it will accept certificates, to the client.

This list must explicitly be set using SSL_CTX_set_client_CA_list() for **ctx** and SSL_set_client_CA_list() for the specific **ssl**. The list specified overrides the previous setting. The CAs listed do not become trusted (**list** only contains the names, not the complete certificates); use SSL_CTX_load_verify_locations(3) to additionally load them for verification.

If the list of acceptable CAs is compiled in a file, the SSL_load_client_CA_file(3) function can be used to help importing the necessary data.

SSL_CTX_add_client_CA() and SSL_add_client_CA() can be used to add additional items the list of client CAs. If no list was specified before using SSL_CTX_set_client_CA_list() or SSL_set_client_CA_list(), a new client CA list for **ctx** or **ssl** (as appropriate) is opened.

These functions are only useful for TLS/SSL servers.

RETURN VALUES

SSL_CTX_set_client_CA_list() and SSL_set_client_CA_list() do not return diagnostic information.

SSL_CTX_add_client_CA() and SSL_add_client_CA() have the following return values:

0 A failure while manipulating the STACK_OF(X509_NAME) object occurred or the X509_NAME could not be extracted from **cacert**. Check the error stack to find out the reason.

1 The operation succeeded.

EXAMPLES

Scan all certificates in **CAfile** and list them as acceptable CAs:

```
SSL_CTX_set_client_CA_list(ctx,SSL_load_client_CA_file(CAfile));
```

SEE ALSO

ssl(3), SSL_get_client_CA_list(3), SSL_load_client_CA_file(3), SSL_CTX_load_verify_locations(3)

Name

SSL_CTX_set_client_cert_cb and SSL_CTX_get_client_cert_cb — handle client certificate callback function

Synopsis

```
#include  <openssl/ssl.h>
```

```
void SSL_CTX_set_client_cert_cb(SSL_CTX *ctx, int (*client_cert_cb)(SSL *ssl, X509 **x509,
    EVP_PKEY **pkey));
int (*SSL_CTX_get_client_cert_cb(SSL_CTX *ctx))(SSL *ssl, X509 **x509, EVP_PKEY **pkey);
int (*client_cert_cb)(SSL *ssl, X509 **x509, EVP_PKEY **pkey);
```

DESCRIPTION

SSL_CTX_set_client_cert_cb() sets the **client_cert_cb()** callback, that is called when a client certificate is requested by a server and no certificate was yet set for the SSL object.

When **client_cert_cb()** is NULL, no callback function is used.

SSL_CTX_get_client_cert_cb() returns a pointer to the currently set callback function.

client_cert_cb() is the application defined callback. If it wants to set a certificate, a certificate/private key combination must be set using the **x509** and **pkey** arguments and "1" must be returned. The certificate will be installed into **ssl**, see the NOTES and BUGS sections. If no certificate should be set, "0" has to be returned and no certificate will be sent. A negative return value will suspend the handshake and the handshake function will return immediately. SSL_get_error(3) will return SSL_ERROR_WANT_X509_LOOKUP to indicate, that the handshake was suspended. The next call to the handshake function will again lead to the call of client_cert_cb(). It is the job of the client_cert_cb() to store information about the state of the last call, if required to continue.

NOTES

During a handshake (or renegotiation) a server may request a certificate from the client. A client certificate must only be sent, when the server did send the request.

When a certificate was set using the SSL_CTX_use_certificate(3) family of functions, it will be sent to the server. The TLS standard requires that only a certificate is sent, if it matches the list of acceptable CAs sent by the server. This constraint is violated by the default behavior of the OpenSSL library. Using the callback function it is possible to implement a proper selection routine or to allow a user interaction to choose the certificate to be sent.

If a callback function is defined and no certificate was yet defined for the SSL object, the callback function will be called. If the callback function returns a certificate, the OpenSSL library will try to load the private key and certificate data into the SSL object using the SSL_use_certificate() and SSL_use_private_key() functions. Thus it will permanently install the certificate and key for this SSL object. It will not be reset by calling SSL_clear(3). If the callback returns no certificate, the OpenSSL library will not send a certificate.

BUGS

The client_cert_cb() cannot return a complete certificate chain, it can only return one client certificate. If the chain only has a length of 2, the root CA certificate may be omitted according to the TLS standard and thus a standard conforming answer can be sent to the server. For a longer chain, the client must send the complete chain (with the option to leave out the root CA certificate). This can only be accomplished by either adding the intermediate CA certificates into the trusted certificate store for the SSL_CTX object (resulting in having to add CA certificates that otherwise maybe would not be trusted), or by adding the chain certificates using the SSL_CTX_add_extra_chain_cert(3) function, which is only available for the SSL_CTX object as a whole and that therefore probably can only apply for one client certificate, making the concept of the callback function (to allow the choice from several certificates) questionable.

Once the SSL object has been used in conjunction with the callback function, the certificate will be set for the SSL object and will not be cleared even when SSL_clear(3) is being called. It is therefore mandatory to destroy the SSL object using SSL_free(3) and create a new one to return to the previous state.

SEE ALSO

ssl(3), SSL_CTX_use_certificate(3), SSL_CTX_add_extra_chain_cert(3), SSL_get_client_CA_list(3), SSL_clear(3), SSL_free(3)

Name

SSL_CTX_set_default_passwd_cb and SSL_CTX_set_default_passwd_cb_userdata — set passwd callback for encrypted PEM file handling

Synopsis

```
#include  <openssl/ssl.h>
```

```
void SSL_CTX_set_default_passwd_cb(SSL_CTX *ctx, pem_password_cb *cb);
void SSL_CTX_set_default_passwd_cb_userdata(SSL_CTX *ctx, void *u);
```

```
int pem_passwd_cb(char *buf, int size, int rwflag, void *userdata);
```

DESCRIPTION

SSL_CTX_set_default_passwd_cb() sets the default password callback called when loading/storing a PEM certificate with encryption.

SSL_CTX_set_default_passwd_cb_userdata() sets a pointer to **userdata** which will be provided to the password callback on invocation.

The pem_passwd_cb(), which must be provided by the application, hands back the password to be used during decryption. On invocation a pointer to **userdata** is provided. The pem_passwd_cb must write the password into the provided buffer **buf** which is of size **size**. The actual length of the password must be returned to the calling function. **rwflag** indicates whether the callback is used for reading/decryption (rwflag=0) or writing/encryption (rwflag=1).

NOTES

When loading or storing private keys, a password might be supplied to protect the private key. The way this password can be supplied may depend on the application. If only one private key is handled, it can be practical to have pem_passwd_cb() handle the password dialog interactively. If several keys have to be handled, it can be practical to ask for the password once, then keep it in memory and use it several times. In the last case, the password could be stored into the **userdata** storage and the pem_passwd_cb() only returns the password already stored.

When asking for the password interactively, pem_passwd_cb() can use **rwflag** to check, whether an item shall be encrypted (rwflag=1). In this case the password dialog may ask for the same password twice for comparison in order to catch typos, that would make decryption impossible.

Other items in PEM formatting (certificates) can also be encrypted, it is however not usual, as certificate information is considered public.

RETURN VALUES

SSL_CTX_set_default_passwd_cb() and SSL_CTX_set_default_passwd_cb_userdata() do not provide diagnostic information.

EXAMPLES

The following example returns the password provided as **userdata** to the calling function. The password is considered to be a '\0' terminated string. If the password does not fit into the buffer, the password is truncated.

```
int pem_passwd_cb(char *buf, int size, int rwflag, void *password)
{
 strncpy(buf, (char *)(password), size);
 buf[size - 1] = '\0';
 return(strlen(buf));
}
```

SEE ALSO

ssl(3), SSL_CTX_use_certificate(3)

Name

SSL_CTX_set_generate_session_id, SSL_set_generate_session_id and SSL_has_matching_session_id — manipulate generation of SSL session IDs (server only)

Synopsis

```
#include  <openssl/ssl.h>
```

```
typedef int (*GEN_SESSION_CB)(const SSL *ssl, unsigned char *id,
                              unsigned int *id_len);
```

```
int SSL_CTX_set_generate_session_id(SSL_CTX *ctx, GEN_SESSION_CB cb);
int SSL_set_generate_session_id(SSL *ssl, GEN_SESSION_CB, cb);
int SSL_has_matching_session_id(const SSL *ssl, const unsigned char *id,
                                unsigned int id_len);
```

DESCRIPTION

SSL_CTX_set_generate_session_id() sets the callback function for generating new session ids for SSL/TLS sessions for **ctx** to be **cb**.

SSL_set_generate_session_id() sets the callback function for generating new session ids for SSL/TLS sessions for **ssl** to be **cb**.

SSL_has_matching_session_id() checks, whether a session with id **id** (of length **id_len**) is already contained in the internal session cache of the parent context of **ssl**.

NOTES

When a new session is established between client and server, the server generates a session id. The session id is an arbitrary sequence of bytes. The length of the session id is 16 bytes for SSLv2 sessions and between 1 and 32 bytes for SSLv3/TLSv1. The session id is not security critical but must be unique for the server. Additionally, the session id is transmitted in the clear when reusing the session so it must not contain sensitive information.

Without a callback being set, an OpenSSL server will generate a unique session id from pseudo random numbers of the maximum possible length. Using the callback function, the session id can be changed to contain additional information like e.g. a host id in order to improve load balancing or external caching techniques.

The callback function receives a pointer to the memory location to put **id** into and a pointer to the maximum allowed length **id_len**. The buffer at location **id** is only guaranteed to have the size **id_len**. The callback is only allowed to generate a shorter id and reduce **id_len**; the callback **must never** increase **id_len** or write to the location **id** exceeding the given limit.

If a SSLv2 session id is generated and **id_len** is reduced, it will be restored after the callback has finished and the session id will be padded with 0x00. It is not recommended to change the **id_len** for SSLv2 sessions. The callback can use the SSL_get_version(3) function to check, whether the session is of type SSLv2.

The location **id** is filled with 0x00 before the callback is called, so the callback may only fill part of the possible length and leave **id_len** untouched while maintaining reproducibility.

Since the sessions must be distinguished, session ids must be unique. Without the callback a random number is used, so that the probability of generating the same session id is extremely small (2^128 possible ids for an SSLv2 session, 2^256 for SSLv3/TLSv1). In order to assure the uniqueness of the generated session id, the callback must call SSL_has_matching_session_id() and generate another id if a conflict occurs. If an id conflict is not resolved, the handshake will fail. If the application codes e.g. a unique host id, a unique process number, and a unique sequence number into the session id, uniqueness could easily be achieved without randomness added (it should however be taken care that no confidential information is leaked this way). If the application can not guarantee uniqueness, it is recommended to use the maximum **id_len** and fill in the bytes not used to code special information with random data to avoid collisions.

SSL_has_matching_session_id() will only query the internal session cache, not the external one. Since the session id is generated before the handshake is completed, it is not immediately added to the cache. If another thread is using the same internal session cache, a race condition can occur in that another thread generates the same session id. Collisions can also occur when using an external session cache, since the external cache is not tested with SSL_has_matching_session_id() and the same race condition applies.

When calling SSL_has_matching_session_id() for an SSLv2 session with reduced **id_len**, the match operation will be performed using the fixed length required and with a 0x00 padded id.

The callback must return 0 if it cannot generate a session id for whatever reason and return 1 on success.

EXAMPLES

The callback function listed will generate a session id with the server id given, and will fill the rest with pseudo random bytes:

```
const char session_id_prefix = "www-18";

#define MAX_SESSION_ID_ATTEMPTS 10
static int generate_session_id(const SSL *ssl, unsigned char *id,
                             unsigned int *id_len)
     {
     unsigned int count = 0;
     const char *version;

version = SSL_get_version(ssl);
if (!strcmp(version, "SSLv2"))
    /* we must not change id_len */;

do      {
        RAND_pseudo_bytes(id, *id_len);
        /* Prefix the session_id with the required prefix. NB: If our
         * prefix is too long, clip it - but there will be worse effects
         * anyway, eg. the server could only possibly create 1 session
         * ID (ie. the prefix!) so all future session negotiations will
         * fail due to conflicts. */
        memcpy(id, session_id_prefix,
                (strlen(session_id_prefix) < *id_len) ?
                strlen(session_id_prefix) : *id_len);
        }
while(SSL_has_matching_session_id(ssl, id, *id_len) &&
        (++count < MAX_SESSION_ID_ATTEMPTS));
if(count >= MAX_SESSION_ID_ATTEMPTS)
        return 0;
return 1;
}
```

RETURN VALUES

SSL_CTX_set_generate_session_id() and SSL_set_generate_session_id() always return 1.

SSL_has_matching_session_id() returns 1 if another session with the same id is already in the cache.

SEE ALSO

ssl(3), SSL_get_version(3)

HISTORY

SSL_CTX_set_generate_session_id(), SSL_set_generate_session_id() and SSL_has_matching_session_id() have been introduced in OpenSSL 0.9.7.

Name

SSL_CTX_set_info_callback, SSL_CTX_get_info_callback, SSL_set_info_callback and SSL_get_info_callback — handle information callback for SSL connections

Synopsis

```
#include  <openssl/ssl.h>
```

```
void SSL_CTX_set_info_callback(SSL_CTX *ctx, void (*callback)());
void (*SSL_CTX_get_info_callback(const SSL_CTX *ctx))();
```

```
void SSL_set_info_callback(SSL *ssl, void (*callback)());
void (*SSL_get_info_callback(const SSL *ssl))();
```

DESCRIPTION

SSL_CTX_set_info_callback() sets the **callback** function, that can be used to obtain state information for SSL objects created from **ctx** during connection setup and use. The setting for **ctx** is overridden from the setting for a specific SSL object, if specified. When **callback** is NULL, not callback function is used.

SSL_set_info_callback() sets the **callback** function, that can be used to obtain state information for **ssl** during connection setup and use. When **callback** is NULL, the callback setting currently valid for **ctx** is used.

SSL_CTX_get_info_callback() returns a pointer to the currently set information callback function for **ctx**.

SSL_get_info_callback() returns a pointer to the currently set information callback function for **ssl**.

NOTES

When setting up a connection and during use, it is possible to obtain state information from the SSL/TLS engine. When set, an information callback function is called whenever the state changes, an alert appears, or an error occurs.

The callback function is called as **callback(SSL *ssl, int where, int ret)**. The **where** argument specifies information about where (in which context) the callback function was called. If **ret** is 0, an error condition occurred. If an alert is handled, SSL_CB_ALERT is set and **ret** specifies the alert information.

where is a bitmask made up of the following bits:

SSL_CB_LOOP

> Callback has been called to indicate state change inside a loop.

SSL_CB_EXIT

> Callback has been called to indicate error exit of a handshake function. (May be soft error with retry option for non-blocking setups.)

SSL_CB_READ

> Callback has been called during read operation.

SSL_CB_WRITE

> Callback has been called during write operation.

SSL_CB_ALERT

> Callback has been called due to an alert being sent or received.

SSL_CB_READ_ALERT (SSL_CB_ALERT|SSL_CB_READ)

SSL_CB_WRITE_ALERT (SSL_CB_ALERT|SSL_CB_WRITE)

SSL_CB_ACCEPT_LOOP (SSL_ST_ACCEPT|SSL_CB_LOOP)

SSL_CB_ACCEPT_EXIT (SSL_ST_ACCEPT|SSL_CB_EXIT)

SSL_CB_CONNECT_LOOP (SSL_ST_CONNECT|SSL_CB_LOOP)

SSL_CB_CONNECT_EXIT (SSL_ST_CONNECT|SSL_CB_EXIT)

SSL_CB_HANDSHAKE_START

> Callback has been called because a new handshake is started.

SSL_CB_HANDSHAKE_DONE 0x20

> Callback has been called because a handshake is finished.

The current state information can be obtained using the SSL_state_string(3) family of functions.

The **ret** information can be evaluated using the SSL_alert_type_string(3) family of functions.

RETURN VALUES

SSL_set_info_callback() does not provide diagnostic information.

SSL_get_info_callback() returns the current setting.

EXAMPLES

The following example callback function prints state strings, information about alerts being handled and error messages to the **bio_err** BIO.

```
void apps_ssl_info_callback(SSL *s, int where, int ret)
       {
       const char *str;
       int w;

w=where& ~SSL_ST_MASK;

if (w & SSL_ST_CONNECT) str="SSL_connect";
else if (w & SSL_ST_ACCEPT) str="SSL_accept";
else str="undefined";

if (where & SSL_CB_LOOP)
        {
        BIO_printf(bio_err,"%s:%s\n",str,SSL_state_string_long(s));
        }
else if (where & SSL_CB_ALERT)
        {
        str=(where & SSL_CB_READ)?"read":"write";
        BIO_printf(bio_err,"SSL3 alert %s:%s:%s\n",
                str,
                SSL_alert_type_string_long(ret),
                SSL_alert_desc_string_long(ret));
        }
else if (where & SSL_CB_EXIT)
        {
        if (ret == 0)
                BIO_printf(bio_err,"%s:failed in %s\n",
                        str,SSL_state_string_long(s));
        else if (ret < 0)
                {
                BIO_printf(bio_err,"%s:error in %s\n",
                        str,SSL_state_string_long(s));
                }
        }
}
```

SEE ALSO

ssl(3), SSL_state_string(3), SSL_alert_type_string(3)

Name

SSL_CTX_set_max_cert_list, SSL_CTX_get_max_cert_list, SSL_set_max_cert_list and SSL_get_max_cert_list — manipulate allowed for the peer's certificate chain

Synopsis

```
#include  <openssl/ssl.h>
```

```
long SSL_CTX_set_max_cert_list(SSL_CTX *ctx, long size);
long SSL_CTX_get_max_cert_list(SSL_CTX *ctx);
```

```
long SSL_set_max_cert_list(SSL *ssl, long size);
long SSL_get_max_cert_list(SSL *ctx);
```

DESCRIPTION

SSL_CTX_set_max_cert_list() sets the maximum size allowed for the peer's certificate chain for all SSL objects created from **ctx** to be <size> bytes. The SSL objects inherit the setting valid for **ctx** at the time SSL_new(3) is being called.

SSL_CTX_get_max_cert_list() returns the currently set maximum size for **ctx**.

SSL_set_max_cert_list() sets the maximum size allowed for the peer's certificate chain for **ssl** to be <size> bytes. This setting stays valid until a new value is set.

SSL_get_max_cert_list() returns the currently set maximum size for **ssl**.

NOTES

During the handshake process, the peer may send a certificate chain. The TLS/SSL standard does not give any maximum size of the certificate chain. The OpenSSL library handles incoming data by a dynamically allocated buffer. In order to prevent this buffer from growing without bounds due to data received from a faulty or malicious peer, a maximum size for the certificate chain is set.

The default value for the maximum certificate chain size is 100kB (30kB on the 16bit DOS platform). This should be sufficient for usual certificate chains (OpenSSL's default maximum chain length is 10, see SSL_CTX_set_verify(3), and certificates without special extensions have a typical size of 1-2kB).

For special applications it can be necessary to extend the maximum certificate chain size allowed to be sent by the peer, see e.g. the work on "Internet X.509 Public Key Infrastructure Proxy Certificate Profile" and "TLS Delegation Protocol" at http://www.ietf.org/ and http://www.globus.org/ .

Under normal conditions it should never be necessary to set a value smaller than the default, as the buffer is handled dynamically and only uses the memory actually required by the data sent by the peer.

If the maximum certificate chain size allowed is exceeded, the handshake will fail with a SSL_R_EXCESSIVE_MESSAGE_SIZE error.

RETURN VALUES

SSL_CTX_set_max_cert_list() and SSL_set_max_cert_list() return the previously set value.

SSL_CTX_get_max_cert_list() and SSL_get_max_cert_list() return the currently set value.

SEE ALSO

ssl(3), SSL_new(3), SSL_CTX_set_verify(3)

HISTORY

SSL*_set/get_max_cert_list() have been introduced in OpenSSL 0.9.7.

Name

SSL_CTX_set_mode, SSL_set_mode, SSL_CTX_get_mode and SSL_get_mode — manipulate SSL engine mode

Synopsis

```
#include  <openssl/ssl.h>
```

```
long SSL_CTX_set_mode(SSL_CTX *ctx, long mode);
long SSL_set_mode(SSL *ssl, long mode);
```

```
long SSL_CTX_get_mode(SSL_CTX *ctx);
long SSL_get_mode(SSL *ssl);
```

DESCRIPTION

SSL_CTX_set_mode() adds the mode set via bitmask in **mode** to **ctx**. Options already set before are not cleared.

SSL_set_mode() adds the mode set via bitmask in **mode** to **ssl**. Options already set before are not cleared.

SSL_CTX_get_mode() returns the mode set for **ctx**.

SSL_get_mode() returns the mode set for **ssl**.

NOTES

The following mode changes are available:

SSL_MODE_ENABLE_PARTIAL_WRITE

> Allow SSL_write(…, n) to return r with 0 < r < n (i.e. report success when just a single record has been written). When not set (the default), SSL_write() will only report success once the complete chunk was written. Once SSL_write() returns with r, r bytes have been successfully written and the next call to SSL_write() must only send the n-r bytes left, imitating the behaviour of write().

SSL_MODE_ACCEPT_MOVING_WRITE_BUFFER

> Make it possible to retry SSL_write() with changed buffer location (the buffer contents must stay the same). This is not the default to avoid the misconception that non-blocking SSL_write() behaves like non-blocking write().

SSL_MODE_AUTO_RETRY

> Never bother the application with retries if the transport is blocking. If a renegotiation take place during normal operation, a SSL_read(3) or SSL_write(3) would return with -1 and indicate the need to retry with SSL_ERROR_WANT_READ. In a non-blocking environment applications must be prepared to handle incomplete read/write operations. In a blocking environment, applications are not always prepared to deal with read/write operations returning without success report. The flag SSL_MODE_AUTO_RETRY will cause read/write operations to only return after the handshake and successful completion.

SSL_MODE_RELEASE_BUFFERS

> When we no longer need a read buffer or a write buffer for a given SSL, then release the memory we were using to hold it. Released memory is either appended to a list of unused RAM chunks on the SSL_CTX, or simply freed if the list of unused chunks would become longer than SSL_CTX->freelist_max_len, which defaults to 32. Using this flag can save around 34k per idle SSL connection. This flag has no effect on SSL v2 connections, or on DTLS connections.

SSL_MODE_SEND_FALLBACK_SCSV

> Send TLS_FALLBACK_SCSV in the ClientHello. To be set only by applications that reconnect with a downgraded protocol version; see draft-ietf-tls-downgrade-scsv-00 for details.

DO NOT ENABLE THIS if your application attempts a normal handshake. Only use this in explicit fallback retries, following the guidance in draft-ietf-tls-downgrade-scsv-00.

RETURN VALUES

SSL_CTX_set_mode() and SSL_set_mode() return the new mode bitmask after adding **mode**.

SSL_CTX_get_mode() and SSL_get_mode() return the current bitmask.

SEE ALSO

ssl(3), SSL_read(3), SSL_write(3)

HISTORY

SSL_MODE_AUTO_RETRY as been added in OpenSSL 0.9.6.

Name

SSL_CTX_set_msg_callback, SSL_CTX_set_msg_callback_arg, SSL_set_msg_callback and SSL_get_msg_callback_arg — install callback for observing protocol messages

Synopsis

```
#include  <openssl/ssl.h>
```

```
void SSL_CTX_set_msg_callback(SSL_CTX *ctx, void (*cb)(int write_p, int version, int content_type,
    const void *buf, size_t len, SSL *ssl, void *arg));
void SSL_CTX_set_msg_callback_arg(SSL_CTX *ctx, void *arg);
```

```
void SSL_set_msg_callback(SSL *ssl, void (*cb)(int write_p, int version, int content_type,
    const void *buf, size_t len, SSL *ssl, void *arg));
void SSL_set_msg_callback_arg(SSL *ssl, void *arg);
```

DESCRIPTION

SSL_CTX_set_msg_callback() or SSL_set_msg_callback() can be used to define a message callback function *cb* for observing all SSL/TLS protocol messages (such as handshake messages) that are received or sent. SSL_CTX_set_msg_callback_arg() and SSL_set_msg_callback_arg() can be used to set argument *arg* to the callback function, which is available for arbitrary application use.

SSL_CTX_set_msg_callback() and SSL_CTX_set_msg_callback_arg() specify default settings that will be copied to new **SSL** objects by SSL_new(3). SSL_set_msg_callback() and SSL_set_msg_callback_arg() modify the actual settings of an **SSL** object. Using a **0** pointer for *cb* disables the message callback.

When *cb* is called by the SSL/TLS library for a protocol message, the function arguments have the following meaning:

write_p
: This flag is **0** when a protocol message has been received and **1** when a protocol message has been sent.

version
: The protocol version according to which the protocol message is interpreted by the library. Currently, this is one of **SSL2_VERSION**, **SSL3_VERSION** and **TLS1_VERSION** (for SSL 2.0, SSL 3.0 and TLS 1.0, respectively).

content_type
: In the case of SSL 2.0, this is always **0**. In the case of SSL 3.0 or TLS 1.0, this is one of the **ContentType** values defined in the protocol specification (**change_cipher_spec(20)**, **alert(21)**, **handshake(22)**; but never **application_data(23)** because the callback will only be called for protocol messages).

buf, len
: *buf* points to a buffer containing the protocol message, which consists of *len* bytes. The buffer is no longer valid after the callback function has returned.

ssl
: The **SSL** object that received or sent the message.

arg
: The user-defined argument optionally defined by SSL_CTX_set_msg_callback_arg() or SSL_set_msg_callback_arg().

NOTES

Protocol messages are passed to the callback function after decryption and fragment collection where applicable. (Thus record boundaries are not visible.)

If processing a received protocol message results in an error, the callback function may not be called. For example, the callback function will never see messages that are considered too large to be processed.

Due to automatic protocol version negotiation, *version* is not necessarily the protocol version used by the sender of the message: If a TLS 1.0 ClientHello message is received by an SSL 3.0-only server, *version* will be **SSL3_VERSION**.

SEE ALSO

ssl(3), SSL_new(3)

HISTORY

SSL_CTX_set_msg_callback(), SSL_CTX_set_msg_callback_arg(), SSL_set_msg_callback() and SSL_get_msg_callback_arg() were added in OpenSSL 0.9.7.

Name

SSL_CTX_set_options, SSL_set_options, SSL_CTX_clear_options, SSL_clear_options, SSL_CTX_get_options, SSL_get_options and SSL_get_secure_renegotiation_support — manipulate SSL options

Synopsis

```
#include  <openssl/ssl.h>
```

```
long SSL_CTX_set_options(SSL_CTX *ctx, long options);
long SSL_set_options(SSL *ssl, long options);
```

```
long SSL_CTX_clear_options(SSL_CTX *ctx, long options);
long SSL_clear_options(SSL *ssl, long options);
```

```
long SSL_CTX_get_options(SSL_CTX *ctx);
long SSL_get_options(SSL *ssl);
```

```
long SSL_get_secure_renegotiation_support(SSL *ssl);
```

DESCRIPTION

Note: all these functions are implemented using macros.

SSL_CTX_set_options() adds the options set via bitmask in **options** to **ctx**. Options already set before are not cleared!

SSL_set_options() adds the options set via bitmask in **options** to **ssl**. Options already set before are not cleared!

SSL_CTX_clear_options() clears the options set via bitmask in **options** to **ctx**.

SSL_clear_options() clears the options set via bitmask in **options** to **ssl**.

SSL_CTX_get_options() returns the options set for **ctx**.

SSL_get_options() returns the options set for **ssl**.

SSL_get_secure_renegotiation_support() indicates whether the peer supports secure renegotiation.

NOTES

The behaviour of the SSL library can be changed by setting several options. The options are coded as bitmasks and can be combined by a logical **or** operation (|).

SSL_CTX_set_options() and SSL_set_options() affect the (external) protocol behaviour of the SSL library. The (internal) behaviour of the API can be changed by using the similar SSL_CTX_set_mode(3) and SSL_set_mode() functions.

During a handshake, the option settings of the SSL object are used. When a new SSL object is created from a context using SSL_new(), the current option setting is copied. Changes to **ctx** do not affect already created SSL objects. SSL_clear() does not affect the settings.

The following **bug workaround** options are available:

SSL_OP_MICROSOFT_SESS_ID_BUG

> www.microsoft.com - when talking SSLv2, if session-id reuse is performed, the session-id passed back in the server-finished message is different from the one decided upon.

SSL_OP_NETSCAPE_CHALLENGE_BUG

> Netscape-Commerce/1.12, when talking SSLv2, accepts a 32 byte challenge but then appears to only use 16 bytes when generating the encryption keys. Using 16 bytes is ok but it should be ok to use 32. According to the SSLv3 spec, one should

use 32 bytes for the challenge when operating in SSLv2/v3 compatibility mode, but as mentioned above, this breaks this server so 16 bytes is the way to go.

SSL_OP_NETSCAPE_REUSE_CIPHER_CHANGE_BUG

As of OpenSSL 0.9.8q and 1.0.0c, this option has no effect.

SSL_OP_SSLREF2_REUSE_CERT_TYPE_BUG

…

SSL_OP_MICROSOFT_BIG_SSLV3_BUFFER

…

SSL_OP_SAFARI_ECDHE_ECDSA_BUG

Don't prefer ECDHE-ECDSA ciphers when the client appears to be Safari on OS X. OS X 10.8..10.8.3 has broken support for ECDHE-ECDSA ciphers.

SSL_OP_SSLEAY_080_CLIENT_DH_BUG

…

SSL_OP_TLS_D5_BUG

…

SSL_OP_TLS_BLOCK_PADDING_BUG

…

SSL_OP_DONT_INSERT_EMPTY_FRAGMENTS

Disables a countermeasure against a SSL 3.0/TLS 1.0 protocol vulnerability affecting CBC ciphers, which cannot be handled by some broken SSL implementations. This option has no effect for connections using other ciphers.

SSL_OP_TLSEXT_PADDING

Adds a padding extension to ensure the ClientHello size is never between 256 and 511 bytes in length. This is needed as a workaround for some implementations.

SSL_OP_ALL

All of the above bug workarounds.

It is usually safe to use **SSL_OP_ALL** to enable the bug workaround options if compatibility with somewhat broken implementations is desired.

The following **modifying** options are available:

SSL_OP_TLS_ROLLBACK_BUG

Disable version rollback attack detection.

During the client key exchange, the client must send the same information about acceptable SSL/TLS protocol levels as during the first hello. Some clients violate this rule by adapting to the server's answer. (Example: the client sends a SSLv2 hello and accepts up to SSLv3.1=TLSv1, the server only understands up to SSLv3. In this case the client must still use the same

SSLv3.1=TLSv1 announcement. Some clients step down to SSLv3 with respect to the server's answer and violate the version rollback protection.)

SSL_OP_SINGLE_DH_USE

Always create a new key when using temporary/ephemeral DH parameters (see SSL_CTX_set_tmp_dh_callback(3)). This option must be used to prevent small subgroup attacks, when the DH parameters were not generated using "strong" primes (e.g. when using DSA-parameters, see dhparam(1)). If "strong" primes were used, it is not strictly necessary to generate a new DH key during each handshake but it is also recommended. **SSL_OP_SINGLE_DH_USE** should therefore be enabled whenever temporary/ephemeral DH parameters are used.

SSL_OP_EPHEMERAL_RSA

This option is no longer implemented and is treated as no op.

SSL_OP_CIPHER_SERVER_PREFERENCE

When choosing a cipher, use the server's preferences instead of the client preferences. When not set, the SSL server will always follow the clients preferences. When set, the SSLv3/TLSv1 server will choose following its own preferences. Because of the different protocol, for SSLv2 the server will send its list of preferences to the client and the client chooses.

SSL_OP_PKCS1_CHECK_1

…

SSL_OP_PKCS1_CHECK_2

…

SSL_OP_NETSCAPE_CA_DN_BUG

If we accept a netscape connection, demand a client cert, have a non-self-signed CA which does not have its CA in netscape, and the browser has a cert, it will crash/hang. Works for 3.x and 4.xbeta

SSL_OP_NETSCAPE_DEMO_CIPHER_CHANGE_BUG

…

SSL_OP_NO_SSLv2

Do not use the SSLv2 protocol. As of OpenSSL 1.0.1s the **SSL_OP_NO_SSLv2** option is set by default.

SSL_OP_NO_SSLv3

Do not use the SSLv3 protocol. It is recommended that applications should set this option.

SSL_OP_NO_TLSv1

Do not use the TLSv1 protocol.

SSL_OP_NO_TLSv1_1

Do not use the TLSv1.1 protocol.

SSL_OP_NO_TLSv1_2

Do not use the TLSv1.2 protocol.

SSL_OP_NO_SESSION_RESUMPTION_ON_RENEGOTIATION

When performing renegotiation as a server, always start a new session (i.e., session resumption requests are only accepted in the initial handshake). This option is not needed for clients.

SSL_OP_NO_TICKET

Normally clients and servers will, where possible, transparently make use of RFC4507bis tickets for stateless session resumption.

If this option is set this functionality is disabled and tickets will not be used by clients or servers.

SSL_OP_ALLOW_UNSAFE_LEGACY_RENEGOTIATION

Allow legacy insecure renegotiation between OpenSSL and unpatched clients or servers. See the **SECURE RENEGOTIATION** section for more details.

SSL_OP_LEGACY_SERVER_CONNECT

Allow legacy insecure renegotiation between OpenSSL and unpatched servers **only**: this option is currently set by default. See the **SECURE RENEGOTIATION** section for more details.

SECURE RENEGOTIATION

OpenSSL 0.9.8m and later always attempts to use secure renegotiation as described in RFC5746. This counters the prefix attack described in CVE-2009-3555 and elsewhere.

The deprecated and highly broken SSLv2 protocol does not support renegotiation at all: its use is **strongly** discouraged.

This attack has far reaching consequences which application writers should be aware of. In the description below an implementation supporting secure renegotiation is referred to as *patched*. A server not supporting secure renegotiation is referred to as *unpatched*.

The following sections describe the operations permitted by OpenSSL's secure renegotiation implementation.

Patched client and server

Connections and renegotiation are always permitted by OpenSSL implementations.

Unpatched client and patched OpenSSL server

The initial connection succeeds but client renegotiation is denied by the server with a **no_renegotiation** warning alert if TLS v1.0 is used or a fatal **handshake_failure** alert in SSL v3.0.

If the patched OpenSSL server attempts to renegotiate a fatal **handshake_failure** alert is sent. This is because the server code may be unaware of the unpatched nature of the client.

If the option **SSL_OP_ALLOW_UNSAFE_LEGACY_RENEGOTIATION** is set then renegotiation **always** succeeds.

NB: a bug in OpenSSL clients earlier than 0.9.8m (all of which are unpatched) will result in the connection hanging if it receives a **no_renegotiation** alert. OpenSSL versions 0.9.8m and later will regard a **no_renegotiation** alert as fatal and respond with a fatal **handshake_failure** alert. This is because the OpenSSL API currently has no provision to indicate to an application that a renegotiation attempt was refused.

Patched OpenSSL client and unpatched server.

SSL_OP_LEGACY_SERVER_CONNECT
SSL_OP_ALLOW_UNSAFE_LEGACY_RENEGOTIATION

If either the two above options are set then initial connections and renegotiation between patched OpenSSL clients and unpatched servers succeeds. If neither option is set then initial connections to unpatched servers will fail.

The option **SSL_OP_LEGACY_SERVER_CONNECT** is currently set by default even though it has security implications: otherwise it would be impossible to connect to unpatched servers (i.e. all of them initially) and this is clearly not acceptable. Renegotiation is permitted because this does not add any additional security issues: during an attack clients do not see any renegotiations anyway.

As more servers become patched the option **SSL_OP_LEGACY_SERVER_CONNECT** will **not** be set by default in a future version of OpenSSL.

OpenSSL client applications wishing to ensure they can connect to unpatched servers should always **set** **SSL_OP_LEGACY_SERVER_CONNECT**

OpenSSL client applications that want to ensure they can **not** connect to unpatched servers (and thus avoid any security issues) should always **clear** **SSL_OP_LEGACY_SERVER_CONNECT** using SSL_CTX_clear_options() or SSL_clear_options().

The difference between the two options is that **SSL_OP_LEGACY_SERVER_CONNECT** enables initial connections and secure renegotiation between OpenSSL clients and unpatched servers **only**, while **SSL_OP_ALLOW_UNSAFE_LEGACY_RENEGOTIATION** allows initial connections and renegotiation between OpenSSL and unpatched clients or servers.

RETURN VALUES

SSL_CTX_set_options() and SSL_set_options() return the new options bitmask after adding **options**.

SSL_CTX_clear_options() and SSL_clear_options() return the new options bitmask after clearing **options**.

SSL_CTX_get_options() and SSL_get_options() return the current bitmask.

SSL_get_secure_renegotiation_support() returns 1 is the peer supports secure renegotiation and 0 if it does not.

SEE ALSO

ssl(3), SSL_new(3), SSL_clear(3), SSL_CTX_set_tmp_dh_callback(3), SSL_CTX_set_tmp_rsa_callback(3), dhparam(1)

HISTORY

SSL_OP_CIPHER_SERVER_PREFERENCE
SSL_OP_NO_SESSION_RESUMPTION_ON_RENEGOTIATION

These two options have been added in OpenSSL 0.9.7.

SSL_OP_TLS_ROLLBACK_BUG has been added in OpenSSL 0.9.6 and was automatically enabled with **SSL_OP_ALL**. As of 0.9.7, it is no longer included in **SSL_OP_ALL** and must be explicitly set.

SSL_OP_DONT_INSERT_EMPTY_FRAGMENTS has been added in OpenSSL 0.9.6e. Versions up to OpenSSL 0.9.6c do not include the countermeasure that can be disabled with this option (in OpenSSL 0.9.6d, it was always enabled).

SSL_CTX_clear_options() and SSL_clear_options() were first added in OpenSSL 0.9.8m.

SSL_OP_ALLOW_UNSAFE_LEGACY_RENEGOTIATION, **SSL_OP_LEGACY_SERVER_CONNECT** and the function SSL_get_secure_renegotiation_support() were first added in OpenSSL 0.9.8m.

Name

SSL_CTX_set_psk_client_callback and SSL_set_psk_client_callback — set PSK client callback

Synopsis

```
#include  <openssl/ssl.h>
```

```
void SSL_CTX_set_psk_client_callback(SSL_CTX *ctx,
       unsigned int (*callback)(SSL *ssl, const char *hint,
       char *identity, unsigned int max_identity_len,
       unsigned char *psk, unsigned int max_psk_len));
void SSL_set_psk_client_callback(SSL *ssl,
       unsigned int (*callback)(SSL *ssl, const char *hint,
       char *identity, unsigned int max_identity_len,
       unsigned char *psk, unsigned int max_psk_len));
```

DESCRIPTION

A client application must provide a callback function which is called when the client is sending the ClientKeyExchange message to the server.

The purpose of the callback function is to select the PSK identity and the pre-shared key to use during the connection setup phase.

The callback is set using functions SSL_CTX_set_psk_client_callback() or SSL_set_psk_client_callback(). The callback function is given the connection in parameter **ssl**, a **NULL**-terminated PSK identity hint sent by the server in parameter **hint**, a buffer **identity** of length **max_identity_len** bytes where the the resulting **NULL**-terminated identity is to be stored, and a buffer **psk** of length **max_psk_len** bytes where the resulting pre-shared key is to be stored.

NOTES

Note that parameter **hint** given to the callback may be **NULL**.

RETURN VALUES

Return values from the client callback are interpreted as follows:

On success (callback found a PSK identity and a pre-shared key to use) the length (> 0) of **psk** in bytes is returned.

Otherwise or on errors callback should return 0. In this case the connection setup fails.

Name

SSL_CTX_set_quiet_shutdown, SSL_CTX_get_quiet_shutdown, SSL_set_quiet_shutdown and SSL_get_quiet_shutdown — manipulate shutdown behaviour

Synopsis

```
#include  <openssl/ssl.h>
```

```
void SSL_CTX_set_quiet_shutdown(SSL_CTX *ctx, int mode);
int SSL_CTX_get_quiet_shutdown(const SSL_CTX *ctx);
```

```
void SSL_set_quiet_shutdown(SSL *ssl, int mode);
int SSL_get_quiet_shutdown(const SSL *ssl);
```

DESCRIPTION

SSL_CTX_set_quiet_shutdown() sets the "quiet shutdown" flag for **ctx** to be **mode**. SSL objects created from **ctx** inherit the **mode** valid at the time SSL_new(3) is called. **mode** may be 0 or 1.

SSL_CTX_get_quiet_shutdown() returns the "quiet shutdown" setting of **ctx**.

SSL_set_quiet_shutdown() sets the "quiet shutdown" flag for **ssl** to be **mode**. The setting stays valid until **ssl** is removed with SSL_free(3) or SSL_set_quiet_shutdown() is called again. It is not changed when SSL_clear(3) is called. **mode** may be 0 or 1.

SSL_get_quiet_shutdown() returns the "quiet shutdown" setting of **ssl**.

NOTES

Normally when a SSL connection is finished, the parties must send out "close notify" alert messages using SSL_shutdown(3) for a clean shutdown.

When setting the "quiet shutdown" flag to 1, SSL_shutdown(3) will set the internal flags to SSL_SENT_SHUTDOWN | SSL_RECEIVED_SHUTDOWN. (SSL_shutdown(3) then behaves like SSL_set_shutdown(3) called with SSL_SENT_SHUTDOWN | SSL_RECEIVED_SHUTDOWN.) The session is thus considered to be shutdown, but no "close notify" alert is sent to the peer. This behaviour violates the TLS standard.

The default is normal shutdown behaviour as described by the TLS standard.

RETURN VALUES

SSL_CTX_set_quiet_shutdown() and SSL_set_quiet_shutdown() do not return diagnostic information.

SSL_CTX_get_quiet_shutdown() and SSL_get_quiet_shutdown return the current setting.

SEE ALSO

ssl(3), SSL_shutdown(3), SSL_set_shutdown(3), SSL_new(3), SSL_clear(3), SSL_free(3)

Name

SSL_CTX_set_read_ahead, SSL_CTX_set_default_read_ahead, SSL_CTX_get_read_ahead, SSL_CTX_get_default_read_ahead, SSL_set_read_ahead and SSL_get_read_ahead — manage whether to read as many input bytes as possible

Synopsis

```
#include  <openssl/ssl.h>
```

```
int SSL_get_read_ahead(const SSL *s);
void SSL_set_read_ahead(SSL *s, int yes);
```

```
#define SSL_CTX_get_default_read_ahead(ctx)
#define SSL_CTX_set_default_read_ahead(ctx,m)
#define SSL_CTX_get_read_ahead(ctx)
#define SSL_CTX_set_read_ahead(ctx,m)
```

DESCRIPTION

SSL_CTX_set_read_ahead() and SSL_set_read_ahead() set whether we should read as many input bytes as possible (for non-blocking reads) or not. For example if **x** bytes are currently required by OpenSSL, but **y** bytes are available from the underlying BIO (where **y** > **x**), then OpenSSL will read all **y** bytes into its buffer (providing that the buffer is large enough) if reading ahead is on, or **x** bytes otherwise. The parameter **yes** or **m** should be 0 to ensure reading ahead is off, or non zero otherwise.

SSL_CTX_set_default_read_ahead is a synonym for SSL_CTX_set_read_ahead, and SSL_CTX_get_default_read_ahead is a synonym for SSL_CTX_get_read_ahead.

SSL_CTX_get_read_ahead() and SSL_get_read_ahead() indicate whether reading ahead has been set or not.

NOTES

These functions have no impact when used with DTLS. The return values for SSL_CTX_get_read_head() and SSL_get_read_ahead() are undefined for DTLS.

RETURN VALUES

SSL_get_read_ahead and SSL_CTX_get_read_ahead return 0 if reading ahead is off, and non zero otherwise.

SEE ALSO

ssl(3)

Name

SSL_CTX_set_session_cache_mode and SSL_CTX_get_session_cache_mode — enable/disable session caching

Synopsis

```
#include  <openssl/ssl.h>
```

```
long SSL_CTX_set_session_cache_mode(SSL_CTX ctx, long mode);
long SSL_CTX_get_session_cache_mode(SSL_CTX ctx);
```

DESCRIPTION

SSL_CTX_set_session_cache_mode() enables/disables session caching by setting the operational mode for **ctx** to <mode>.

SSL_CTX_get_session_cache_mode() returns the currently used cache mode.

NOTES

The OpenSSL library can store/retrieve SSL/TLS sessions for later reuse. The sessions can be held in memory for each **ctx**, if more than one SSL_CTX object is being maintained, the sessions are unique for each SSL_CTX object.

In order to reuse a session, a client must send the session's id to the server. It can only send exactly one id. The server then either agrees to reuse the session or it starts a full handshake (to create a new session).

A server will lookup up the session in its internal session storage. If the session is not found in internal storage or lookups for the internal storage have been deactivated (SSL_SESS_CACHE_NO_INTERNAL_LOOKUP), the server will try the external storage if available.

Since a client may try to reuse a session intended for use in a different context, the session id context must be set by the server (see SSL_CTX_set_session_id_context(3)).

The following session cache modes and modifiers are available:

SSL_SESS_CACHE_OFF

No session caching for client or server takes place.

SSL_SESS_CACHE_CLIENT

Client sessions are added to the session cache. As there is no reliable way for the OpenSSL library to know whether a session should be reused or which session to choose (due to the abstract BIO layer the SSL engine does not have details about the connection), the application must select the session to be reused by using the SSL_set_session(3) function. This option is not activated by default.

SSL_SESS_CACHE_SERVER

Server sessions are added to the session cache. When a client proposes a session to be reused, the server looks for the corresponding session in (first) the internal session cache (unless SSL_SESS_CACHE_NO_INTERNAL_LOOKUP is set), then (second) in the external cache if available. If the session is found, the server will try to reuse the session. This is the default.

SSL_SESS_CACHE_BOTH

Enable both SSL_SESS_CACHE_CLIENT and SSL_SESS_CACHE_SERVER at the same time.

SSL_SESS_CACHE_NO_AUTO_CLEAR

Normally the session cache is checked for expired sessions every 255 connections using the SSL_CTX_flush_sessions(3) function. Since this may lead to a delay which cannot be controlled, the automatic flushing may be disabled and SSL_CTX_flush_sessions(3) can be called explicitly by the application.

SSL_SESS_CACHE_NO_INTERNAL_LOOKUP

By setting this flag, session-resume operations in an SSL/TLS server will not automatically look up sessions in the internal cache, even if sessions are automatically stored there. If external session caching callbacks are in use, this flag guarantees that all lookups are directed to the external cache. As automatic lookup only applies for SSL/TLS servers, the flag has no effect on clients.

SSL_SESS_CACHE_NO_INTERNAL_STORE

Depending on the presence of SSL_SESS_CACHE_CLIENT and/or SSL_SESS_CACHE_SERVER, sessions negotiated in an SSL/TLS handshake may be cached for possible reuse. Normally a new session is added to the internal cache as well as any external session caching (callback) that is configured for the SSL_CTX. This flag will prevent sessions being stored in the internal cache (though the application can add them manually using SSL_CTX_add_session(3)). Note: in any SSL/TLS servers where external caching is configured, any successful session lookups in the external cache (ie. for session-resume requests) would normally be copied into the local cache before processing continues - this flag prevents these additions to the internal cache as well.

SSL_SESS_CACHE_NO_INTERNAL

Enable both SSL_SESS_CACHE_NO_INTERNAL_LOOKUP and SSL_SESS_CACHE_NO_INTERNAL_STORE at the same time.

The default mode is SSL_SESS_CACHE_SERVER.

RETURN VALUES

SSL_CTX_set_session_cache_mode() returns the previously set cache mode.

SSL_CTX_get_session_cache_mode() returns the currently set cache mode.

SEE ALSO

ssl(3), SSL_set_session(3), SSL_session_reused(3), SSL_CTX_add_session(3), SSL_CTX_sess_number(3), SSL_CTX_sess_set_cache_size(3), SSL_CTX_sess_set_get_cb(3), SSL_CTX_set_session_id_context(3), SSL_CTX_set_timeout(3), SSL_CTX_flush_sessions(3)

HISTORY

SSL_SESS_CACHE_NO_INTERNAL_STORE and SSL_SESS_CACHE_NO_INTERNAL were introduced in OpenSSL 0.9.6h.

Name

SSL_CTX_set_session_id_context and SSL_set_session_id_context — set context within which session can be reused (server side only)

Synopsis

```
#include  <openssl/ssl.h>
```

```
int SSL_CTX_set_session_id_context(SSL_CTX *ctx, const unsigned char *sid_ctx,
                                   unsigned int sid_ctx_len);
int SSL_set_session_id_context(SSL *ssl, const unsigned char *sid_ctx,
                               unsigned int sid_ctx_len);
```

DESCRIPTION

SSL_CTX_set_session_id_context() sets the context **sid_ctx** of length **sid_ctx_len** within which a session can be reused for the **ctx** object.

SSL_set_session_id_context() sets the context **sid_ctx** of length **sid_ctx_len** within which a session can be reused for the **ssl** object.

NOTES

Sessions are generated within a certain context. When exporting/importing sessions with **i2d_SSL_SESSION/d2i_SSL_SESSION** it would be possible, to re-import a session generated from another context (e.g. another application), which might lead to malfunctions. Therefore each application must set its own session id context **sid_ctx** which is used to distinguish the contexts and is stored in exported sessions. The **sid_ctx** can be any kind of binary data with a given length, it is therefore possible to use e.g. the name of the application and/or the hostname and/or service name …

The session id context becomes part of the session. The session id context is set by the SSL/TLS server. The SSL_CTX_set_session_id_context() and SSL_set_session_id_context() functions are therefore only useful on the server side.

OpenSSL clients will check the session id context returned by the server when reusing a session.

The maximum length of the **sid_ctx** is limited to **SSL_MAX_SSL_SESSION_ID_LENGTH**.

WARNINGS

If the session id context is not set on an SSL/TLS server and client certificates are used, stored sessions will not be reused but a fatal error will be flagged and the handshake will fail.

If a server returns a different session id context to an OpenSSL client when reusing a session, an error will be flagged and the handshake will fail. OpenSSL servers will always return the correct session id context, as an OpenSSL server checks the session id context itself before reusing a session as described above.

RETURN VALUES

SSL_CTX_set_session_id_context() and SSL_set_session_id_context() return the following values:

0 The length **sid_ctx_len** of the session id context **sid_ctx** exceeded the maximum allowed length of **SSL_MAX_SSL_SESSION_ID_LENGTH**. The error is logged to the error stack.

1 The operation succeeded.

SEE ALSO

ssl(3)

Name

SSL_CTX_set_ssl_version, SSL_set_ssl_method and SSL_get_ssl_method — choose a new TLS/SSL method

Synopsis

```
#include  <openssl/ssl.h>
```

```
int SSL_CTX_set_ssl_version(SSL_CTX *ctx, const SSL_METHOD *method);
int SSL_set_ssl_method(SSL *s, const SSL_METHOD *method);
const SSL_METHOD *SSL_get_ssl_method(SSL *ssl);
```

DESCRIPTION

SSL_CTX_set_ssl_version() sets a new default TLS/SSL **method** for SSL objects newly created from this **ctx**. SSL objects already created with SSL_new(3) are not affected, except when SSL_clear(3) is being called.

SSL_set_ssl_method() sets a new TLS/SSL **method** for a particular **ssl** object. It may be reset, when SSL_clear() is called.

SSL_get_ssl_method() returns a function pointer to the TLS/SSL method set in **ssl**.

NOTES

The available **method** choices are described in SSL_CTX_new(3).

When SSL_clear(3) is called and no session is connected to an SSL object, the method of the SSL object is reset to the method currently set in the corresponding SSL_CTX object.

RETURN VALUES

The following return values can occur for SSL_CTX_set_ssl_version() and SSL_set_ssl_method():

0 The new choice failed, check the error stack to find out the reason.

1 The operation succeeded.

SEE ALSO

SSL_CTX_new(3), SSL_new(3), SSL_clear(3), ssl(3), SSL_set_connect_state(3)

Name

SSL_CTX_set_timeout and SSL_CTX_get_timeout — manipulate timeout values for session caching

Synopsis

```
#include  <openssl/ssl.h>
```

```
long SSL_CTX_set_timeout(SSL_CTX *ctx, long t);
long SSL_CTX_get_timeout(SSL_CTX *ctx);
```

DESCRIPTION

SSL_CTX_set_timeout() sets the timeout for newly created sessions for **ctx** to **t**. The timeout value **t** must be given in seconds.

SSL_CTX_get_timeout() returns the currently set timeout value for **ctx**.

NOTES

Whenever a new session is created, it is assigned a maximum lifetime. This lifetime is specified by storing the creation time of the session and the timeout value valid at this time. If the actual time is later than creation time plus timeout, the session is not reused.

Due to this realization, all sessions behave according to the timeout value valid at the time of the session negotiation. Changes of the timeout value do not affect already established sessions.

The expiration time of a single session can be modified using the SSL_SESSION_get_time(3) family of functions.

Expired sessions are removed from the internal session cache, whenever SSL_CTX_flush_sessions(3) is called, either directly by the application or automatically (see SSL_CTX_set_session_cache_mode(3))

The default value for session timeout is decided on a per protocol basis, see SSL_get_default_timeout(3). All currently supported protocols have the same default timeout value of 300 seconds.

RETURN VALUES

SSL_CTX_set_timeout() returns the previously set timeout value.

SSL_CTX_get_timeout() returns the currently set timeout value.

SEE ALSO

ssl(3), SSL_CTX_set_session_cache_mode(3), SSL_SESSION_get_time(3), SSL_CTX_flush_sessions(3), SSL_get_default_timeout(3)

Name

SSL_CTX_set_tlsext_status_cb, SSL_CTX_set_tlsext_status_arg, SSL_set_tlsext_status_type, SSL_get_tlsext_status_ocsp_resp and SSL_set_tlsext_status_ocsp_resp — OCSP Certificate Status Request functions

Synopsis

```
#include  <openssl/tls1.h>
```

```
long SSL_CTX_set_tlsext_status_cb(SSL_CTX *ctx,
                                  int (*callback)(SSL *, void *));
long SSL_CTX_set_tlsext_status_arg(SSL_CTX *ctx, void *arg);
```

```
long SSL_set_tlsext_status_type(SSL *s, int type);
```

```
long SSL_get_tlsext_status_ocsp_resp(ssl, unsigned char **resp);
long SSL_set_tlsext_status_ocsp_resp(ssl, unsigned char *resp, int len);
```

DESCRIPTION

A client application may request that a server send back an OCSP status response (also known as OCSP stapling). To do so the client should call the SSL_set_tlsext_status_type() function prior to the start of the handshake. Currently the only supported type is **TLSEXT_STATUSTYPE_ocsp**. This value should be passed in the **type** argument. The client should additionally provide a callback function to decide what to do with the returned OCSP response by calling SSL_CTX_set_tlsext_status_cb(). The callback function should determine whether the returned OCSP response is acceptable or not. The callback will be passed as an argument the value previously set via a call to SSL_CTX_set_tlsext_status_arg(). Note that the callback will not be called in the event of a handshake where session resumption occurs (because there are no Certificates exchanged in such a handshake).

The response returned by the server can be obtained via a call to SSL_get_tlsext_status_ocsp_resp(). The value ***resp** will be updated to point to the OCSP response data and the return value will be the length of that data. Typically a callback would obtain an OCSP_RESPONSE object from this data via a call to the d2i_OCSP_RESPONSE() function. If the server has not provided any response data then ***resp** will be NULL and the return value from SSL_get_tlsext_status_ocsp_resp() will be -1.

A server application must also call the SSL_CTX_set_tlsext_status_cb() function if it wants to be able to provide clients with OCSP Certificate Status responses. Typically the server callback would obtain the server certificate that is being sent back to the client via a call to SSL_get_certificate(); obtain the OCSP response to be sent back; and then set that response data by calling SSL_set_tlsext_status_ocsp_resp(). A pointer to the response data should be provided in the **resp** argument, and the length of that data should be in the **len** argument.

RETURN VALUES

The callback when used on the client side should return a negative value on error; 0 if the response is not acceptable (in which case the handshake will fail) or a positive value if it is acceptable.

The callback when used on the server side should return with either SSL_TLSEXT_ERR_OK (meaning that the OCSP response that has been set should be returned), SSL_TLSEXT_ERR_NOACK (meaning that an OCSP response should not be returned) or SSL_TLSEXT_ERR_ALERT_FATAL (meaning that a fatal error has occurred).

SSL_CTX_set_tlsext_status_cb(), SSL_CTX_set_tlsext_status_arg(), SSL_set_tlsext_status_type() and SSL_set_tlsext_status_ocsp_resp() return 0 on error or 1 on success.

SSL_get_tlsext_status_ocsp_resp() returns the length of the OCSP response data or -1 if there is no OCSP response data.

Name

SSL_CTX_set_tmp_dh_callback, SSL_CTX_set_tmp_dh, SSL_set_tmp_dh_callback and SSL_set_tmp_dh — handle DH keys for ephemeral key exchange

Synopsis

```
#include  <openssl/ssl.h>
```

```
void SSL_CTX_set_tmp_dh_callback(SSL_CTX *ctx,
           DH *(*tmp_dh_callback)(SSL *ssl, int is_export, int keylength));
long SSL_CTX_set_tmp_dh(SSL_CTX *ctx, DH *dh);
```

```
void SSL_set_tmp_dh_callback(SSL *ctx,
           DH *(*tmp_dh_callback)(SSL *ssl, int is_export, int keylength));
long SSL_set_tmp_dh(SSL *ssl, DH *dh)
```

DESCRIPTION

SSL_CTX_set_tmp_dh_callback() sets the callback function for **ctx** to be used when a DH parameters are required to **tmp_dh_callback**. The callback is inherited by all **ssl** objects created from **ctx**.

SSL_CTX_set_tmp_dh() sets DH parameters to be used to be **dh**. The key is inherited by all **ssl** objects created from **ctx**.

SSL_set_tmp_dh_callback() sets the callback only for **ssl**.

SSL_set_tmp_dh() sets the parameters only for **ssl**.

These functions apply to SSL/TLS servers only.

NOTES

When using a cipher with RSA authentication, an ephemeral DH key exchange can take place. Ciphers with DSA keys always use ephemeral DH keys as well. In these cases, the session data are negotiated using the ephemeral/temporary DH key and the key supplied and certified by the certificate chain is only used for signing. Anonymous ciphers (without a permanent server key) also use ephemeral DH keys.

Using ephemeral DH key exchange yields forward secrecy, as the connection can only be decrypted, when the DH key is known. By generating a temporary DH key inside the server application that is lost when the application is left, it becomes impossible for an attacker to decrypt past sessions, even if he gets hold of the normal (certified) key, as this key was only used for signing.

In order to perform a DH key exchange the server must use a DH group (DH parameters) and generate a DH key. The server will always generate a new DH key during the negotiation.

As generating DH parameters is extremely time consuming, an application should not generate the parameters on the fly but supply the parameters. DH parameters can be reused, as the actual key is newly generated during the negotiation. The risk in reusing DH parameters is that an attacker may specialize on a very often used DH group. Applications should therefore generate their own DH parameters during the installation process using the openssl dhparam(1) application. This application guarantees that "strong" primes are used.

Files dh2048.pem, and dh4096.pem in the 'apps' directory of the current version of the OpenSSL distribution contain the 'SKIP' DH parameters, which use safe primes and were generated verifiably pseudo-randomly. These files can be converted into C code using the **-C** option of the dhparam(1) application. Generation of custom DH parameters during installation should still be preferred to stop an attacker from specializing on a commonly used group. Files dh1024.pem and dh512.pem contain old parameters that must not be used by applications.

An application may either directly specify the DH parameters or can supply the DH parameters via a callback function.

Previous versions of the callback used **is_export** and **keylength** parameters to control parameter generation for export and non-export cipher suites. Modern servers that do not support export ciphersuites are advised to either use SSL_CTX_set_tmp_dh() or alternatively, use the callback but ignore **keylength** and **is_export** and simply supply at least 2048-bit parameters in the callback.

EXAMPLES

Setup DH parameters with a key length of 2048 bits. (Error handling partly left out.)

```
Command-line parameter generation:
$ openssl dhparam -out dh_param_2048.pem 2048

Code for setting up parameters during server initialization:

…
SSL_CTX ctx = SSL_CTX_new();
…

/* Set up ephemeral DH parameters. */
DH *dh_2048 = NULL;
FILE *paramfile;
paramfile = fopen("dh_param_2048.pem", "r");
if (paramfile) {
  dh_2048 = PEM_read_DHparams(paramfile, NULL, NULL, NULL);
  fclose(paramfile);
} else {
  /* Error. */
}
if (dh_2048 == NULL) {
 /* Error. */
}
if (SSL_CTX_set_tmp_dh(ctx, dh_2048) != 1) {
  /* Error. */
}
…
```

RETURN VALUES

SSL_CTX_set_tmp_dh_callback() and SSL_set_tmp_dh_callback() do not return diagnostic output.

SSL_CTX_set_tmp_dh() and SSL_set_tmp_dh() do return 1 on success and 0 on failure. Check the error queue to find out the reason of failure.

SEE ALSO

ssl(3), SSL_CTX_set_cipher_list(3), SSL_CTX_set_tmp_rsa_callback(3), SSL_CTX_set_options(3), ciphers(1), dhparam(1)

Name

SSL_CTX_set_tmp_rsa_callback, SSL_CTX_set_tmp_rsa, SSL_CTX_need_tmp_rsa, SSL_set_tmp_rsa_callback, SSL_set_tmp_rsa and SSL_need_tmp_rsa — handle RSA keys for ephemeral key exchange

Synopsis

```
#include  <openssl/ssl.h>
```

```
void SSL_CTX_set_tmp_rsa_callback(SSL_CTX *ctx,
           RSA *(*tmp_rsa_callback)(SSL *ssl, int is_export, int keylength));
long SSL_CTX_set_tmp_rsa(SSL_CTX *ctx, RSA *rsa);
long SSL_CTX_need_tmp_rsa(SSL_CTX *ctx);
```

```
void SSL_set_tmp_rsa_callback(SSL_CTX *ctx,
           RSA *(*tmp_rsa_callback)(SSL *ssl, int is_export, int keylength));
long SSL_set_tmp_rsa(SSL *ssl, RSA *rsa)
long SSL_need_tmp_rsa(SSL *ssl)
```

```
RSA *(*tmp_rsa_callback)(SSL *ssl, int is_export, int keylength);
```

DESCRIPTION

SSL_CTX_set_tmp_rsa_callback() sets the callback function for **ctx** to be used when a temporary/ephemeral RSA key is required to **tmp_rsa_callback**. The callback is inherited by all SSL objects newly created from **ctx** with <SSL_new(3)|SSL_new(3)>. Already created SSL objects are not affected.

SSL_CTX_set_tmp_rsa() sets the temporary/ephemeral RSA key to be used to be **rsa**. The key is inherited by all SSL objects newly created from **ctx** with <SSL_new(3)|SSL_new(3)>. Already created SSL objects are not affected.

SSL_CTX_need_tmp_rsa() returns 1, if a temporary/ephemeral RSA key is needed for RSA-based strength-limited 'exportable' ciphersuites because a RSA key with a keysize larger than 512 bits is installed.

SSL_set_tmp_rsa_callback() sets the callback only for **ssl**.

SSL_set_tmp_rsa() sets the key only for **ssl**.

SSL_need_tmp_rsa() returns 1, if a temporary/ephemeral RSA key is needed, for RSA-based strength-limited 'exportable' ciphersuites because a RSA key with a keysize larger than 512 bits is installed.

These functions apply to SSL/TLS servers only.

NOTES

When using a cipher with RSA authentication, an ephemeral RSA key exchange can take place. In this case the session data are negotiated using the ephemeral/temporary RSA key and the RSA key supplied and certified by the certificate chain is only used for signing.

Under previous export restrictions, ciphers with RSA keys shorter (512 bits) than the usual key length of 1024 bits were created. To use these ciphers with RSA keys of usual length, an ephemeral key exchange must be performed, as the normal (certified) key cannot be directly used.

Using ephemeral RSA key exchange yields forward secrecy, as the connection can only be decrypted, when the RSA key is known. By generating a temporary RSA key inside the server application that is lost when the application is left, it becomes impossible for an attacker to decrypt past sessions, even if he gets hold of the normal (certified) RSA key, as this key was used for signing only. The downside is that creating a RSA key is computationally expensive.

Additionally, the use of ephemeral RSA key exchange is only allowed in the TLS standard, when the RSA key can be used for signing only, that is for export ciphers. Using ephemeral RSA key exchange for other purposes violates the standard and can break interoperability with clients. It is therefore strongly recommended to not use ephemeral RSA key exchange and use EDH (Ephemeral Diffie-Hellman) key exchange instead in order to achieve forward secrecy (see SSL_CTX_set_tmp_dh_callback(3)).

An application may either directly specify the key or can supply the key via a callback function. The callback approach has the advantage, that the callback may generate the key only in case it is actually needed. As the generation of a RSA key is however costly, it will lead to a significant delay in the handshake procedure. Another advantage of the callback function is that it can supply keys of different size while the explicit setting of the key is only useful for key size of 512 bits to satisfy the export restricted ciphers and does give away key length if a longer key would be allowed.

The **tmp_rsa_callback** is called with the **keylength** needed and the **is_export** information. The **is_export** flag is set, when the ephemeral RSA key exchange is performed with an export cipher.

EXAMPLES

Generate temporary RSA keys to prepare ephemeral RSA key exchange. As the generation of a RSA key costs a lot of computer time, they saved for later reuse. For demonstration purposes, two keys for 512 bits and 1024 bits respectively are generated.

```
…
/* Set up ephemeral RSA stuff */
RSA *rsa_512 = NULL;
RSA *rsa_1024 = NULL;

rsa_512 = RSA_generate_key(512,RSA_F4,NULL,NULL);
if (rsa_512 == NULL)
    evaluate_error_queue();

rsa_1024 = RSA_generate_key(1024,RSA_F4,NULL,NULL);
if (rsa_1024 == NULL)
  evaluate_error_queue();

…

RSA *tmp_rsa_callback(SSL *s, int is_export, int keylength)
{
   RSA *rsa_tmp=NULL;

   switch (keylength) {
   case 512:
     if (rsa_512)
       rsa_tmp = rsa_512;
     else { /* generate on the fly, should not happen in this example */
       rsa_tmp = RSA_generate_key(keylength,RSA_F4,NULL,NULL);
       rsa_512 = rsa_tmp; /* Remember for later reuse */
     }
     break;
   case 1024:
     if (rsa_1024)
       rsa_tmp=rsa_1024;
     else
       should_not_happen_in_this_example();
     break;
   default:
     /* Generating a key on the fly is very costly, so use what is there */
     if (rsa_1024)
       rsa_tmp=rsa_1024;
     else
       rsa_tmp=rsa_512; /* Use at least a shorter key */
   }
   return(rsa_tmp);
}
```

RETURN VALUES

SSL_CTX_set_tmp_rsa_callback() and SSL_set_tmp_rsa_callback() do not return diagnostic output.

SSL_CTX_set_tmp_rsa() and SSL_set_tmp_rsa() do return 1 on success and 0 on failure. Check the error queue to find out the reason of failure.

SSL_CTX_need_tmp_rsa() and SSL_need_tmp_rsa() return 1 if a temporary RSA key is needed and 0 otherwise.

SEE ALSO

ssl(3), SSL_CTX_set_cipher_list(3), SSL_CTX_set_options(3), SSL_CTX_set_tmp_dh_callback(3), SSL_new(3), ciphers(1)

Name

SSL_CTX_set_verify, SSL_set_verify, SSL_CTX_set_verify_depth and SSL_set_verify_depth — set peer certificate verification parameters

Synopsis

```
#include  <openssl/ssl.h>
```

```
void SSL_CTX_set_verify(SSL_CTX *ctx, int mode,
                        int (*verify_callback)(int, X509_STORE_CTX *));
void SSL_set_verify(SSL *s, int mode,
                    int (*verify_callback)(int, X509_STORE_CTX *));
void SSL_CTX_set_verify_depth(SSL_CTX *ctx,int depth);
void SSL_set_verify_depth(SSL *s, int depth);
```

```
int verify_callback(int preverify_ok, X509_STORE_CTX *x509_ctx);
```

DESCRIPTION

SSL_CTX_set_verify() sets the verification flags for **ctx** to be **mode** and specifies the **verify_callback** function to be used. If no callback function shall be specified, the NULL pointer can be used for **verify_callback**.

SSL_set_verify() sets the verification flags for **ssl** to be **mode** and specifies the **verify_callback** function to be used. If no callback function shall be specified, the NULL pointer can be used for **verify_callback**. In this case last **verify_callback** set specifically for this **ssl** remains. If no special **callback** was set before, the default callback for the underlying **ctx** is used, that was valid at the time **ssl** was created with SSL_new(3).

SSL_CTX_set_verify_depth() sets the maximum **depth** for the certificate chain verification that shall be allowed for **ctx**. (See the BUGS section.)

SSL_set_verify_depth() sets the maximum **depth** for the certificate chain verification that shall be allowed for **ssl**. (See the BUGS section.)

NOTES

The verification of certificates can be controlled by a set of logically or'ed **mode** flags:

SSL_VERIFY_NONE

> **Server mode:** the server will not send a client certificate request to the client, so the client will not send a certificate.
>
> **Client mode:** if not using an anonymous cipher (by default disabled), the server will send a certificate which will be checked. The result of the certificate verification process can be checked after the TLS/SSL handshake using the SSL_get_verify_result(3) function. The handshake will be continued regardless of the verification result.

SSL_VERIFY_PEER

> **Server mode:** the server sends a client certificate request to the client. The certificate returned (if any) is checked. If the verification process fails, the TLS/SSL handshake is immediately terminated with an alert message containing the reason for the verification failure. The behaviour can be controlled by the additional SSL_VERIFY_FAIL_IF_NO_PEER_CERT and SSL_VERIFY_CLIENT_ONCE flags.
>
> **Client mode:** the server certificate is verified. If the verification process fails, the TLS/SSL handshake is immediately terminated with an alert message containing the reason for the verification failure. If no server certificate is sent, because an anonymous cipher is used, SSL_VERIFY_PEER is ignored.

SSL_VERIFY_FAIL_IF_NO_PEER_CERT

> **Server mode:** if the client did not return a certificate, the TLS/SSL handshake is immediately terminated with a "handshake failure" alert. This flag must be used together with SSL_VERIFY_PEER.
>
> **Client mode:** ignored

SSL_VERIFY_CLIENT_ONCE

> **Server mode:** only request a client certificate on the initial TLS/SSL handshake. Do not ask for a client certificate again in case of a renegotiation. This flag must be used together with SSL_VERIFY_PEER.
>
> **Client mode:** ignored

Exactly one of the **mode** flags SSL_VERIFY_NONE and SSL_VERIFY_PEER must be set at any time.

The actual verification procedure is performed either using the built-in verification procedure or using another application provided verification function set with SSL_CTX_set_cert_verify_callback(3). The following descriptions apply in the case of the built-in procedure. An application provided procedure also has access to the verify depth information and the verify_callback() function, but the way this information is used may be different.

SSL_CTX_set_verify_depth() and SSL_set_verify_depth() set the limit up to which depth certificates in a chain are used during the verification procedure. If the certificate chain is longer than allowed, the certificates above the limit are ignored. Error messages are generated as if these certificates would not be present, most likely a X509_V_ERR_UNABLE_TO_GET_ISSUER_CERT_LOCALLY will be issued. The depth count is "level 0:peer certificate", "level 1: CA certificate", "level 2: higher level CA certificate", and so on. Setting the maximum depth to 2 allows the levels 0, 1, and 2. The default depth limit is 100, allowing for the peer certificate and additional 100 CA certificates.

The **verify_callback** function is used to control the behaviour when the SSL_VERIFY_PEER flag is set. It must be supplied by the application and receives two arguments: **preverify_ok** indicates, whether the verification of the certificate in question was passed (preverify_ok=1) or not (preverify_ok=0). **x509_ctx** is a pointer to the complete context used for the certificate chain verification.

The certificate chain is checked starting with the deepest nesting level (the root CA certificate) and worked upward to the peer's certificate. At each level signatures and issuer attributes are checked. Whenever a verification error is found, the error number is stored in **x509_ctx** and **verify_callback** is called with **preverify_ok**=0. By applying X509_CTX_store_* functions **verify_callback** can locate the certificate in question and perform additional steps (see EXAMPLES). If no error is found for a certificate, **verify_callback** is called with **preverify_ok**=1 before advancing to the next level.

The return value of **verify_callback** controls the strategy of the further verification process. If **verify_callback** returns 0, the verification process is immediately stopped with "verification failed" state. If SSL_VERIFY_PEER is set, a verification failure alert is sent to the peer and the TLS/SSL handshake is terminated. If **verify_callback** returns 1, the verification process is continued. If **verify_callback** always returns 1, the TLS/SSL handshake will not be terminated with respect to verification failures and the connection will be established. The calling process can however retrieve the error code of the last verification error using SSL_get_verify_result(3) or by maintaining its own error storage managed by **verify_callback**.

If no **verify_callback** is specified, the default callback will be used. Its return value is identical to **preverify_ok**, so that any verification failure will lead to a termination of the TLS/SSL handshake with an alert message, if SSL_VERIFY_PEER is set.

BUGS

In client mode, it is not checked whether the SSL_VERIFY_PEER flag is set, but whether SSL_VERIFY_NONE is not set. This can lead to unexpected behaviour, if the SSL_VERIFY_PEER and SSL_VERIFY_NONE are not used as required (exactly one must be set at any time).

The certificate verification depth set with SSL[_CTX]_verify_depth() stops the verification at a certain depth. The error message produced will be that of an incomplete certificate chain and not X509_V_ERR_CERT_CHAIN_TOO_LONG as may be expected.

RETURN VALUES

The SSL*_set_verify*() functions do not provide diagnostic information.

EXAMPLES

The following code sequence realizes an example **verify_callback** function that will always continue the TLS/SSL handshake regardless of verification failure, if wished. The callback realizes a verification depth limit with more informational output.

All verification errors are printed; information about the certificate chain is printed on request. The example is realized for a server that does allow but not require client certificates.

The example makes use of the ex_data technique to store application data into/retrieve application data from the SSL structure (see SSL_get_ex_new_index(3), SSL_get_ex_data_X509_STORE_CTX_idx(3)).

```
…
typedef struct {
  int verbose_mode;
  int verify_depth;
  int always_continue;
} mydata_t;
int mydata_index;
…
static int verify_callback(int preverify_ok, X509_STORE_CTX *ctx)
{
   char    buf[256];
   X509   *err_cert;
   int     err, depth;
   SSL    *ssl;
   mydata_t *mydata;

err_cert = X509_STORE_CTX_get_current_cert(ctx);
err = X509_STORE_CTX_get_error(ctx);
depth = X509_STORE_CTX_get_error_depth(ctx);

/*
 * Retrieve the pointer to the SSL of the connection currently treated
 * and the application specific data stored into the SSL object.
 */
ssl = X509_STORE_CTX_get_ex_data(ctx, SSL_get_ex_data_X509_STORE_CTX_idx());
mydata = SSL_get_ex_data(ssl, mydata_index);

X509_NAME_oneline(X509_get_subject_name(err_cert), buf, 256);

/*
 * Catch a too long certificate chain. The depth limit set using
 * SSL_CTX_set_verify_depth() is by purpose set to "limit+1" so
 * that whenever the "depth>verify_depth" condition is met, we
 * have violated the limit and want to log this error condition.
 * We must do it here, because the CHAIN_TOO_LONG error would not
 * be found explicitly; only errors introduced by cutting off the
 * additional certificates would be logged.
 */
if (depth > mydata->verify_depth) {
    preverify_ok = 0;
    err = X509_V_ERR_CERT_CHAIN_TOO_LONG;
    X509_STORE_CTX_set_error(ctx, err);
}
if (!preverify_ok) {
    printf("verify error:num=%d:%s:depth=%d:%s\n", err,
             X509_verify_cert_error_string(err), depth, buf);
}
else if (mydata->verbose_mode)
{
    printf("depth=%d:%s\n", depth, buf);
}

/*
 * At this point, err contains the last verification error. We can use
 * it for something special
 */
if (!preverify_ok && (err == X509_V_ERR_UNABLE_TO_GET_ISSUER_CERT))
{
  X509_NAME_oneline(X509_get_issuer_name(ctx->current_cert), buf, 256);
  printf("issuer= %s\n", buf);
}

   if (mydata->always_continue)
     return 1;
   else
     return preverify_ok;
}
…

mydata_t mydata;

…
mydata_index = SSL_get_ex_new_index(0, "mydata index", NULL, NULL, NULL);

…
SSL_CTX_set_verify(ctx, SSL_VERIFY_PEER|SSL_VERIFY_CLIENT_ONCE,
                   verify_callback);

/*
 * Let the verify_callback catch the verify_depth error so that we get
 * an appropriate error in the logfile.
 */
SSL_CTX_set_verify_depth(verify_depth + 1);

/*
 * Set up the SSL specific data into "mydata" and store it into th SSL
 * structure.
 */
mydata.verify_depth = verify_depth; …
SSL_set_ex_data(ssl, mydata_index, &mydata);

…
SSL_accept(ssl);       /* check of success left out for clarity */
if (peer = SSL_get_peer_certificate(ssl))
{
  if (SSL_get_verify_result(ssl) == X509_V_OK)
  {
    /* The client sent a certificate which verified OK */
  }
}
```

SEE ALSO

ssl(3), SSL_new(3), SSL_CTX_get_verify_mode(3), SSL_get_verify_result(3), SSL_CTX_load_verify_locations(3), SSL_get_peer_certificate(3), SSL_CTX_set_cert_verify_callback(3), SSL_get_ex_data_X509_STORE_CTX_idx(3), SSL_get_ex_new_index(3)

Name

SSL_CTX_use_certificate, SSL_CTX_use_certificate_ASN1, SSL_CTX_use_certificate_file, SSL_use_certificate, SSL_use_certificate_ASN1, SSL_use_certificate_file, SSL_CTX_use_certificate_chain_file, SSL_CTX_use_PrivateKey, SSL_CTX_use_PrivateKey_ASN1, SSL_CTX_use_PrivateKey_file, SSL_CTX_use_RSAPrivateKey, SSL_CTX_use_RSAPrivateKey_ASN1, SSL_CTX_use_RSAPrivateKey_file, SSL_use_PrivateKey_file, SSL_use_PrivateKey_ASN1, SSL_use_PrivateKey, SSL_use_RSAPrivateKey, SSL_use_RSAPrivateKey_ASN1, SSL_use_RSAPrivateKey_file, SSL_CTX_check_private_key and SSL_check_private_key — load certificate and key data

Synopsis

```
#include  <openssl/ssl.h>
```

```
int SSL_CTX_use_certificate(SSL_CTX *ctx, X509 *x);
int SSL_CTX_use_certificate_ASN1(SSL_CTX *ctx, int len, unsigned char *d);
int SSL_CTX_use_certificate_file(SSL_CTX *ctx, const char *file, int type);
int SSL_use_certificate(SSL *ssl, X509 *x);
int SSL_use_certificate_ASN1(SSL *ssl, unsigned char *d, int len);
int SSL_use_certificate_file(SSL *ssl, const char *file, int type);
```

```
int SSL_CTX_use_certificate_chain_file(SSL_CTX *ctx, const char *file);
```

```
int SSL_CTX_use_PrivateKey(SSL_CTX *ctx, EVP_PKEY *pkey);
int SSL_CTX_use_PrivateKey_ASN1(int pk, SSL_CTX *ctx, unsigned char *d,
                                long len);
int SSL_CTX_use_PrivateKey_file(SSL_CTX *ctx, const char *file, int type);
int SSL_CTX_use_RSAPrivateKey(SSL_CTX *ctx, RSA *rsa);
int SSL_CTX_use_RSAPrivateKey_ASN1(SSL_CTX *ctx, unsigned char *d, long len);
int SSL_CTX_use_RSAPrivateKey_file(SSL_CTX *ctx, const char *file, int type);
int SSL_use_PrivateKey(SSL *ssl, EVP_PKEY *pkey);
int SSL_use_PrivateKey_ASN1(int pk,SSL *ssl, unsigned char *d, long len);
int SSL_use_PrivateKey_file(SSL *ssl, const char *file, int type);
int SSL_use_RSAPrivateKey(SSL *ssl, RSA *rsa);
int SSL_use_RSAPrivateKey_ASN1(SSL *ssl, unsigned char *d, long len);
int SSL_use_RSAPrivateKey_file(SSL *ssl, const char *file, int type);
```

```
int SSL_CTX_check_private_key(const SSL_CTX *ctx);
int SSL_check_private_key(const SSL *ssl);
```

DESCRIPTION

These functions load the certificates and private keys into the SSL_CTX or SSL object, respectively.

The SSL_CTX_* class of functions loads the certificates and keys into the SSL_CTX object **ctx**. The information is passed to SSL objects **ssl** created from **ctx** with SSL_new(3) by copying, so that changes applied to **ctx** do not propagate to already existing SSL objects.

The SSL_* class of functions only loads certificates and keys into a specific SSL object. The specific information is kept, when SSL_clear(3) is called for this SSL object.

SSL_CTX_use_certificate() loads the certificate **x** into **ctx**, SSL_use_certificate() loads **x** into **ssl**. The rest of the certificates needed to form the complete certificate chain can be specified using the SSL_CTX_add_extra_chain_cert(3) function.

SSL_CTX_use_certificate_ASN1() loads the ASN1 encoded certificate from the memory location **d** (with length **len**) into **ctx**, SSL_use_certificate_ASN1() loads the ASN1 encoded certificate into **ssl**.

SSL_CTX_use_certificate_file() loads the first certificate stored in **file** into **ctx**. The formatting **type** of the certificate must be specified from the known types SSL_FILETYPE_PEM, SSL_FILETYPE_ASN1. SSL_use_certificate_file() loads the certificate from **file** into **ssl**. See the NOTES section on why SSL_CTX_use_certificate_chain_file() should be preferred.

SSL_CTX_use_certificate_chain_file() loads a certificate chain from **file** into **ctx**. The certificates must be in PEM format and must be sorted starting with the subject's certificate (actual client or server certificate), followed by intermediate CA certificates if applicable, and ending at the highest level (root) CA. There is no corresponding function working on a single SSL object.

SSL_CTX_use_PrivateKey() adds **pkey** as private key to **ctx**. SSL_CTX_use_RSAPrivateKey() adds the private key **rsa** of type RSA to **ctx**. SSL_use_PrivateKey() adds **pkey** as private key to **ssl**; SSL_use_RSAPrivateKey() adds **rsa** as private key of type RSA to **ssl**. If a certificate has already been set and the private does not belong to the certificate an error is returned. To change a certificate, private key pair the new certificate needs to be set with SSL_use_certificate() or SSL_CTX_use_certificate() before setting the private key with SSL_CTX_use_PrivateKey() or SSL_use_PrivateKey().

SSL_CTX_use_PrivateKey_ASN1() adds the private key of type **pk** stored at memory location **d** (length **len**) to **ctx**. SSL_CTX_use_RSAPrivateKey_ASN1() adds the private key of type RSA stored at memory location **d** (length **len**) to **ctx**. SSL_use_PrivateKey_ASN1() and SSL_use_RSAPrivateKey_ASN1() add the private key to **ssl**.

SSL_CTX_use_PrivateKey_file() adds the first private key found in **file** to **ctx**. The formatting **type** of the certificate must be specified from the known types SSL_FILETYPE_PEM, SSL_FILETYPE_ASN1. SSL_CTX_use_RSAPrivateKey_file() adds the first private RSA key found in **file** to **ctx**. SSL_use_PrivateKey_file() adds the first private key found in **file** to **ssl**; SSL_use_RSAPrivateKey_file() adds the first private RSA key found to **ssl**.

SSL_CTX_check_private_key() checks the consistency of a private key with the corresponding certificate loaded into **ctx**. If more than one key/certificate pair (RSA/DSA) is installed, the last item installed will be checked. If e.g. the last item was a RSA certificate or key, the RSA key/certificate pair will be checked. SSL_check_private_key() performs the same check for **ssl**. If no key/certificate was explicitly added for this **ssl**, the last item added into **ctx** will be checked.

NOTES

The internal certificate store of OpenSSL can hold two private key/certificate pairs at a time: one key/certificate of type RSA and one key/certificate of type DSA. The certificate used depends on the cipher select, see also SSL_CTX_set_cipher_list(3).

When reading certificates and private keys from file, files of type SSL_FILETYPE_ASN1 (also known as **DER**, binary encoding) can only contain one certificate or private key, consequently SSL_CTX_use_certificate_chain_file() is only applicable to PEM formatting. Files of type SSL_FILETYPE_PEM can contain more than one item.

SSL_CTX_use_certificate_chain_file() adds the first certificate found in the file to the certificate store. The other certificates are added to the store of chain certificates using SSL_CTX_add_extra_chain_cert(3). There exists only one extra chain store, so that the same chain is appended to both types of certificates, RSA and DSA! If it is not intended to use both type of certificate at the same time, it is recommended to use the SSL_CTX_use_certificate_chain_file() instead of the SSL_CTX_use_certificate_file() function in order to allow the use of complete certificate chains even when no trusted CA storage is used or when the CA issuing the certificate shall not be added to the trusted CA storage.

If additional certificates are needed to complete the chain during the TLS negotiation, CA certificates are additionally looked up in the locations of trusted CA certificates, see SSL_CTX_load_verify_locations(3).

The private keys loaded from file can be encrypted. In order to successfully load encrypted keys, a function returning the passphrase must have been supplied, see SSL_CTX_set_default_passwd_cb(3). (Certificate files might be encrypted as well from the technical point of view, it however does not make sense as the data in the certificate is considered public anyway.)

RETURN VALUES

On success, the functions return 1. Otherwise check out the error stack to find out the reason.

SEE ALSO

ssl(3), SSL_new(3), SSL_clear(3), SSL_CTX_load_verify_locations(3), SSL_CTX_set_default_passwd_cb(3), SSL_CTX_set_cipher_list(3), SSL_CTX_set_client_cert_cb(3), SSL_CTX_add_extra_chain_cert(3)

HISTORY

Support for DER encoded private keys (SSL_FILETYPE_ASN1) in SSL_CTX_use_PrivateKey_file() and SSL_use_PrivateKey_file() was added in 0.9.8 .

Name

SSL_CTX_use_psk_identity_hint, SSL_use_psk_identity_hint, SSL_CTX_set_psk_server_callback and SSL_set_psk_server_callback — set PSK identity hint to use

Synopsis

```
#include  <openssl/ssl.h>
```

```
int SSL_CTX_use_psk_identity_hint(SSL_CTX *ctx, const char *hint);
int SSL_use_psk_identity_hint(SSL *ssl, const char *hint);
```

```
void SSL_CTX_set_psk_server_callback(SSL_CTX *ctx,
       unsigned int (*callback)(SSL *ssl, const char *identity,
       unsigned char *psk, int max_psk_len));
void SSL_set_psk_server_callback(SSL *ssl,
       unsigned int (*callback)(SSL *ssl, const char *identity,
       unsigned char *psk, int max_psk_len));
```

DESCRIPTION

SSL_CTX_use_psk_identity_hint() sets the given **NULL**-terminated PSK identity hint **hint** to SSL context object **ctx**. SSL_use_psk_identity_hint() sets the given **NULL**-terminated PSK identity hint **hint** to SSL connection object **ssl**. If **hint** is **NULL** the current hint from **ctx** or **ssl** is deleted.

In the case where PSK identity hint is **NULL**, the server does not send the ServerKeyExchange message to the client.

A server application must provide a callback function which is called when the server receives the ClientKeyExchange message from the client. The purpose of the callback function is to validate the received PSK identity and to fetch the pre-shared key used during the connection setup phase. The callback is set using functions SSL_CTX_set_psk_server_callback() or SSL_set_psk_server_callback(). The callback function is given the connection in parameter **ssl**, **NULL**-terminated PSK identity sent by the client in parameter **identity**, and a buffer **psk** of length **max_psk_len** bytes where the pre-shared key is to be stored.

RETURN VALUES

SSL_CTX_use_psk_identity_hint() and SSL_use_psk_identity_hint() return 1 on success, 0 otherwise.

Return values from the server callback are interpreted as follows:

> 0 PSK identity was found and the server callback has provided the PSK successfully in parameter **psk**. Return value is the length of **psk** in bytes. It is an error to return a value greater than **max_psk_len**.

If the PSK identity was not found but the callback instructs the protocol to continue anyway, the callback must provide some random data to **psk** and return the length of the random data, so the connection will fail with decryption_error before it will be finished completely.

0 PSK identity was not found. An "unknown_psk_identity" alert message will be sent and the connection setup fails.

Name

SSL_do_handshake — perform a TLS/SSL handshake

Synopsis

```
#include  <openssl/ssl.h>
```

```
int SSL_do_handshake(SSL *ssl);
```

DESCRIPTION

SSL_do_handshake() will wait for a SSL/TLS handshake to take place. If the connection is in client mode, the handshake will be started. The handshake routines may have to be explicitly set in advance using either SSL_set_connect_state(3) or SSL_set_accept_state(3).

NOTES

The behaviour of SSL_do_handshake() depends on the underlying BIO.

If the underlying BIO is **blocking**, SSL_do_handshake() will only return once the handshake has been finished or an error occurred, except for SGC (Server Gated Cryptography). For SGC, SSL_do_handshake() may return with -1, but SSL_get_error() will yield **SSL_ERROR_WANT_READ/WRITE** and SSL_do_handshake() should be called again.

If the underlying BIO is **non-blocking**, SSL_do_handshake() will also return when the underlying BIO could not satisfy the needs of SSL_do_handshake() to continue the handshake. In this case a call to SSL_get_error() with the return value of SSL_do_handshake() will yield **SSL_ERROR_WANT_READ** or **SSL_ERROR_WANT_WRITE**. The calling process then must repeat the call after taking appropriate action to satisfy the needs of SSL_do_handshake(). The action depends on the underlying BIO. When using a non-blocking socket, nothing is to be done, but select() can be used to check for the required condition. When using a buffering BIO, like a BIO pair, data must be written into or retrieved out of the BIO before being able to continue.

RETURN VALUES

The following return values can occur:

0 The TLS/SSL handshake was not successful but was shut down controlled and by the specifications of the TLS/SSL protocol. Call SSL_get_error() with the return value **ret** to find out the reason.

1 The TLS/SSL handshake was successfully completed, a TLS/SSL connection has been established.

<0 The TLS/SSL handshake was not successful because a fatal error occurred either at the protocol level or a connection failure occurred. The shutdown was not clean. It can also occur of action is need to continue the operation for non-blocking BIOs. Call SSL_get_error() with the return value **ret** to find out the reason.

SEE ALSO

SSL_get_error(3), SSL_connect(3), SSL_accept(3), ssl(3), bio(3), SSL_set_connect_state(3)

Name

SSL_free — free an allocated SSL structure

Synopsis

```
#include  <openssl/ssl.h>
```

```
void SSL_free(SSL *ssl);
```

DESCRIPTION

SSL_free() decrements the reference count of **ssl**, and removes the SSL structure pointed to by **ssl** and frees up the allocated memory if the reference count has reached 0.

NOTES

SSL_free() also calls the free()ing procedures for indirectly affected items, if applicable: the buffering BIO, the read and write BIOs, cipher lists specially created for this **ssl**, the **SSL_SESSION**. Do not explicitly free these indirectly freed up items before or after calling SSL_free(), as trying to free things twice may lead to program failure.

The ssl session has reference counts from two users: the SSL object, for which the reference count is removed by SSL_free() and the internal session cache. If the session is considered bad, because SSL_shutdown(3) was not called for the connection and SSL_set_shutdown(3) was not used to set the SSL_SENT_SHUTDOWN state, the session will also be removed from the session cache as required by RFC2246.

RETURN VALUES

SSL_free() does not provide diagnostic information.

SSL_new(3), SSL_clear(3), SSL_shutdown(3), SSL_set_shutdown(3), ssl(3)

Name

SSL_get_ciphers and SSL_get_cipher_list — get list of available SSL_CIPHERs

Synopsis

```
#include  <openssl/ssl.h>
```

```
STACK_OF(SSL_CIPHER) *SSL_get_ciphers(const SSL *ssl);
const char *SSL_get_cipher_list(const SSL *ssl, int priority);
```

DESCRIPTION

SSL_get_ciphers() returns the stack of available SSL_CIPHERs for **ssl**, sorted by preference. If **ssl** is NULL or no ciphers are available, NULL is returned.

SSL_get_cipher_list() returns a pointer to the name of the SSL_CIPHER listed for **ssl** with **priority**. If **ssl** is NULL, no ciphers are available, or there are less ciphers than **priority** available, NULL is returned.

NOTES

The details of the ciphers obtained by SSL_get_ciphers() can be obtained using the SSL_CIPHER_get_name(3) family of functions.

Call SSL_get_cipher_list() with **priority** starting from 0 to obtain the sorted list of available ciphers, until NULL is returned.

RETURN VALUES

See DESCRIPTION

SEE ALSO

ssl(3), SSL_CTX_set_cipher_list(3), SSL_CIPHER_get_name(3)

Name

SSL_get_client_CA_list and SSL_CTX_get_client_CA_list — get list of client CAs

Synopsis

```
#include  <openssl/ssl.h>
```

```
STACK_OF(X509_NAME) *SSL_get_client_CA_list(const SSL *s);
STACK_OF(X509_NAME) *SSL_CTX_get_client_CA_list(const SSL_CTX *ctx);
```

DESCRIPTION

SSL_CTX_get_client_CA_list() returns the list of client CAs explicitly set for **ctx** using SSL_CTX_set_client_CA_list(3).

SSL_get_client_CA_list() returns the list of client CAs explicitly set for **ssl** using SSL_set_client_CA_list() or **ssl**'s SSL_CTX object with SSL_CTX_set_client_CA_list(3), when in server mode. In client mode, SSL_get_client_CA_list returns the list of client CAs sent from the server, if any.

RETURN VALUES

SSL_CTX_set_client_CA_list() and SSL_set_client_CA_list() do not return diagnostic information.

SSL_CTX_add_client_CA() and SSL_add_client_CA() have the following return values:

STACK_OF(X509_NAMES) List of CA names explicitly set (for **ctx** or in server mode) or send by the server (client mode).

NULL No client CA list was explicitly set (for **ctx** or in server mode) or the server did not send a list of CAs (client mode).

SEE ALSO

ssl(3), SSL_CTX_set_client_CA_list(3), SSL_CTX_set_client_cert_cb(3)

Name

SSL_get_current_cipher, SSL_get_cipher, SSL_get_cipher_name, SSL_get_cipher_bits and SSL_get_cipher_version — get SSL_CIPHER of a connection

Synopsis

```
#include  <openssl/ssl.h>
```

```
SSL_CIPHER *SSL_get_current_cipher(const SSL *ssl);
#define SSL_get_cipher(s) \
               SSL_CIPHER_get_name(SSL_get_current_cipher(s))
#define SSL_get_cipher_name(s) \
               SSL_CIPHER_get_name(SSL_get_current_cipher(s))
#define SSL_get_cipher_bits(s,np) \
               SSL_CIPHER_get_bits(SSL_get_current_cipher(s),np)
#define SSL_get_cipher_version(s) \
               SSL_CIPHER_get_version(SSL_get_current_cipher(s))
```

DESCRIPTION

SSL_get_current_cipher() returns a pointer to an SSL_CIPHER object containing the description of the actually used cipher of a connection established with the **ssl** object.

SSL_get_cipher() and SSL_get_cipher_name() are identical macros to obtain the name of the currently used cipher. SSL_get_cipher_bits() is a macro to obtain the number of secret/algorithm bits used and SSL_get_cipher_version() returns the protocol name. See SSL_CIPHER_get_name(3) for more details.

RETURN VALUES

SSL_get_current_cipher() returns the cipher actually used or NULL, when no session has been established.

SEE ALSO

ssl(3), SSL_CIPHER_get_name(3)

Name

SSL_get_default_timeout — get default session timeout value

Synopsis

```
#include  <openssl/ssl.h>
```

```
long SSL_get_default_timeout(const SSL *ssl);
```

DESCRIPTION

SSL_get_default_timeout() returns the default timeout value assigned to SSL_SESSION objects negotiated for the protocol valid for **ssl**.

NOTES

Whenever a new session is negotiated, it is assigned a timeout value, after which it will not be accepted for session reuse. If the timeout value was not explicitly set using SSL_CTX_set_timeout(3), the hardcoded default timeout for the protocol will be used.

SSL_get_default_timeout() return this hardcoded value, which is 300 seconds for all currently supported protocols (SSLv2, SSLv3, and TLSv1).

RETURN VALUES

See description.

SEE ALSO

ssl(3), SSL_CTX_set_session_cache_mode(3), SSL_SESSION_get_time(3), SSL_CTX_flush_sessions(3), SSL_get_default_timeout(3)

Name

SSL_get_error — obtain result code for TLS/SSL I/O operation

Synopsis

```
#include  <openssl/ssl.h>
```

```
int SSL_get_error(const SSL *ssl, int ret);
```

DESCRIPTION

SSL_get_error() returns a result code (suitable for the C "switch" statement) for a preceding call to SSL_connect(), SSL_accept(), SSL_do_handshake(), SSL_read(), SSL_peek(), or SSL_write() on **ssl**. The value returned by that TLS/SSL I/O function must be passed to SSL_get_error() in parameter **ret**.

In addition to **ssl** and **ret**, SSL_get_error() inspects the current thread's OpenSSL error queue. Thus, SSL_get_error() must be used in the same thread that performed the TLS/SSL I/O operation, and no other OpenSSL function calls should appear in between. The current thread's error queue must be empty before the TLS/SSL I/O operation is attempted, or SSL_get_error() will not work reliably.

RETURN VALUES

The following return values can currently occur:

SSL_ERROR_NONE

> The TLS/SSL I/O operation completed. This result code is returned if and only if **ret > 0**.

SSL_ERROR_ZERO_RETURN

> The TLS/SSL connection has been closed. If the protocol version is SSL 3.0 or TLS 1.0, this result code is returned only if a closure alert has occurred in the protocol, i.e. if the connection has been closed cleanly. Note that in this case **SSL_ERROR_ZERO_RETURN** does not necessarily indicate that the underlying transport has been closed.

SSL_ERROR_WANT_READ, SSL_ERROR_WANT_WRITE

> The operation did not complete; the same TLS/SSL I/O function should be called again later. If, by then, the underlying **BIO** has data available for reading (if the result code is **SSL_ERROR_WANT_READ**) or allows writing data (**SSL_ERROR_WANT_WRITE**), then some TLS/SSL protocol progress will take place, i.e. at least part of an TLS/SSL record will be read or written. Note that the retry may again lead to a **SSL_ERROR_WANT_READ** or **SSL_ERROR_WANT_WRITE** condition. There is no fixed upper limit for the number of iterations that may be necessary until progress becomes visible at application protocol level.
>
> For socket **BIO**s (e.g. when SSL_set_fd() was used), select() or poll() on the underlying socket can be used to find out when the TLS/SSL I/O function should be retried.
>
> Caveat: Any TLS/SSL I/O function can lead to either of **SSL_ERROR_WANT_READ** and **SSL_ERROR_WANT_WRITE**. In particular, SSL_read() or SSL_peek() may want to write data and SSL_write() may want to read data. This is mainly because TLS/SSL handshakes may occur at any time during the protocol (initiated by either the client or the server); SSL_read(), SSL_peek(), and SSL_write() will handle any pending handshakes.

SSL_ERROR_WANT_CONNECT, SSL_ERROR_WANT_ACCEPT

> The operation did not complete; the same TLS/SSL I/O function should be called again later. The underlying BIO was not connected yet to the peer and the call would block in connect()/accept(). The SSL function should be called again when the

connection is established. These messages can only appear with a BIO_s_connect() or BIO_s_accept() BIO, respectively. In order to find out, when the connection has been successfully established, on many platforms select() or poll() for writing on the socket file descriptor can be used.

SSL_ERROR_WANT_X509_LOOKUP

The operation did not complete because an application callback set by SSL_CTX_set_client_cert_cb() has asked to be called again. The TLS/SSL I/O function should be called again later. Details depend on the application.

SSL_ERROR_SYSCALL

Some I/O error occurred. The OpenSSL error queue may contain more information on the error. If the error queue is empty (i.e. ERR_get_error() returns 0), **ret** can be used to find out more about the error: If **ret == 0**, an EOF was observed that violates the protocol. If **ret == -1**, the underlying **BIO** reported an I/O error (for socket I/O on Unix systems, consult **errno** for details).

SSL_ERROR_SSL

A failure in the SSL library occurred, usually a protocol error. The OpenSSL error queue contains more information on the error.

SEE ALSO

ssl(3), err(3)

HISTORY

SSL_get_error() was added in SSLeay 0.8.

Name

SSL_get_ex_data_X509_STORE_CTX_idx — get ex_data index to access SSL structure from X509_STORE_CTX

Synopsis

```
#include  <openssl/ssl.h>
```

```
int SSL_get_ex_data_X509_STORE_CTX_idx(void);
```

DESCRIPTION

SSL_get_ex_data_X509_STORE_CTX_idx() returns the index number under which the pointer to the SSL object is stored into the X509_STORE_CTX object.

NOTES

Whenever a X509_STORE_CTX object is created for the verification of the peers certificate during a handshake, a pointer to the SSL object is stored into the X509_STORE_CTX object to identify the connection affected. To retrieve this pointer the X509_STORE_CTX_get_ex_data() function can be used with the correct index. This index is globally the same for all X509_STORE_CTX objects and can be retrieved using SSL_get_ex_data_X509_STORE_CTX_idx(). The index value is set when SSL_get_ex_data_X509_STORE_CTX_idx() is first called either by the application program directly or indirectly during other SSL setup functions or during the handshake.

The value depends on other index values defined for X509_STORE_CTX objects before the SSL index is created.

RETURN VALUES

>=0 The index value to access the pointer.

<0 An error occurred, check the error stack for a detailed error message.

EXAMPLES

The index returned from SSL_get_ex_data_X509_STORE_CTX_idx() allows to access the SSL object for the connection to be accessed during the verify_callback() when checking the peers certificate. Please check the example in SSL_CTX_set_verify(3),

SEE ALSO

ssl(3), SSL_CTX_set_verify(3), CRYPTO_set_ex_data(3)

Name

SSL_get_ex_new_index, SSL_set_ex_data and SSL_get_ex_data — internal application specific data functions

Synopsis

```
#include  <openssl/ssl.h>
```

```
int SSL_get_ex_new_index(long argl, void *argp,
               CRYPTO_EX_new *new_func,
               CRYPTO_EX_dup *dup_func,
               CRYPTO_EX_free *free_func);
```

```
int SSL_set_ex_data(SSL *ssl, int idx, void *arg);
```

```
void *SSL_get_ex_data(const SSL *ssl, int idx);
```

```
typedef int new_func(void *parent, void *ptr, CRYPTO_EX_DATA *ad,
               int idx, long argl, void *argp);
typedef void free_func(void *parent, void *ptr, CRYPTO_EX_DATA *ad,
               int idx, long argl, void *argp);
typedef int dup_func(CRYPTO_EX_DATA *to, CRYPTO_EX_DATA *from, void *from_d,
               int idx, long argl, void *argp);
```

DESCRIPTION

Several OpenSSL structures can have application specific data attached to them. These functions are used internally by OpenSSL to manipulate application specific data attached to a specific structure.

SSL_get_ex_new_index() is used to register a new index for application specific data.

SSL_set_ex_data() is used to store application data at **arg** for **idx** into the **ssl** object.

SSL_get_ex_data() is used to retrieve the information for **idx** from **ssl**.

A detailed description for the ***_get_ex_new_index()** functionality can be found in RSA_get_ex_new_index(3). The ***_get_ex_data()** and ***_set_ex_data()** functionality is described in CRYPTO_set_ex_data(3).

EXAMPLES

An example on how to use the functionality is included in the example verify_callback() in SSL_CTX_set_verify(3).

SEE ALSO

ssl(3), RSA_get_ex_new_index(3), CRYPTO_set_ex_data(3), SSL_CTX_set_verify(3)

Name

SSL_get_fd — get file descriptor linked to an SSL object

Synopsis

```
#include  <openssl/ssl.h>
```

```
int SSL_get_fd(const SSL *ssl);
int SSL_get_rfd(const SSL *ssl);
int SSL_get_wfd(const SSL *ssl);
```

DESCRIPTION

SSL_get_fd() returns the file descriptor which is linked to **ssl**. SSL_get_rfd() and SSL_get_wfd() return the file descriptors for the read or the write channel, which can be different. If the read and the write channel are different, SSL_get_fd() will return the file descriptor of the read channel.

RETURN VALUES

The following return values can occur:

-1 The operation failed, because the underlying BIO is not of the correct type (suitable for file descriptors).

>=0 The file descriptor linked to **ssl**.

SEE ALSO

SSL_set_fd(3), ssl(3) , bio(3)

Name

SSL_get_peer_cert_chain — get the X509 certificate chain of the peer

Synopsis

```
#include  <openssl/ssl.h>
```

```
STACK_OF(X509) *SSL_get_peer_cert_chain(const SSL *ssl);
```

DESCRIPTION

SSL_get_peer_cert_chain() returns a pointer to STACK_OF(X509) certificates forming the certificate chain of the peer. If called on the client side, the stack also contains the peer's certificate; if called on the server side, the peer's certificate must be obtained separately using SSL_get_peer_certificate(3). If the peer did not present a certificate, NULL is returned.

NOTES

The peer certificate chain is not necessarily available after reusing a session, in which case a NULL pointer is returned.

The reference count of the STACK_OF(X509) object is not incremented. If the corresponding session is freed, the pointer must not be used any longer.

RETURN VALUES

The following return values can occur:

NULL

> No certificate was presented by the peer or no connection was established or the certificate chain is no longer available when a session is reused.

Pointer to a STACK_OF(X509)

> The return value points to the certificate chain presented by the peer.

SEE ALSO

ssl(3), SSL_get_peer_certificate(3)

Name

SSL_get_peer_certificate — get the X509 certificate of the peer

Synopsis

```
#include  <openssl/ssl.h>
```

```
X509 *SSL_get_peer_certificate(const SSL *ssl);
```

DESCRIPTION

SSL_get_peer_certificate() returns a pointer to the X509 certificate the peer presented. If the peer did not present a certificate, NULL is returned.

NOTES

Due to the protocol definition, a TLS/SSL server will always send a certificate, if present. A client will only send a certificate when explicitly requested to do so by the server (see SSL_CTX_set_verify(3)). If an anonymous cipher is used, no certificates are sent.

That a certificate is returned does not indicate information about the verification state, use SSL_get_verify_result(3) to check the verification state.

The reference count of the X509 object is incremented by one, so that it will not be destroyed when the session containing the peer certificate is freed. The X509 object must be explicitly freed using X509_free().

RETURN VALUES

The following return values can occur:

NULL

> No certificate was presented by the peer or no connection was established.

Pointer to an X509 certificate

> The return value points to the certificate presented by the peer.

SEE ALSO

ssl(3), SSL_get_verify_result(3), SSL_CTX_set_verify(3)

Name

SSL_get_psk_identity and SSL_get_psk_identity_hint — get PSK client identity and hint

Synopsis

```
#include  <openssl/ssl.h>
```

```
const char *SSL_get_psk_identity_hint(const SSL *ssl);
const char *SSL_get_psk_identity(const SSL *ssl);
```

DESCRIPTION

SSL_get_psk_identity_hint() is used to retrieve the PSK identity hint used during the connection setup related to SSL object **ssl**. Similarly, SSL_get_psk_identity() is used to retrieve the PSK identity used during the connection setup.

RETURN VALUES

If non-**NULL**, SSL_get_psk_identity_hint() returns the PSK identity hint and SSL_get_psk_identity() returns the PSK identity. Both are **NULL**-terminated. SSL_get_psk_identity_hint() may return **NULL** if no PSK identity hint was used during the connection setup.

Note that the return value is valid only during the lifetime of the SSL object **ssl**.

Name

SSL_get_rbio — get BIO linked to an SSL object

Synopsis

```
#include  <openssl/ssl.h>
```

```
BIO *SSL_get_rbio(SSL *ssl);
BIO *SSL_get_wbio(SSL *ssl);
```

DESCRIPTION

SSL_get_rbio() and SSL_get_wbio() return pointers to the BIOs for the read or the write channel, which can be different. The reference count of the BIO is not incremented.

RETURN VALUES

The following return values can occur:

NULL

No BIO was connected to the SSL object

Any other pointer

The BIO linked to **ssl**.

SEE ALSO

SSL_set_bio(3), ssl(3) , bio(3)

Name

SSL_get_session — retrieve TLS/SSL session data

Synopsis

```
#include  <openssl/ssl.h>
```

```
SSL_SESSION *SSL_get_session(const SSL *ssl);
SSL_SESSION *SSL_get0_session(const SSL *ssl);
SSL_SESSION *SSL_get1_session(SSL *ssl);
```

DESCRIPTION

SSL_get_session() returns a pointer to the **SSL_SESSION** actually used in **ssl**. The reference count of the **SSL_SESSION** is not incremented, so that the pointer can become invalid by other operations.

SSL_get0_session() is the same as SSL_get_session().

SSL_get1_session() is the same as SSL_get_session(), but the reference count of the **SSL_SESSION** is incremented by one.

NOTES

The ssl session contains all information required to re-establish the connection without a new handshake.

SSL_get0_session() returns a pointer to the actual session. As the reference counter is not incremented, the pointer is only valid while the connection is in use. If SSL_clear(3) or SSL_free(3) is called, the session may be removed completely (if considered bad), and the pointer obtained will become invalid. Even if the session is valid, it can be removed at any time due to timeout during SSL_CTX_flush_sessions(3).

If the data is to be kept, SSL_get1_session() will increment the reference count, so that the session will not be implicitly removed by other operations but stays in memory. In order to remove the session SSL_SESSION_free(3) must be explicitly called once to decrement the reference count again.

SSL_SESSION objects keep internal link information about the session cache list, when being inserted into one SSL_CTX object's session cache. One SSL_SESSION object, regardless of its reference count, must therefore only be used with one SSL_CTX object (and the SSL objects created from this SSL_CTX object).

RETURN VALUES

The following return values can occur:

NULL

> There is no session available in **ssl**.

Pointer to an SSL

> The return value points to the data of an SSL session.

SEE ALSO

ssl(3), SSL_free(3), SSL_clear(3), SSL_SESSION_free(3)

Name

SSL_get_SSL_CTX — get the SSL_CTX from which an SSL is created

Synopsis

```
#include  <openssl/ssl.h>
```

```
SSL_CTX *SSL_get_SSL_CTX(const SSL *ssl);
```

DESCRIPTION

SSL_get_SSL_CTX() returns a pointer to the SSL_CTX object, from which **ssl** was created with SSL_new(3).

RETURN VALUES

The pointer to the SSL_CTX object is returned.

SEE ALSO

ssl(3), SSL_new(3)

Name

SSL_get_verify_result — get result of peer certificate verification

Synopsis

```
#include  <openssl/ssl.h>
```

```
long SSL_get_verify_result(const SSL *ssl);
```

DESCRIPTION

SSL_get_verify_result() returns the result of the verification of the X509 certificate presented by the peer, if any.

NOTES

SSL_get_verify_result() can only return one error code while the verification of a certificate can fail because of many reasons at the same time. Only the last verification error that occurred during the processing is available from SSL_get_verify_result().

The verification result is part of the established session and is restored when a session is reused.

BUGS

If no peer certificate was presented, the returned result code is X509_V_OK. This is because no verification error occurred, it does however not indicate success. SSL_get_verify_result() is only useful in connection with SSL_get_peer_certificate(3).

RETURN VALUES

The following return values can currently occur:

X509_V_OK

> The verification succeeded or no peer certificate was presented.

Any other value

> Documented in verify(1).

SEE ALSO

ssl(3), SSL_set_verify_result(3), SSL_get_peer_certificate(3), verify(1)

Name

SSL_get_version — get the protocol version of a connection.

Synopsis

```
#include  <openssl/ssl.h>
```

```
const char *SSL_get_version(const SSL *ssl);
```

DESCRIPTION

SSL_get_version() returns the name of the protocol used for the connection **ssl**.

RETURN VALUES

The following strings can be returned:

SSLv2

> The connection uses the SSLv2 protocol.

SSLv3

> The connection uses the SSLv3 protocol.

TLSv1

> The connection uses the TLSv1.0 protocol.

TLSv1.1

> The connection uses the TLSv1.1 protocol.

TLSv1.2

> The connection uses the TLSv1.2 protocol.

unknown

> This indicates that no version has been set (no connection established).

SEE ALSO

ssl(3)

Name

SSL_library_init, OpenSSL_add_ssl_algorithms and SSLeay_add_ssl_algorithms — initialize SSL library by registering algorithms

Synopsis

```
#include  <openssl/ssl.h>
```

```
int SSL_library_init(void);
#define OpenSSL_add_ssl_algorithms()    SSL_library_init()
#define SSLeay_add_ssl_algorithms()     SSL_library_init()
```

DESCRIPTION

SSL_library_init() registers the available SSL/TLS ciphers and digests.

OpenSSL_add_ssl_algorithms() and SSLeay_add_ssl_algorithms() are synonyms for SSL_library_init().

NOTES

SSL_library_init() must be called before any other action takes place. SSL_library_init() is not reentrant.

WARNING

SSL_library_init() adds ciphers and digests used directly and indirectly by SSL/TLS.

EXAMPLES

A typical TLS/SSL application will start with the library initialization, and provide readable error messages.

```
SSL_load_error_strings();                /* readable error messages */
SSL_library_init();                      /* initialize library */
```

RETURN VALUES

SSL_library_init() always returns "1", so it is safe to discard the return value.

NOTES

OpenSSL 0.9.8o and 1.0.0a and later added SHA2 algorithms to SSL_library_init(). Applications which need to use SHA2 in earlier versions of OpenSSL should call OpenSSL_add_all_algorithms() as well.

SEE ALSO

ssl(3), SSL_load_error_strings(3), RAND_add(3)

Name

SSL_load_client_CA_file — load certificate names from file

Synopsis

```
#include  <openssl/ssl.h>
```

```
STACK_OF(X509_NAME) *SSL_load_client_CA_file(const char *file);
```

DESCRIPTION

SSL_load_client_CA_file() reads certificates from **file** and returns a STACK_OF(X509_NAME) with the subject names found.

NOTES

SSL_load_client_CA_file() reads a file of PEM formatted certificates and extracts the X509_NAMES of the certificates found. While the name suggests the specific usage as support function for SSL_CTX_set_client_CA_list(3), it is not limited to CA certificates.

EXAMPLES

Load names of CAs from file and use it as a client CA list:

```
SSL_CTX *ctx;
STACK_OF(X509_NAME) *cert_names;

…
cert_names = SSL_load_client_CA_file("/path/to/CAfile.pem");
if (cert_names != NULL)
  SSL_CTX_set_client_CA_list(ctx, cert_names);
else
  error_handling();
…
```

RETURN VALUES

The following return values can occur:

NULL

> The operation failed, check out the error stack for the reason.

Pointer to STACK_OF(X509_NAME)

> Pointer to the subject names of the successfully read certificates.

SEE ALSO

ssl(3), SSL_CTX_set_client_CA_list(3)

Name

SSL_new — create a new SSL structure for a connection

Synopsis

```
#include  <openssl/ssl.h>
```

```
SSL *SSL_new(SSL_CTX *ctx);
```

DESCRIPTION

SSL_new() creates a new **SSL** structure which is needed to hold the data for a TLS/SSL connection. The new structure inherits the settings of the underlying context **ctx**: connection method (SSLv2/v3/TLSv1), options, verification settings, timeout settings.

RETURN VALUES

The following return values can occur:

NULL

> The creation of a new SSL structure failed. Check the error stack to find out the reason.

Pointer to an SSL structure

> The return value points to an allocated SSL structure.

SEE ALSO

SSL_free(3), SSL_clear(3), SSL_CTX_set_options(3), SSL_get_SSL_CTX(3), ssl(3)

Name

SSL_pending — obtain number of readable bytes buffered in an SSL object

Synopsis

```
#include  <openssl/ssl.h>
```

```
int SSL_pending(const SSL *ssl);
```

DESCRIPTION

SSL_pending() returns the number of bytes which are available inside **ssl** for immediate read.

NOTES

Data are received in blocks from the peer. Therefore data can be buffered inside **ssl** and are ready for immediate retrieval with SSL_read(3).

RETURN VALUES

The number of bytes pending is returned.

BUGS

SSL_pending() takes into account only bytes from the TLS/SSL record that is currently being processed (if any). If the **SSL** object's *read_ahead* flag is set (see SSL_CTX_set_read_ahead(3)), additional protocol bytes may have been read containing more TLS/SSL records; these are ignored by SSL_pending().

Up to OpenSSL 0.9.6, SSL_pending() does not check if the record type of pending data is application data.

SEE ALSO

SSL_read(3), SSL_CTX_set_read_ahead(3), ssl(3)

Name

SSL_read — read bytes from a TLS/SSL connection.

Synopsis

```
#include  <openssl/ssl.h>
```

```
int SSL_read(SSL *ssl, void *buf, int num);
```

DESCRIPTION

SSL_read() tries to read **num** bytes from the specified **ssl** into the buffer **buf**.

NOTES

If necessary, SSL_read() will negotiate a TLS/SSL session, if not already explicitly performed by SSL_connect(3) or SSL_accept(3). If the peer requests a re-negotiation, it will be performed transparently during the SSL_read() operation. The behaviour of SSL_read() depends on the underlying BIO.

For the transparent negotiation to succeed, the **ssl** must have been initialized to client or server mode. This is being done by calling SSL_set_connect_state(3) or SSL_set_accept_state() before the first call to an SSL_read() or SSL_write(3) function.

SSL_read() works based on the SSL/TLS records. The data are received in records (with a maximum record size of 16kB for SSLv3/TLSv1). Only when a record has been completely received, it can be processed (decryption and check of integrity). Therefore data that was not retrieved at the last call of SSL_read() can still be buffered inside the SSL layer and will be retrieved on the next call to SSL_read(). If **num** is higher than the number of bytes buffered, SSL_read() will return with the bytes buffered. If no more bytes are in the buffer, SSL_read() will trigger the processing of the next record. Only when the record has been received and processed completely, SSL_read() will return reporting success. At most the contents of the record will be returned. As the size of an SSL/TLS record may exceed the maximum packet size of the underlying transport (e.g. TCP), it may be necessary to read several packets from the transport layer before the record is complete and SSL_read() can succeed.

If the underlying BIO is **blocking**, SSL_read() will only return, once the read operation has been finished or an error occurred, except when a renegotiation take place, in which case a SSL_ERROR_WANT_READ may occur. This behaviour can be controlled with the SSL_MODE_AUTO_RETRY flag of the SSL_CTX_set_mode(3) call.

If the underlying BIO is **non-blocking**, SSL_read() will also return when the underlying BIO could not satisfy the needs of SSL_read() to continue the operation. In this case a call to SSL_get_error(3) with the return value of SSL_read() will yield **SSL_ERROR_WANT_READ** or **SSL_ERROR_WANT_WRITE**. As at any time a re-negotiation is possible, a call to SSL_read() can also cause write operations! The calling process then must repeat the call after taking appropriate action to satisfy the needs of SSL_read(). The action depends on the underlying BIO. When using a non-blocking socket, nothing is to be done, but select() can be used to check for the required condition. When using a buffering BIO, like a BIO pair, data must be written into or retrieved out of the BIO before being able to continue.

SSL_pending(3) can be used to find out whether there are buffered bytes available for immediate retrieval. In this case SSL_read() can be called without blocking or actually receiving new data from the underlying socket.

WARNING

When an SSL_read() operation has to be repeated because of **SSL_ERROR_WANT_READ** or **SSL_ERROR_WANT_WRITE**, it must be repeated with the same arguments.

RETURN VALUES

The following return values can occur:

>0 The read operation was successful; the return value is the number of bytes actually read from the TLS/SSL connection.

0 The read operation was not successful. The reason may either be a clean shutdown due to a "close notify" alert sent by the peer (in which case the SSL_RECEIVED_SHUTDOWN flag in the ssl shutdown state is set (see SSL_shutdown(3), SSL_set_shutdown(3)). It is also possible, that the peer simply shut down the underlying transport and the shutdown is incomplete. Call SSL_get_error() with the return value **ret** to find out, whether an error occurred or the connection was shut down cleanly (SSL_ERROR_ZERO_RETURN).

SSLv2 (deprecated) does not support a shutdown alert protocol, so it can only be detected, whether the underlying connection was closed. It cannot be checked, whether the closure was initiated by the peer or by something else.

<0 The read operation was not successful, because either an error occurred or action must be taken by the calling process. Call SSL_get_error() with the return value **ret** to find out the reason.

SEE ALSO

SSL_get_error(3), SSL_write(3), SSL_CTX_set_mode(3), SSL_CTX_new(3), SSL_connect(3), SSL_accept(3)SSL_set_connect_state(3), SSL_pending(3), SSL_shutdown(3), SSL_set_shutdown(3), ssl(3), bio(3)

Name

SSL_rstate_string and SSL_rstate_string_long — get textual description of state of an SSL object during read operation

Synopsis

```
#include  <openssl/ssl.h>
```

```
const char *SSL_rstate_string(SSL *ssl);
const char *SSL_rstate_string_long(SSL *ssl);
```

DESCRIPTION

SSL_rstate_string() returns a 2 letter string indicating the current read state of the SSL object **ssl**.

SSL_rstate_string_long() returns a string indicating the current read state of the SSL object **ssl**.

NOTES

When performing a read operation, the SSL/TLS engine must parse the record, consisting of header and body. When working in a blocking environment, SSL_rstate_string[_long]() should always return "RD"/"read done".

This function should only seldom be needed in applications.

RETURN VALUES

SSL_rstate_string() and SSL_rstate_string_long() can return the following values:

"RH"/"read header"	The header of the record is being evaluated.
"RB"/"read body"	The body of the record is being evaluated.
"RD"/"read done"	The record has been completely processed.
"unknown"/"unknown"	The read state is unknown. This should never happen.

SEE ALSO

ssl(3)

Name

SSL_SESSION_free — free an allocated SSL_SESSION structure

Synopsis

```
#include  <openssl/ssl.h>
```

```
void SSL_SESSION_free(SSL_SESSION *session);
```

DESCRIPTION

SSL_SESSION_free() decrements the reference count of **session** and removes the **SSL_SESSION** structure pointed to by **session** and frees up the allocated memory, if the reference count has reached 0.

NOTES

SSL_SESSION objects are allocated, when a TLS/SSL handshake operation is successfully completed. Depending on the settings, see SSL_CTX_set_session_cache_mode(3), the SSL_SESSION objects are internally referenced by the SSL_CTX and linked into its session cache. SSL objects may be using the SSL_SESSION object; as a session may be reused, several SSL objects may be using one SSL_SESSION object at the same time. It is therefore crucial to keep the reference count (usage information) correct and not delete a SSL_SESSION object that is still used, as this may lead to program failures due to dangling pointers. These failures may also appear delayed, e.g. when an SSL_SESSION object was completely freed as the reference count incorrectly became 0, but it is still referenced in the internal session cache and the cache list is processed during a SSL_CTX_flush_sessions(3) operation.

SSL_SESSION_free() must only be called for SSL_SESSION objects, for which the reference count was explicitly incremented (e.g. by calling SSL_get1_session(), see SSL_get_session(3)) or when the SSL_SESSION object was generated outside a TLS handshake operation, e.g. by using d2i_SSL_SESSION(3). It must not be called on other SSL_SESSION objects, as this would cause incorrect reference counts and therefore program failures.

RETURN VALUES

SSL_SESSION_free() does not provide diagnostic information.

SEE ALSO

ssl(3), SSL_get_session(3), SSL_CTX_set_session_cache_mode(3), SSL_CTX_flush_sessions(3), d2i_SSL_SESSION(3)

Name

SSL_SESSION_get_ex_new_index, SSL_SESSION_set_ex_data and SSL_SESSION_get_ex_data — internal application specific data functions

Synopsis

```
#include  <openssl/ssl.h>
```

```
int SSL_SESSION_get_ex_new_index(long argl, void *argp,
               CRYPTO_EX_new *new_func,
               CRYPTO_EX_dup *dup_func,
               CRYPTO_EX_free *free_func);
```

```
int SSL_SESSION_set_ex_data(SSL_SESSION *session, int idx, void *arg);
```

```
void *SSL_SESSION_get_ex_data(const SSL_SESSION *session, int idx);
```

```
typedef int new_func(void *parent, void *ptr, CRYPTO_EX_DATA *ad,
               int idx, long argl, void *argp);
typedef void free_func(void *parent, void *ptr, CRYPTO_EX_DATA *ad,
               int idx, long argl, void *argp);
typedef int dup_func(CRYPTO_EX_DATA *to, CRYPTO_EX_DATA *from, void *from_d,
               int idx, long argl, void *argp);
```

DESCRIPTION

Several OpenSSL structures can have application specific data attached to them. These functions are used internally by OpenSSL to manipulate application specific data attached to a specific structure.

SSL_SESSION_get_ex_new_index() is used to register a new index for application specific data.

SSL_SESSION_set_ex_data() is used to store application data at **arg** for **idx** into the **session** object.

SSL_SESSION_get_ex_data() is used to retrieve the information for **idx** from **session**.

A detailed description for the ***_get_ex_new_index()** functionality can be found in RSA_get_ex_new_index(3). The ***_get_ex_data()** and ***_set_ex_data()** functionality is described in CRYPTO_set_ex_data(3).

WARNINGS

The application data is only maintained for sessions held in memory. The application data is not included when dumping the session with i2d_SSL_SESSION() (and all functions indirectly calling the dump functions like PEM_write_SSL_SESSION() and PEM_write_bio_SSL_SESSION()) and can therefore not be restored.

SEE ALSO

ssl(3), RSA_get_ex_new_index(3), CRYPTO_set_ex_data(3)

Name

SSL_SESSION_get_time, SSL_SESSION_set_time, SSL_SESSION_get_timeout and SSL_SESSION_set_timeout — retrieve and manipulate session time and timeout settings

Synopsis

```
#include  <openssl/ssl.h>
```

```
long SSL_SESSION_get_time(const SSL_SESSION *s);
long SSL_SESSION_set_time(SSL_SESSION *s, long tm);
long SSL_SESSION_get_timeout(const SSL_SESSION *s);
long SSL_SESSION_set_timeout(SSL_SESSION *s, long tm);
```

```
long SSL_get_time(const SSL_SESSION *s);
long SSL_set_time(SSL_SESSION *s, long tm);
long SSL_get_timeout(const SSL_SESSION *s);
long SSL_set_timeout(SSL_SESSION *s, long tm);
```

DESCRIPTION

SSL_SESSION_get_time() returns the time at which the session **s** was established. The time is given in seconds since the Epoch and therefore compatible to the time delivered by the time() call.

SSL_SESSION_set_time() replaces the creation time of the session **s** with the chosen value **tm**.

SSL_SESSION_get_timeout() returns the timeout value set for session **s** in seconds.

SSL_SESSION_set_timeout() sets the timeout value for session **s** in seconds to **tm**.

The SSL_get_time(), SSL_set_time(), SSL_get_timeout(), and SSL_set_timeout() functions are synonyms for the SSL_SESSION_*() counterparts.

NOTES

Sessions are expired by examining the creation time and the timeout value. Both are set at creation time of the session to the actual time and the default timeout value at creation, respectively, as set by SSL_CTX_set_timeout(3). Using these functions it is possible to extend or shorten the lifetime of the session.

RETURN VALUES

SSL_SESSION_get_time() and SSL_SESSION_get_timeout() return the currently valid values.

SSL_SESSION_set_time() and SSL_SESSION_set_timeout() return 1 on success.

If any of the function is passed the NULL pointer for the session **s**, 0 is returned.

SEE ALSO

ssl(3), SSL_CTX_set_timeout(3), SSL_get_default_timeout(3)

Name

SSL_session_reused — query whether a reused session was negotiated during handshake

Synopsis

```
#include  <openssl/ssl.h>
```

```
int SSL_session_reused(SSL *ssl);
```

DESCRIPTION

Query, whether a reused session was negotiated during the handshake.

NOTES

During the negotiation, a client can propose to reuse a session. The server then looks up the session in its cache. If both client and server agree on the session, it will be reused and a flag is being set that can be queried by the application.

RETURN VALUES

The following return values can occur:

0 A new session was negotiated.

1 A session was reused.

SEE ALSO

ssl(3), SSL_set_session(3), SSL_CTX_set_session_cache_mode(3)

Name

SSL_set_bio — connect the SSL object with a BIO

Synopsis

```
#include  <openssl/ssl.h>
```

```
void SSL_set_bio(SSL *ssl, BIO *rbio, BIO *wbio);
```

DESCRIPTION

SSL_set_bio() connects the BIOs **rbio** and **wbio** for the read and write operations of the TLS/SSL (encrypted) side of **ssl**.

The SSL engine inherits the behaviour of **rbio** and **wbio**, respectively. If a BIO is non-blocking, the **ssl** will also have non-blocking behaviour.

If there was already a BIO connected to **ssl**, BIO_free() will be called (for both the reading and writing side, if different).

RETURN VALUES

SSL_set_bio() cannot fail.

SEE ALSO

SSL_get_rbio(3), SSL_connect(3), SSL_accept(3), SSL_shutdown(3), ssl(3), bio(3)

Name

SSL_set_connect_state and SSL_get_accept_state — prepare SSL object to work in client or server mode

Synopsis

```
#include  <openssl/ssl.h>
```

```
void SSL_set_connect_state(SSL *ssl);
```

```
void SSL_set_accept_state(SSL *ssl);
```

DESCRIPTION

SSL_set_connect_state() sets **ssl** to work in client mode.

SSL_set_accept_state() sets **ssl** to work in server mode.

NOTES

When the SSL_CTX object was created with SSL_CTX_new(3), it was either assigned a dedicated client method, a dedicated server method, or a generic method, that can be used for both client and server connections. (The method might have been changed with SSL_CTX_set_ssl_version(3) or SSL_set_ssl_method().)

When beginning a new handshake, the SSL engine must know whether it must call the connect (client) or accept (server) routines. Even though it may be clear from the method chosen, whether client or server mode was requested, the handshake routines must be explicitly set.

When using the SSL_connect(3) or SSL_accept(3) routines, the correct handshake routines are automatically set. When performing a transparent negotiation using SSL_write(3) or SSL_read(3), the handshake routines must be explicitly set in advance using either SSL_set_connect_state() or SSL_set_accept_state().

RETURN VALUES

SSL_set_connect_state() and SSL_set_accept_state() do not return diagnostic information.

SEE ALSO

ssl(3), SSL_new(3), SSL_CTX_new(3), SSL_connect(3), SSL_accept(3), SSL_write(3), SSL_read(3), SSL_do_handshake(3), SSL_CTX_set_ssl_version(3)

Name

SSL_set_fd — connect the SSL object with a file descriptor

Synopsis

```
#include  <openssl/ssl.h>
```

```
int SSL_set_fd(SSL *ssl, int fd);
int SSL_set_rfd(SSL *ssl, int fd);
int SSL_set_wfd(SSL *ssl, int fd);
```

DESCRIPTION

SSL_set_fd() sets the file descriptor **fd** as the input/output facility for the TLS/SSL (encrypted) side of **ssl**. **fd** will typically be the socket file descriptor of a network connection.

When performing the operation, a **socket BIO** is automatically created to interface between the **ssl** and **fd**. The BIO and hence the SSL engine inherit the behaviour of **fd**. If **fd** is non-blocking, the **ssl** will also have non-blocking behaviour.

If there was already a BIO connected to **ssl**, BIO_free() will be called (for both the reading and writing side, if different).

SSL_set_rfd() and SSL_set_wfd() perform the respective action, but only for the read channel or the write channel, which can be set independently.

RETURN VALUES

The following return values can occur:

0 The operation failed. Check the error stack to find out why.

1 The operation succeeded.

SEE ALSO

SSL_get_fd(3), SSL_set_bio(3), SSL_connect(3), SSL_accept(3), SSL_shutdown(3), ssl(3) , bio(3)

Name

SSL_set_session — set a TLS/SSL session to be used during TLS/SSL connect

Synopsis

```
#include  <openssl/ssl.h>
```

```
int SSL_set_session(SSL *ssl, SSL_SESSION *session);
```

DESCRIPTION

SSL_set_session() sets **session** to be used when the TLS/SSL connection is to be established. SSL_set_session() is only useful for TLS/SSL clients. When the session is set, the reference count of **session** is incremented by 1. If the session is not reused, the reference count is decremented again during SSL_connect(). Whether the session was reused can be queried with the SSL_session_reused(3) call.

If there is already a session set inside **ssl** (because it was set with SSL_set_session() before or because the same **ssl** was already used for a connection), SSL_SESSION_free() will be called for that session.

NOTES

SSL_SESSION objects keep internal link information about the session cache list, when being inserted into one SSL_CTX object's session cache. One SSL_SESSION object, regardless of its reference count, must therefore only be used with one SSL_CTX object (and the SSL objects created from this SSL_CTX object).

RETURN VALUES

The following return values can occur:

0 The operation failed; check the error stack to find out the reason.

1 The operation succeeded.

SEE ALSO

ssl(3), SSL_SESSION_free(3), SSL_get_session(3), SSL_session_reused(3), SSL_CTX_set_session_cache_mode(3)

Name

SSL_set_shutdown and SSL_get_shutdown — manipulate shutdown state of an SSL connection

Synopsis

```
#include  <openssl/ssl.h>
```

```
void SSL_set_shutdown(SSL *ssl, int mode);
```

```
int SSL_get_shutdown(const SSL *ssl);
```

DESCRIPTION

SSL_set_shutdown() sets the shutdown state of **ssl** to **mode**.

SSL_get_shutdown() returns the shutdown mode of **ssl**.

NOTES

The shutdown state of an ssl connection is a bitmask of:

0

No shutdown setting, yet.

SSL_SENT_SHUTDOWN

A "close notify" shutdown alert was sent to the peer, the connection is being considered closed and the session is closed and correct.

SSL_RECEIVED_SHUTDOWN

A shutdown alert was received form the peer, either a normal "close notify" or a fatal error.

SSL_SENT_SHUTDOWN and SSL_RECEIVED_SHUTDOWN can be set at the same time.

The shutdown state of the connection is used to determine the state of the ssl session. If the session is still open, when SSL_clear(3) or SSL_free(3) is called, it is considered bad and removed according to RFC2246. The actual condition for a correctly closed session is SSL_SENT_SHUTDOWN (according to the TLS RFC, it is acceptable to only send the "close notify" alert but to not wait for the peer's answer, when the underlying connection is closed). SSL_set_shutdown() can be used to set this state without sending a close alert to the peer (see SSL_shutdown(3)).

If a "close notify" was received, SSL_RECEIVED_SHUTDOWN will be set, for setting SSL_SENT_SHUTDOWN the application must however still call SSL_shutdown(3) or SSL_set_shutdown() itself.

RETURN VALUES

SSL_set_shutdown() does not return diagnostic information.

SSL_get_shutdown() returns the current setting.

SEE ALSO

ssl(3), SSL_shutdown(3), SSL_CTX_set_quiet_shutdown(3), SSL_clear(3), SSL_free(3)

Name

SSL_set_verify_result — override result of peer certificate verification

Synopsis

```
#include  <openssl/ssl.h>
```

```
void SSL_set_verify_result(SSL *ssl, long verify_result);
```

DESCRIPTION

SSL_set_verify_result() sets **verify_result** of the object **ssl** to be the result of the verification of the X509 certificate presented by the peer, if any.

NOTES

SSL_set_verify_result() overrides the verification result. It only changes the verification result of the **ssl** object. It does not become part of the established session, so if the session is to be reused later, the original value will reappear.

The valid codes for **verify_result** are documented in verify(1).

RETURN VALUES

SSL_set_verify_result() does not provide a return value.

SEE ALSO

ssl(3), SSL_get_verify_result(3), SSL_get_peer_certificate(3), verify(1)

Name

SSL_shutdown — shut down a TLS/SSL connection

Synopsis

```
#include  <openssl/ssl.h>
```

```
int SSL_shutdown(SSL *ssl);
```

DESCRIPTION

SSL_shutdown() shuts down an active TLS/SSL connection. It sends the "close notify" shutdown alert to the peer.

NOTES

SSL_shutdown() tries to send the "close notify" shutdown alert to the peer. Whether the operation succeeds or not, the SSL_SENT_SHUTDOWN flag is set and a currently open session is considered closed and good and will be kept in the session cache for further reuse.

The shutdown procedure consists of 2 steps: the sending of the "close notify" shutdown alert and the reception of the peer's "close notify" shutdown alert. According to the TLS standard, it is acceptable for an application to only send its shutdown alert and then close the underlying connection without waiting for the peer's response (this way resources can be saved, as the process can already terminate or serve another connection). When the underlying connection shall be used for more communications, the complete shutdown procedure (bidirectional "close notify" alerts) must be performed, so that the peers stay synchronized.

SSL_shutdown() supports both uni- and bidirectional shutdown by its 2 step behaviour.

- When the application is the first party to send the "close notify" alert, SSL_shutdown() will only send the alert and then set the SSL_SENT_SHUTDOWN flag (so that the session is considered good and will be kept in cache). SSL_shutdown() will then return with 0. If a unidirectional shutdown is enough (the underlying connection shall be closed anyway), this first call to SSL_shutdown() is sufficient. In order to complete the bidirectional shutdown handshake, SSL_shutdown() must be called again. The second call will make SSL_shutdown() wait for the peer's "close notify" shutdown alert. On success, the second call to SSL_shutdown() will return with 1.
- If the peer already sent the "close notify" alert **and** it was already processed implicitly inside another function (SSL_read(3)), the SSL_RECEIVED_SHUTDOWN flag is set. SSL_shutdown() will send the "close notify" alert, set the SSL_SENT_SHUTDOWN flag and will immediately return with 1. Whether SSL_RECEIVED_SHUTDOWN is already set can be checked using the SSL_get_shutdown() (see also SSL_set_shutdown(3) call.

It is therefore recommended, to check the return value of SSL_shutdown() and call SSL_shutdown() again, if the bidirectional shutdown is not yet complete (return value of the first call is 0). As the shutdown is not specially handled in the SSLv2 protocol, SSL_shutdown() will succeed on the first call.

The behaviour of SSL_shutdown() additionally depends on the underlying BIO.

If the underlying BIO is **blocking**, SSL_shutdown() will only return once the handshake step has been finished or an error occurred.

If the underlying BIO is **non-blocking**, SSL_shutdown() will also return when the underlying BIO could not satisfy the needs of SSL_shutdown() to continue the handshake. In this case a call to SSL_get_error() with the return value of SSL_shutdown() will yield **SSL_ERROR_WANT_READ** or **SSL_ERROR_WANT_WRITE**. The calling process then must repeat the call after taking appropriate action to satisfy the needs of SSL_shutdown(). The action depends on the underlying BIO. When using a non-blocking socket, nothing is to be done, but select() can be used to check for the required condition. When using a buffering BIO, like a BIO pair, data must be written into or retrieved out of the BIO before being able to continue.

SSL_shutdown() can be modified to only set the connection to "shutdown" state but not actually send the "close notify" alert messages, see SSL_CTX_set_quiet_shutdown(3). When "quiet shutdown" is enabled, SSL_shutdown() will always succeed and return 1.

RETURN VALUES

The following return values can occur:

0 The shutdown is not yet finished. Call SSL_shutdown() for a second time, if a bidirectional shutdown shall be performed. The output of SSL_get_error(3) may be misleading, as an erroneous SSL_ERROR_SYSCALL may be flagged even though no error occurred.

1 The shutdown was successfully completed. The "close notify" alert was sent and the peer's "close notify" alert was received.

-1 The shutdown was not successful because a fatal error occurred either at the protocol level or a connection failure occurred. It can also occur if action is need to continue the operation for non-blocking BIOs. Call SSL_get_error(3) with the return value **ret** to find out the reason.

SEE ALSO

SSL_get_error(3), SSL_connect(3), SSL_accept(3), SSL_set_shutdown(3), SSL_CTX_set_quiet_shutdown(3), SSL_clear(3), SSL_free(3), ssl(3), bio(3)

Name

SSL_state_string and SSL_state_string_long — get textual description of state of an SSL object

Synopsis

```
#include  <openssl/ssl.h>
```

```
const char *SSL_state_string(const SSL *ssl);
const char *SSL_state_string_long(const SSL *ssl);
```

DESCRIPTION

SSL_state_string() returns a 6 letter string indicating the current state of the SSL object **ssl**.

SSL_state_string_long() returns a string indicating the current state of the SSL object **ssl**.

NOTES

During its use, an SSL objects passes several states. The state is internally maintained. Querying the state information is not very informative before or when a connection has been established. It however can be of significant interest during the handshake.

When using non-blocking sockets, the function call performing the handshake may return with SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE condition, so that SSL_state_string[_long]() may be called.

For both blocking or non-blocking sockets, the details state information can be used within the info_callback function set with the SSL_set_info_callback() call.

RETURN VALUES

Detailed description of possible states to be included later.

SEE ALSO

ssl(3), SSL_CTX_set_info_callback(3)

Name

SSL_want, SSL_want_nothing, SSL_want_read, SSL_want_write and SSL_want_x509_lookup — obtain state information TLS/SSL I/O operation

Synopsis

```
#include  <openssl/ssl.h>
```

```
int SSL_want(const SSL *ssl);
int SSL_want_nothing(const SSL *ssl);
int SSL_want_read(const SSL *ssl);
int SSL_want_write(const SSL *ssl);
int SSL_want_x509_lookup(const SSL *ssl);
```

DESCRIPTION

SSL_want() returns state information for the SSL object **ssl**.

The other SSL_want_*() calls are shortcuts for the possible states returned by SSL_want().

NOTES

SSL_want() examines the internal state information of the SSL object. Its return values are similar to that of SSL_get_error(3). Unlike SSL_get_error(3), which also evaluates the error queue, the results are obtained by examining an internal state flag only. The information must therefore only be used for normal operation under non-blocking I/O. Error conditions are not handled and must be treated using SSL_get_error(3).

The result returned by SSL_want() should always be consistent with the result of SSL_get_error(3).

RETURN VALUES

The following return values can currently occur for SSL_want():

SSL_NOTHING

> There is no data to be written or to be read.

SSL_WRITING

> There are data in the SSL buffer that must be written to the underlying **BIO** layer in order to complete the actual SSL_*() operation. A call to SSL_get_error(3) should return SSL_ERROR_WANT_WRITE.

SSL_READING

> More data must be read from the underlying **BIO** layer in order to complete the actual SSL_*() operation. A call to SSL_get_error(3) should return SSL_ERROR_WANT_READ.

SSL_X509_LOOKUP

> The operation did not complete because an application callback set by SSL_CTX_set_client_cert_cb() has asked to be called again. A call to SSL_get_error(3) should return SSL_ERROR_WANT_X509_LOOKUP.

SSL_want_nothing(), SSL_want_read(), SSL_want_write(), SSL_want_x509_lookup() return 1, when the corresponding condition is true or 0 otherwise.

SEE ALSO

ssl(3), err(3), SSL_get_error(3)

Name

SSL_write — write bytes to a TLS/SSL connection.

Synopsis

```
#include  <openssl/ssl.h>
```

```
int SSL_write(SSL *ssl, const void *buf, int num);
```

DESCRIPTION

SSL_write() writes **num** bytes from the buffer **buf** into the specified **ssl** connection.

NOTES

If necessary, SSL_write() will negotiate a TLS/SSL session, if not already explicitly performed by SSL_connect(3) or SSL_accept(3). If the peer requests a re-negotiation, it will be performed transparently during the SSL_write() operation. The behaviour of SSL_write() depends on the underlying BIO.

For the transparent negotiation to succeed, the **ssl** must have been initialized to client or server mode. This is being done by calling SSL_set_connect_state(3) or SSL_set_accept_state() before the first call to an SSL_read(3) or SSL_write() function.

If the underlying BIO is **blocking**, SSL_write() will only return, once the write operation has been finished or an error occurred, except when a renegotiation take place, in which case a SSL_ERROR_WANT_READ may occur. This behaviour can be controlled with the SSL_MODE_AUTO_RETRY flag of the SSL_CTX_set_mode(3) call.

If the underlying BIO is **non-blocking**, SSL_write() will also return, when the underlying BIO could not satisfy the needs of SSL_write() to continue the operation. In this case a call to SSL_get_error(3) with the return value of SSL_write() will yield **SSL_ERROR_WANT_READ** or **SSL_ERROR_WANT_WRITE**. As at any time a re-negotiation is possible, a call to SSL_write() can also cause read operations! The calling process then must repeat the call after taking appropriate action to satisfy the needs of SSL_write(). The action depends on the underlying BIO. When using a non-blocking socket, nothing is to be done, but select() can be used to check for the required condition. When using a buffering BIO, like a BIO pair, data must be written into or retrieved out of the BIO before being able to continue.

SSL_write() will only return with success, when the complete contents of **buf** of length **num** has been written. This default behaviour can be changed with the SSL_MODE_ENABLE_PARTIAL_WRITE option of SSL_CTX_set_mode(3). When this flag is set, SSL_write() will also return with success, when a partial write has been successfully completed. In this case the SSL_write() operation is considered completed. The bytes are sent and a new SSL_write() operation with a new buffer (with the already sent bytes removed) must be started. A partial write is performed with the size of a message block, which is 16kB for SSLv3/TLSv1.

WARNING

When an SSL_write() operation has to be repeated because of **SSL_ERROR_WANT_READ** or **SSL_ERROR_WANT_WRITE**, it must be repeated with the same arguments.

When calling SSL_write() with num=0 bytes to be sent the behaviour is undefined.

RETURN VALUES

The following return values can occur:

>0 The write operation was successful, the return value is the number of bytes actually written to the TLS/SSL connection.

0 The write operation was not successful. Probably the underlying connection was closed. Call SSL_get_error() with the return value **ret** to find out, whether an error occurred or the connection was shut down cleanly (SSL_ERROR_ZERO_RETURN).

SSLv2 (deprecated) does not support a shutdown alert protocol, so it can only be detected, whether the underlying connection was closed. It cannot be checked, why the closure happened.

<0 The write operation was not successful, because either an error occurred or action must be taken by the calling process. Call SSL_get_error() with the return value **ret** to find out the reason.

SEE ALSO

SSL_get_error(3), SSL_read(3), SSL_CTX_set_mode(3), SSL_CTX_new(3), SSL_connect(3), SSL_accept(3)SSL_set_connect_state(3), ssl(3), bio(3)

Part XLVIII. Mbed TLS

Table of Contents

Chapter 180. Mbed TLS overview

Introduction

The CYGPKG_MBEDTLS package provides a standard Mbed TLS world to eCos applications.

This package is covered by the Apache 2.0 license as distributed in the original Mbed TLS package:

Example 180.1. Apache 2.0 License

```
                                 Apache License
                           Version 2.0, January 2004
                        http://www.apache.org/licenses/

   TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

   1. Definitions.

      "License" shall mean the terms and conditions for use, reproduction,
      and distribution as defined by Sections 1 through 9 of this document.

      "Licensor" shall mean the copyright owner or entity authorized by
      the copyright owner that is granting the License.

      "Legal Entity" shall mean the union of the acting entity and all
      other entities that control, are controlled by, or are under common
      control with that entity. For the purposes of this definition,
      "control" means (i) the power, direct or indirect, to cause the
      direction or management of such entity, whether by contract or
      otherwise, or (ii) ownership of fifty percent (50%) or more of the
      outstanding shares, or (iii) beneficial ownership of such entity.

      "You" (or "Your") shall mean an individual or Legal Entity
      exercising permissions granted by this License.

      "Source" form shall mean the preferred form for making modifications,
      including but not limited to software source code, documentation
      source, and configuration files.

      "Object" form shall mean any form resulting from mechanical
      transformation or translation of a Source form, including but
      not limited to compiled object code, generated documentation,
      and conversions to other media types.

      "Work" shall mean the work of authorship, whether in Source or
      Object form, made available under the License, as indicated by a
      copyright notice that is included in or attached to the work
      (an example is provided in the Appendix below).

      "Derivative Works" shall mean any work, whether in Source or Object
      form, that is based on (or derived from) the Work and for which the
      editorial revisions, annotations, elaborations, or other modifications
      represent, as a whole, an original work of authorship. For the purposes
      of this License, Derivative Works shall not include works that remain
      separable from, or merely link (or bind by name) to the interfaces of,
      the Work and Derivative Works thereof.

      "Contribution" shall mean any work of authorship, including
      the original version of the Work and any modifications or additions
      to that Work or Derivative Works thereof, that is intentionally
      submitted to Licensor for inclusion in the Work by the copyright owner
      or by an individual or Legal Entity authorized to submit on behalf of
      the copyright owner. For the purposes of this definition, "submitted"
      means any form of electronic, verbal, or written communication sent
      to the Licensor or its representatives, including but not limited to
      communication on electronic mailing lists, source code control systems,
      and issue tracking systems that are managed by, or on behalf of, the
      Licensor for the purpose of discussing and improving the Work, but
      excluding communication that is conspicuously marked or otherwise
      designated in writing by the copyright owner as "Not a Contribution."

      "Contributor" shall mean Licensor and any individual or Legal Entity
      on behalf of whom a Contribution has been received by Licensor and
      subsequently incorporated within the Work.

   2. Grant of Copyright License. Subject to the terms and conditions of
      this License, each Contributor hereby grants to You a perpetual,
      worldwide, non-exclusive, no-charge, royalty-free, irrevocable
      copyright license to reproduce, prepare Derivative Works of,
      publicly display, publicly perform, sublicense, and distribute the
      Work and such Derivative Works in Source or Object form.

   3. Grant of Patent License. Subject to the terms and conditions of
      this License, each Contributor hereby grants to You a perpetual,
      worldwide, non-exclusive, no-charge, royalty-free, irrevocable
      (except as stated in this section) patent license to make, have made,
      use, offer to sell, sell, import, and otherwise transfer the Work,
      where such license applies only to those patent claims licensable
      by such Contributor that are necessarily infringed by their
      Contribution(s) alone or by combination of their Contribution(s)
      with the Work to which such Contribution(s) was submitted. If You
      institute patent litigation against any entity (including a
      cross-claim or counterclaim in a lawsuit) alleging that the Work
      or a Contribution incorporated within the Work constitutes direct
      or contributory patent infringement, then any patent licenses
      granted to You under this License for that Work shall terminate
      as of the date such litigation is filed.

   4. Redistribution. You may reproduce and distribute copies of the
      Work or Derivative Works thereof in any medium, with or without
      modifications, and in Source or Object form, provided that You
      meet the following conditions:

      (a) You must give any other recipients of the Work or
          Derivative Works a copy of this License; and

      (b) You must cause any modified files to carry prominent notices
          stating that You changed the files; and

      (c) You must retain, in the Source form of any Derivative Works
          that You distribute, all copyright, patent, trademark, and
          attribution notices from the Source form of the Work,
          excluding those notices that do not pertain to any part of
          the Derivative Works; and

      (d) If the Work includes a "NOTICE" text file as part of its
          distribution, then any Derivative Works that You distribute must
          include a readable copy of the attribution notices contained
          within such NOTICE file, excluding those notices that do not
          pertain to any part of the Derivative Works, in at least one
          of the following places: within a NOTICE text file distributed
          as part of the Derivative Works; within the Source form or
          documentation, if provided along with the Derivative Works; or,
          within a display generated by the Derivative Works, if and
          wherever such third-party notices normally appear. The contents
          of the NOTICE file are for informational purposes only and
          do not modify the License. You may add Your own attribution
          notices within Derivative Works that You distribute, alongside
          or as an addendum to the NOTICE text from the Work, provided
          that such additional attribution notices cannot be construed
          as modifying the License.

      You may add Your own copyright statement to Your modifications and
      may provide additional or different license terms and conditions
      for use, reproduction, or distribution of Your modifications, or
      for any such Derivative Works as a whole, provided Your use,
      reproduction, and distribution of the Work otherwise complies with
      the conditions stated in this License.

   5. Submission of Contributions. Unless You explicitly state otherwise,
      any Contribution intentionally submitted for inclusion in the Work
      by You to the Licensor shall be under the terms and conditions of
      this License, without any additional terms or conditions.
      Notwithstanding the above, nothing herein shall supersede or modify
      the terms of any separate license agreement you may have executed
      with Licensor regarding such Contributions.

   6. Trademarks. This License does not grant permission to use the trade
      names, trademarks, service marks, or product names of the Licensor,
      except as required for reasonable and customary use in describing the
      origin of the Work and reproducing the content of the NOTICE file.

   7. Disclaimer of Warranty. Unless required by applicable law or
      agreed to in writing, Licensor provides the Work (and each
      Contributor provides its Contributions) on an "AS IS" BASIS,
      WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
      implied, including, without limitation, any warranties or conditions
      of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A
      PARTICULAR PURPOSE. You are solely responsible for determining the
      appropriateness of using or redistributing the Work and assume any
      risks associated with Your exercise of permissions under this License.

   8. Limitation of Liability. In no event and under no legal theory,
      whether in tort (including negligence), contract, or otherwise,
      unless required by applicable law (such as deliberate and grossly
      negligent acts) or agreed to in writing, shall any Contributor be
      liable to You for damages, including any direct, indirect, special,
      incidental, or consequential damages of any character arising as a
      result of this License or out of the use or inability to use the
      Work (including but not limited to damages for loss of goodwill,
      work stoppage, computer failure or malfunction, or any and all
      other commercial damages or losses), even if such Contributor
      has been advised of the possibility of such damages.

   9. Accepting Warranty or Additional Liability. While redistributing
      the Work or Derivative Works thereof, You may choose to offer,
      and charge a fee for, acceptance of support, warranty, indemnity,
      or other liability obligations and/or rights consistent with this
      License. However, in accepting such obligations, You may act only
      on Your own behalf and on Your sole responsibility, not on behalf
      of any other Contributor, and only if You agree to indemnify,
      defend, and hold each Contributor harmless for any liability
      incurred by, or claims asserted against, such Contributor by reason
      of your accepting any such warranty or additional liability.

   END OF TERMS AND CONDITIONS

   APPENDIX: How to apply the Apache License to your work.

      To apply the Apache License to your work, attach the following
      boilerplate notice, with the fields enclosed by brackets "[]"
      replaced with your own identifying information. (Don't include
      the brackets!)  The text should be enclosed in the appropriate
      comment syntax for the file format. We also recommend that a
      file or class name and description of purpose be included on the
      same "printed page" as the copyright notice for easier
      identification within third-party archives.

   Copyright [yyyy] [name of copyright owner]

   Licensed under the Apache License, Version 2.0 (the "License");
   you may not use this file except in compliance with the License.
   You may obtain a copy of the License at

       http://www.apache.org/licenses/LICENSE-2.0

   Unless required by applicable law or agreed to in writing, software
   distributed under the License is distributed on an "AS IS" BASIS,
   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
   See the License for the specific language governing permissions and
   limitations under the License.
```

For definitive Mbed TLS documentation please refer to the main Mbed TLS Dev Corner website. We do not duplicate that documentation here.

Mbed TLS is now part of Trusted Firmware. The Trusted Firmware Mbed TLS wiki providing a useful overview.

Chapter 181. Configuration

This chapter shows how to incorporate the Mbed TLS support into an eCos configuration, and how to configure it once included.

Configuration Overview

The Mbed TLS support is contained in a single eCos package `CYGPKG_MBEDTLS`. However, some functionality is dependant on other eCos features. e.g. the eCos networking stack support.

Quick Start

Incorporating the Mbed TLS support into your application is straightforward. The essential starting point is to incorporate the Mbed TLS eCos package (`CYGPKG_MBEDTLS`) into your configuration.

This may be achieved directly using **ecosconfig add** on the command line, or the **Build->Packages…** menu item within the eCos Configuration Tool.

Depending on the Mbed TLS package configuration other packages may be required (e.g. network stack support). The package requires that the `CYGPKG_INFRA` and `CYGPKG_MEMALLOC` packages are included in the eCos application configuration.

Chapter 182. eCos port

Overview

The goal for the `CYGPKG_MBEDTLS` package is to avoid where possible having to have any core Mbed TLS source file changes made specifically for eCos. This is to ensure that re-imports of newer versions of the Mbed TLS world involve minimal effort. The files are as provided in the official Mbed TLS release package as imported, with the following exceptions:

1. Files have been moved, unmodified, to create a standard eCos package tree structure to integrate with the eCosPro build environment
2. The files `src/platform.c`, `include/platform.h` and `include/config.h` have changes to allow compilation as an eCos package
3. The file `src/ecos_net_sockets.c` provides the internal Mbed TLS network API (as originally provided by `src/net_sockets.c`) to support the eCos network stacks (e.g. `CYGPKG_NET_LWIP` or `CYGPKG_NET_FREEBSD_STACK`).
4. The file `tests/selftest.c` has been modified with some eCos specific code to allow use in the eCos automated test farm. The core of the selftest functionality is unchanged however

The current Mbed TLS version provided by the eCos package is `2.28.7` (with `2.28` being a Long Term Support (LTS) release supported until at least 2024Q4). Please read the package `ChangeLog` file for the original Mbed TLS release notes which are interleaved with the commments covering eCosCentric changes.

Entropy

Cryptographic security is dependent on the use of **strong** keys. Strong keys are dependent on the quality of entropy used in their generation.

The eCosPro port of Mbed TLS provides both a default portable weak entropy mechanism, and the ability to implement a target platform specific mechanism to gather high-quality entropy.

We strongly recommend that you use as high a quality source of entropy as possible when using Mbed TLS on your target platform. The use of the generic weak entropy mechanism (provided by `MBEDTLS_TIMING_C`) is **NOT** recommended for anything other than simplifying initial bring-up and testing.

For eCos we define `MBEDTLS_ENTROPY_HARDWARE_ALT` for the configuration (in `include/mbedtls/config.h`) since the default Mbed TLS platform entropy implementation is Un*x/Windows only. The use of `MBEDTLS_ENTROPY_HARDWARE_ALT` provides for a link-time entropy source to be provided by the platform support. Other entropy sources are dynamically added at runtime via the `mbedtls_entropy_add_source()` functionality.

Note

In the following *weak* refers to the linker preference when linking an application and not to a weak entropy source.

The Mbed TLS package `src/platform.c` source file provides a **weak** function definition for the `mbedtls_hardware_poll()` implementation that will return `MBEDTLS_ERR_ENTROPY_SOURCE_FAILED` since we expect the platform support or application code to provide a cryptographically strong implementation. Ideally such an implementation will use TRNG (True Random Number Generator) hardware support by the HAL, variant, architecture, etc. as appropriate.

The function prototype:

#include <mbedtls/entropy_poll.h>

`int **mbedtls_hardware_poll**(*data*, *output*, *len*, *olen*);`

conforms to the entropy provision API for routines declared via the `mbedtls_entropy_add_source()` API (in fact the Mbed TLS run-time will add `mbedtls_hardware_poll()` automatically via that mechanism).

The callback specific (context data pointer) parameter *`data`* will normally be `NULL` for the Mbed TLS registered implementation. The *`output`* parameter references a buffer of at least *`len`* bytes to receive random data. The *`olen`* pointer is filled by the function to report the number of **valid** bytes actually written, and may be 0 (zero) if the function is not able to satisfy the request for true random data. The function should return 0 to indicate no errors occurred, or `MBEDTLS_ERR_ENTROPY_SOURCE_FAILED` on error.

As an example of a platform specific implementation, the STM32 port of eCos provides an Mbed TLS entropy source based on the STM32's `RNG` hardware.

Chapter 183. Test Programs

Test Programs

Some Mbed TLS specific tests are built and can be used to verify correct operation of the Mbed TLS library.

1. `selftest`

   This test executes the internal Mbed TLS sanity tests to verify correct operation of the various features.

2. `cpp_dummy_build`

   This is a simple sanity test that the Mbed TLS headers can be included in C++ compilations.

3. `lb_ssl`

   When an eCos network configuration is available (lwIP or FreeBSD) then this test will perform a client/server local loopback SSL connection using certificates. This is a complete test of a secure `HTTPS` connection using BSD style sockets. To make it easier to work with either the client or the server side the source for the test is split into multiple files, with the client and server side implementations being in their own source files. The relevant client and server side sources are, lightly modified for eCos, versions of the original Mbed TLS source files found respectively in `mbedtls-mbedtls-2.28.5/programs/ssl/ssl_client1.c` and `mbedtls-mbedtls-2.28.5/programs/ssl/ssl_server.c`.

   The following is example output from a run of the test:

   ### Example 183.1. lb_ssl test run

```
INFO:<code from 0x60000008 -> 0x6006d524, CRC 601c>
INFO:<SSL certificate based connection test using mbedTLS v2.24.0>
INFO:<Target time OK for X.509 verification>
INFO:<Initialising network interfaces>
lwIP i/f[e0] (default): (hwaddr 12:34:DA:A5:69:F9) IP 192.168.1.226
lwIP i/f driver state NOT initialised
lwIP i/f[lo]: (No hwaddr) IP 127.0.0.1
INFO:<Waiting for server to start>

  . Loading the server cert. and key... ok
  . Bind on https://localhost:4433/ ... ok
  . Seeding the random number generator... ok
  . Setting up the SSL data.... ok
  . Waiting for a remote connection ...
  . Seeding the random number generator... ok
  . Loading the CA root certificate ... ok (0 skipped)
  . Connecting to tcp/localhost/4433... ok
  . Performing the SSL/TLS handshake... ok
  . Setting up the SSL/TLS structure... ok
  . Performing the SSL/TLS handshake... ok
  < Read from client: ok
  . Verifying peer X.509 certificate... ok
  > Write to server: 18 bytes read

GET / HTTP/1.0

  > Write to client: 156 bytes written

HTTP/1.0 200 OK
Content-Type: text/html

<h2>mbed TLS Test Server</h2>
<p>Successful connection using: TLS-ECDHE-RSA-WITH-CHACHA20-POLY1305-SHA256</p>
```

```
  . Closing the connection... ok
  . Waiting for a remote connection ... 18 bytes written

GET / HTTP/1.0

  < Read from server: 156 bytes read

HTTP/1.0 200 OK
Content-Type: text/html

<h2>mbed TLS Test Server</h2>
<p>Successful connection using: TLS-ECDHE-RSA-WITH-CHACHA20-POLY1305-SHA256</p>
PASS:<SSL certificate based client/server test>
PASS:<Done>
EXIT:<done>
```

Part XLIX. eCosPro-SecureShell

Important

eCosPro-SecureShell is distributed as an optional set of eCos add-on packages that may not be included in your release of eCosPro. If these packages are not listed in either the graphical or command line eCos Configuration tool, please contact eCosCentric for availability and pricing.

Name

CYGPKG_NET_DROPBEAR — provide ssh support

Description

Note

The *eCosPro-SecureShell* package is the formal product name of the eCos Dropbear Port and the two can be used interchangeably to refer to this package.

CYGPKG_NET_DROPBEAR is a port to eCos of some of the ssh functionality of the dropbear code. It supports the following:

1. Server support. This allows remote clients to log in to an eCos system and run commands. Of course eCos does not have a full-blown shell and the ability to run arbitrary commands loaded from disk. Instead the ssh connection is passed on to functions within the application code which can read the data coming from the remote ssh client and take appropriate action. The package ships with two examples: a simple shell-like application and an interactive game.

2. Client support. This allows the eCos application to establish a secure connection to a remote server, for example a PC running Linux and openssh, run a command on that server, and interact with that command.

3. Client-side scp support. This builds on the generic client support. It allows eCos applications to read and write files on a remote server over a secure connection.

The port only provides a core subset of the standard dropbear functionality. For example more advanced features like agent forwarding and X11 forwarding are not supported because those would add significantly to the overhead and complexity of the code, and would rarely be used in practice.

Ssh secure communication comes at a price. Depending on the architecture it will typically add 100-200K to the application's code size. The data requirements are considerable, including a need for 32K data buffers and multiple threads. The code will require a lot of cpu cycles. A typical embedded processor running eCos is much slower than the typical cpu of a desktop PC, and the dropbear code will take correspondingly longer to perform a given operation. Establishing an ssh connection is especially expensive and may take some seconds or even tens of seconds of cpu time. Once the connection has been established the cpu overheads are lower, but still significant. Finally the dropbear code makes extensive demands on the lower-level TCP/IP and I/O layers and various configuration options in those layers may need adjusting, as described below.

Configuration

The eCos dropbear port is intended to work in conjunction with the full BSD TCP/IP package and has numerous dependencies. Most of these can be satisfied simply by creating the eCos configuration using the net template. The dropbear package has additional dependencies on the LibTomMath multi-precision arithmetic package CYGPKG_MATH_LIBTOMMATH and the LibTomCrypt cryptography library CYGPKG_CRYPT_LIBTOMCRYPT, so those packages will have to be added explicitly to the configuration alongside CYGPKG_NET_DROPBEAR.

Usually the dropbear code depends on the presence of a file system for holding public and private keys and other data. In the eCos port this dependency has been eliminated and no file system is required. Instead all the required data is embedded directly in the eCos application and passed to the dropbear code as function arguments.

Ssh connections impose considerable demands on the lower-level TCP/IP and I/O layers, and various configuration options in those layers may need adjusting from their small default values. For example each outgoing ssh connection involves five sockets, plus one statically allocated socket shared between all connections. By default the file I/O package only supports 16 open file descriptors, three of which are used for stdin/stdout/stderr and some of the remainder may be used by other packages like DNS. That should leave enough free file descriptors for one or two ssh connections, but only if the application does not use them for other networking or file I/O activities. Increasing the configuration options CYGNUM_FILEIO_NFD and CYGNUM_FILEIO_NFILE would avoid problems in this area.

When it comes to the TCP/IP stack, the first option to consider is CYGPKG_NET_MAXSOCKETS. Closing down a network connection does not immediately free all resources associated with that connection because it is necessary to synchronize with the other end and make sure that that will not send any more packets. Hence if the application attempts multiple ssh connections in quick succession then the TCP/IP stack may run out socket resources. Increasing CYGPKG_NET_MAXSOCKETS avoids this problem. If the connections involve large amount of data then it may also be necessary to increase CYGPKG_NET_MEMPOOL_SIZE.

Port

Porting dropbear to eCos involved non-trivial modifications to the source code. The package's src subdirectory corresponds to the contents of a standard dropbear tarball. New files ecosmain.c, ecos.h and config.h have been added, and various existing files have had to be modified. A CDL script, documentation and an example application have been added to the appropriate package subdirectories, and a new header dropbear.h has been written to export the API provided by the eCos port. Two example server-side applications can be found in the package's misc subdirectory, and testcases can be found in the tests subdirectory.

Name

Dropbear — Ssh daemon support

Synopsis

#include <dropbear.h> extern int cyg_dropbear_connections;

void **cyg_dropbear_init**(*data*);

void **cyg_dropbear_done**(*handle*, *exit_code*);

int **cyg_dropbear_get_stderr**(*handle*);

const char* **cyg_dropbear_get_username**(*handle*);

const struct sockaddr_storage* **cyg_dropbear_get_addr**(*handle*);

Description

This document assumes the reader has a basic understanding of both ssh operation and of TCP/IP network programming. Information on both of these can be readily obtained from other sources. The eCos port of dropbear does behave rather differently from dropbear or other ssh implementations in a conventional setup such as a network of Linux workstations. eCos does not support multiple processes or shell executables. The dropbear code runs in separate threads within the eCos application, started by a call to `cyg_dropbear_init`, rather than as a separate daemon process. Authentication does not involve reading in files such as `~/.ssh/authorized_keys2` or `/etc/passwd`. Instead the application developer has to write a number of callback functions to supply the necessary information and can decide how best to implement those functions - which may involve file I/O if that happens to be convenient. When an ssh connection is established the dropbear code will not start up a new shell process inside a pseudo terminal. Instead the dropbear code will create a local TCP/IP socket and pass one end of this to the application. Data coming from the network will be decrypted and sent down the local socket, to be read by application code. Data written by the application to the local socket will be encrypted and sent over the network to the ssh client.

The eCos dropbear package comes with two working examples in the `misc/hangman` and `misc/shell` directories which should be examined in conjunction with this documentation.

API

The internals of the dropbear code are not directly accessible to application code. Instead dropbear runs in a number of internal threads, interacting with the application via local TCP/IP sockets and a number of callback functions.

Initialization

The main entry point to dropbear functionality is the function `cyg_dropbear_init`. This should be called by the application when the dropbear code can start running, any time after the network connection has been activated. It takes a single argument, a pointer to a cyg_dropbear_data data structure. That structure should be filled in by the application with information needed by the dropbear code, including pointers to various callback functions.

```
typedef struct cyg_dropbear_data {
    const char*     db_rsa_private_host_key;
    int             db_rsa_private_host_keylen;
    const char*     db_dss_private_host_key;
    int             db_dss_private_host_keylen;
    cyg_bool        (*db_accept_connection)(struct sockaddr_storage*);
    void            (*db_connected)(int, cyg_dropbear_handle);
    void            (*db_get_public_keys)(char*, char*,
                                 struct sockaddr_storage*, char **);
    void            (*db_free_public_keys)(char**);
    cyg_bool        (*db_authenticate_password)(char*,
                                  struct sockaddr_storage *, char*);
    void            (*db_logger)(const char*, va_list);
} cyg_dropbear_data;
```

The *cyg_dropbear_data* structure will be accessed regularly by the dropbear code so must not be overwritten or freed by the application. Typically it will be statically allocated.

The initialization function will start the main dropbear thread, running at configurable priority `CYGNUM_NET_DROPBEAR_THREAD_PRI`. This thread will initialize the dropbear package. It will then bind to the desired TCP/IP port on address `INADDR_ANY`. The default port is 22 as per ssh but this can be changed via `CYGNUM_NET_DROPBEAR_NETWORK_PORT`. The thread will accept new connections from remote clients and start a separate worker thread for each connection, again running at priority `CYGNUM_NET_DROPBEAR_THREAD_PRI`. Usually the main dropbear thread will also assist with setting up the local connection. `cyg_dropbear_init` will return once the main thread has been started.

Host Keys

As an ssh daemon the dropbear code requires a private host key. This can be either a DSS key or an RSA key, but preferably both should be supplied. Private keys can be generated using the **dropbearkey** host utility. The host key is used primarily by clients to uniquely identify a given machine, in conjunction with the `~/.ssh/known_hosts` file.

The application should supply one or two host keys via the *db_rsa_private_host_key*, *db_rsa_private_host_keylen*, *db_dss_private_host_key* and *db_dss_private_host_keylen* fields of the cyg_dropbear_data structure passed to `cyg_dropbear_init`.

Conventional ssh usage dictates that all units should have unique private host keys. That implies that the host keys are held in a file system or in non-volatile flash memory and customized on a per-unit basis. Generating new host keys on startup is undesirable because it would result in different keys after every reboot, defeating the purpose of the `known_hosts` file. In practice it may be acceptable to use a single set of host keys for all units, but it is up to the application developer to consider the security implications.

New Connections Callback

When the main dropbear thread accepts a new ssh connection from the network it will start a new worker thread to handle that connection. Before any data is exchanged over the TCP/IP socket the worker thread will call back into the application so that the latter can decide whether or not a new connection should be accepted at this time.

```
cyg_bool
application_accept_connection_callback(struct sockaddr_storage* addr)
{
    …
}
```

A pointer to this callback function should be stored in the *db_accept_connection* field of the cyg_dropbear_data structure passed to `cyg_dropbear_init`. The callback's argument is the address of the connecting client. Typically this will be an IPv4 sockaddr_in address but the *ss_family* field of the sockaddr_storage structure should be checked. The client's address can be used to restrict incoming connections to certain networks or even to individual hosts. This offers some increased security along similar lines to a firewall, albeit only for ssh traffic, but it may cause problems in future if it ever becomes necessary to access the unit from a different network or host.

Another common use for the new connections callback is to limit the number of concurrent connections, and thus limit the resources consumed. To facilitate this the dropbear code exports an integer `cyg_dropbear_connections` which holds the current number of client connections.

The callback should return zero if the new connection should be declined, non-zero if the dropbear code should proceed with key exchange and authentication.

The use of a new connection callback is optional. If the application does not want to perform any checks at this time then it can use a NULL pointer for the *db_accept_connection* field.

Key Authentication Callback

After the new connections callback the dropbear worker thread will engage in a key exchange with the client. This allows the client to verify information in the `known_hosts` file with the daemon's private host key or keys. The exchange also involves negotiating a mutually acceptable encryption protocol and deciding what user authentication mechanisms are supported. The eCos dropbear daemon supports public/private key authentication and password authentication.

For public/private key authentication the client will generate a message using the private key, and the daemon can validate this message using the public key. In a conventional ssh setup the public keys come from a file `~/.ssh/authorized_keys` or `~/.ssh/authorized_keys2`, but the eCos dropbear port does not assume the presence of a file system. Instead it is the application's responsibility to provide the public keys, and a callback mechanism is used.

```
void
application_get_public_keys(char* algo,
                            char* username,
                            struct sockaddr_storage* addr,
                            char** keys)
{
    …
}

void
application_free_public_keys(char** keys)
{
    …
}
```

Pointers to these callback functions should be stored in the *`db_get_public_keys`* and *`db_free_public_keys`* fields of the cyg_dropbear_data structure passed to `cyg_dropbear_init`. The first argument to the get_public_keys callback identifies the type of key, usually `ssh-dss` or `ssh-rsa`. The second and third arguments supply the user name and the client's network address. Together these three arguments can be used to restrict which public keys are supplied to the dropbear code for validation. Strictly this is optional since the public keys for e.g. a different user are not going to match anyway, but the validation requires significant cpu cycles.

The fourth argument points at a static array of up to `CYGNUM_NET_DROPBEAR_MAXIMUM_PUBLIC_KEYS` entries. This array will be initialized to NULL pointers. The get_public_keys callback should fill in zero or more array entries with pointers to public key strings. These strings may be statically or dynamically allocated. In case of the latter, the free_public_keys callback will be invoked to release the memory when the validation has finished.

Within the dropbear code a mutex is used to ensure that only one worker thread at a time will attempt public key authentication. Hence the callback functions can assume that there will be only one call to get_public_keys at a time and that free_public_keys will be called before the next get_public_keys. This allows the callbacks to use static data with no need for additional locking.

Support for public/private key authentication is optional. If the application is only interested in password authentication then it can use a NULL pointer for the *`db_get_public_keys`* field. If all public keys are statically allocated then a NULL pointer can be used for *`db_free_public_keys`* field.

Password Authentication Callback

If public/private key authentication fails then the dropbear code can support password authentication instead. The ssh client will encrypt the password before sending it over the wires. The dropbear code will decrypt it and will pass the plain text password to an application callback for verification.

```
cyg_bool
application_authenticate_password(char* username,
                                  struct sockaddr_storage* addr,
                                  char* password)
{
    …
}
```

A pointer to this callback function should be stored in the *`db_authenticate_password`* field of the cyg_dropbear_data structure passed to `cyg_dropbear_init`. It is up to the application developer to decide how to validate the password. A very simple approach would just involve comparing it with a constant string. However that approach is not very secure: if there is any way to read the unit's memory directly, for example via jtag debug hardware, then it will be possible to extract the password directly from memory. A more advanced approach would involve encryption, for example using much the same techniques as for a Unix `/etc/passwd` file.

Support for password authentication is optional and can be disabled simply by not supplying the callback function.

Connected Callback

In a conventional ssh environment, once authentication has succeeded the daemon will open a pseudo tty, fork a new process, and start a shell running in that process. eCos does not have pseudo ttys, processes, or shells, so a different approach is needed. The dropbear worker thread for this ssh session will establish a local socket connection and will pass this socket on to the application. Encrypted data coming from the network will be decrypted by the dropbear worker thread and then sent down the local socket to the application. When the application writes plain text data down the local socket the dropbear worker thread will read it, encrypt it, and sent it to the ssh client over the network. As far as the application is concerned it can just `read` and `write` data via the local socket, and it need not concern itself with ssh protocols or encryption. Other standard network programming techniques such as `select` can be used as desired.

To inform the application that authentication has succeeded and that the local socket connection has been established, the dropbear code will invoke another callback function:

```
void
application_connected(int socket, cyg_dropbear_handle handle)
{
    …
}
```

A pointer to this function should be placed in the *`db_connected`* field of the cyg_dropbear_data structure passed to `cyg_dropbear_init`. The first argument is a file descriptor for the local socket, for use by the application code. The second argument is a handle that can be used to get additional information about the connection, and that must be used to shut down the connection.

The connected callback runs from inside the main dropbear thread. It should not run the relevant application code directly, since that would prevent the dropbear code from accepting any new ssh connections. Instead the callback should arrange for some other part of the application to handle traffic for this connection. This can be done in various ways, for example by waking up an existing thread, creating a new one, or adding the socket file descriptor to an fd_set structure which gets monitored via `select` elsewhere in the application.

The socket passed to the connected callback is still an ordinary blocking socket, but can be turned into a non-blocking socket if desired using the standard `FIONBIO` `ioctl`. In the absence of pseudo tty support the socket provides raw terminal functionality. Typically input will appear a character at a time rather than a line at a time, and any line editing functionality has to be supplied by the application. It is also up to the application to implement carriage return/linefeed processing, to handle special input characters like ctrl-C or ctrl-D, and so on.

Strictly the use of the connected callback is optional. Application developers can choose to accept the local socket connection in their own code rather than leave it to the main dropbear thread, as that may make it easier to port existing code. It is still necessary to obtain the cyg_dropbear_handle so that the connection can be closed down correctly. The hangman example program illustrates how to do this.

`stderr` channel

By default the dropbear code only sets up stdin and stdout channels. If a stderr channel is needed as well then the configuration option `CYGIMP_NET_DROPBEAR_SUPPORT_STDERR` should be enabled. An additional function `cyg_dropbear_get_stderr` then becomes available. This function should be called only from the connected callback. It returns a file descriptor for a new local socket which can be used for stderr output, or -1 if the system has run out of sockets or file descriptors.

Use of `cyg_dropbear_get_stderr` is optional, so applications can decide on a per-connection basis whether or not a stderr channel is needed.

Connection Information

Once a connection has been established and the connected callback has been invoked, application code can retrieve some additional information about this connection. `cyg_dropbear_get_username` will provide the user name supplied during authentication, and `cyg_dropbear_get_addr` will return the network address of the ssh client.

Closing Connection

An ssh connection can be shut down in one of two ways. If the client is terminated then the eCos application will detect an end of file condition on the local socket and can close its end of the connection. Alternatively the eCos application can decide to close its end unilaterally. Either way closing the connection involves two steps. The `close` system call should be used on the file descriptor for the local socket, and on the one for the stderr socket if `cyg_dropbear_get_stderr` has been used. Next `cyg_dropbear_done` should be invoked using the cyg_dropbear_handle provided by the connected callback. Calling `cyg_dropbear_done` is analogous to the shell process exiting in a conventional ssh environment. The second argument corresponds to the exit code of that process, so a value of 0 indicates no error. The cyg_dropbear_handle ceases to be valid during the call to `cyg_dropbear_done` so the application must ensure the handle will not be used concurrently by another thread or after the done call.

Note

If `cyg_dropbear_done` is not called when closing down an ssh connection then the dropbear worker thread associated with that connection will never terminate, and any resources allocated will not be released. If this happens repeatedly then the system will run out of memory and fail.

Logging Callback

The dropbear code includes support for logging various events. In a conventional ssh environment these log messages will end up in the system log files, but when running under eCos that approach is usually inappropriate. Instead by default the log messages will just be discarded, but if desired the application can capture the messages by installing a logging callback in the *`db_logger`* field of the cyg_dropbear_data structure:

```
void
application_logger(char* format, va_list args)
{
    …
}
```

Configuration

The dropbear package provides a number of additional configuration options.

`CYGIMP_NET_DROPBEAR_SUPPORT_STDERR`

When a new secure connection is established, by default the dropbear code only provides stdin and stdout channels. If this option is enabled then application code can also request a stderr channel for certain connections. This consumes one extra file descriptor allocated during initialization, plus another two file descriptors for every connection that uses stderr.

`CYGNUM_NET_DROPBEAR_MAX_PAYLOAD_LEN`

While establishing a secure connection the client and daemon negotiate an upper limit for the packet size. This has to be large enough to cope with the initial key exchange and related information. For eCos the default size is set to 1K, which should be sufficient but is still rather smaller than the dropbear default on other platforms. If the application involves transferring large amounts of data over ssh then throughput may be improved by increasing this packet size, at the cost of increased memory usage.

`CYGNUM_NET_DROPBEAR_MAXIMUM_PUBLIC_KEYS`

During the authentication phase the dropbear code may invoke a get_public_keys callback function provided by the application. One of the arguments to that function is a fixed-size array of pointers to public key strings. This configuration option determines the size of that array, and has a default value of 8. If for some reason the application supports large numbers of public keys then it may be necessary to increase this setting.

`CYGNUM_NET_DROPBEAR_NETWORK_PORT`

By default the dropbear main thread listens on port 22, the standard TCP/IP port for ssh. An alternative network port can be specified if desired. This provides a rather limited amount of protection against network attacks which attempt to connect directly to the ssh port, but will be of no use against a determined attack. Using a non-standard port will also be an inconvenience on the client side: the user has to either specify the port number on the command line or modify a configuration file such as `~/.ssh/config`.

`CYGNUM_NET_DROPBEAR_LOCAL_PORT`

This configuration option controls the TCP/IP port used to establish the local socket for each ssh connection.

`CYGNUM_NET_DROPBEAR_STDERR_PORT`

This configuration option controls the TCP/IP port used to establish the stderr socket during a call to `cyg_dropbear_get_stderr`.

`CYGNUM_NET_DROPBEAR_THREAD_PRI`

The dropbear package involves one thread started by `cyg_dropbear_init` plus additional worker threads, one per ssh connection. These threads all run at the same priority, controlled by this configuration option. The ssh protocol involves computationally intensive operations such as encrypting and decrypting packets, so these threads may have a significiant impact on the number of cpu cycles available to the rest of the system. It may be necessary to manipulate the thread priority to achieve an acceptable balance between overall system performance and the achievable ssh bandwidth.

`CYGNUM_NET_DROPBEAR_THREAD_STACKSIZE`

This configuration option controls the size of the stacks allocated for the main dropbear thread and for each worker thread. Fairly large stacks are needed to implement the ssh protocol, and in addition the various application callback functions will also run on these stacks. The default value should suffice for all architectures, but this can be checked by using a debug build with stack checking enabled. If the stack sizes are reduced to save memory then it is strongly recommended that the system be tested with stack checking enabled.

`CYGNUM_NET_DROPBEAR_CONNECTION_DELAY`

The main dropbear thread can delay for a configurable number of system clock ticks between accepting each ssh connection from the network. This provides a limited amount of protection against denial of service attacks, especially when used in conjunction with an accept_connection callback function which limits the number of open connections. Obviously if the eCos system is the target of a denial of service attack then it will still be very hard for legitimate users to get ssh access to the unit, but there is an increased chance that the unit will continue operating rather than run out of resources and fail.

`CYGNUM_NET_DROPBEAR_RETRY_DELAY`

Within the main dropbear code, if the system runs out of resources and for example a `malloc` call fails then this is treated as a serious failure and the ssh connection should get shut down, which will release resources back to the system. Where appropriate some of the eCos-specific code will instead sleep for a while and then retry, in the hope that other parts of the system have released sufficient resources during the delay. This configuration option specifies the number of system clock ticks that should be spent sleeping before retrying.

A further three configuration options in other packages may have a significant impact on the behaviour of the dropbear code. The dropbear code involves a fixed overhead of two sockets, one for the network port and one for the local port, plus an additional three sockets for each ssh connection, one for the network side and two locally. If stderr support is enabled then there is an additional fixed overhead of another socket plus another two sockets for each connection. Depending on how many concurrent ssh connections the application is expected to support it may be necessary to increase the limits on the number of file descriptors and sockets, controlled by `CYGNUM_FILEIO_NFILE`, `CYGNUM_FILEIO_NFD` and `CYGPKG_NET_MAXSOCKETS`. If C library stdio functions are used then `FOPEN_MAX` may also need to be increased. If the system runs out of sockets or file descriptors when a client attempts to establish a new connection, this will normally be detected and the connection will be shut down immediately. However there are edge and race conditions which may cause the client to hang until a timeout occurs.

Name

Dropbear — Ssh client support

Synopsis

#include <dropbear.h>

int **cyg_dropbear_ssh_connect**(*handle*, *addr*, *auth*, *command*);

void **cyg_dropbear_ssh_close**(*handle*, *wait*);

Description

The client-side API allows an eCos application to establish a secure connection to a remote ssh server and run commands on the remote machine. This requires that the application authenticate itself as a valid user on that system. Once the remote command is running the eCos application can interact with its stdin/stdout/stderr stream over sockets.

The client-side code has only been tested against openssh running on a Linux server and and as such descriptions of host-side server configuration settings and files in the remainder of this section refer to Linux. Interoperability with other ssh implementations cannot be guaranteed.

Application developers should be aware that establishing an ssh connection is a complicated business. Even if the eCos application is working correctly there are many things completely outside its control that could go wrong and prevent a secure connection from being established. Some of these are: firewalls intercepting and discarding packets to the ssh server; tcp wrappers intercepting and rejecting requests before they even reach the ssh server, courtesy of settings in the `/etc/hosts.allow` and `/etc/hosts.deny` files on the host ssh server; ssh server settings in `/etc/ssh/sshd_config` which are incompatible with the application's requirements; problems with the user account specified by the application; or problems with the ssh keys in the `~/.ssh/authorized_keys2` file of the user's account on the host server. It is recommended that when experiencing connectivity problems from an eCos application the developer first checks the server's setup, for example by using **ssh** or **dbclient** commands on a suitable Linux box on the same network as the eCos system and specifying the same account and keys.

Application developers should also be aware that allowing remote systems running eCos to access an ssh server has security implications. For example if an attacker has physical access to a remote system, that attacker could use technology like jtag to examine the contents of the flash memory and search for plain text passwords or private keys. It is the developers' responsibility to understand the security issues associated with ssh technology and decide whether the risks are acceptable.

API

There are only two functions in the client-side API, one to establish a secure connection and run a command on the remote machine, the other to shut down the connection cleanly. The key data structure is a cyg_dropbear_cli_handle which holds all application-level state relevant to the connection.

```
typedef struct cyg_dropbear_cli_handle {
    int         db_stdin_stdout;
    int         db_stderr;
    int         db_exitcode;
    char        db_error[CYG_DROPBEAR_MAX_ERROR];
    …
} cyg_dropbear_cli_handle;
```

Each ssh connection requires its own instance of this data structure, and the instance must exist for the duration of the connection. All of the fields are managed by the dropbear code and the application should only read them, not modify them in any way. The *`db_stdin_stdout`* and *`db_stderr`* fields are file descriptors corresponding to sockets. Any data written to *`db_stdin_stdout`* will appear on the remote application's stdin stream. Any data written by the remote application to its stdout will appear as input on *db_stdin_stdout*. Any data written to its stderr will appear as input on *db_stderr*. The *db_exitcode* field is only valid once the remote application has exited and hold its exit code, usually 0 for a successful run and non-zero to indicate some kind of error. Most error conditions associated with the ssh connection itself will result in an error message being placed in *db_error*. However error conditions within the remote command will typically be reported via the stderr stream.

A typical client-side application will look like this:

```
void
run_remote_program(…)
{
    cyg_dropbear_cli_handle handle;

    <Fill in a struct sockaddr with the server's network address>

    <Fill in an authentication structure>

    if (!cyg_dropbear_ssh_connect(&handle, …)) {
        <Something has gone wrong during the connect process>
        <If there is a user, report the handle's db_error message>
        return;
    }

    if ( <reading from remote application> ) {
        while (! <EOF detected on db_stdin_stdout> ) {
            <Use read() on db_stdin_stdout>
            <Optionally, for robustness, look for errors on db_stderr>
        }
    } else { // writing to remote application
        while ( <there is data to be written> ) {
            <use write() on db_stdin_stdout>
            <Optionally, for robustness, look for errors on db_stderr>
        }
    }

    cyg_dropbear_ssh_close(&handle, 1);
    <Optionally check the exit code>
}
```

More complicated behaviour is possible, and the `clitest1.c` testcase in the package's `tests` subdirectory provides numerous examples.

Connecting

The `cyg_dropbear_ssh_connect` function takes four arguments:

`handle`

A pointer to a cyg_dropbear_cli_handle structure. This structure will be filled in and managed by the dropbear code, and should only be read by the eCos application. The structure must remain valid for the duration of the ssh connection, until after the call to `cyg_dropbear_ssh_close`.

`addr`

The full address of the ssh server on the remote machine. Typically this will actually be a sockaddr_in or sockaddr_in6 structure (assuming `CYGPKG_NET_INET6` is enabled). The ssh server must be accessible via this address irrespective of any firewall filtering, tcpwrapper settings (`/etc/hosts.allow` and `/etc/hosts.deny`), and ssh server settings (`/etc/ssh/sshd_config` on the host, especially the `AddressFamily`, `ListenAddress` and `Port` settings). Usually the port number will be `htons(22)` but an alternative ssh server listening on a different port may also be used. The dropbear code does not examine the contents of the address, it simply passes the address on to the TCP/IP stack's `connect` function.

`auth`

This structure holds all the authentication information, and is discussed in detail below.

`command` The command to be executed on the remote server. Typically this will be run using the account's default shell with a `-c <command>` option. If `NULL` is passed then the remote ssh server will start an interactive shell.

If the connection attempt succeeds and the remote ssh server starts the remote shell then `cyg_dropbear_ssh_connect` will return 1 and the *`db_stdin_stdout`* and *`db_stderr`* fields in the handle structure will be filled in with suitable sockets. Note that this does not mean that the remote shell has successfully started the requested command. If that part of the operation fails then the shell will output an error message on stderr and exit.

If the connection attempt fails because of a lack of resources, because the remote ssh server is not accessible, or because of an authentication failure, then `cyg_dropbear_ssh_connect` will return 0 and the handle's *`db_error`* field will contain an error message.

Internally, establishing an ssh connection involves starting a separate worker thread and it is the worker thread which runs the main dropbear code. It accepts messages over the network socket from the remote ssh server, decrypts them, and forwards them over local sockets to the application's *`db_stdin_stdout`* and *`db_stderr`*. It also accepts data written to *`db_stdin_stdout`* via a local socket, encrypts it, and passes it on to the remote application via the network socket and the ssh server.

Authentication

The cyg_dropbear_authenticate structure passed to `cyg_dropbear_ssh_connect` holds the information needed to authenticate the connection with the remote ssh server. Most of the fields are optional, as long as at least one valid authentication mechanism is provided. The structure contains the following fields:

```
typedef struct cyg_dropbear_authenticate {
    const char*     db_username;
    const char*     db_host_rsa_key_pub;
    const char*     db_host_dsa_key_pub;
    const char*     db_id_rsa;
    int             db_id_rsa_keylen;
    const char*     db_id_dsa;
    int             db_id_dsa_keylen;
    const char*     db_password;
} cyg_dropbear_authenticate;
```

The structure can be constructed at run-time or statically allocated. The dropbear code only reads the various fields during the call to `cyg_dropbear_ssh_connect`.

The *`db_username`* field must be filled in. It should be a simple string corresponding to a valid account name on the ssh server machine, for example:

```
    struct cyg_dropbear_authenticate auth;
    …
    auth.db_username = "dropbeartest";
```

The account name must also be one allowed by the ssh server, as per the `/etc/ssh/sshd_config` file's `AllowUsers` setting on the host.

The rsa and dsa host keys can be used to validate the identity of the remote ssh server, preventing certain man-in-the-middle attacks. These fields serve much the same purpose as the `~/.ssh/known_hosts` file when using the **ssh** command on a Linux system. During the authentication stage of establishing a connection the remote ssh server will send a signature encrypted using the server's private key, and the public keys can be used to validate this signature. The fields should be initialized using the contents of the `/etc/ssh/ssh_host_dsa_key.pub` and `/etc/ssh/ssh_host_rsa_key.pub` files on the host server, for example:

```
    …
    auth.db_host_rsa_key_pub = "ssh-rsa AAAAB3Nz…rb8=";
    auth.db_host_dsa_key_pub = "ssh-dss AAAABDNz…v7s=";
```

Note that the host's `/etc/ssh/sshd_config` file can specify alternative keys using `HostKey` settings. It is not necessary to supply the public host keys to the dropbear code. If neither host key is supplied then the code will simply not attempt to validate the identity of the remote ssh server and the `known_hosts` protection is not applied.

Specifying the public host keys in the authentication structure has one major disadvantage. If it ever becomes necessary to change the host keys on the ssh server then the eCos boxes will not be able to connect to the remote ssh server until the boxes are updated with a new host key. This behaviour is different from running **ssh** interactively on the Linux command line where the user will be given the choice of accepting the new key and updating the `known_hosts` file.

The *`db_id_rsa`*, *`db_id_rsa_keylen`*, *`db_id_dsa`* and *`db_id_dsakeylen`* fields are used to hold the private keys for public key authentication. If neither key is supplied then the dropbear code will only attempt password authentication. It should be noted that establishing a secure connection using an RSA private key requires many more cpu cycles than using a DSA private key. If both keys are supplied then the dropbear code will try the DSA key first, then the RSA key. The overheads can be reduced by using a smaller keysize, but obviously that has security implications.

The dropbear code requires its private keys in a different format from the Linux **ssh** command, and embedding these keys requires a somewhat convoluted process. One approach involves generating the keys using the host's **ssh-keygen** utility with *`-t rsa`* or *`-t dsa`*. No passphrase must be used, and great care should be taken not to overwrite the user's default private key file. The ssh private key can then be converted into a dropbear key using the **dropbearconvert** utility, either built from the sources or installed via e.g. the standard dropbear package.

Notes

1. **dropbearconvert** requires the private ssh key to be in legacy PEM private key format, while the default format of private keys for OpenSSH's **ssh-keygen** command is its own internal format. The option *`-m PEM`* to **ssh-keygen** may be used to specify the private key format of PEM when creating a new private key, or to convert an existing key from internal to PEM format using the *`-p`* option (traditionally used to remove or change a key's passphrase). For example:

   ```
   ssh-keygen -m PEM -t rsa -C "dropbear@example.com"  # Create new rsa key
   ssh-keygen -m PEM -p -f ~/.ssh/id_rsa               # Convert existing key to PEM format
   ```

2. OpenSSH 7.0 and greater disables the ssh-dss (DSA) public key algorithm by default as it is considered too weak. It's use is not recommended. For additional details see:

   - https://www.openssh.com/legacy.html
   - https://security.stackexchange.com/questions/112802/why-openssh-deprecated-dsa-keys

The dropbear-format `db_id_dsa` file can then be converted into a C array using the **privatekey2c** utility supplied in the package's `misc/hangman` and `misc/shell` directories.

For example, to generate a set of private keys for both dsa and rsa, convert them into dropbear format, and generate a C array for inclusion into an eCos client:

```
ssh-keygen -m PEM -t rsa -C "dropbear@ecoscentric.com"      # Create rsa key
ssh-keygen -m PEM -t dsa -C "dropbear@ecoscentric.com"      # Create dsa key
dropbearconvert openssh dropbear ~/.ssh/id_rsa db_id_rsa    # Convert to dropbear format
dropbearconvert openssh dropbear ~/.ssh/id_dsa db_id_dsa    # Convert to dropbear format
privatekey2c id_rsa db_id_rsa > id_rsa.c                    # Create C code for eCos
privatekey2c id_dsa db_id_dsa > id_dsa.c                    # Create C code for eCos
```

The resulting `id_dsa.c` file will contain code like the following:

```
#define ID_DSA_LEN 457
static const char id_dsa[ID_DSA_LEN] = {
    0x00, 0x00, 0x00, 0x07, \
    … \
    0xd3 \
};
```

The resulting file can be #include'd by the application code and used to fill in the auth structure's *`db_id_dsa`* and *`db_id_dsa_keylen`* fields.

As an alternative to using the **ssh-keygen** and **dropbearconvert** utilities, dropbear's **dropbearkey** can be used to generate the private key files directly in the dropbear format. It will still be necessary to use **privatekey2c** to turn the keys into a C array.

The remote ssh server will use the account's `~/.ssh/authorized_keys` file to validate the private keys supplied by the eCos application. This file must be created or updated with the corresponding public keys. For example:

```
cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
cat ~/.ssh/id_dsa.pub >> ~/.ssh/authorized_keys
chmod go-rw ~/.ssh/authorized_keys
```

Note

Prior to version 3 of OpenSSH, public key authentication used the file `authorized_keys` for SSH Protocols 1.3 and 1.5, while `authorized_keys2` was used for SSH protocol 2.0. ersion 3 and above of OpenSSH deprecated the use of `authorized_keys2`, allowing all public keys to be put into `authorized_keys`.

It is also necessary to have `PubkeyAuthentication` enabled in `/etc/ssh/sshd_config` on the ssh server host, and if DSA is to be permitted, `PubkeyAcceptedKeyTypes` must include *`ssh-dss`*.

The final field in the authentication structure is *`db_password`*. Again this should be a C string holding the plaintext password, or NULL if password authentication should not be attempted. The ssh server will only allow plaintext passwords if `PasswordAuthentication` is enabled in the server's configuration file.

I/O with the remote process

Once a secure connection has been established using `cyg_dropbear_ssh_connect` the eCos application code can interact with the remote command using the *`db_stdin_stdout`* and *`db_stderr`* fields in the connection's handle structure. These fields hold file descriptors corresponding to local sockets, and the other ends of these local sockets are managed by an internal worker thread started by the connect operation and running the bulk of the dropbear code. Typically I/O involves the `read`, `write` and `select` calls. Any data written to *`db_stdin_stdout`* will be read by the worker thread, encrypted, forwarded to the remote ssh server over the network connection, decrypted, and can be read by the remote command using its stdin stream. Any data written by the remote command to stdout or stderr will follow the reverse path and can be read by the eCos application code using the *`db_stin_stdout`* and *`db_stderr`* file descriptors.

If the remote command exits while the eCos application is still reading data, this will result in an end-of-file condition on the *`db_stdin_stdout`* file descriptor. In other words a `read` call will return 0. At that point the application should close the connection using `cyg_dropbear_ssh_close`, and not by using the `close` call on the file descriptor. Similarly if the eCos application wants to close the connection while the remote command is still running then it should use `cyg_dropbear_ssh_close`.

Given the complexity of the data flow, details of any error conditions such as broken network connections will not get as far as the application code. Specifically, the value of the `errno` variable will not correspond to the underlying error condition. Instead error conditions are likely to manifest as end-of-file conditions indistinguishable from the command exiting.

Closing a connection

When the eCos application wants to close down the ssh connection it should call `cyg_dropbear_ssh_close`. This takes two arguments, the handle filled in by `cyg_dropbear_ssh_connect` and a flag to indicate whether or not the application wants to wait for the remote command to exit. If the application does not wait then the *`db_exitcode`* field may not be valid, and the internal worker thread will continue to run in the background and consume resources until the remote command has finished.

Configuration

There are no configuration options specific to the client-side API, other than ones related to testing as described below. However there are a number of options common to the server-side and client-side API.

CYGNUM_NET_DROPBEAR_MAX_PAYLOAD_LEN

While establishing a secure connection the client and daemon negotiate an upper limit for the outgoing packet size. This has to be large enough to cope with the initial key exchange and related information. For eCos the default size is set to 1K, which should be sufficient but is still rather smaller than the dropbear default on other platforms. If the application involves transferring large amounts of data over ssh then throughput may be improved by increasing this packet size, at the cost of increased memory usage.

CYGNUM_NET_DROPBEAR_THREAD_PRI

The client-side dropbear code involves an internal worker thread for each secure connection. These worker threads run at the same priority controlled by this configuration option. The ssh protocol involves computationally intensive operations such as encrypting and decrypting packets, so these threads may have a significiant impact on the number of cpu cycles available to the rest of the system. It may be necessary to manipulate the thread priority to achieve an acceptable balance between overall system performance and the achievable ssh bandwidth.

CYGNUM_NET_DROPBEAR_THREAD_STACKSIZE

This configuration option controls the size of the stacks allocated for internal worker threads. Fairly large stacks are needed to implement the ssh protocol, and in addition the various application callback functions will also run on these stacks. The default value should suffice for all architectures, but this can be checked by using a debug build with stack checking enabled. If the stack sizes are reduced to save memory then it is strongly recommended that the system be tested with stack checking enabled.

Testing

The client-side code comes with an extensive testcase tests/clitest1.c. However this testcase is not built by default. It requires information about the testing environment such as the address of a suitable ssh server, as well as authentication information such as private keys and plain-text passwords. Embedding that kind of information directly into readily-available source code would have obvious security implications. Instead it is necessary to set a variety of configuration options before the testcase can be built.

Not all relevant configurations options need to be set. For example if the booldata option CYGTST_NET_DROPBEAR_TESTS_CLI_PASSWORD is not enabled then the testcase will not attempt any tests related to plain-text passwords. Details of the authentication mechanisms and relevant server settings can be found in the authentication section above.

Some of the configuration values are long, for example complete private ssh keys. Entering these using the graphical configuration tool may be problematical. Instead the ecos.ecc savefile can be edited directly. Alternatively the dropbear package comes with a file misc/clitest1.ecm, containing the configuration options related to testing. This file can be edited and then imported into the eCos configuration. Where appropriate the testcase will turn the configuration values into C strings automatically, so there is no need to add extra quotes to the values. The relevant options are:

CYGPKG_NET_DROPBEAR_TESTS_CLI

Unless this option is enabled the client-side testcase will not be built.

CYGTST_NET_DROPBEAR_TESTS_CLI_SERVER_ADDR

This should be the address of the remote ssh server in a format acceptable to the standard inet_pton function, for example 192.168.0.42.

```
cdl_option CYGTST_NET_DROPBEAR_TESTS_CLI_SERVER_ADDR {
    user_value 192.168.0.42
};
```

CYGTST_NET_DROPBEAR_TESTS_CLI_SERVER_PORT

This should be the tcp port number for the remote ssh server. It should be 22 when interacting with the system's standard ssh server.

```
cdl_option CYGTST_NET_DROPBEAR_TESTS_CLI_SERVER_PORT {
#    user_value 22
};
```

CYGTST_NET_DROPBEAR_TESTS_CLI_AF

This should be the address family, `AF_INET` for IPv4 testing or `AF_INET6` for IPv6 testing.

```
cdl_option CYGTST_NET_DROPBEAR_TESTS_CLI_AF {
#    user_value AF_INET
};
```

CYGTST_NET_DROPBEAR_TESTS_CLI_USERNAME

This should be the name of the testing account on the ssh server machine.

```
cdl_option CYGTST_NET_DROPBEAR_TESTS_CLI_USERNAME {
     user_value dropbeartest
};
```

CYGTST_NET_DROPBEAR_TESTS_CLI_HOST_RSA_KEY_PUB

The public RSA host key for the remote ssh daemon. On a modern debian Linux systems, this can be found in `/etc/ssh/ssh_host_rsa_key.pub`. Note that the trailing comment, normally the hostname, should be ommitted.

```
cdl_option CYGTST_NET_DROPBEAR_TESTS_CLI_HOST_RSA_KEY_PUB {
    user_value 1 "ssh-rsa AAAA…rb8="
};
```

CYGTST_NET_DROPBEAR_TESTS_CLI_HOST_DSA_KEY_PUB

The public DSA host key for the remote ssh daemon. On a modern debian Linux systems, this can be found in `/etc/ssh/ssh_host_dsa_key.pub`. Note that the trailing comment, normally the hostname, should be ommitted.

```
cdl_option CYGTST_NET_DROPBEAR_TESTS_CLI_HOST_DSA_KEY_PUB {
    user_value 1 "ssh-dss AAAA…v7s="
};
```

CYGTST_NET_DROPBEAR_TESTS_CLI_ID_RSA

This value should be the contents of a private RSA key. This can be obtained using the **privatekey2c** utility as described above in the authentication section. The contents of the C array in the resulting source file can then be copied and pasted into the .ecm or .ecc file.

```
cdl_option CYGTST_NET_DROPBEAR_TESTS_CLI_ID_RSA {
    user_value 1 " \
    0x00, 0x00, 0x00, 0x07, 0x73, 0x73, 0x68, 0x2d, \
    … \
    0x76, 0xfe, 0x15"
};
```

CYGTST_NET_DROPBEAR_TESTS_CLI_ID_DSA

This value should be the contents of a private DSA key. This can be obtained using the **privatekey2c** utility as described above in the authentication section. The contents of the C array in the resulting source file can then be copied and pasted into the .ecm or .ecc file.

```
cdl_option CYGTST_NET_DROPBEAR_TESTS_CLI_ID_DSA {
    user_value 1 " \
    0x00, 0x00, 0x00, 0x07, 0x73, 0x73, 0x68, 0x2d, \
    … \
    0x1d"
};
```

CYGTST_NET_DROPBEAR_TESTS_CLI_PASSWORD

This value should be the plain-text password used for authenticating the user.

```
cdl_option CYGTST_NET_DROPBEAR_TESTS_CLI_PASSWORD {
    user_value 1 opensesame
};
```

SSH Server Host Configuration

Modern Linux hosts are normally tightly configured with respect to ssh access so as to minimise risk of being compromised. When using an OpenSSH server on Linux as the test host, you may wish to ensure access to the host is not compromised and is restricted. For example, you may only wish to permit access by the dropbear clitest1 and scptest1 tests to the user specified in CYGTST_NET_DROPBEAR_TESTS_CLI_USERNAME from internal network addresses used by the test hardware. For example, if the test hardware were located on the internal IPv4 segment 192.168.8.0/24 and internal IPv6 segment fd54:5cd1:6fc7:1619::/64, the configuration file /etc/ssh/sshd_config could contain the following configuration extract that permits password and DSA autentication for the user dropbeartest from either network:

```
# Permit dropbear test user to connect from internal test networks and use password authentication
Match User dropbeartest Address 192.168.8.0/24,fd54:5cd1:6fc7:1619::/64
        PasswordAuthentication yes
        PubkeyAcceptedKeyTypes=+ssh-dss
```

When testing the user account from another Linux server using ssh, the user file ~/.ssh/config also should contain the following line to permit DSA private key authentication as that client host may likely have such authentication prohibited:

```
PubkeyAcceptedKeyTypes +ssh-dss
```

Name

Dropbear — scp client support

Synopsis

#include <dropbear.h>

int **cyg_dropbear_scp_open**(*handle*, *addr*, *auth*, *path*, *flags*, *len*, *mode*);

ssize_t **cyg_dropbear_scp_write**(*handle*, *buf*, *len*);

ssize_t **cyg_dropbear_scp_read**(*handle*, *buf*, *len*);

void **cyg_dropbear_scp_close**(*handle*);

Description

The scp client support allows eCos applications to read and write files on a remote server over a secure channel. It is implemented as a thin layer over the generic client-side support and the same caveats regarding security implications etc. are applicable. The package comes with a testcase `tests/scptest1.c` which can serve as example code.

API

The scp API consists of just four functions. `cyg_dropbear_scp_open` is used to establish a secure connection to a remote ssh server, run the scp command on that server to handle the remote file I/O. and perform some initial protocol operations. The data can then be transferred using repeated calls to `cyg_dropbear_scp_read` and `cyg_dropbear_scp_write`. Finally `cyg_dropbear_scp_close` can be used to shut down the connection. All calls make use of a handle structure to hold per-connection state:

```
typedef struct cyg_dropbear_scp_handle {
    cyg_dropbear_cli_handle     db_cli_handle;
    …
} cyg_dropbear_scp_handle;
```

The main field of interest is *`db_cli_handle.db_error`* which will contain a suitable error message if a connect operation fails. Typical code to write to a remote file would look like this:

```
<global sockaddr_storage structure containing a suitable address>

<global cyg_dropbear_authenticate structure appropriately filled in>

void
write_remote_file(char* buf, int len)
{
    cyg_dropbear_scp_handle  handle;
    int                      xfrd;

    if (!cyg_dropbear_scp_open(&handle, <addr>, <auth>,
                               "/tmp/out", O_WRONLY, &len,
                               S_IRUSR | S_IWUSR)) {
        <report handle.db_cli_handle.db_error to the user>
        return
    }

    for (xfrd = 0; xfrd < len; ) {
        <use cyg_dropbear_scp_write to send a chunk to the remote server>
    }

    cyg_dropbear_scp_close(&handle);
}
```

The code for reading a remote file is very similar:

```
void
read_remote_file(char* buf, int maxlen)
{
    cyg_dropbear_scp_handle handle;
    int len;

    if (!cyg_dropbear_scp_open(&handle, <addr>, <auth>,
                               "/tmp/in", O_RDONLY, &len, 0)) {
        <report handle.db_cli_handle.db_error to the user>
        return
    }
    if (len > maxlen) {
        <decide what to do>
    }

    for (xfrd = 0; xfrd < len; ) {
        <use cyg_dropbear_scp_read to read a chunk from the remote server>
    }

    cyg_dropbear_scp_close(&handle);
}
```

Connecting

The function cyg_dropbear_scp_open is used to establish a secure connection to a remote server and to open a file on that remote system. It takes seven arguments:

handle
: A pointer to a cyg_dropbear_scp_handle structure. This will be filled in and managed by the dropbear code, and should only be read by the eCos application. The structure must remain valid for the duration of the scp operation, until after the call to cyg_dropbear_scp_close.

addr
: The full address of the ssh server on the remote machine. Typically this will actually be a sockaddr_in or sockaddr_in6 structure (assuming CYGPKG_NET_INET6 is enabled). The ssh server must be accessible via this address irrespective of any firewall filtering, tcpwrapper settings (/etc/hosts.allow and /etc/hosts.deny, if enabled), and ssh server settings (/etc/ssh/sshd_config, especially the AddressFamily, ListenAddress and Port settings). Usually the port number will be htons(22) but it is possible to connect to an alternative ssh server listening on a different port, if desired. The dropbear code does not examine the contents of the address, it simply passes the address on to the TCP/IP stack's connect function.

auth
: This structure holds all the authentication information and is discussed in detail in the documentation for the Ssh client support.

path
: The full path of a file on the remote system.

flags
: This should be O_RDONLY to read a file on the remote system, or O_WRONLY to write a file.

len
: The scp protocol requires that the total amount of data to be transferred is known at the start of the transfer. len should be a pointer to an ssize_t variable. For a write operation the application should initialize that variable with the total transfer size before calling cyg_dropbear_scp_open. For a read operation the dropbear code will set that variable to the file size, thus letting the application know how much data it should read.

mode
: This field is only relevant when writing a file, and is used to set the access mask for the file on the remote system as per e.g. the Linux chmod system call. It will be some combination of the S_IRUSR, S_IWUSR, S_IXUSR, S_IRGRP, S_IWGRP, S_IXGRP, S_IROTH,

S_IWOTH, and S_IXOTH constants defined in the <sys/stat.h> header file. Note that the application should use the eCos values for these constants and the dropbear code will automatically translate them to the Linux equivalents. Also note that the settings are subject to the account's umask value on the remote server.

The cyg_dropbear_scp_open function will attempt to make a secure connection to the remote server, start the **scp** on that server, and perform the initial protocol operations. If it returns successfully then the application can proceed with the data transfers using cyg_dropbear_scp_write or cyg_dropbear_scp_read. If it fails for any reason then an error message will be written to the *db_cli_handle.db_error* field of the handle. Due to the complexity of the operation and the implementation's need for a background worker thread that runs the bulk of the dropbear code, the value in errno may not give an accurate indication of the error(s) that occurred.

Transferring Data

Once an scp connection has been established the application can transfer data using cyg_dropbear_scp_write if the remote file was opened with O_WRONLY, or cyg_dropbear_scp_read if O_RDONLY was used. In addition to transferring the data these functions catch certain error conditions and manage the scp protocol, so their use is preferable to any attempt to read or write the data directly over sockets.

cyg_dropbear_scp_read will return the amount of data actually read during this call, which may be less than the amount requested because of buffering effects. Typically this function will be called in a loop until all required data has been transferred. A return value of 0 indicates an end-of-file condition, usually because the transfer is complete but possibly because the connection has been broken. A return value of -1 indicates some unexpected and indeterminate error condition.

cyg_dropbear_scp_write will return the amount of data actually written during this call. Usually this will be the amount requested, but may be less because of buffering effects. Typically this function will be called in a loop until all required data has been transferred. For large transfers it may be desirable to split the transfer into a number of smaller chunks, effectively spreading the cpu cycle and buffering costs over a longer period of time. A return value of 0 or -1 indicates an unexpected and indeterminate error condition.

Closing a Connection

At the end of a transfer cyg_dropbear_scp_close should be used to shut down the connection. The handle structure will no longer be needed after this call returns.

Normally the application should transfer exactly the amount of data requested. For a write this is the size specified during the open call. For a read this is the size filled in by the dropbear code during the open call. It is possible to call cyg_dropbear_scp_close before the transfer has completed. For a read this is harmless. For a write, some or all of the data transferred so far may be discarded by the remote scp command, and the exact behaviour is unpredictable.

Configuration

There are no configuration options specific to the client-side scp support. However this support is built on top of the generic ssh client-side API so all configuration options relevant to that also affect scp operations.

Testing

The dropbear package comes with a testcase tests/scptest1.c. However this testcase is not built by default. It will only be built if the configuration enables the building of the generic client-side testcase tests/clitest1.c, and will use the same configuration options to identify the remote server and to provide the authentication information.

Part L. FTP Client for eCos TCP/IP Stack

The ftpclient package provides an FTP (File Transfer Protocol) client for use with the TCP/IP stack in eCos. It supports both IPv4 and IPv6 and will use the DNS client, when its is part of the eCos configuration. The API allows for active (server->client) or passive (client->server) transfer connections to be used.

The data operations support binary mode transfers only. The supplied filename paths only support Unix-style '/, directory separators.

The API provides support for either direct buffer transfers, or for call-back routines to be used to copy data as required. Care must be taken, especially when receiving files, since the direct buffer support calls are limited to handling files that will fit within the single buffer passed through the API. The API call-back based implementation can avoid this limitation by allowing the client to control the data reception and buffering as appropriate.

By default the data transfer operations will initially attempt a connection using the newer “extended” (IPv6 capable) FTP commands (`EPRT` or `EPSV` as appropriate). If those commands are not supported by the FTP server then the client will fallback to using the original FTP connection commands (`PORT` or `PASV` respectively).

Table of Contents

Chapter 184. FTP Client API and Configuration

FTP Client API

The FTP Client API is provided by the include file `install/include/ftpclient.h` and it can be used thus:

```
#include <network.h>
#include <ftpclient.h>
```

Note

The reason for the following variety of API operations performing the `get` and `put` operations is for backwards compatibility with earlier releases of the FTP Client package. Existing API parameter requirements have not been changed, with new features adding new API calls as required.

Support API

Support functions are available irrespective of the type of data transfer operations used.

ftp_delete

```
int ftp_delete(char * hostname,
               char * username,
               char * passwd,
               char * filename,
               ftp_printf_t ftp_printf);
```

Delete a file from the FTP server. The *`filename`* should be the full pathname of the file to be deleted.

ftpclient_printf

```
void ftpclient_printf(unsigned error, const char *fmt, …);
```

The `get` and `put` API operations take a pointer to a function to use for printing out diagnostic and error messages. This is a sample implementation which can be used if you don't want to implement the function yourself. *`error`* will be true when the message to print is an error message. Otherwise the message is diagnostic, eg. the commands sent and received from the server.

Basic FTP Client API

The basic API only supports active connections, and is provided for backwards compatability with earlier releases. It is superceded by the extended API described below, which provides greater control and extra features not available with the original basic API. The basic API consists of the following exported functions:

ftp_get

```
int ftp_get(char * hostname,
            char * username,
            char * passwd,
            char * filename,
            char * buf,
            unsigned buf_size,
            ftp_printf_t ftp_printf);
```

Use the FTP protocol to retrieve a file from a server. Only binary mode is supported. The filename can include a directory name. Only use unix style ’/, file separators, not ’\,. The file is placed into *`buf`*. *`buf`* has maximum size *`buf_size`*. If the file is bigger than this, the transfer fails and FTP_TOOBIG is returned. Other error codes listed in the header can also be returned. If the transfer is successful the number of bytes received is returned.

ftp_put

```
int ftp_put(char * hostname,
            char * username,
            char * passwd,
            char * filename,
            char * buf,
            unsigned buf_size,
            ftp_printf_t ftp_printf);
```

Use the FTP protocol to send a file to a server. Only binary mode is supported. The filename can include a directory name. Only use unix style ’/, file separators, not ’\,. The contents of *`buf`* are placed into the file on the server. If an error occurs one of the codes listed will be returned. If the transfer is successful zero is returned.

ftp_get_var

```
int ftp_get(char * hostname,
            char * username,
            char * passwd,
            char * filename,
            ftp_write_t ftp_write,
            void * ftp_write_priv,
            ftp_printf_t ftp_printf);
```

Use the FTP protocol to retrieve a file from a server. Only binary mode is supported. The filename can include a directory name. Only use unix style ’/, file separators, not ’\,. The *`ftp_write`* function is called as data arrives, using the supplied *`ftp_write_priv`* private context. If the transfer is successful the total number of bytes received is returned.

ftp_put_var

```
int ftp_put(char * hostname,
            char * username,
            char * passwd,
            char * filename,
            ftp_read_t ftp_read,
            void * ftp_read_priv,
            ftp_printf_t ftp_printf);
```

Use the FTP protocol to send a file to a server. Only binary mode is supported. The filename can include a directory name. Only use unix style ’/, file separators, not ’\,. The *`ftp_read`* function is called to fetch data to be written, using the supplied *`ftp_read_priv`* private context. The call returns the total amount of data written, or a negative error indication.

Extended FTP Client API

The extended FTP Client API provides for more control of the FTP connection, including passive mode selection and timeout configuration. The extended API uses the `ftp_extended_info` structure to pass information into the handler functions. The use of a structure allows the data to be re-used, or individual fields modified, between calls using the API. The extended API also provides a boolean `passive` that if `true` causes the relevant API call to use a passive FTP connection, with `false` indicating an active FTP connection (as per the original API).

The extended API also allows for RX and TX timeouts (specified as a number of seconds) to be used for connection and data transfers. If either of the timeout fields is set to 0 then the respective CDL configuration option

`CYGNUM_NET_FTPCLIENT_TIMEOUT_RX` and `CYGNUM_NET_FTPCLIENT_TIMEOUT_TX` will be used by the API operation instead.

The structure contains the following fields, which mostly mirror the individual parameter functions as used by the older (active-only) API:

```
struct ftp_extended_info {
  cyg_bool passive;
  char *hostname;
  char *username;
  char *passwd;
  char *filename;
  ftp_printf_t ftp_printf;
  unsigned int rx_timeout;
  unsigned int tx_timeout;
};
```

The extended API consists of the following functions:

ftp_get_extended

```
int ftp_get_extended(struct ftp_extended_info * info,
                     char * buf,
                     unsigned buf_size);
```

Use the FTP protocol to retrieve a file from a server. Only binary mode is supported. The filename can include a directory name. Only use unix style '/, file separators, not '\,. The *`info`* parameter provides the server connection and credentials information, as well as indicating the type of connection to establish, the diagnostic and error output routine and the RX and TX timeouts. The file is placed into *`buf`*. *`buf`* has maximum size *`buf_size`*. If the file is bigger than this, the transfer fails and FTP_TOOBIG is returned. Other error codes listed in the header can also be returned. If the transfer is successful the number of bytes received is returned.

ftp_put_extended

```
int ftp_put_extended(struct ftp_extended_info * info,
                     char * buf,
                     unsigned buf_size);
```

Use the FTP protocol to send a file to a server. Only binary mode is supported. The filename can include a directory name. Only use unix style '/, file separators, not '\,. The *`info`* parameter provides the server connection and credentials information, as well as indicating the type of connection to establish, the diagnostic and error output routine and the RX and TX timeouts. The contents of *`buf`* are placed into the file on the server. If an error occurs one of the codes listed will be returned. If the transfer is successful zero is returned.

ftp_get_extended_var

```
int ftp_get_extended_var(struct ftp_extended_info * info,
                         ftp_write_t ftp_write,
                         void * ftp_wrote_priv);
```

Use the FTP protocol to retrieve a file from a server. Only binary mode is supported. The filename can include a directory name. Only use unix style '/, file separators, not '\,. The *`info`* parameter provides the server connection and credentials information, as well as indicating the type of connection to establish, the diagnostic and error output routine and the RX and TX timeouts. The *`ftp_write`* function is called as data arrives, using the supplied *`ftp_write_priv`* private context. If the transfer is successful the total number of bytes received is returned.

ftp_put_extended_var

```
int ftp_put_extended_var(struct ftp_extended_info * info,
                         ftp_read_t ftp_read,
                         void * ftp_read_priv);
```

Use the FTP protocol to send a file to a server. Only binary mode is supported. The filename can include a directory name. Only use unix style ’/, file separators, not ’\,. The *`info`* parameter provides the server connection and credentials information, as well as indicating the type of connection to establish, the diagnostic and error output routine and the RX and TX timeouts. The *`ftp_read`* function is called to fetch data to be written, using the supplied *`ftp_read_priv`* private context. The call returns the total amount of data written, or a negative error indication.

FTP Client Configuration

The FTP Client provides some CDL configuration items that can be used to tune the performance and behaviour of the client.

CYGNUM_NET_FTPCLIENT_BUFSIZE

> The FTP data transfer functions buffer the data as it passes between systems and this option controls the size of the dynamically allocated buffer. The buffers are allocated using the standard `malloc()` interface.

CYGNUM_NET_FTPCLIENT_TIMEOUT_RX

> This option controls the default timeout in seconds used when waiting for connection or data reception. It is used to set the `SO_RCVTIMEO` for the overall control and data socket connections, as well as the individual `read()` operations. The CDL value can be over-ridden by a run-time supplied value when using the extended API.

CYGNUM_NET_FTPCLIENT_TIMEOUT_TX

> This option controls the default timeout in seconds used when waiting for data transmissions to complete. It is used to set the `SO_SNDTIMEO` for the overall control and data socket connections, as well as the individual `write()` operations. The CDL value can be over-ridden by a run-time supplied value when using the extended API.

Part LI. FTP Server Support

Name

eCosPro Support FTP Service — Overview

Description

The ftpserver package provides an FTP (File Transfer Protocol) server for use with the TCP/IP stack in eCos. It is currently restricted to IPv4 networks only and can only use the BSD IP stack.

The server implements a restricted subset of the FTP protocol, sufficient to transfer binary files to and from an eCos system. Only passive mode transfers are supported. Directory creation ('**mkdir**'), changing ('**cd**') or listing ('**ls**') are not supported. User and password authentication is supported if the user supplies a means of checking these, otherwise access is permitted to all clients. As authentication is not securely encrypted, it is recommended that the FTP server is only used on an internal trusted network.

The server has been tested against the default Linux **ftp** command for both storing and retrieving files. It has also been tested with **wget** and **Firefox** for retrieving files. The server should work with any command-line based FTP client. Graphical clients are less likely to work since these will want to fetch file listings, which the server does not currently support.

Name

FTP Server API — describe FTP server API, callback and configuration

FTP Server API

The FTP server provides a single entry point:

```
    #include <ftpserver.h>

     __externC int cyg_ftpserver_start( cyg_ftpserver *server );
```

This function starts an FTP server running using parameters defined in the *server* argument. It should be called from a thread with a stack with at least CYGNUM_NET_FTPSERVER_STACK_SIZE bytes available. Under normal circumstances this function will not return, so the application should create a thread dedicated to the server. There are examples of this in the test programs.

The *server* argument contains a number of fields that the user may set to control the behaviour of the FTP server.

const char *address

> This is a string giving the IP address on which the server will listen. If not set this will default to listening on all network interface, which is usually what is wanted. This option is only useful if a target has more than one network interface and the FTP server should only listen on one of them.

int port

> This defines the port number on which the server will listen. Leaving this value unset, zero, will cause the server to choose the default FTP port of 21.

const char *greeting

> This is the string that the server will use in its initial response to a client connection. Leaving this NULL will cause the server to use a default message: "Welcome to eCosPro FTP service.".

const char *root

> This string defines a filesystem pathname to the root directory of the FTP server. Files will only be transferred to and from this directory or any subdirectories. A NULL pointer here will cause the server to use "/" as the root.

int data_port

> This defines the first port at which the server will create passive data connections. It will try successive ports from this value until a free port is found, limited to the following 20 ports. Leaving this value NULL will cause the server to start from port 5000.

The normal idiom for use of the FTP server is to define the cyg_ftpserver structure statically and to then call cyg_ftpserver_start() with a pointer to that. The following is a somewhat contrived example:

```
#include <ftpserver.h>

//==========================================================================

static cyg_ftpserver ftp_server =
{
    .port       = 2121,                            // Listen on port 2121
    .root       = "/fatfs/ftp",                    // FTP to/from here
    .greeting   = "Welcome to Fubar FTP service.", // Welcome string
    .data_port  = 40000                            // Move data ports to 40000+
};

//==========================================================================
static void ftpd(CYG_ADDRWORD p)
{
    init_all_network_interfaces();

    cyg_ftpserver_start( &ftp_server );
}

//==========================================================================

#define STACK_SIZE CYGNUM_NET_FTPSERVER_STACK_SIZE
static char stack[STACK_SIZE] CYGBLD_ATTRIB_ALIGN_MAX;
static cyg_thread thread_data;
static cyg_handle_t thread_handle;

void start_ftpd_thread(void)
{
    cyg_thread_create(10,               // Priority
                    ftpd,               // entry
                    0,                  // entry parameter
                    "FTP test",         // Name
                    &stack[0],          // Stack
                    STACK_SIZE,         // Size
                    &thread_handle,     // Handle
                    &thread_data        // Thread data structure
                    );
  cyg_thread_resume(thread_handle);  /* Start it */
}
```

The defaults for the configuration parameters are such that leaving them all zero or NULL will cause the server to configure itself on the standard port serving files to/from the root of the filesystem.

Callback Interface

The default server operates to and from a directory in a filesystem, in the same way that most FTP servers operate. However, not all embedded systems have filesystem support. To allow such systems to provide an FTP service, all file access operations go through a set of callback functions in the cyg_ftpserver structure. The user can define these to redirect file I/O operations to other devices such as memory buffers or flash memory regions.

The callback interface defines a type, ftpserver_cookie that is passed to the callback functions. It is a word or pointer sized value and is typically used by the callback functions to contain a data structure pointer, or an index into a table that uniquely identifies an open file.

The callbacks for a notional *example* interface are defined as follows:

`int example_open( const char *path, cyg_bool write, ftpserver_cookie *cookie );`

> Called to open a file for data transfer. The *path* is a fully rooted path formed from the *root* field of the server configuration plus the client supplied path. The *write* argument is true if the file is to be opened for writing and false if for reading. If successful the function may store a private value to **cookie*.
>
> If the file is opened successfully this function should return FTPSERVER_OK. If the file cannot be opened for reading it should return FTPSERVER_NO_FILE. If the file cannot be written to or created it should return FTPSERVER_NOT_ALLOWED.

`int example_write( ftpserver_cookie cookie, const cyg_uint8 *buf, cyg_uint32 size );`

> Called to write data to the file. The *cookie* argument is the value stored by the open callback. Arguments *buf* and *size* define the data to be written.
>
> If the data is successfully written this function should return FTPSERVER_OK. If any error occurs it should return FTPSERVER_WRITE_FAIL.

int example_read(ftpserver_cookie cookie, cyg_uint8 *buf, cyg_uint32 *size);

Called to read data from the file. The cookie argument is the value stored by the open callback. Arguments buf and *size define a buffer into which the data should be stored, *size should be updated with the amount of data written to the buffer. If no more data is available, *size should be set to zero.

If the function is successful it should return FTPSERVER_OK. If any error occurs it should return FTPSERVER_READ_FAIL.

int example_close(ftpserver_cookie cookie);

Called to close the file for further data transfer. The cookie argument is the value stored by the open callback. After this call the cookie value will not be used in further calls.

This function should return FTPSERVER_OK under all circumstance, there are no error conditions of interest to the FTP server.

int example_auth(const char *user, const char *passwd);

This function is called to authenticate a user ID and password supplied by the client. The FTP server will only request a password, by returning a 331 response, if a user supplied authentication function is installed.

If the user and password are accepted, this function should return FTPSERVER_LOGIN_OK; it should return FTPSERVER_BADUSER if they do not.

int example_delete(const char *path);

Called to delete a file. The path is a fully rooted path formed from the root field of the server configuration plus the client supplied path.

If the file is deleted successfully this function should return FTPSERVER_OPDONE. If the file cannot be deleted or is not found, FTPSERVER_NO_FILE should be returned.

Extending the previous example, the cyg_ftpserver might look like this:

```
static cyg_ftpserver ftp_server =
{
    .port       = 2121,                           // Listen on port 2121
    .root       = "/fatfs/ftp",                   // FTP to/from here
    .greeting   = "Welcome to Fubar FTP service.", // Welcome string
    .data_port  = 40000,                          // Move data ports to 40000+

    .open       = example_open,
    .read       = example_read,
    .write      = example_write,
    .close      = example_close,
    .auth       = example_auth,
    .delete     = example_delete,

};
```

Configuration Options

The FTP server package contains a number of configuration options:

CYGNUM_NET_FTPSERVER_BUFFER_SIZE

This option defines the size of the buffer used for data transfers. Buffer size is a tradeoff between memory use and transfer speed. The default should be a reasonable compromise between these two.

CYGNUM_NET_FTPSERVER_USER_SIZE

This option defines the size of the buffer used for storing the user name supplied by the client.

`CYGNUM_NET_FTPSERVER_PASSWD_SIZE`

This option defines the size of the buffer used for storing the password supplied by the client. The default reflects the habit of using a user's email address as a password for anonymous FTP.

`CYGNUM_NET_FTPSERVER_CONTROL_TIMEOUT`

This option defines the timeout applied to control streams. If after this number of seconds the client has not interacted with the server, the connection is closed down. Since the server only allows a single client at a time, any client that fails to close a control stream down will block it for others.

`CYGNUM_NET_FTPSERVER_DATA_TIMEOUT`

This option defines the timeout applied to data streams. If after this number of seconds the client has not transferred data to or from the server, the connection is closed down. Clients are expected to use a data connection immediately after it is created so this timeout can be considerably smaller than the control timeout.

Name

FTP Server Test Programs — describe the test programs and their host-side support

Test Programs

There are three test programs in the `tests` subdirectory of this package that exercise the FTP server and do duty as examples of its use. These are supported by an eCos configuration and **expect** scripts in the `misc` subdirectory which call the standard linux **ftp** client to exercise the FTP server.

To build an eCos library suitably configured to support the test programs, import the `ftpserver.ecm` minimal configuration located in the `misc` subdirectory of this package (at `packages/net/ftpserver/`*`VERSION`* within your eCos package repository) into your target configuration. This minimal configuration loads the `CYGPKG_NET`, `CYGPKG_IO`, `CYGPKG_NET_FTPSERVER` and `CYGPKG_FS_RAM` packages, enables the building of the test programs as well as increases the sizes of `CYGPKG_NET_MAXSOCKETS`, `CYGNUM_FILEIO_NFILE` and `CYGNUM_FILEIO_NFD` from their default values to values large enough to allow the test scripts to operate in a reasonable amount of time.

When each test program is executed it will output on either the **gdb** console, or a debug channel, the filename of a corresponding **expect** script to be executed on your development PC, along with any additional parameters to provide the script. This **expect** script is used to exercise each FTP server test application. The script name and parameter will be contained within angle braces prefixed by `CMD:`. Each script can be executed on a suitable Linux host which has **expect** and the standard Linux text-mode **ftp** client installed and on the PATH. For example, the output below suggests that the script `ftpserver1.exp` is executed with the argument `192.168.1.54`.

```
  INFO:<Create file1>
  CMD:<ftpserver1.exp 192.168.1.54>
  INFO:<Start FTP server>
```

These scripts are located within the `misc` subdirectory of the FTP server package and on its completion will terminate the ftp test program on the target.

Note

All tests can be stopped manually by logging onto the target platform running the test with a standard ftp client using the user ID "shutdown" and password "now".

ftpserver1

The `ftpserver1.c` test creates a default FTP server using the RAM filesystem. A default file, `file1`, is created that can be retrieved, and further files can be uploaded, retrieved or overwritten.

ftpserver2

The `ftpserver2.c` test uses the callback interface to serve files in memory. It creates a read-only file, `text`, and a 32KB read/write buffer, `buffer`.

ftpserver3

The `ftpserver3.c` test is identical to `ftpserver1.c` except that it expects a specific user and password, "test" and "barnwell", to be supplied before it permits access.

Part LII. Embedded HTTP Server

Table of Contents

Chapter 185. Embedded HTTP Server

Introduction

The *eCos* HTTPD package provides a simple HTTP server for use with applications in eCos. This server is specifically aimed at the remote control and monitoring requirements of embedded applications. For this reason the emphasis is on dynamically generated content, simple forms handling and a basic CGI interface. It is *not* intended to be a general purpose server for delivering arbitrary web content. For these purposes a port of the GoAhead web server is available from www.goahead.com.

This server is also capable of serving content using IPv6 when the eCos configuration contains IPv6.

Server Organization

The server consists of one or more threads running in parallel to any application threads and which serve web pages to clients. Apart from defining content, the application does not need to do anything to start the HTTP server.

The HTTP server is, by default, started by a static constructor. This simply creates an initial thread and sets it running. Since this is called before the scheduler is started, nothing will happen until the application calls `cyg_scheduler_start()`. The server thread can also be started explicitly by the application, see the `CYGNUM_HTTPD_SERVER_AUTO_START` option for details.

When the thread gets to run it first optionally delays for some period of time. This is to allow the application to perform any initialization free of any interference from the HTTP server. When the thread does finally run it creates a socket, binds it to the HTTP server port, and puts it into listen mode. It will then create any additional HTTPD server threads that have been configured before becoming a server thread itself.

Each HTTPD server thread simply waits for a connection to be made to the server port. When the connection is made it reads the HTTP request and extracts the filename being accessed. If the request also contains form data, this is also preserved. The filename is then looked up in a table.

Each table entry contains a filename pattern string, a pointer to a handler function, and a user defined argument for the function. Table entries are defined using the same link-time table building mechanism used to generate device tables. This is all handled by the `CYG_HTTPD_TABLE_ENTRY()` macro which has the following format:

```
#include <cyg/httpd/httpd.h>

CYG_HTTPD_TABLE_ENTRY( __name, __pattern, __handler, __arg )
```

The `__name` argument is a variable name for the table entry since C does not allow us to define anonymous data structures. This name should be chosen so that it is unique and does not pollute the name space. The `__pattern` argument is the match pattern. The `__handler` argument is a pointer to the handler function and `__arg` the user defined value.

The link-time table building means that several different pieces of code can define server table entries, and so long as the patterns do not clash they can be totally oblivious of each other. However, note also that this mechanism does not guarantee the order in which entries appear, this depends on the order of object files in the link, which could vary from one build to the next. So any tricky pattern matching that relies on this may not always work.

A request filename matches an entry in the table if either it exactly matches the pattern string, or if the pattern ends in an asterisk, and it matches everything up to that point. So for example the pattern "/monitor/threads.html" will only match that exact filename, but the pattern "/monitor/thread-*" will match "/monitor/thread-0040.html", "/monitor/thread-0100.html" and any other filename starting with "/monitor/thread-".

When a pattern is matched, the hander function is called. It has the following prototype:

```
cyg_bool cyg_httpd_handler(FILE *client,
                           char *filename,
                           char *formdata,
                           void *arg);
```

The *client* argument is the TCP connection to the client: anything output through this stream will be returned to the browser. The *filename* argument is the filename from the HTTP request and the *formdata* argument is any form response data, or NULL if none was sent. The *arg* argument is the user defined value from the table entry.

The handler is entirely responsible for generating the response to the client, both HTTP header and content. If the handler decides that it does not want to generate a response it can return `false`, in which case the table scan is resumed for another match. If no match is found, or no handler returns true, then a default response page is generated indicating that the requested page cannot be found.

Finally, the server thread closes the connection to the client and loops back to accept a new connection.

Server Configuration

The HTTP server has a number of configuration options:

`CYGNUM_HTTPD_SERVER_PORT`

> This option defines the TCP port that the server will listen on. It defaults to the standard HTTP port number 80. It may be changed to a different number if, for example, another HTTP server is using the main HTTP port.

`CYGDAT_HTTPD_SERVER_ID`

> This is the string that is reported to the client in the "Server:" field of the HTTP header.

`CYGNUM_HTTPD_THREAD_COUNT`

> The HTTP server can be configured to use more than one thread to service HTTP requests. If you expect to serve complex pages with many images or other components that are fetched separately, or if any pages may take a long time to send, then it may be useful to increase the number of server threads. For most uses, however, the connection queuing in the TCP/IP stack and the speed with which each page is generated, means that a single thread is usually adequate.

`CYGNUM_HTTPD_THREAD_PRIORITY`

> The HTTP server threads can be run at any priority. The exact priority depends on the importance of the server relative to the rest of the system. The default is to put them in the middle of the priority range to provide reasonable response without impacting genuine high priority threads.

`CYGNUM_HTTPD_THREAD_STACK_SIZE`

> This is the amount of stack to be allocated for each of the HTTPD threads. The actual stack size allocated will be this value plus the values of `CYGNUM_HAL_STACK_SIZE_MINIMUM` and `CYGNUM_HTTPD_SERVER_BUFFER_SIZE`.

`CYGNUM_HTTPD_SERVER_BUFFER_SIZE`

> This defines the size of the buffer used to receive the first line of each HTTP request. If you expect to use particularly long URLs or have very complex forms, this should be increased.

`CYGNUM_HTTPD_SERVER_AUTO_START`

> This option causes the HTTP Daemon to be started automatically during system initialization. If this option is not set then the application must start the daemon explicitly by calling `cyg_httpd_startup()`. This option is set by default.

`CYGNUM_HTTPD_SERVER_DELAY`

> This defines the number of system clock ticks that the HTTP server will wait before initializing itself and spawning any extra server threads. This is to give the application a chance to initialize properly without any interference from the HTTPD.

Support Functions and Macros

The emphasis of this server is on dynamically generated content, rather than fetching it from a filesystem. To do this the handler functions make calls to `fprintf()` and `fputs()`. Such handler functions would end up a mass of print calls, with the actual structure of the HTML page hidden in the format strings and arguments, making maintenance and debugging very difficult. Such an approach would also result in the definition of many, often only slightly different, format strings, leading to unnecessary bloat.

In an effort to expose the structure of the HTML in the structure of the C code, and to maximize the sharing of string constants, the `cyg/httpd/httpd.h` header file defines a set of helper functions and macros. Most of these are wrappers for predefined print calls on the *`client`* stream passed to the hander function. For examples of their use, see the System Monitor example.

Note

All arguments to macros are pointers to strings, unless otherwise stated. In general, wherever a function or macro has an *`attr`* or *`__attr`* parameter, then the contents of this string will be inserted into the tag being defined as HTML attributes. If it is a NULL or empty string it will be ignored.

HTTP Support

```
void cyg_http_start( FILE *client, char *content_type, int content_length );
void cyg_http_finish( FILE *client );
#define html_begin(__client)
#define html_end( __client )
```

The function `cyg_http_start()` generates a simple HTTP response header containing the value of `CYGDAT_HTTPD_SERVER_ID` in the "Server" field, and the values of *`content_type`* and *`content_length`* in the "Content-type" and "Content-length" field respectively. The function `cyg_http_finish()` just adds an extra newline to the end of the output and then flushes it to force the data out to the client.

The macro `html_begin()` generates an HTTP header with a "text/html" content type followed by an opening "<html>" tag. `html_end()` generates a closing "</html>" tag and calls `cyg_http_finish()`.

General HTML Support

```
void cyg_html_tag_begin( FILE *client, char *tag, char *attr );
void cyg_html_tag_end( FILE *client, char *tag );
#define html_tag_begin( __client, __tag, __attr )
#define html_tag_end( __client, __tag )
#define html_head( __client, __title, __meta )
#define html_body_begin( __client, __attr )
#define html_body_end( __client )
#define html_heading( __client, __level, __heading )
#define html_para_begin( __client, __attr )
#define html_url( __client, __text, __link )
#define html_image( __client, __source, __alt, __attr )
```

The function `cyg_html_tag_begin()` generates an opening tag with the given name. The function `cyg_html_tag_end()` generates a closing tag with the given name. The macros `html_tag_begin()` and `html_tag_end` are just wrappers for these functions.

The macro `html_head()` generates an HTML header section with *`__title`* as the title. The *`__meta`* argument defines any meta tags that will be inserted into the header. `html_body_begin()` and `html_body_end` generate HTML body begin and end tags.

`html_heading()` generates a complete HTML header where *`__level`* is a numerical level, between 1 and 6, and *`__heading`* is the heading text. `html_para_begin()` generates a paragraph break.

`html_url()` inserts a URL where `__text` is the displayed text and `__link` is the URL of the linked page. `html_image()` inserts an image tag where `__source` is the URL of the image to be included and `__alt` is the alternative text for when the image is not displayed.

Table Support

```
#define html_table_begin( __client, __attr )
#define html_table_end( __client )
#define html_table_header( __client, __content, __attr )
#define html_table_row_begin( __client, __attr )
#define html_table_row_end( __client )
#define html_table_data_begin( __client, __attr )
#define html_table_data_end( __client )
```

`html_table_begin()` starts a table and `html_table_end()` end it. `html_table_header()` generates a simple table column header containg the string `__content`.

`html_table_row_begin()` and `html_table_row_end()` begin and end a table row, and similarly `html_table_data_begin()` and `html_table_data_end()` begin and end a table entry.

Forms Support

```
#define html_form_begin( __client, __url, __attr )
#define html_form_end( __client )
#define html_form_input( __client, __type, __name, __value, __attr )
#define html_form_input_radio( __client, __name, __value, __checked )
#define html_form_input_checkbox( __client, __name, __value, __checked )
#define html_form_input_hidden( __client, __name, __value )
#define html_form_select_begin( __client, __name, __attr )
#define html_form_option( __client, __value, __label, __selected )
#define html_form_select_end( __client )
void cyg_formdata_parse( char *data, char *list[], int size );
char *cyg_formlist_find( char *list[], char *name );
```

`html_form_begin()` begins a form, the `__url` argument is the value for the `action` attribute. `html_form_end()` ends the form.

`html_form_input()` defines a general form input element with the given type, name and value. `html_form_input_radio` creates a radio button with the given name and value; the `__checked` argument is a boolean expression that is used to determine whether the `checked` attribute is added to the tag. Similarly `html_form_input_checkbox()` defines a checkbox element. `html_form_input_hidden()` defines a hidden form element with the given name and value.

`html_form_select_begin()` begins a multiple choice menu with the given name. `html_form_select_end()` end it. `html_form_option()` defines a menu entry with the given value and label; the `__selected` argument is a boolean expression controlling whether the `selected` attribute is added to the tag.

`cyg_formdata_parse()` converts a form response string into an `NULL`-terminated array of "name=value" entries. The `data` argument is the string as passed to the handler function; note that this string is not copied and will be updated in place to form the list entries. `list` is a pointer to an array of character pointers, and is `size` elements long. `cyg_formlist_find()` searches a list generated by `cyg_formdata_parse()` and returns a pointer to the value part of the string whose name part matches `name`; if there is no match it will return `NULL`.

Predefined Handlers

```
cyg_bool cyg_httpd_send_html( FILE *client, char *filename, char *request, void *arg );

typedef struct
{
    char        *content_type;
    cyg_uint32  content_length;
    cyg_uint8   *data;
} cyg_httpd_data;
#define CYG_HTTPD_DATA( __name, __type, __length, __data )

cyg_bool cyg_httpd_send_data( FILE *client, char *filename, char *request, void *arg );
```

The HTTP server defines a couple of predefined handers to make it easier to deliver simple, static content.

`cyg_httpd_send_html()` takes a `NULL`-terminated string as the argument and sends it to the client with an HTTP header indicating that it is HTML. The following is an example of its use:

```
char cyg_html_message[] = "<head><title>Welcome</title></head>\n"
                          "<body><h2>Welcome to my Web Page</h2></body>\n"

CYG_HTTPD_TABLE_ENTRY( cyg_html_message_entry,
                       "/message.html",
                       cyg_httpd_send_html,
                       cyg_html_message );
```

`cyg_httpd_send_data()` Sends arbitrary data to the client. The argument is a pointer to a cyg_httpd_data structure that defines the content type and length of the data, and a pointer to the data itself. The `CYG_HTTPD_DATA()` macro automates the definition of the structure. Here is a typical example of its use:

```
static cyg_uint8 ecos_logo_gif[] = {
    …
};

CYG_HTTPD_DATA( cyg_monitor_ecos_logo_data,
                "image/gif",
                sizeof(ecos_logo_gif),
                ecos_logo_gif );

CYG_HTTPD_TABLE_ENTRY( cyg_monitor_ecos_logo,
                       "/monitor/ecos.gif",
                       cyg_httpd_send_data,
                       &cyg_monitor_ecos_logo_data );
```

System Monitor

Included in the HTTPD package is a simple System Monitor that is intended to act as a test and an example of how to produce servers. It is also hoped that it might be of some use in and of itself.

The System Monitor is intended to work in the background of any application. Adding the network stack and the HTTPD package to any configuration will enable the monitor by default. It may be disabled by disabling the `CYGPKG_HTTPD_MONITOR` option.

The monitor is intended to be simple and self-explanatory in use. It consists of four main pages. The thread monitor page presents a table of all current threads showing such things as id, state, priority, name and stack dimensions. Clicking on the thread ID will link to a thread edit page where the thread's state and priority may be manipulated. The interrupt monitor just shows a table of the current interrupts and indicates which are active. The memory monitor shows a 256 byte page of memory, with controls to change the base address and display element size. Note: Accessing invalid memory locations can cause memory exceptions and the program to crash. The network monitor page shows information extracted from the active network interfaces and protocols. Finally, if kernel instrumentation is enabled, the instrumentation page provides some controls over the instrumentation mechanism, and displays the instrumentation buffer.

Part LIII. SNMP

Table of Contents

Chapter 186. SNMP for *eCos*

Version

This is a port of UCD-SNMP-4.1.2

Originally this document said: See http://ucd-snmp.ucdavis.edu/ for details. And send them a postcard.

The project has since been renamed “net-snmp” and re-homed at http://net-snmp.sourceforge.net/ where various new releases (of the original, not *eCos* ports) are available.

The original source base from which we worked to create the *eCos* port is available from various archive sites such as ftp://ftp.freesnmp.com/mirrors/net-snmp/ generally with this filename and details:

```
        ucd-snmp-4.1.2.tar.gz. . . . . . Nov  2  2000   1164k
```

SNMP packages in the *eCos* source repository

The SNMP/eCos package consists of two eCos packages; the SNMP library and the SNMP agent.

The sources are arranged this way partly for consistency with the original release from UCD, and so as to accommodate possible future use of the SNMP library without having an agent present. That could be used to build an eCos-based SNMP client application.

The library contains support code for talking SNMP over the net - the SNMP protocol itself - and a MIB file parser (ASN-1) which is not used in the agent case.

The agent contains the application specific handler files to get information about the system into the SNMP world, together with the SNMP agent thread (`snmpd` in `UNIX` terms).

MIBs supported

The standard set in MIB-II, together with the Ether-Like MIB, are supported by default. The MIB files used to compile the handlers in the agent and to “drive” the testing (**snmpwalk** *et al* under `LINUX`) are those acquired from that same UCD distribution.

These are the supported MIBs; all are below mib2 == 1.3.6.1.2.1:

```
system        { mib2 1 }
interfaces    { mib2 2 }
              [ address-translation “at” { mib2 3 } is deprecated ]
ip            { mib2 4 }
icmp          { mib2 5 }
tcp           { mib2 6 }
udp           { mib2 7 }
              [ exterior gateway protocol “egp” { mib2 8 } not supported ]
              [ cmot { mib2 9 } is “historic”, just a placeholder ]
dot3          { mib2 10 7 } == { transmission 7 } “EtherLike MIB”
snmp          { mib2 11 }
```

On inclusion of SNMPv3 support packages, the following MIBs are added to the default set of MIBs enumerated above :

```
snmpEngine    { snmpFrameworkMIBObjects 1 }  SNMP-FRAMEWORK-MIB, as described in
                                             RFC-2571 for support of SNMPv3
                                             framework.

usmStats      {           usmMIBObjects 1 }  SNMP-USER-BASED-SM-MIB, as
usmUser       {           usmMIBObjects 2 }  specified in RFC-2574 for support
                                             of user based security model in
                                             SNMPv3 management domains.
```

Note

Not every MIB variable is necessarily supported - some don't really apply to eCos, some are simply not yet implemented, and some would be overly complex to implement to be worth it in an embedded system. Similarly writing to some variables may be permitted by the MIB definition, but may not produce any effect. For example trying to set an interface administratively up or down with IF-MIB::ifAdminStatus at present has no effect.

Changes to eCos sources

Small changes were made in three areas to accomodate SNMP originally:

1. Various hardware-specific ethernet drivers.
2. The generic ethernet device driver.
3. The OpenBSD TCP/IP networking package (Deprecated and removed from current eCosPro releases).

These changes were made in order to export information about the driver and the network that the SNMP agent must report. The changes were trivial in the case of the network stack, since it was already SNMP-friendly. The generic ethernet device driver was re-organized to have an extensive header file and to add a couple of APIs to extract statistics that the hardware-specific device drivers keep within themselves.

There may be a performance hit for recording that data; disabling a config option named something like `CYGDBG_DEVS_ETH_xxxx_xxxx_KEEP_STATISTICS` depending on the specific device driver will prevent that.

Not all platform ethernet device drivers export complete SNMP statistical information; if the exported information is missing, SNMP will report zero values for such data (in the dot3 MIB).

The interface chipset has an ID which is an OID; not all the latest greatest devices are listed in the available database, so new chipsets may need to be added to the client MIB, if not defined in those from UCD.

Starting the SNMP Agent

A routine to instantiate and start the SNMP agent thread in the default configuration is provided in `PACKAGES/net/snmp/agent/current/src/snmptask.c`

It starts the snmpd thread at priority `CYGPKG_NET_THREAD_PRIORITY+1` by default, ie. one step less important than the TCP/IP stack service thread. It also statically creates and uses a very large stack of around 100 KiloBytes. To use that convenience function, this code fragment may be copied (in plain C).

```
#ifdef CYGPKG_SNMPAGENT
{
                extern void cyg_net_snmp_init(void);
                cyg_net_snmp_init();

}
#endif
```

In case you need to perform initialization, for example setting up SNMPv3 security features, when the snmp agent starts and every time it restarts, you can register a callback function by simply writing the global variable:

```
externC void (*snmpd_reinit_function)( void );
```

with a suitable function pointer.

The entry point to the SNMP agent is:

```
externC void snmpd( void (*initfunc)( void ) );
```

so you can of course easily start it in a thread of your choice at another priority instead if required, after performing whatever other initialization your SNMP MIBs need. A larger than default stacksize is required. The `initfunc` parameter is the callback function mentioned above — a NULL parameter there is safe and obviously means no callback is registered.

Note that if you call `snmpd();` yourself and do *not* call `cyg_net_snmp_init();` then that routine, global variable, and the default large stack will not be used. This is the recommended way control such features from your application; create and start the thread yourself at the appropriate moment.

Other APIs from the `snmpd` module are available, specifically:

```
void SnmpdShutDown(int a);
```

which causes the `snmpd` to restart itself — including the callback to your init function — as soon as possible.

The parameter *a* is ignored. It is there because in `snmpd`'s “natural environment” this routine is a `UNIX` signal handler.

The helper functions in the network stack for managing `DHCP` leases will call `SnmpdShutDown()` when necessary, for example if network interfaces go down and/or come up again.

Configuring eCos

To use the SNMP agent, the SNMP library and agent packages must be included in your configuration. To incorporate the stack into your configuration select the SNMP library and SNMP agent packages in the eCos Configuration Tool, or at the command line type:

```
$ ecosconfig add snmplib snmpagent
```

After adding the networking, common ethernet device drivers, snmp library and snmp agent packages, there is no configuration required. However there are a number of configuration options that can be set such as some details for the System MIB, and disabling SNMPv3 support (see below).

Starting the SNMP agent is not integrated into network tests other than `snmpping` below, nor is it started automatically in normal eCos startup - it is up to the application to start the agent when it is ready, at least after the network interfaces are both ’up,.

Version usage (v1, v2 or v3)

The default build supports all three versions of the SNMP protocol, but without any dispatcher functionality (rfc 2571, section 3.1.1.2). This has the following implications :

1. There is no community authentication for v1 and v2c.

2. Security provided by v3 can be bypassed by using v1/v2c protocol.

To provide the dispatcher with rfc 2571 type functionality, it is required to set up security models and access profiles. This can be provided in the normal Unix style by writing the required configurations in `snmpd.conf` file. Application code may setup profiles in `snmpd.conf` and optionally set the environment variable `SNMPCONFPATH` to point to the file if it is not in the usual location. The whole concept works in the usual way as with the standard UCD-SNMP distribution.

Traps

The support of the `trapsink` command in the snmpd.conf file is not tested and there may be problems for it working as expected. Moreover, in systems that do not have filesystem support, there is no way to configure a trap-session in the conventional way.

For reasons mentioned above, applications need to initialize their own trap sessions and pass it the details of trap-sink. The following is a small sample for initializing a v1 trap session :

```
typedef struct trap {
        unsigned char ip [4];
        unsigned int  port;
        unsigned char community [256];
}

trap            trapsink;
unsinged char   sink [16];

…
…

if (trapsink.ip != 0) {
        sprintf (sink, "%d.%d.%d.%d",
                 trapsink[0], trapsink[1], trapsink[2], trapsink[3]);
        if (create_trap_session (sink,
                trapsink.port,
                (char *)trapsink.community,
                SNMP_VERSION_1,
                SNMP_MSG_TRAP) == 0) {
                log_error ("Creation of trap session failed \n");
        }
}
```

`snmpd.conf` file

Using snmpd.conf requires the inclusion of one of the file-system packages (eg. CYGPKG_RAMFS) and CYGPKG_FILEIO. With these two packages included, the SNMP sub-system will read the snmpd.conf file from the location specified in `SNMPCONFPATH`, or the standard builtin locations, and use these profiles. Only the profiles specified in the `ACCESS-CONTROL` section of snmpd.conf file have been tested and shown to work. Other profiles which have been implemented in `UCD-SNMP-4.1.2`'s `snmpd.conf` may not work because the sole purpose of adding support for the snmpd.conf file has been to set up `ACCESS-CONTROL` models.

At startup, the SNMP module tries to look for file `snmp.conf`. If this file is not available, the module successively looks for files `snmpd.conf`, `snmp.local.conf` and `snmpd.local.conf` at the locations pointed to by `SNMPCONFPATH` environment variable. In case `SNMPCONFPATH` is not defined, the search sequence is carried out in default directories. The default directories are :`/usr/share/snmp`, `/usr/local/share/snmp` and `$(HOME)/.snmp`. The configurations read from these files are used to control both, SNMP applications and the SNMP agent; in the usual UNIX fashion.

The inclusion of snmpd.conf support is enabled by default when suitable filesystems and FILEIO packages are active.

Test cases

Currently only one test program is provided which uses SNMP.

"snmpping" in the SNMP agent package runs the ping test from the TCPIP package, with the snmpd running also. This allows you to interrogate it using host tools of your choice. It supports MIBs as documented above, so eg. **snmpwalk <*`hostname`*> public dot3** under Linux/UNIX should have the desired effect.

For serious testing, you should increase the length of time the test runs by setting CYGNUM_SNMPAGENT_TESTS_ITERATIONS to something big (e.g., 999999). Build the test (**make -C net/snmp/agent/current tests**) and run it on the target.

Then start several jobs, some for pinging the board (to make the stats change) and some for interrogating the snmpd. Set $IP to whatever IP address the board has:

```
# in a root shell, for flood ping
while(1)
date
ping -f -c 3001 $IP
sleep 5
ping -c 32 -s 2345 $IP
end
```

```
# have more than one of these going at once
```

```
setenv MIBS all
while (1)
snmpwalk -OS $IP  public
date
end
```

Leave to run for a couple of days or so to test stability.

The test program can also test snmpd.conf support. It tries to build a minimal snmpd.conf file on a RAM filesystem and passes it to the snmp sub-system. With this profile on target, the following snmp[cmd] (cmd=walk, get, set) should work :

```
snmp[cmd] -v1 $IP crux $OID
snmp[cmd] -v2 $IP crux $OID
snmp[cmd] -v3 $IP -u root -L noAuthNoPriv $OID
snmp[cmd] -v3 $IP -u root -L authNoPriv -A MD5 -a md5passwd $OID
```

The following commands would however fail since they violate the access model :

```
snmp[cmd] $IP public $OID
snmp[cmd] -v1 $IP public $OID
snmp[cmd] -v2c $IP public $OID
snmp[cmd] -v3 $IP -u no_user -L noAuthNoPriv $OID
snmp[cmd] -v3 $IP -u root -L authNoPriv -A MD5 -a badpasswd $OID
```

SNMP clients and package use

SNMP clients may use these packages, but this usage is currently untested: the reason why this port to eCos exists is to acquire the SNMP agent. The fact that that the SNMP API (for clients) exists is a side-effect. See the standard man page SNMP_API(3) for details. There are further caveats below about client-side use of the SNMP library.

All of the SNMP header files are installed beneath .../include/ucd-snmp in the install tree. The SNMP code itself assumes that directory is on its include path, so we recommend that client code does the same. Further, like the TCP/IP stack, compiling SNMP code requires definition of _KERNEL and __ECOS, and additionally IN_UCD_SNMP_SOURCE.

Therefore, add all of these to your compile lines if you wish to include SNMP header files:

```
-D_KERNEL
-D__ECOS
-DIN_UCD_SNMP_SOURCE=1
-I$(PREFIX)/include/ucd-snmp
```

Unimplemented features

Currently, the filesystem and persistent storage areas are left undone, to be implemented by the application.

The SNMP library package is intended to support client and agent code alike. It therefore contains lots of assumptions about the presence of persistent storage ie. a filesystem. Currently, by default, eCos has no such thing, so those areas have been simply commented out and made to return empty lists or say “no data here.”

Specifically the following files have omitted/unimplemented code :

`PACKAGES/net/snmp/lib/current/src/parse.c`

contains code to enumerate MIB files discovered in the system MIB directories (“`/usr/share/snmp/mibs`”), and read them all in, building data structures that are used by client programs to interrogate an agent. This is not required in an agent, so the routine which enumerates the directories returns an empty list.

`PACKAGES/net/snmp/lib/current/src/read_config.c` contains two systems:

The first tries to read the configuration file as described in the snmpd.conf file section and the second system contains code to record persistent data as files in a directory (typically `/var/ucd-snmp`) thus preserving the state permanently.

The first part is partially implemented to support multiple profiles and enables dispatcher functionality as discussed in the section called "Version usage (v1, v2 or v3)". The second part is not supported at all in the default implementation. As required, a cleaner interface to permit application code to manage persistent data will be developed in consultation with customers.

MIB Compiler

In the directory `/snmp/agent/current/utils/mib2c`, there are the following files:

```
README-eCos              notes about running with a nonstandard
                         perl path.
README.mib2c             the README from UCD; full instructions on
                         using mib2c
mib2c                    the perl program
mib2c.conf               a configuration file altered to include the
                         eCos/UCD
mib2c.conf-ORIG          copyright and better #include paths; and
                         the ORIGinal.
mib2c.storage.conf       other config files, not modified.
mib2c.vartypes.conf
```

mib2c is provided BUT it requires the SNMP perl package SNMP-3.1.0, and that in turn requires perl nsPerl5.005_03 (part of Red Hat Linux from 6.0, April 1999).

These are available from the CPAN ("the Comprehensive Perl Archive Network") as usual; http://www.cpan.org/ and links from there. Specifically:

- PERL itself: http://people.netscape.com/kristian/nsPerl/
- http://people.netscape.com/richm/nsPerl/nsPerl5.005_03-11-i686-linux.tar.gz
- SNMP.pl http://www.cpan.org/modules/01modules.index.html
- http://cpan.valueclick.com/modules/by-category/05_Networking_Devices_IPC/SNMP/
- http://www.cpan.org/authors/id/G/GS/GSM/SNMP.tar.gz

(note that the .tar.gz files are not browsable)

For documentation on the files produced, see the documentation available at http://ucd-snmp.ucdavis.edu/ in general, and file `AGENT.txt` in particular.

It is likely that the output of mib2c will be further customized depending on eCos customer needs; it,s easy to do this by editing the mib2c.conf file to add or remove whatever you need with the resulting C sources.

The UCD autoconf-style configuration does not apply to eCos. So if you add a completely new MIB to the agent, and support it using mib2c so that the my_new_mib.c file contains a init_my_new_mib() routine to register the MIB handler, you will also need to edit a couple of control files; these claim to be auto-generated, but in the eCos release, they,re not, don,t worry.

```
PACKAGES/net/snmp/agent/current/include/mib_module_includes.h
```

contains a number of lines like

```
#include "mibgroup/mibII/interfaces.h"
```

so add your new MIB thus:

```
#include "mibgroup/mibII/my_new_mib.h"
```

```
PACKAGES/net/snmp/agent/current/include/mib_module_inits.h
```

contains a number of lines like

```
init_interfaces();
init_dot3();
```

and so on; add your new MIB as follows:

```
init_my_new_mib();
```

and this should work correctly.

snmpd.conf

```
SNMPD.CONF(5)                                       SNMPD.CONF(5)



NAME
       share/snmp/snmpd.conf  -  configuration  file for the ucd-
       snmp SNMP agent.

DESCRIPTION
       snmpd.conf is the configuration file which defines how the
       ucd-smnp SNMP agent operates.  These files may contain any
       of the directives found in the DIRECTIVES  section  below.
       This  file  is  not  required for the agent to operate and
       report mib entries.

PLEASE READ FIRST
       First, make sure you have read the  snmp_config(5)  manual
       page  that  describes how the ucd-snmp configuration files
       operate, where they are located  and  how  they  all  work
       together.

EXTENSIBLE-MIB
       The  ucd-snmp  SNMP  agent reports much of its information
       through queries to the 1.3.6.1.4.1.2021 section of the mib
       tree.   Every  mib in this section has the following table
       entries in it.

       .1 -- index
              This is the table's index numbers for each  of  the
              DIRECTIVES listed below.

       .2 -- name
              The  name of the given table entry.  This should be
              unique, but is not required to be.

       .100 -- errorFlag
              This is a flag returning either the integer value 1
              or  0 if an error is detected for this table entry.

       .101 -- errorMsg
              This is a DISPLAY-STRING describing any error trig-
              gering the errorFlag above.

       .102 -- errorFix
              If  this entry is SNMPset to the integer value of 1
              AND the errorFlag defined above is indeed  a  1,  a
              program  or script will get executed with the table
              entry name from above as the argument.  The program
              to  be  executed is configured in the config.h file
              at compile time.

   Directives
       proc NAME

       proc NAME MAX

       proc NAME MAX MIN
```

```
              Checks to see if the NAME'd processes  are  running
              on  the  agent's  machine.  An error flag (1) and a
              description  message  are  then   passed   to   the
              1.3.6.1.4.1.2021.2.100  and  1.3.6.1.4.1.2021.2.101
              mib tables (respectively) if the NAME'd program  is
              not  found  in  the  process  table  as reported by
              "/bin/ps -e".

              If MAX and MIN are not specified, MAX is assumed to
              be infinity and MIN is assumed to be 1.

              If  MAX  is specified but MIN is not specified, MIN
              is assumed to be 0.

       procfix NAME PROG ARGS
              This registers a command  that  knows  how  to  fix
              errors   with   the   given   process  NAME.   When
              1.3.6.1.4.1.2021.2.102 for a given NAMEd program is
              set to the integer value of 1, this command will be
              called.  It defaults to a compiled value set  using
              the PROCFIXCMD definition in the config.h file.

       exec NAME PROG ARGS

       exec MIBNUM NAME PROG ARGS

              If  MIBNUM is not specified, the agent executes the
              named PROG with arguments of ARGS and  returns  the
              exit status and the first line of the STDOUT output
              of   the   PROG   program   to   queries   of   the
              1.3.6.1.4.1.2021.8.100  and  1.3.6.1.4.1.2021.8.101
              mib  tables  (respectively).   All  STDOUT   output
              beyond the first line is silently truncated.

              If  MIBNUM  is  specified,  it  acts  as  above but
              returns the exit status  to  MIBNUM.100.0  and  the
              entire  STDOUT  output to the table MIBNUM.101 in a
              mib table.  In this case, the MIBNUM.101  mib  con-
              tains the entire STDOUT output, one mib table entry
              per line of output (ie, the first line is output as
              MIBNUM.101.1,  the second at MIBNUM.101.2, etc…).

              Note:  The MIBNUM must be specified in dotted-inte-
                     ger  notation  and  can  not be specified as
                     ".iso.org.dod.internet…"  (should  instead
                     be

              Note:  The  agent caches the exit status and STDOUT
                     of the executed program for 30 seconds after
                     the  initial  query.   This  is  to increase
                     speed and maintain consistency  of  informa-
                     tion  for  consecutive  table  queries.  The
                     cache can be flushed by a  snmp-set  request
                     of  integer(1)  to 1.3.6.1.4.1.2021.100.VER-
                     CLEARCACHE.

       execfix NAME PROG ARGS
              This registers a command  that  knows  how  to  fix
              errors  with  the  given  exec  or  sh  NAME.  When
              1.3.6.1.4.1.2021.8.102 for a given NAMEd  entry  is
              set to the integer value of 1, this command will be
              called.  It defaults to a compiled value set  using
              the EXECFIXCMD definition in the config.h file.

       disk PATH

       disk PATH [ MINSPACE | MINPERCENT% ]
```

```
              Checks  the  named disks mounted at PATH for avail-
              able disk space.  If the disk space  is  less  than
              MINSPACE  (kB) if specified or less than MINPERCENT
              (%) if a  %  sign  is  specified,  or  DEFDISKMINI-
              MUMSPACE  (kB)  if  not  specified,  the associated
              entry in the 1.3.6.1.4.1.2021.9.100 mib table  will
              be  set to (1) and a descriptive error message will
              be returned to queries of 1.3.6.1.4.1.2021.9.101.

       load MAX1

       load MAX1 MAX5

       load MAX1 MAX5 MAX15

              Checks the load average of the machine and  returns
              an error flag (1), and an text-string error message
              to   queries   of    1.3.6.1.4.1.2021.10.100    and
              1.3.6.1.4.1.2021.10.101   (respectively)  when  the
              1-minute, 5-minute, or  15-minute  averages  exceed
              the associated maximum values.  If any of the MAX1,
              MAX5, or MAX15 values are unspecified, they default
              to a value of DEFMAXLOADAVE.

       file FILE [MAXSIZE]
              Monitors  file sizes and makes sure they don't grow
              beyond a certain size.  MAXSIZE defaults  to  infi-
              nite  if  not specified, and only monitors the size
              without reporting errors about it.

   Errors
       Any errors in obtaining the above information are reported
       via    the    1.3.6.1.4.1.2021.101.100    flag   and   the
       1.3.6.1.4.1.2021.101.101 text-string description.

SMUX SUB-AGENTS
       To enable and SMUX based sub-agent, such as gated, use the
       smuxpeer configuration entry

       smuxpeer OID PASS
              For gated a sensible entry might be

       .1.3.6.1.4.1.4.1.3 secret

ACCESS CONTROL
       snmpd  supports the View-Based Access Control Model (vacm)
       as defined in RFC 2275.  To this end,  it  recognizes  the
       following  keywords  in  the  configuration file: com2sec,
       group, access, and view  as  well  as  some  easier-to-use
       wrapper   directives:  rocommunity,  rwcommunity,  rouser,
       rwuser.

       rocommunity COMMUNITY [SOURCE] [OID]

       rwcommunity COMMUNITY [SOURCE] [OID]
              These create read-only and  read-write  communities
              that  can  be used to access the agent.  They are a
              quick method of using the following com2sec, group,
              access,  and view directive lines.  They are not as
              efficient either, as groups aren't created  so  the
              tables  are possibly larger.  In other words: don't
              use these if you have complex situations to set up.

              The  format  of the SOURCE is token is described in
              the com2sec directive section below.  The OID token
              restricts  access  for that community to everything
              below that given OID.

       rouser USER [noauth|auth|priv] [OID]

       rwuser USER [noauth|auth|priv] [OID]
              Creates a  SNMPv3  USM  user  in  the  VACM  access
              configuration  tables.   Again,  its more efficient
              (and powerful) to use the combined com2sec,  group,
              access, and view directives instead.

              The minimum level of authentication and privacy the
              user must use  is  specified  by  the  first  token
              (which  defaults  to  "auth").   The  OID parameter
              restricts access for that user to everything  below
              the given OID.

       com2sec NAME SOURCE COMMUNITY
              This   directive   specifies  the  mapping  from  a
              source/community pair to a  security  name.  SOURCE
              can be a hostname, a subnet, or the word "default".
              A subnet can be specified as  IP/MASK  or  IP/BITS.
              The first source/community combination that matches
              the incoming packet is selected.

       group NAME MODEL SECURITY
              This directive defines the mapping  from  security-
              model/securityname  to  group.  MODEL is one of v1,
              v2c, or usm.

       access NAME CONTEXT MODEL LEVEL PREFX READ WRITE NOTIFY
              The  access  directive  maps  from   group/security
              model/security  level  to  a view.  MODEL is one of
              any, v1, v2c, or usm.   LEVEL  is  one  of  noauth,
              auth,  or priv.  PREFX specifies how CONTEXT should
              be matched against the context of the incoming pdu,
              either  exact  or  prefix.   READ, WRITE and NOTIFY
              specifies the view to be used for the corresponding
              access.   For  v1  or  v2c  access,  LEVEL  will be
              noauth, and CONTEXT will be empty.

       view NAME TYPE SUBTREE [MASK]
              The defines the named view. TYPE is either included
              or  excluded.   MASK is a list of hex octets, sepa-
              rated by '.' or ':'.  The MASK defaults to "ff"  if
              not specified.

              The  reason  for the mask is, that it allows you to
              control access to one row in a table,  in  a  rela-
              tively  simple  way.  As  an example, as an ISP you
              might consider giving each customer access  to  his
              or her own interface:

              view cust1 included interfaces.ifTable.ifEntry.ifIndex.1 ff.a0
              view cust2 included interfaces.ifTable.ifEntry.ifIndex.2 ff.a0

              (interfaces.ifTable.ifEntry.ifIndex.1 == .1.3.6.1.2.1.2.2.1.1.1,
              ff.a0 == 11111111.10100000. which nicely covers up and including
              the row index, but lets the user vary the field of the row)

       VACM Examples:
              #       sec.name  source          community
              com2sec local     localhost       private
              com2sec mynet     10.10.10.0/24   public
              com2sec public    default         public

              #             sec.model  sec.name
              group mygroup v1         mynet
              group mygroup v2c        mynet
              group mygroup usm        mynet
              group local   v1         local
              group local   v2c        local
              group local   usm        local
              group public  v1         public
              group public  v2c        public
              group public  usm        public

              #           incl/excl subtree                          mask
              view all    included  .1                               80
              view system included  system                           fe
              view mib2   included  .iso.org.dod.internet.mgmt.mib-2 fc

              #              context sec.model sec.level prefix read   write notify
              access mygroup ""      any       noauth    exact  mib2   none  none
              access public  ""      any       noauth    exact  system none  none
              access local   ""      any       noauth    exact  all    all   all

       Default VACM model
              The default configuration of the agent, as shipped, is functionally
              equivalent to the following entries:
              com2sec   public    default   public
              group     public    v1   public
              group     public    v2c  public
              group     public    usm  public
              view      all  included  .1
              access    public    ""   any  noauth    exact     all  none none

SNMPv3 CONFIGURATION
       engineID STRING
              The  snmpd  agent  needs  to  be configured with an
              engineID to be able to respond to SNMPv3  messages.
              With  this  configuration  file  line, the engineID
              will be configured from STRING.  The default  value
              of  the  engineID  is  configured with the first IP
              address found for the hostname of the machine.

       createUser username (MD5|SHA) authpassphrase [DES]  [priv-
       passphrase]
              This directive should be placed into the "/var/ucd-
              snmp"/snmpd.conf  file  instead of the other normal
              locations.  The reason is that the  information  is
              read  from  the  file  and then the line is removed
              (eliminating the storage of the master password for
              that  user)  and  replaced  with  the  key  that is
              derived from it.  This key is a localized  key,  so
              that  if  it is stolen it can not be used to access
              other agents.  If the password is stolen,  however,
              it can be.

              MD5  and  SHA  are the authentication types to use,
              but you must have built the  package  with  openssl
              installed  in  order  to use SHA.  The only privacy
              protocol currently supported is DES.  If  the  pri-
              vacy  passphrase is not specified, it is assumed to
              be the same as the authentication passphrase.  Note
              that  the users created will be useless unless they
              are also added to the VACM  access  control  tables
              described above.

              Warning:  the minimum pass phrase length is 8 char-
              acters.

              SNMPv3 users can be created at  runtime  using  the
              snmpusm command.
```

```
SETTING SYSTEM INFORMATION
       syslocation STRING

       syscontact STRING

              Sets the system location and the system contact for
              the agent.  This information  is  reported  by  the
              'system' table in the mibII tree.

       authtrapenable NUMBER
              Setting  authtrapenable  to 1 enables generation of
              authentication failure traps. The default value  is
              2 (disable).

       trapcommunity STRING
              This  defines  the  default  community string to be
              used when sending traps.  Note  that  this  command
              must  be  used  prior to any of the following three
              commands  that  are  intended  use  this  community
              string.

       trapsink HOST [COMMUNITY [PORT]]

       trap2sink HOST [COMMUNITY [PORT]]

       informsink HOST [COMMUNITY [PORT]]
              These  commands  define  the hosts to receive traps
              (and/or inform notifications). The daemon  sends  a
              Cold  Start  trap when it starts up. If enabled, it
              also sends traps on authentication failures.   Mul-
              tiple  trapsink, trap2sink and informsink lines may
              be specified to specify multiple destinations.  Use
              trap2sink  to  send  SNMPv2 traps and informsink to
              send inform notifications.   If  COMMUNITY  is  not
              specified,  the  string from a preceding trapcommu-
              nity directive will be used. If PORT is not  speci-
              fied,  the  well known SNMP trap port (162) will be
              used.

PASS-THROUGH CONTROL
       pass MIBOID EXEC
              Passes entire control of MIBOID to  the  EXEC  pro-
              gram.   The  EXEC  program  is called in one of the
              following three ways:

              EXEC -g MIBOID

              EXEC -n MIBOID

                     These call lines match to SNMP get and  get-
                     next requests.  It is expected that the EXEC
                     program will take the arguments passed to it
                     and  return the appropriate response through
                     it's stdout.

                     The first line of stdout should be  the  mib
                     OID of the returning value.  The second line
                     should be the TYPE of value returned,  where
                     TYPE  is  one  of  the text strings: string,
                     integer,  unsigned,   objectid,   timeticks,
                     ipaddress,  counter,  or  gauge.   The third
                     line of stdout should be  the  VALUE  corre-
                     sponding with the returned TYPE.

                     For  instance, if a script was to return the
                     value integer value "42" when a request  for
                     .1.3.6.1.4.100  was  requested,  the  script
                     should return the following 3 lines:
                       .1.3.6.1.4.100
                       integer
                       42

                     To indicate that the  script  is  unable  to
                     comply with the request due to an end-of-mib
                     condition or an invalid request, simple exit
                     and  return  no  output to stdout at all.  A
                     snmp error will be  generated  corresponding
                     to the SNMP NO-SUCH-NAME response.

              EXEC -s MIBOID TYPE VALUE

                     For SNMP set requests, the above call method
                     is used.  The TYPE passed to the  EXEC  pro-
                     gram  is  one  of the text strings: integer,
                     counter, gauge, timeticks, ipaddress, objid,
                     or  string,  indicating  the  type  of value
                     passed in the next argument.

                     Return nothing to stdout, and the  set  will
                     assumed to have been successful.  Otherwise,
                     return one of the following error strings to
                     signal an error: not-writable, or wrong-type
                     and the appropriate error response  will  be
                     generated instead.

                      Note:  By   default,   the  only  community
                             allowed to  write  (ie  snmpset)  to
                             your  script  will  be the "private"
                             community,or community #2 if defined
                             differently by the "community" token
                             discussed above.  Which  communities
                             are  allowed  write  access are con-
                             trolled by the RWRITE definition  in
                             the snmplib/snmp_impl.h source file.

EXAMPLE
       See the EXAMPLE.CONF file in the top level  source  direc-
       tory for a more detailed example of how the above informa-
       tion is used in real examples.

RE-READING snmpd.conf and snmpd.local.conf
       The ucd-snmp agent can be forced to re-read its configura-
       tion files.  It can be told to do so by one of two ways:

       1.     An       snmpset       of       integer(1)       to
              1.3.6.1.4.1.2021.100.VERUPDATECONFIG.

       2.     A "kill -HUP" signal sent to the snmpd  agent  pro-
              cess.

FILES
       share/snmp/snmpd.conf

SEE ALSO
       snmp_config(5), snmpd(1), EXAMPLE.conf, read_config(3).



                           27 Jan 2000              SNMPD.CONF(5)
```

Part LIV. mDNS Responder and DNS-SD

Important

This eCosPro-mDNS Middleware package is STRICTLY LICENSED FOR NON-COMMERCIAL PURPOSES ONLY. It may not be used for Commercial purposes in full or in part in any format, including source code, binary code and object code format.

A Commercial eCosPro License version 3 (or above) which explicity includes this Middleware Package is required for Commercial use.

Table of Contents

Chapter 187. mDNS overview

Introduction

eCosPro-mDNS is eCosCentric's commercial name for the `CYGPKG_NET_MDNS` package. The `CYGPKG_NET_MDNS` package implements a small, lightweight, link-local scope mDNS Responder designed to provide Service Discovery (DNS-SD) support for services announced to the network. It optionally provides a DNS-SD API to allow network services to be discovered by the client application. The current mDNS implementation makes use of the lwIP TCP/IP stack.

mDNS Responder Features:

- Probing and Announcement of the local hostname and of registered services.

  These are repeated as necessary on cable change, mDNS naming conflict, and so on.

- Configuration of local DNS records, and responding to external mDNS queries for the local hostname and registered services.

  ### Note

  Only the class IN "the Internet" is supported.

- Support for both IPv4 and IPv6 addressing.

  ### Note

  Only link-local scope.

The goal of the mDNS Responder support is to have as small a dynamic memory footprint as possible, whilst still being flexible enough for a variety of real-world scenarios. The mDNS Responder currently has a simple single-packet probe/announce scheme. This limits the number of services that can be registered depending on the lwIP heap available and the limitations that lwIP imposes on individual UDP packet transmissions. It is important that the mDNS configuration matches the client application expectations as regards number of services, and the resources available (and configured) for the target platform.

DNS-SD Features:

- Simple, low resource footprint, callback mechanism for Resource Record processing.
- Ability to generate network queries for specific Resource Record types, to trigger responses.

Similar to the goal for the mDNS Responder, the DNS-SD API allows for clients to be coded with as small a dynamic memory footprint as possible, whilst not limiting the possibilities for resource-rich applications.

The mDNS implementation is based on the Multicast DNS RFC6762 and DNS-Based Service Discovery RFC6763 standards.

These in turn reference many other RFCs, for example RFC2782 for DNS SRV records, and RFC1035 covering DNS TXT entries.

The MCASTDNS section 16 "Multicast DNS Character Set" explicitly states that all names in mDNS MUST be encoded as precomposed UTF-8 [RFC3629] "Net-Unicode" [RFC5198] text. When an application provides strings to the mDNS Responder it should ensure that they conform to the standard naming requirements.

A good introduction to the features and benefits of Bonjour/Zeroconf can be found in the O'Reilly book "Zero Configuration Networking - The Definitive Guide" written by Stuart Chesire and Daniel H. Steinberg.

Chapter 188. API

The main mDNS API provides a serialisation layer between the low-level (lwIP) networking based operations and client application threads.

API

Name

cyg_mdns_init — Initialise mDNS Responder

Synopsis

#include <mdns.h>

```
cyg_bool cyg_mdns_init();
```

Description

This initialises the common mDNS support. It will normally be called from the application shortly after lwIP initialisation, and performs the basic "start-of-day" initialisation.

Note

All of the required lwIP network interfaces should have been added (normally via a call to the `init_all_network_interfaces()`) prior to initialising the mDNS Responder. If network interfaces are subsequently removed then the mDNS daemon should be disabled and re-enabled with the functions `cyg_mdns_disable()` and `cyg_mdns_enable()` to ensure the correct interface configurations are used. If network interfaces are subsequently added, it is sufficient to call solely `cyg_mdns_enable()` so that it configures the new interface for mDNS requests.

Once initialised, it is possible to register services with `cyg_mdns_service_register()`, and then allow the mDNS responder to respond to incoming requests with `cyg_mdns_enable()`.

Return value

Boolean `true` if the mDNS sub-system has initialised OK, or `false` on failure.

Name

cyg_mdns_terminate — Terminate mDNS Responder

Synopsis

#include <mdns.h>

```
void cyg_mdns_terminate();
```

Description

Provides clean soft-shutdown of the mDNS Responder. In reality most deeply embedded systems will never actually need to call this routine, since they will be CPU-reset-restart systems.

This call will unregister all attached service vectors before disabling enabled interfaces. So any leaving (TTL=0) announcements will be transmitted and IGMP/MLD filtering will be removed prior to disabling the mDNS packet reception.

Name

cyg_mdns_enable — Enable mDNS

Synopsis

#include <mdns.h>

```
cyg_bool cyg_mdns_enable();
```

Description

Start the mDNS Responder listening on all of the network interfaces. This function enables the listener “daemon” to respond to mDNS queries. Usually, this call does nothing if the mDNS Responder is already enabled, except in the case where new network interfaces have been added, in which case this function is called in order to receive mDNS requests on the new interfaces.

Return value

Boolean `true` if mDNS successfully initialised. On error `false` is returned.

Name

cyg_mdns_disable — Disable mDNS

Synopsis

#include <mdns.h>

```
cyg_bool cyg_mdns_disable();
```

Description

This function stops the mDNS Responder from listening on the network interfaces. This function does nothing if the mDNS Responder is not enabled.

Return value

Boolean `true` if mDNS successfully disabled. On error `false` is returned.

Name

cyg_mdns_service_register — Register set of services

Synopsis

#include <mdns.h>

cyg_bool **cyg_mdns_service_register**(*count*, *vector*);

Description

This function registers the set of service(s) in the supplied *vector* array. *count* gives the number of vector entries. If any of the referenced services were already registered for a different service then an error is returned and none of the services in the supplied *vector* are registered. If there are no free (CDL CYGNUM_MDNS_NUM_SERVICE_VECTORS controlled) vector slots available then an error is returned.

A unique 3-tuple will be used to identify services registered. The tuple consists of the UTF-8 *service* and *proto* strings, plus the 16-bit service *port* number. A simple descriptor structure is defined to hold these values, along with the “Weight” and “Priority” values (which are not considered part of the unique identification). It is the responsibility of the caller to ensure the lifetime of the string objects referenced, and that they are immutable when registered with the mDNS Responder; similarly that the passed service descriptors in *vector* are immutable.

If all of the *vector* tuples match an already registered *vector* then the held pointer is updated. This allows the application to supply new “Priority” and “Weight” values.

Or to put it another way, you can update the “Priority” and “Weight” values for a service *only* if you pass in a vector that replaces all the same services as the vector that was used to previously register that service.

Note

Furthermore, since the API requires that the registered *vector* is immutable this updating of “Weight” and “Priority” can only happen when the supplied *vector* pointer is different. This ensures that the application client code is not tempted to update the registered services data (which should be immutable) under-the-feet of the mDNS world.

We do not need to return a handle when registering services since the service tuple is used to uniquely identify each service.

Normally, for simple systems, the application will just have a single cyg_mdns_service_register() call with the fixed, hardwired, vector of service descriptors to be announced for the application.

The cyg_mdns_service structure is used to define a service tuple. It is the responsibility of the client application to ensure valid data is passed to the mDNS sub-system.

```
typedef struct cyg_mdns_service {
  const struct cyg_mdns_service_identity *id;
  const cyg_uint8 *default_label;
  const cyg_mdns_txt_record *txt_vector;
  cyg_uint16 priority;
  cyg_uint16 weight;
  cyg_uint8 txt_count;
} cyg_mdns_service;
```

The service descriptor *id* field is a pointer to a cyg_mdns_service_identity structure descriptor.

```
typedef struct cyg_mdns_service_identity {
  const cyg_uint8 *service;
  const cyg_uint8 *proto;
  cyg_uint16 port;
} cyg_mdns_service_identity;
```

The *service* and *proto* fields are pointers to a vector containing the 1-byte label length followed by 1..`MDNS_MAX_LABEL` UTF-8 characters. The client *must* ensure that the labels are prefixed with the '_' character. Defining these vectors using constant literal strings in C is unlikely to be appropriate as strings will include an additional NUL terminator character.

Note

For simplicity, and to avoid having to maintain an internal database, it is the responsibility of the client application to ensure the correct service and protocol label assignments.

The mDNS system exports the `cyg_mdns_label_tcp` and `cyg_mdns_label_udp` labels since they are common and are likely to be needed by most applications. They also avoid the cost associated with the developer having to provide duplicate string definitions in the application when defining their service vectors. In fact, for DNS-SD conforming worlds, only the “_tcp” and “_udp” protocol specifiers should be used.

If the *txt_vector* pointer is NULL then whenever a TXT field is required then a single 1-byte NUL character TXT response will be provided. If it is non-NULL then the *txt_count* field specifies the number of `cyg_mdns_txt_record` TXT table entries provided.

```
typedef struct cyg_mdns_txt_record {
  const cyg_uint8 *table;
  cyg_uint8 len;
} cyg_mdns_txt_record;
```

The *table* field of each TXT descriptor references a sequence of 1-byte length followed by 1…`MDNS_MAX_LABEL` UTF-8 character strings. The *len* field specifies the overall individual table length. The mDNS standard does NOT require the table to be terminated by a zero-length (`NUL`) indicator, since the TXT resource record contains the length. The following is an incomplete service example, but does show the provision of two TXT records (though, in reality, this simple example would be merged into a single TXT entry). Due to the use of the type `cyg_uint8` for the *len* field individual TXT tables are limited to a maximum of 255-bytes in length. Care must also be taken if registering services with multiple, large, TXT tables that the resulting data will fit within the memory available to the mDNS packet transmission code.

```
static const cyg_uint8 txttab_example_1[] = {
  0x09,'t','x','t','v','e','r','s','=','1',
  0x0B,'O','p','t','i','o','n','s','=','o','n','e'
};

static const cyg_uint8 txttab_example_2[] = {
  0x09,'F','e','a','t','u','r','e','O','n'
};

static const cyg_mdns_txt_record txt_records[] = {
  {
    .table = txttab_example_1,
    .len = sizeof(txttab_example_1)
  },
  {
    .table = txttab_example_2,
    .len = sizeof(txttab_example_2)
  }
};

static const cyg_mdns_service services[] = {
  {
    .txt_vector = txt_records,
    .txt_count = (sizeof(txt_records) / sizeof(txt_records[0])),
    // ... other fields as required
  }
};
```

Return value

Boolean `true` on success, and `false` on error.

Name

cyg_mdns_service_unregister — Remove registered services

Synopsis

#include <mdns.h>

`cyg_bool` **`cyg_mdns_service_unregister`**`(`*`count`*`, `*`vector`*`);`

Description

This function removes the services listed in *`vector`* from the set of registered services. *`vector`* must be a service vector previously passed to `cyg_mdns_service_register()`. An error is returned if the *`vector`* does not match a complete held vector. In other words, when a list of services is registered by a vector, they may only be de-registered together with that same vector.

In reality (due to the API restriction on immutable “vector”s being registered), the *`vector`* and *`count`* parameters are validated by checking that the *`vector`* matches a previously registered vector, and *`count`* matches the count of vectors previously passed when that vector was registered.

If the vector is indeed valid, then mDNS announcements are sent immediately to advertise to attached networks that the service has been removed (Resource Records with TTL of 0).

Return value

Boolean `true` on success, and `false` on error.

Name

cyg_mdns_sethostname — Set base hostname

Synopsis

#include <mdns.h>

```
cyg_bool cyg_mdns_sethostname(hostname);
```

Description

Request that the given NUL terminated UTF-8 string be used as the device hostname. This is only a request and depending on the external network devices the name may not be available. The hostname callback function interface provides, if required, application control over the alternative naming strategy when a hostname is not available, or becomes (due to external network changes) unavailable.

Return value

Boolean `true` on success, and `false` on error.

Name

cyg_mdns_hostname_callback_register — Register hostname generation callback

Synopsis

#include <mdns.h>

void **cyg_mdns_hostname_callback_register**(*fn*, *private*);

Description

This function is used to attach a hostname callback handler function. By default the mDNS sub-system provides a callback handler which attempts to acquire a unique hostname via monotonically increasing a suffix appended to the base hostname value, and automatically trying to claim the amended hostname. e.g. “ecospro” -> “ecospro-1”, “ecospro-2”, … “ecospro-999”, … etc. The application can over-ride this default behaviour by registering an alternative handler function using this API.

The DNS-SD standard Appendix D “Choice of Factory-Default Names” recommends that names are user-friendly, instead of, for example, having something like the 24-bit (non-OUI) MAC address appended. This however means that when a device is first connected to a network containing multiple similarly CDL configured devices then it can take some time to negotiate a unique name. It is the responsibility of the application, using whatever persistant storage schemes it has access to, to ideally store any claimed unique name for subsequent restarts of the device.

If the passed *fn* is `NULL` then the code reverts to the default mDNS callback handler.

```
typedef void (*cyg_mdns_hostname_callback_fn)(void *private, cyg_bool success,
                                              const cyg_uint8 *hostname, cyg_uint8 len);
```

The callback function is executed from the main lwIP networking thread and should be implemented like an eCos DSR and must *NEVER* block, and should run for as little time as possible (for example by waking other threads to perform lengthier tasks).

The hostname callback function is called with *success* set to `true` when the name given has been successfully claimed. At this point an application that has provided its own callback handler can, if desired, record the new name in its persistent storage. The ability for the application to track the last allocated name over reboots, and to use a stored name with a call to the `cyg_mdns_sethostname()` on startup will minimise subsequent delays claiming a name.

Note

The lifetime of the UTF-8 *hostname* pointer is for the active call to the callback function only, and if required for later application use the contents should be copied into a suitable thread-safe buffer accordingly.

If *success* is `false` then the callback indicates that the given *hostname* is not valid and an alternative name should be requested (via another call to `cyg_mdns_sethostname()` from a different thread as appropriate). NOTE: The callback handler may be called at *any* point whilst registered if the hostname is no longer valid (e.g. due to a device appearing on the network and claiming the hostname that was being used). If the callback is no longer required it should be released by attaching a new callback, or by passing NULL for the default handler to be re-instated. The callback function is executed similarly to an eCos DSR and is limited in the operations it can perform and must never block. If higher level processing is required by an application due to a *hostname* conflict then suitable eCos primitives should be used to notify such code.

Note

The callback is implemented as described above (instead of, for example, returning an alternative name to its caller) so that the DSR-like nature of the function is made clear, and that any slow operations should (and can) be performed by higher application layers as needed. It is envisaged that application specific hostname conflict handlers will signal a controlling thread, which when re-scheduled will subsequently call `cyg_mdns_sethostname()` with the next name to be tried.

Name

cyg_mdns_gethostname — Get current hostname value

Synopsis

#include <mdns.h>

cyg_bool **cyg_mdns_gethostname**(*dstbuf*, *len*);

Description

This function provides access to the currently configured mDNS UTF-8 hostname (shared across all mDNS configured lwIP network interfaces) and is intended for debug or UI usage. The active hostname is copied into the supplied buffer, where the passed `*len` specifies the valid buffer length of *dstbuf*. Normally the referenced *dstbuf* should have at least `MDNS_MAX_LABEL` available space, and `*len` set accordingly. The return boolean state indicates success or failure, with `*len` updated with the number of bytes written/required. On failure the contents of *dstbuf* are undefined. If *dstbuf* is `NULL` then the call can be used with a valid *len* pointer to ascertain the amount of storage required to hold the name.

Note

It is possible due to a dynamically changing external network that any returned hostname may already be invalid by the time the call returns the filled buffer to the application. For normal real-world operations the hostname callback function should be tracked by the application to cope with the actual active/current state with suitable cross-thread synchronisation implemented as required.

Return value

If a non-null *len* is supplied then the referenced location is updated with the hostname length at the time of the function call. Boolean `true` is returned if *dstbuf* is NULL or is a pointer to a `*len` buffer large enough to hold the hostname at the time of the call, which is filled with the hostname. On error boolean `false` is returned.

Name

cyg_mdns_setservicelabel — Set service label

Synopsis

#include <mdns.h>

```
cyg_bool cyg_mdns_setservicelabel(id, label);
```

Description

Request that the given NUL terminated UTF-8 string be used as the label for the service specified by *`id`*. This is only a request and depending on the external network devices the name may not be available. The servicelabel callback function interface provides, if required, application control over the alternative naming strategy used when the complete <label>.<service>.<proto>.local name is not available.

Note

This function is not available if the system is configured with `CYGFUN_MDNS_COMMON_NAME` enabled.

Return value

Boolean `true` on success, and `false` on error.

Name

cyg_mdns_getservicelabel — Get current service label value

Synopsis

#include <mdns.h>

```
cyg_bool cyg_mdns_getservicelabel(id, dstbuf, len);
```

Description

This function provides access to the currently configured mDNS UTF-8 label for the specified *`id`* service descriptor. This function is intended for debug or non-time critical UI usage. The active label is copied into the supplied buffer, where the passed *`*len`* specifies the valid buffer length of *`dstbuf`*. Normally the referenced *`dstbuf`* should have at least `MDNS_MAX_LABEL` available space, and *`*len`* set accordingly. The return boolean state indicates success or failure, with *`*len`* updated with the number of bytes written/required. On failure the contents of *`dstbuf`* are undefined. If *`dstbuf`* is `NULL` then the call can be used with a valid *`len`* pointer to ascertain the amount of storage required to hold the name.

Note

It is possible due to a dynamically changing external network that any returned label may already be invalid by the time the call returns the filled buffer to the application. For normal real-world operations the servicelabel callback function should be tracked by the application to cope with the actual active/current state with suitable cross-thread synchronisation implemented as required.

Note

This function is not available if the system is configured with `CYGFUN_MDNS_COMMON_NAME` enabled.

Return value

If a non-null *`len`* is supplied then the referenced location is updated with the label length at the time of the function call. Boolean `true` is returned if *`dstbuf`* is NULL or is a pointer to a *`*len`* buffer large enough to hold the service label at the time of the call, which is filled with the service label. On error boolean `false` is returned.

Name

cyg_mdns_servicelabel_callback_register — Register service label generation callback

Synopsis

#include <mdns.h>

```
void cyg_mdns_servicelabel_callback_register(fn, private);
```

Description

This function is used to attach a servicelabel callback handler function. By default the mDNS sub-system provides a callback handler which attempts to acquire a unique label via monotonically increasing a suffix appended to the current label value, and then automatically attempting to claim the amended <label>.<service>.<proto>.local name.

Note

This function is not available if the system is configured with `CYGFUN_MDNS_COMMON_NAME` enabled.

The application can over-ride this default behaviour by registering an alternative handler function using this API. If the passed *fn* is `NULL` then the code reverts to the default mDNS callback handler.

The same limitations for the callback handler apply as for the hostname callback interface as documented by cyg_mdns_hostname_callback_register.

Name

cyg_mdns_hinfo_register — Register HINFO record data

Synopsis

#include <mdns.h>

`cyg_bool` **`cyg_mdns_hinfo_register`**`(`*`data`*`, `*`len`*`);`

Description

This optional function (controlled by the CDL option `CYGFUN_MDNS_HINFO_OVERRIDE`) allows the application to supply an immutable data vector to be used as the HINFO record data when announcing that a hostname has been claimed. The passed *`len`* parameter specifies the length of the valid data from the supplied *`data`* pointer. Passing the `NULL` pointer allows the application to restore the default mDNS HINFO response.

The passed *`data`* vector should consist of two (2) length-prefixed strings conforming to the mDNS HINFO rules:

- Each length-prefixed string must start with a letter.
- Only upper-case letters, digits, hyphen and forward slash characters are permitted.

The following is a very simple fixed data example:

```
static const cyg_uint8 record_hinfo[] = {
  /* CPU */
  0x09,'C','O','R','T','E','X','-','M','3',
  /* OS */
  0x07,'E','C','O','S','P','R','O',
};

if (!cyg_mdns_hinfo_register(record_hinfo,sizeof(record_hinfo))) {
  /* generate error */
}
```

In some systems the CPU information may need to be derived dynamically at run-time (if multiple platforms are being supported by the application source) and so a RAM structure would be used that is then filled with the CPU and OS fields. The `tests/mdns_example.c` source, located in this package in the eCos source repository, provides an example dynamic implementation.

Return value

The boolean `true` is returned when the referenced data has been registered successfully. If an invalid HINFO data structure is supplied then boolean `false` is returned and the current mDNS HINFO state is not updated.

Name

cyg_mdns_discovery_callback_register — Register DNS-SD response callback

Synopsis

#include <mdns.h>

`cyg_mdns_discovery_context *`**`cyg_mdns_discovery_callback_register`**`(`*`name`*`, `*`fn`*`, `*`private`*`);`

Description

This function is used to attach a callback handler function to be called when (by default) ANSWER, and optionally ADDITIONAL, responses matching the supplied *`name`* are received. The `cyg_mdns_discovery_callback_flags()` function can be used to update the control flags from the default ANSWERS-only configuration once the callback has been registered.

Note

This function is not available if the system is not configured with `CYGIMP_NET_MDNS_DNSSD` enabled.

The *`name`* is an encoded string-table specifying the name that will be matched against mDNS responses when deciding if the registered callback function *`fn`* should be called. For example to match “_http._tcp.local”:

```
const cyg_uint8 label_http[] = { 0x05,'_','h','t','t','p' };
const cyg_uint8 * const services_http[] = { label_http, cyg_mdns_label_tcp, cyg_mdns_label_local, NULL };
```

The *`private`* value is used to pass data (if needed) to the callback function, and is never interpreted by the callback API.

The `<mdns.h>` header defines the function prototype for the callback *`fn`* function:

```
typedef void (*cyg_mdns_discovery_callback_fn)( void *private, cyg_uint32 state, const cyg_mdns_resource *phrr,
                                                struct pbuf *p, cyg_uint16 offname, cyg_uint16 offdata );
```

The *`private`* is the same value as supplied when registering the callback function, and is used by the client to hold any state needed to implement any interaction between the DSR-alike callback function and the application foreground client code.

The *`state`* parameter is a bitmask of flags and are used by the callback to determine what processing is required. The currently defined flags are:

`MDNS_DISCOVERY_CB_NOMORE`

Set to indicate no more data available for the active mDNS response being processed. When this flag is set *NO* actual Resource Record is actually being supplied. This call allows application that require, for example, a U/I update to be notified that a complete response has been processed and no more records are available for the callback. It may be used by a callback that has been caching information whilst constructing a “complete” image of a service to pass the data gathered to another client component.

`MDNS_DISCOVERY_CB_ADDITIONAL`

Set when the callback is being passed an Additional record, and not a direct Answer record.

`MDNS_DISCOVERY_CB_FLUSH`

Informational flag indicating whether this is a “unique” response and any cache should be flushed. See RFC 6762 section 10.2 for more information.

The *`phrr`* pointer references a structure containing, primarily, the type of Resource Record that the callback is being passed, and the TTL (Time-To-Live) of the record, where a TTL value of zero indicates a service is no longer available.

```
typedef struct cyg_mdns_resource {
    cyg_uint16 type;
    cyg_uint16 class;
    cyg_uint32 ttl;
    cyg_uint16 rdlength;
} cyg_mdns_resource;
```

The *`p`* pointer references the lwIP packet buffer containing the Resource Record entry to be processed by the callback, with the *`offname`* and *`offdata`* indices referencing, respectively, the offsets to the resource name and Resource Record type (`phrr->type`) specific data if relevant. The callback should *NEVER* modify the referenced lwIP packet buffer, and should use the available Support API routines to extract the (possibly) compressed data from the lwIP packet buffer for local processing/caching as required. This is because the packet buffer may be shared by multiple Resource Record responses, and other callbacks may also be executed as a result of a single mDNS response as well as the mDNS Responder requirements.

By default only ANSWER records are passed through, but if needed the `cyg_mdns_discovery_callback_flags()` function can be used to manipulate the control flags used to decide which response records are passed to the callback.

When the callback is no longer required it can be detached by using the `cyg_mdns_discovery_callback_unregister()` function. It is the responsibility of the controlling application to ensure that any resources owned by the callback routine are in a safe state prior to unregistering the callback.

Note

The callback function registered should *NEVER* block and should be treated like a DSR function. The function is called within the context of the mDNS lwIP handler as the result of low-level mDNS packet processing.

Return value

A non-`NULL` pointer to an abstract *`handle`* used for subsequent interaction with the successfully registered callback, or `NULL` on failure. For example, the call may fail if all the available callback slots are currently in use.

Note

The number of concurrently registered callbacks is controlled by the `CYGNUM_MDNS_DNSSD_NUM_CALLBACKS` configuration option.

Name

cyg_mdns_discovery_callback_unregister — Unregister DNS-SD response callback

Synopsis

#include <mdns.h>

```
cyg_bool cyg_mdns_discovery_callback_unregister(handle);
```

Description

This function is used to detach a previously registered callback handler function using its abstract *handle*.

Note

This function is not available if the system is not configured with `CYGIMP_NET_MDNS_DNSSD` enabled.

This will ensure that the callback function is no longer called by the mDNS response processing, and makes the callback slot available for re-use.

Return value

Boolean `true` if the operation has completed OK, or `false` on failure.

Name

cyg_mdns_discovery_callback_flags — Read/Modify DNS-SD response callback control flags

Synopsis

#include <mdns.h>

cyg_bool **cyg_mdns_discovery_callback_flags**(*handle*, *and*, *eor*, *flags*);

Description

This function is used to read or modify the control flags used by the mDNS code to decide which received response records are passed to the registered callback function referenced by the abstract *handle*.

Note

This function is not available if the system is not configured with CYGIMP_NET_MDNS_DNSSD enabled.

For simplicity, and code size, this function uses the standard AND/EOR scheme to allow the function to be used for non-destructive interrogation, as well as for flag modification. First the supplied “*and*” value is used as a mask to select the bits to be preserved, prior to the supplied “*eor*” parameter being used to modify specific flag bits. For example, to read the current state without affecting the flags:

```
cyg_mdns_discovery_callback_flags(handle,0xFF,0x00,&current); // READ
```

Whereas the following will ignore the existing flag state and only set the MDNS_DISCOVERY_ADDITIONAL flag value:

```
cyg_mdns_discovery_callback_flags(handle,0x00,MDNS_DISCOVERY_ADDITIONAL,&explicit); // SET
```

The following example explicitly clears the single MDNS_DISCOVERY_ALL flag:

```
cyg_mdns_discovery_callback_flags(handle,~MDNS_DISCOVERY_ALL,0,&modified); // CLEAR FLAG
```

The use of the AND/EOR scheme allows for a single operation to be used to read some flags, explicitly set/clear some flags and toggle other flags all in a single operation if required.

The full set of flags available are:

```
#define MDNS_DISCOVERY_ANSWER     (1 << 0) // Callback will be passed Answer records (default)
#define MDNS_DISCOVERY_ADDITIONAL (1 << 1) // Callback will be passed Additional records
#define MDNS_DISCOVERY_ALL        (1 << 2) // Callback will be passed ALL Additional records if
                                           //   ANSWER callback executed
```

Return value

Boolean true if the operation has completed OK, or false on failure. On success the location referenced by the passed *flags* parameter is updated with the current (as optionally modified) flag state.

Name

cyg_mdns_discovery_query — Issue a DNS-SD query

Synopsis

#include <mdns.h>

`cyg_bool` **`cyg_mdns_discovery_query`**`(`*`name`*`, `*`cnetif`*`, `*`type`*`);`

Description

This function is used to transmit a query to the selected lwIP network interface. The *`name`* should reference a `NULL` terminated set of pointers to the individual field entries for the query name. The *`type`* should be a RFC1035 defined resource record value. If the supplied *`cnetif`* is `NULL` then the code will transmit the query to all active interfaces. An IPv4 query will always be generated, and if lwIP IPv6 is configured than the code will also generate an IPv6 query.

Note

This function is not available if the system is not configured with `CYGIMP_NET_MDNS_DNSSD` enabled.

The following example source fragment demonstrates the “vector of pointers” structure used to hold the *`name`* to be used for the query. This example will issue a query across all interfaces asking for HTTP server information:

```
static const cyg_uint8 label_http[] = { 0x05,'_','h','t','t','p' };

static const cyg_uint8 * const services_http[] = {
    label_http,cyg_mdns_label_tcp,cyg_mdns_label_local,NULL
};

if (cyg_mdns_discovery_query(services_http,NULL,MDNS_RRTYPE_PTR)) {
    // success
} else {
    // failed
}
```

The `<mdns.h>` header file contains manifests (prefixed `MDNS_RRTYPE_`) for the main RFC1035 defined Resource Record types. Though, for DNS-SD use, normally only the `MDNS_RRTYPE_PTR`, `MDNS_RRTYPE_SRV`, `MDNS_RRTYPE_A` and `MDNS_RRTYPE_AAAA` types would be used during network service discovery.

Note

As discussed previously the process of generating queries is distinct from the system for registering callbacks to process responses. It is perfectly possible to generate requests when no callbacks are in place to handle responses (though this would not be of much use), and similarly it is expected that a callback may be registered for the lifetime of the application, with it asynchronously processing relevant responses that may have been triggered by the use of this query function.

Return value

Boolean `true` if the operation has completed OK, or `false` on failure. A failure may result if the operation cannot be posted to the lwIP layer for processing (this will trigger an assert when an `ASSERT` enabled configuration is being used), or (more likely) if there are no active (“link up”) network interfaces on which to send the query.

Example Responder

The `tests/mdns_example.c` source file included in the package provides a complete, simple, real-world example of the mDNS Responder usage. It registers two services, a dummy (port 9) service `_workstation._tcp.local` and a HTTP daemon using service `_http._tcp.local` on port 80.

Normally the first task after the lwIP networking has been initialised is to call the C function:

```
cyg_bool cyg_mdns_init();
```

This initialises the core of the mDNS support. After the mDNS Responder has been initialised, services can be registered, and the network interfaces enabled as required.

The following is a very simple example of declaring a `<hostname>._http._tcp.local` service for the `<hostname>.local` address, where `<hostname>` is the default as configured by the `CYGDAT_NET_MDNS_HOSTNAME` CDL option.

```
static const cyg_uint8 label_http[] = { 0x05,'_','h','t','t','p' };

static const cyg_mdns_service_identity httpd80 = {
  .service = label_http,
  .proto = cyg_mdns_label_tcp,
  .port = 80
};

static const struct cyg_mdns_service httpd80_service = {
  .id = &httpd80,
  .txt_vector = NULL,
  .txt_count = 0,
};

…

if (cyg_mdns_init()) {
  if (cyg_mdns_service_register(1,(struct mdns_service *)&httpd80_service)) {
    if (cyg_mdns_enable()) {
      // other application processing as needed ...
    } else {
      (void)cyg_mdns_service_unregister(1,(struct mdns_service *)&httpd80_service);
      // report error
    }
  } else {
    cyg_mdns_terminate();
    // report error
  }
} else {
  // report error
}
```

Example DNS-SD Queries

The `tests/dnssd_example.c` source file included in the package provides a complete, simple, real-world example of the DNS-SD API usage. See the section called “DNS-SD Example” for more detail.

Chapter 189. Support API

Some support functions are provided by the mDNS code for use by client applications. These functions do not by themselves affect the mDNS state and are thread-safe. Some functions are only present when `CYGIMP_NET_MDNS_DNSSD` support is configured.

Support API

Name

cyg_mdns_strlen — Calculate uncompressed length of (possibly compressed) string

Synopsis

#include <mdns.h>

```
cyg_uint16 cyg_mdns_strlen(p, index, raw);
```

Description

This function is used to calculate the space needed to hold an uncompressed copy of a Resource Record encoded string. The *p* pointer references the lwIP packet buffer containing the original string structure, with the *index* specifying the starting offset within the lwIP packet for the string.

The boolean *raw* flag is used to control the style of uncompressed string for which a length is being calculated.

Return value

The return value depends on the supplied boolean *raw* flag parameter. When *raw*=`true` the return value is the size needed for an uncompressed “raw” copy of the string still using the mDNS encoding style (length byte followed by UTF-8 character data). When *raw*=`false` is specified then the function returns the size needed to hold a final string representation using '.' to separate fields, and also including the terminating `NUL` character. If invalid parameters, or data, are given then the error limit value `0xFFFF` is returned.

Name

cyg_mdns_name_uncompress — Uncompress encoded string

Synopsis

#include <mdns.h>

cyg_uint16 **cyg_mdns_name_uncompress**(*dst*, *dstlimit*, *p*, *index*, *raw*);

Description

This function is used to build a copy of the Resource Record encoded string in the supplied destination buffer. The *p* pointer references a lwIP packet buffer containing the encoded string to be processed at the given *index* offset within the packet. If the supplied *dst* pointer is `NULL` then the function does not copy the data, but can be used to step over an encoded string. When *dst* is not-`NULL` then the *dstlimit* value defines the amount of space available in the destination buffer. When copying the function will not copy more than *dstlimit* characters.

When the boolean *raw* parameter is `true`, the function will copy the uncompressed individual fields to the destination buffer, still using the encoded string style (length byte followed by UTF-8 character data). Whereas *raw*=`false` will build a '.' separated name in the destination buffer.

Return value

The return value is the lwIP packet buffer index for the first data byte after the encoded string.

Name

cyg_mdns_strlen_vector — Calculate uncompressed length of string vector

Synopsis

#include <mdns.h>

```
cyg_uint16 cyg_mdns_strlen_vector(iname, raw);
```

Description

This function is used to calculate the space needed to hold an uncompressed copy of an encoded string described by the *iname* vector. The last entry of the *iname* vector must be `NULL`.

Note

This function is not available if the system is not configured with `CYGIMP_NET_MDNS_DNSSD` enabled.

When *raw*=`true` the function counts the space needed to hold a copy of all the vector entries in an uncompressed form (for supporting “compressed->uncompressed” copying in the original “string-table” format, i.e. length byte followed by UTF-8 character data). The returned value is the space for the referenced encoded strings, and is not the size of the vector describing the name.

When *raw*=`false` we are counting the space needed to hold the “final” string representation with '.' inserted to separate fields.

Return value

Number of bytes needed to hold the uncompressed string in the selected boolean parameter *raw* style.

Name

cyg_mdns_strlen_uncompressed — Length of uncompressed encoded string

Synopsis

#include <mdns.h>

`cyg_uint16` **`cyg_mdns_strlen_uncompressed`**`(`*`us`*`);`

Description

This function is used to calculate the length of an uncompressed, encoded string terminated by a zero-length (`NUL`) field.

Note

This function is not available if the system is not configured with `CYGIMP_NET_MDNS_DNSSD` enabled.

```
static const cyg_uint8 encoded_string[] = {
  0x08,'t','e','s','t','n','a','m','e',
  0x05,'_','h','t','t','p',
  0x00
};

…

cyg_uint16 elen = cyg_mdns_strlen_uncompressed(encoded_string);
// elen == ((1 + 8) + (1 + 5) + 1) == 16
```

Return value

The number of bytes occupied by the encoded string.

Name

cyg_mdns_name — Convert encoded name into dot-notation

Synopsis

#include <mdns.h>

cyg_uint16 **cyg_mdns_name**(*dst*, *dstlimit*, *strtab*);

Description

This function is used to convert an encoded name into a “simple” dot-notation `NUL`-terminated string.

Note

This function is not available if the system is not configured with `CYGIMP_NET_MDNS_DNSSD` enabled.

If the passed *dst* pointer is `NULL` then the routine can be used to calculate the space needed to hold the converted name.

Pseudo-code example:

```
cyg_uint16 needed = cyg_mdns_name(NULL,0,hostname);
char buffer[needed];
cyg_uint16 used = cyg_mdns_name(buffer,needed,hostname);
if (used == needed) {
  diag_printf("  http://%s:%u/\n",buffer,port);
}
```

The routine will not write more than the passed *dstlimit* number of characters, and as such if the passed buffer is too small then the string will *not* be `NUL` terminated.

Return value

The number of characters needed to hold the converted string (when *dst*=`NULL`), or the number of characters used in the supplied *dst* destination buffer.

Name

cyg_mdns_build_txt_vector — Build vector of pointers to individual TXT Record fields

Synopsis

#include <mdns.h>

`cyg_uint8` **`cyg_mdns_build_txt_vector`**`(`*`txt`*`, `*`txtlen`*`, `*`txtkeys`*`, `*`vlen`*`);`

Description

This helper function is used to construct a vector of the individual TXT Resource Record fields for easier client processing.

Note

This function is not available if the system is not configured with `CYGIMP_NET_MDNS_DNSSD` enabled.

The *`txt`* parameter should reference a contiguous buffer containing a TXT Record string-table - length byte followed by character data, possibly followed by another length byte and more character data and so on. NOTE: An empty TXT string-table is marked with a `NUL` character, so *`txtlen`* should always be `1` or more, but a string-table containing entries does *NOT* have a terminating `NUL`, which is why the *`txtlen`* parameter is required.

If the passed *`txtkeys`* value is `NULL` then the function counts the number of entries needed to hold the individual `key[=value]` entries within the TXT Record, so can be used to ascertain the size of vector needed for a specific TXT Record. When *`txtkeys`* is non-`NULL` it points to the beginning of an array of cyg_uint8 pointers. The *`vlen`* parameter specifies the number of vector slots available in this array to receive pointers into the *`txt`* buffer for the individual entries (which for TXT Records will be in the form `key` or `key=value`). When populating the supplied *`txtkeys`* vector a terminating `NULL` pointer is added if the supplied *`vlen`* indicates the supplied vector is larger than the number of TXT fields found.

For example:

```
static const cyg_uint8 example_txt[] = {
  0x04,'a','b','c','d',
  0x0C,'s','o','m','e','k','e','y','=','d','a','t','a'
};
cyg_uint16 txtlen = (cyg_uint16)sizeof(example_txt);

cyg_uint8 entries = cyg_mdns_build_txt_vector(example_txt,txtlen,NULL,0);
cyg_uint8 *txtkeys[entries + 1]; // extra “+1” for terminating NULL

(void)cyg_mdns_build_txt_vector(example_txt,txtlen,txtkeys,(entries + 1));
diag_printf("  TXT-%u contains %u entr%s\n",ti,entries,((entries == 1) ? "y" : "ies"));

unsigned int idx;
for (idx = 0; (idx < entries); idx++) {
  cyg_uint8 klen = txtkeys[idx][0];
  if (klen) {
    cyg_uint8 key[klen + 1];
    memcpy(key,&txtkeys[idx][1],klen);
    key[klen] = '\0';
    diag_printf("    [%u]=\"%s\"\n",idx,key);
  } else {
    diag_printf("    [%u]={EMPTY}\n",idx);
  }
}
```

Return value

Returns a count of the number of `key[=value]` entries in the referenced TXT Record.

Name

cyg_mdns_build_strtab_vector — Build vector of pointers to individual encoded string fields

Synopsis

#include <mdns.h>

`cyg_uint8` **`cyg_mdns_build_strtab_vector`**`(`*`strtab`*`, `*`labels`*`, `*`vlen`*`);`

Description

This helper function is used to construct a vector of the individual label fields present in the supplied, uncompressed, contiguous *`strtab`* string-table.

Note

This function is not available if the system is not configured with `CYGIMP_NET_MDNS_DNSSD` enabled.

If the passed *`labels`* value is `NULL` then the function counts the number of entries needed to hold pointers to the individual label field entries within the `NUL`-terminated string-table, so can be used to ascertain the size of vector needed for a specific encoded string. When *`labels`* is non-`NULL` the *`vlen`* parameter specifies the number of vector slots available to receive pointers into the *`strtab`* buffer for the individual label field entries. When populating the supplied *`labels`* vector a terminating `NULL` pointer is added.

This function is similar in operation to the `cyg_mdns_build_txt_vector()` function, so a similar approach it its example can be used to build and subsequently process a vector of pointers.

Return value

Returns a count of the number of fields within the referenced uncompressed encoded string.

Name

cyg_mdns_strcasecmp_strtab — Compare two encoded string-tables ignoring case

Synopsis

#include <mdns.h>

`int` **`cyg_mdns_strcasecmp_strtab`**`(`*`t1`*`, `*`t2`*`);`

Description

This function allows a field-by-field comparison of two encoded string-table buffers to be performed.

Note

This function is not available if the system is not configured with `CYGIMP_NET_MDNS_DNSSD` enabled.

The referenced string-table parameters should be pointers to uncompressed, contiguous, `NUL`-terminated encoded strings of the form:

```
static const cyg_uint8 encoded_string_example[] = {
  0x08,'t','e','s','t','n','a','m','e',
  0x05,'_','h','t','t','p',
  0x00
};
```

Return value

Returns an integer value equal to zero if *`t1`* is found to exactly match *`t2`* (ignoring case), otherwise a non-zero value is returned on a failure to match.

Chapter 190. Configuration

This chapter shows how to include the mDNS support into an eCos configuration, and how to configure it once installed.

Configuration Overview

The mDNS Responder is contained in a single eCos package `CYGPKG_NET_MDNS`. However, it depends on the services of a collection of other packages for complete functionality. Currently the mDNS Responder implementation is tightly bound with the lwIP TCP/IP networking stack provided by the `CYGPKG_NET_LWIP` package. The lwIP package provides the packet buffer support (allocation and access routines) as well as the underlying multicast UDP transport support.

Quick Start

Incorporating the mDNS Responder into your application is straightforward. The essential starting point is to incorporate the mDNS eCos package (`CYGPKG_NET_MDNS`) into your configuration.

This may be achieved directly using **ecosconfig add** on the command line, or the **Build->Packages…** menu item within the eCos Configuration Tool.

Alternatively, as a convenience, minimal configuration files (`.ecm` files) have been provided to permit an easy starting point for creating a configuration incorporating the mDNS Responder. Two configuration files, `mdns_bct` and `mdns_minimal`, are provided for those intending to use the Bonjour Conformance Test (BCT), and these can be found in the directory *`ECOSPRO`*`/packages/net/mdns/`*`VERSION`*`/misc` where *`ECOSPRO`* is the base directory of the eCosPro installation, and *`VERSION`* is the eCosPro (or package) version. The `mdns_minimal.ecm` file is provided as an example of a low footprint configuration for resource limited target platforms. These files may be used either by providing the configuration file name with the command line **ecosconfig import**; or with the **File->Import…** menu item within the eCos Configuration Tool. Both these files are basic, incorporating only those packages which are essential for mDNS operation.

> **Note**
>
> When building the tests for a limited RAM platform it may be necessary to build only the mDNS tests. The tests for some standard packages may require too much RAM to successfully link. For example to build the mDNS tests for a low resource platform:
>
> ```
> $ mkdir minimal_bct
> $ cd minimal_bct
> $ ecosconfig new PLATFORM
> [ … ecosconfig output elided … ]
> $ ecosconfig import $ECOS_REPOSITORY/net/mdns/VERSION/misc/mdns_minimal.ecm
> [ … ecosconfig output elided … ]
> $ ecosconfig resolve
> $ ecosconfig tree
> $ make
> [ … make output elided … ]
> $ cd net/mdns/current
> $ make tests
> [ … make output elided … ]
> ```

Configuring the mDNS Responder

Once added to the eCos configuration, the mDNS Responder package has a number of configuration options.

`CYGDAT_NET_MDNS_HOSTNAME`

The default mDNS *`hostname`* used when announcing services is provided as part of the standard eCos CDL, but it is expected that real-world applications will manage (via whatever means is appropriate for the target application/platform) their own default `hostname` configuration, and will then use the mDNS API as required to configure the required `hostname` prior to enabling a network interface.

The mDNS `hostname` is, when assigned, shared across all the active mDNS network interfaces. This is to ensure that a named service is common across all the configured interfaces.

Note

The actual `hostname` in use may change asynchronously as part of the handling of mDNS name conflicts and rejections. It is possible for another device to attach to the network and cause a new `hostname` to be chosen. The `hostname` callback support allows for application control of the namespace used. By default the mDNS Responder provides a simple conflict resolution approach of appending a suffix with a monotonically increasing number.

The DNS-SD standard Appendix D “Choice of Factory-Default Names” suggests that the starting name is a simple, user-friendly, human-decipherable, name; and is not automatically derived from some unique identification (e.g. the OUI MAC address).

`CYGNUM_MDNS_NUM_SERVICE_VECTORS`

The number of “sets of service vectors” that can be supported. Each service vector specifies one or more services. To minimise the memory footprint required to support mDNS, the service descriptors are provided by the controlling application as a set of immutable contiguous descriptors. This option controls the number of such sets (minimum 1) that can be held by the mDNS instance.

The CDL configuration should match the target application requirements, and this setting relates to the number of distinct cyg_mdns_service_register calls that can be made. For most embedded applications a value of `1` should suffice since the services to be advertised are known and fixed. However, if an application needs to register and de-register differing sets of services dynamically then this configuration value needs to reflect the number of distinct service sets required.

`CYGNUM_MDNS_MAX_SERVICE_COUNT`

This option specifies the number (count) of service descriptors allowed in an individual service vector. To minimise the memory footprint required to support mDNS this option allows the number of services per vector to be tuned to reflect the requirements of the application.

`CYGFUN_MDNS_COMMON_NAME`

If enabled this option will use the claimed hostname for all registered services. This requires a much smaller RAM footprint than allowing for per-service labels, and may be required for deeply-embedded, low memory, targets if more than a few services are registered. In some environments it may be sufficient to have a single name used for the host and all the announced services.

Note

If this feature is enabled then the mDNS Responder will *NOT* pass the Bonjour Conformance Test (BCT) since it explicitly requests a service label containing spaces, which will pass but then the BCT it then subsequently raises a FAILURE when it sees the hostname label contain spaces.

`CYGFUN_MDNS_HINFO_OVERRIDE`

By default the mDNS Responder provides an HINFO record using the CPU name “CPU” and the operating system name “ECOSPRO” when announcing the claimed hostname. If this option is enabled then support is provided to allow the application to provide its own simple HINFO record data structure.

`CYGNUM_MDNS_ANNOUNCE_COUNT`

The mDNS standard allows for between 2 and 8 announcement packets (with increasing inter-packet delays) to be transmitted when a hostname or service is claimed. Since mDNS is based on UDP multicasts it is possible in a congested system, or for limited bandwidth targets, for packets to be missed. This option allows an increased number of packets to be transmitted to minimise the chances of remote systems missing an announcement.

`CYGNUM_MDNS_TTL_SERVICE`

This option specifies the default “time to live” (TTL) value in seconds that is published for registered services.

`CYGNUM_MDNS_TTL_ANNOUNCEMENT`

This option specifies the default “time to live” (TTL) value in seconds for announcements.

`CYGFUN_MDNS_STATS`

If enabled then this option adds code to track information that may be useful when ascertaining the resources required for a configuration or when debugging the mDNS Responder.

The information is held in the exported structure object *`cyg_mdns_statistics`*. This allows the contents to be manually inspected via a debug tool, or for the application to access the data using the structure definition as provided by the header file `mdns.h`.

See the the section called “Statistics” section for more information on use of this configuration option.

`CYGDBG_MDNS_DEBUG`

If this option is enabled then it provides access to individually controlled CDL debug options for various sub-systems or package features. This allows the detail and amount of debug information to be controlled.

Configuring the mDNS DNS-SD support

Optional support for the DNS-SD API can be enabled in a configuration by enabling the `CYGIMP_NET_MDNS_DNSSD` option. By default, for backwards compatibility, this option is disabled. This ensures the extra memory footprint required to support the DNS-SD API does not impact existing configurations unless explicitly enabled.

`CYGIMP_NET_MDNS_DNSSD`

If this feature is enabled then the DNS-SD API is provided to allow client applications to perform service discovery.

`CYGNUM_MDNS_DNSSD_NUM_CALLBACKS`

This option controls the number of active, concurrent, application callbacks supported by the DNS-SD API. A lower value will help minimise the memory footprint required by the mDNS implementation. The value should be tuned to the application requirements. It is possible for an application to be written requiring only a single active callback handler, though multiple handlers may allow different application sub-systems to be maintaining their own “view” of the specific network services or hosts that the particular sub-system is interested in.

Tuning

Footprint

The current implementation relies on the lwIP heap for allocating the response packets. This requires that lwIP has been configured with a large enough `CYGNUM_LWIP_MEM_SIZE` heap size. If ASSERTS are enabled for the build then failure to allocate the required response packet space will trigger an assert failure, otherwise the packet transmission operation will be dropped in that hope that subsequent operations will succeed once some lwIP heap has become available.

Service Labels

The `CYGFUN_MDNS_COMMON_NAME` controls whether a single, common, name is used to announce the host and all registered services. This option reduces the RAM footprint required, at the expense of not supporting per-service labels.

For very simple target applications, where maybe only a single service is being announced, it may be deemed that it is not an issue if the hostname and service-label are the same. However, a conflict with either causing a renaming will therefore affect both the hostname and service-label.

Statistics

The `CYGFUN_MDNS_STATS` option can be enabled to allow for information counts and sizes to be gathered during execution. These statistics can help with the tuning of the mDNS world during development, since monitoring the minimum and maximum usage counts of resources along with the error counts can indicate resource starvation issues.

For the network packet memory tracking, a common *`cyg_mdns_statistics_memory`* structure is defined and used for each of the packet size records being tracked:

```
struct cyg_mdns_statistics_memory {
  cyg_uint32 count; // number of tracked items
  cyg_uint16 min; // minimum size seen
  cyg_uint16 max; // maximum size seen
};
```

For example the basic statistics structure is defined as:

```
struct cyg_mdns_statistics {
    struct cyg_mdns_statistics_memory errors; // failed packet allocation information
    struct cyg_mdns_statistics_memory hostname; // hostname probe/announce packets
    struct cyg_mdns_statistics_memory service; // service probe/announce packets
    struct cyg_mdns_statistics_memory responses; // query response packets
    struct cyg_mdns_statistics_memory tx; // packets transmitted
};
```

The *`errors`* information records the number of failures to allocate a packet buffer, along with the smallest allocation attempt that failed reported in the *`min`* field and the largest allocation attempt that failed in the *`max`* field.

The *`hostname`*, *`service`* and *`responses`* elements record the number of buffer allocations made for each of the respective mDNS packet type generators along with the minimum and maximum sizes seen.

The *`tx`* field tracks the actual generated packet size as transmitted for all of the packet generators. The maximum recorded by this statistic will normally be smaller than the largest of the packet generator maximums since it tracks the size of packet actually transmitted after any name compression has been applied.

Real-World Example

As an example, with careful tuning, it is possible to implement a simple mDNS Responder and webserver with a very small RAM footprint.

For a Cortex-M3 (STM32F207x) based platform, the default mDNS Responder configuration occupies ~14K of code and requires less than 512-bytes of RAM for the mDNS Responder specific state. If `CYGFUN_MDNS_COMMON_NAME` is enabled then the code occupies less than 14K and requires less than 300-bytes of RAM to hold the mDNS specific context.

The code (normally ROM) and RAM footprint for a complete application will be higher depending on the eCos features configured and application code requirements (e.g. number and size of thread stacks, network buffers, etc.).

Chapter 191. Debug and Test

Debugging

Asserts

If the target platform resources allow, the first step in debugging should be to enable ASSERTs. The inclusion of assert checking will increase the code footprint and lower the performance, but does allow the code to catch internal errors from unexpected data values. e.g. when the application/client is not able to guarantee the validity of data passed into the mDNS Responder.

The mDNS Responder asserts are controlled via the standard eCos Infrastructure CYGPKG_INFRA package CYGDBG_USE_ASSERTS option. If enabled, then run-time assertion checks are performed by the mDNS Responder.

If assertions are enabled, and a debugger is being used it is normally worthwhile setting a breakpoint on the cyg_assert_fail symbol so that the debugger will stop prior to entering the default busy-loop processing.

Diagnostic Output

In conjunction with the CYGDBG_MDNS_DEBUG CDL configuration setting, the header-file src/mdns_debug.h implements the mDNS specific debug control.

When CYGDBG_MDNS_DEBUG is enabled a set of individually selectable sub-systems are available to control the diagnostic output generated.

However, when developing or debugging the mDNS Responder implementation, it may be simpler (with fewer build side-effects) to control the debugging output via uncommenting the necessary manifests at the head of the src/mdns_debug.h source file than re-configuring the complete eCos configuration via the CDL. That way only the mDNS package will be re-built.

Note

Some diagnostic output, if enabled, may adversely affect the operation of the mDNS Responder as seen by 3rd-party code. For example, “slow” serial diagnostic output of the packet parsing and response generation could mean that a significant amount of time passes, such that the mDNS Responder no longer adheres to the timings as specified by the mDNS/DNS-SD standards.

Testing

If the configuration option CYGPKG_NET_MDNS_TESTS is enabled then a set of simple tests are built.

Note

If the target platform has limited memory and is unable to execute the tests then the eCos synthetic Linux target can be used to execute tests and verify the behaviour of the mDNS implementation when debugging, assertions or large test executables are required.

mdns_example

The mdns_example provides a very basic HTTP daemon, with enough functionality to pass the Bonjour Conformance Test (see the section called “Bonjour Conformance Test”).

On startup the application initialises a HTTP daemon listening on port 80, and registers and enables the mDNS announcing that service. A `HTTP GET` request for the default root page will return an HTML page listing the sub-pages provided. The application is manually terminated by requesting the `exit` page. The `config` page has specific support for the features needed for the BCT.

The mdns_example is, however, a simple example of announcing a DNS-SD service, as well as a simple lwIP HTTP daemon demonstration.

dnssd_example

The dnssd_example application demonstrates the use of the DNS-SD API to discover network services (see the section called “DNS-SD Example” for more detail). The application will enumerate local services being announced to the network, before performing specific queries for “_http” and “_ipp” services to ascertain server and TXT information.

mdns_testp

The mdns_testp uses dummy network interfaces to monitor the responses generated by the mDNS Responder in a known clean environment. The physical network connection is not used after the startup initialisation. The test application controls the packets injected into the networking stack and provides its own artificial `driver` layer for packet transmission. This allows for specific mDNS Responder features to be tested without the noise and interference of a real network setup, and without specifying a stand-alone, closed, network like the 3rd-party Bonjour conformance testing.

mdns_farm

The mdns_farm test is primarily for use in the eCosCentric® automated testfarm. It can however be manually executed by starting the application, and then manually executing the simple bash test script `misc/mdnstest1.sh` on a suitable host, passing the network address of the target executing the mdns_farm executable. For example:

```
$ ./misc/mdnstest1.sh 192.168.7.165
```

Note

The host used to execute the script should have a suitable avahi configuration.

Bonjour Conformance Test

If the configuration option `CYGPKG_NET_MDNS_TESTS_STANDALONE` is enabled then the mdns_example is built. This implements a simple mDNS application providing, by default, a limited functionality HTTP daemon on port 80. This test program, when used with a suitable eCos system configuration, can be used for execution against the Bonjour Conformance Test (BCT).

Note

For the full BCT test, which includes link-local address verification, the eCos configuration should have the lwIP IPv4 AutoIP (`CYGFUN_LWIP_AUTOIP`) support enabled and the network interface being used for the Ethernet connection configured to obtain its network address using AutoIP (`CYGOPT_LWIP_ETH_DEV_ADDR_AUTOIP#`).

For the BCT to function, a very specific closed network setup is required. For the mDNS Responder package testing the following equipment was used:

- Unit-Under-Test (UUT)

  The target platform being tested, executing a suitable eCosPro configuration.

- MacBookPro (OS: 10.8.2)

  Executing the BonjourConformanceTest v1.2.8 in a Terminal shell window, and the Safari v6.0.1 web-browser.

- Airport Extreme (HW: A1408 FW: 7.6.1)

  10base-T (wired) Ethernet connections to the UUT and the MacBookPro.

The HTTP daemon provided by the mdns_example application provides support for the necessary BCT interaction using specific CGI parameters to the `/config` page.

The BCT initially tests the IPv4 link-local address support before starting the mDNS Responder interaction.

The following example URLs are based on the BCT v1.2.8 interaction with a default mDNS configuration. If the application being tested uses a different starting hostname, or later BCT executables change the processing, then the target address will need to be modified to suit the state reached at the specific point in the BCT sequence.

For the `MANUAL NAME CHANGE` component the unit-under-test (UUT) needs to be supplied with a specific service label. The `config?service=` option allows a new service-label to be specified for the registered HTTP daemon.

```
ecospro-21.local./config?service=New - Bonjour Service Name
```

Note

The mdns_example provides the HTTP daemon as the first registered service, and the simple `config?service=` option interacts with that specific service in the knowledge that the BCT uses the first announced service as its test `SRV`.

For the `Mixed-Network Interoperability Test` `ROUTABLE TO LINK-LOCAL COMMUNICATION` component the UUT needs to be configured with a routable IP address. The `config?ip=` option allows an IPv4 address to be specified. This will re-configure the default network interface to the specified IPv4 address with an explicit netmask of `255.255.255.0`. For example:

```
ecospro-42.local./config?ip=17.1.1.1
```

For the final `CHATTINESS` component of the BCT the UUT needs to be re-configured with a link-local address. The `config?ll=` option allows an IPv4 link-local address to be supplied, explicitly setting the netmask `255.255.0.0`. For example:

```
ecospro-42.local./config?ll=169.254.1.0
```

Note

For the final `CHATTINESS` component of the BCT, after changing the UUT IPv4 address to a link-local address described above it is important to exit the Safari web-browser being executed on the test host to avoid its Bonjour support interfering with the BCT execution.

When the BCT execution finishes it prompts the operator regarding the generation of an execution report. When applying for Bonjour Conformance the report should be created, and in conjunction with the generated `debug.log` file, sent to Apple for validation as documented in their procedures.

The results for an actual BCT run using the `mdns_example`, executing on a STM32F207xx (Cortex-M3) based platform, are available in the Example 191.1, “ `doc/bct_stm32f207_result.txt` ” and `doc/bct_stm32f207_debug.log` files. These files are original, as produced by the BCT application. For reference, the `doc/bct_stm32f207_terminal.txt` file contains the execution output captured from the MacOS Terminal.

The following is a listing of the `doc/bct_result.txt` file as provided in the package.

Note

The BCT actually finished its testing at 18:05 on Tue Nov 20th, but the `Completed` timestamp reflects when the report was written and does not reflect the time taken for the actual BCT run.

Example 191.1. `doc/bct_stm32f207_result.txt`

```
Bonjour Conformance Test Version 1.2.8
Started   Tue Nov 20 12:57:22 2012
Completed Wed Nov 21 09:00:08 2012


Link-Local Address Allocation
-----------------------------
 PASSED: INITIAL PROBING
 PASSED: PROBING: RATE LIMITING
 PASSED: PROBING: CONFLICTING SIMULTANEOUS PROBES
 PASSED: PROBING: PROBE DENIALS
 PASSED: PROBING COMPLETION
 WARNING: SUBSEQUENT CONFLICTS: RE-PROBE AFTER FIRST CONFLICT
 WARNING: SUBSEQUENT CONFLICTS: RE-PROBE AFTER FIRST CONFLICT
 WARNING: SUBSEQUENT CONFLICTS: RE-PROBE AFTER FIRST CONFLICT
 WARNING: SUBSEQUENT CONFLICTS: RE-PROBE AFTER FIRST CONFLICT
 WARNING: SUBSEQUENT CONFLICTS: RE-PROBE AFTER FIRST CONFLICT
 WARNING: SUBSEQUENT CONFLICTS: RE-PROBE AFTER FIRST CONFLICT
 WARNING: SUBSEQUENT CONFLICTS: RE-PROBE AFTER FIRST CONFLICT
 WARNING: SUBSEQUENT CONFLICTS: RE-PROBE AFTER FIRST CONFLICT
 WARNING: SUBSEQUENT CONFLICTS: RE-PROBE AFTER FIRST CONFLICT

 PASSED: SUBSEQUENT CONFLICTS
 PASSED: HOT-PLUG: USE OF PREVIOUS ADDRESS AS FIRST PROBE CANDIDATE
 PASSED: CABLE CHANGE HANDLING
 PASSED: PREMATURE MDNS PROBING
PASSED with 9 warning(s).

Multicast DNS
-----------------------------
 PASSED: INITIAL PROBING
 PASSED: PROBING: SIMULTANEOUS PROBE CONFLICT
 PASSED: PROBING: RATE LIMITING
 PASSED: PROBING: PROBE DENIALS
 PASSED: WINNING SIMULTANEOUS PROBES - ANNOUNCEMENTS
 PASSED: WINNING SIMULTANEOUS PROBES: WINNING SIMULTANEOUS PROBES
 PASSED: SRV PROBING/ANNOUNCEMENTS
 PASSED: SUBSEQUENT CONFLICT - ANNOUNCEMENTS
 PASSED: SUBSEQUENT CONFLICT - A
 PASSED: SUBSEQUENT CONFLICT - ANNOUNCEMENTS
 PASSED: SUBSEQUENT CONFLICT - SRV
 PASSED: SIMPLE REPLY VERIFICATION
 PASSED: SHARED REPLY TIMING - UNIFORM RANDOM REPLY TIME DISTRIBUTION
 PASSED: SHARED REPLY TIMING
 PASSED: MULTIPLE QUESTIONS - SHARED REPLY TIMING - UNIFORM RANDOM REPLY TIME DISTRIBUTION
 PASSED: MULTIPLE QUESTIONS - SHARED REPLY TIMING
 PASSED: REPLY AGGREGATION
 PASSED: MANUAL NAME CHANGE - ANNOUNCEMENTS
 PASSED: HOT-PLUGGING: INITIAL PROBING
 PASSED: HOT-PLUGGING: PROBING: SIMULTANEOUS PROBE CONFLICT
 PASSED: HOT-PLUGGING: PROBING: RATE LIMITING
 PASSED: HOT-PLUGGING: PROBING: PROBE DENIALS
 PASSED: HOT-PLUGGING: WINNING SIMULTANEOUS PROBES - ANNOUNCEMENTS
 PASSED: HOT-PLUGGING: WINNING SIMULTANEOUS PROBES: WINNING SIMULTANEOUS PROBES
 PASSED: HOT-PLUGGING: SRV PROBING/ANNOUNCEMENTS
 PASSED: HOT-PLUGGING: SUBSEQUENT CONFLICT - ANNOUNCEMENTS
 PASSED: HOT-PLUGGING: SUBSEQUENT CONFLICT - A
 PASSED: HOT-PLUGGING: SUBSEQUENT CONFLICT - ANNOUNCEMENTS
 PASSED: HOT-PLUGGING: SUBSEQUENT CONFLICT - SRV
 PASSED: HOT-PLUGGING
 PASSED: NO DUPLICATE RECORDS IN PACKETS
 PASSED: REQUIRED ADDITIONAL RECORDS IN ANSWERS
 PASSED: LEGAL CHARACTERS IN ADDRESS RECORD NAMES
 PASSED: CACHE FLUSH BIT SET IN NON-SHARED RESPONSES
 PASSED: CACHE FLUSH BIT NOT SET IN PROPOSED ANSWER OF PROBES
PASSED with 0 warning(s).

Mixed-Network Interoperability
-----------------------------
 PASSED: LINK-LOCAL TO ROUTABLE COMMUNICATION
 PASSED: ROUTABLE TO LINK-LOCAL COMMUNICATION
 PASSED: CACHE FLUSH BIT NOT SET IN UNICAST RESPONSE
 PASSED: UNICAST INTEROPERABILITY
 PASSED: CHATTINESS
 PASSED: mDNS IP TTL CHECK
 PASSED: DUPLICATE RECORDS CHECK
 PASSED: ADDITIONAL RECORDS IN ANSWER CHECK
PASSED with 0 warning(s).


******************************************************************************
 CONGRATULATIONS: You successfully passed the Bonjour Conformance test
******************************************************************************
```

DNS-SD Example

If required the dnssd_example application can be used as a starting point for a real-world application. The test starts a thread which performs some DNS-SD operations, as likely to be needed by an application that needs to search for mDNS announced network services. The test is contained within the function:

void **thread_dnssd**(*arg*);

Since the callback function (as described in cyg_mdns_discovery_callback_register()) must *NEVER* block, this example uses the eCos mailbox and fixed-size-memory-allocator functionality as the mechanism to communicate between the callback (which was registered with the mDNS world with the aforementioned function) and the application code. It uses the non-blocking functionality of these core eCos features to ensure the callback will not block.

The core of the foreground functionality is performed in the function:

void **example_browse**(*tag*, *services*, *cflags*, *waitsecs*, *cb_hostport*);

which performs a `PTR` service browse operation, and then waits for callback events to the function registered against the specific *services* name. As documented, a query can result in multiple registered callbacks being called since the queries are common, and not specific to a registered *services* name. The *waitsecs* just specifies how long (in seconds) the example code will wait for responses before returning. The code currently waits a relatively long time after issuing the single query, whereas a real-world application would more likely have its own control loop to re-issue periodic queries if needed; and to maintain whatever data structures for discovered services it needs to track. However, whilst the callback is active, any matching mDNS response will be passed through the callback, irrespective of whether that response is the result of a query generated by this “host”, or another network device. Another example in the function:

service_table_entry ***example_query**(*query*, *type*, *ctx*);

implements functionality where the query is re-issued every second, but the function completes as soon as a valid result is received (which would normally be a very short time after the initial query is transmitted to the network).

The example callback is provided by the `cb_service()` function, and is coded to support both the basic service browsing, but also the individual specific queries. Simpler applications could implement a simpler function just responding to the information they need. Fundamentally the callback implements two phases: Processing records, and marking the end of a “set” of responses:

```
void cb_service(void *priv, cyg_uint32 state, const cyg_mdns_resource *phrr,
                struct pbuf *p, u16_t offname, u16_t offdata)
{
  if (state & MDNS_DISCOVERY_CB_NOMORE) { // No record for this callback
    /* If required notify foreground.
```

```
       This could trigger a U/I update with recently acquired data, or
       just be the point at which captured data is passed through. */
  } else { // Process record
    switch (phrr->type) {
      …
    }
  }

  return;
}
```

A fundamental task of the callback is to extract the encoded information from the referenced lwIP packet buffer. The following example fragment shows how the referenced response name can be copied into a local buffer. This example uses `raw=true` so that the result is an uncompressed version in the mDNS encoded string format (length byte followed by the field data, with the set of fields terminated by a `0x00` byte indicating a zero-length field):

```
cyg_uint16 nlen = cyg_mdns_strlen(p,offname,true);
cyg_uint8 name[nlen];
u16_t endindex;

endindex = cyg_mdns_name_uncompress((char *)name,sizeof(name),p,offname,true);
CYG_ASSERT((endindex <= offdata),"Name length mismatch");
```

Note

The callback should *NEVER* modify the passed lwIP packet buffer. A mDNS response may contain multiple records that are relevant to more than one callback as well as to the internal mDNS Responder operation. Also mDNS responses may be compressed, where data is shared between records. This is why a DNS-SD query client that needs to preserve information needs to make its own uncompressed copies without changing the referenced mDNS response data.

Back at the application foreground level, the first substantial operation performed within `thread_dnssd()` is to use the specific DNS-SD “wildcard” query to ascertain what service types are being announced to the network. This uses the `cyg_mdns_name_services_dnssd_udp_local` variable (built-in to the mDNS package) to reference the standard “_services._dns-sd._udp.local” name. The function outputs each service type discovered, for example:

```
INFO:<Browsing for “Local services” (wait 4-seconds)>
Service: _http._tcp.local
Service: _pdl-datastream._tcp.local
Service: _ipp._tcp.local
Service: _printer._tcp.local
Service: _udisks-ssh._tcp.local
Service: _ssh._tcp.local
Service: _sftp-ssh._tcp.local
Service: _workstation._tcp.local
Service: _http-alt._tcp.local
Service: _scanner._tcp.local
PASS:<Service Browse>
```

The example then performs three passes each against the “_http” and “_ipp” local service names, with each pass allowing more records to be passed to the registered callback for processing.

The first pass uses the default control flags, which will pass only the explicit matching ANSWER records for any active PTR answers. The application will output a line for each specific service type that provides a response:

```
INFO:<Browsing for “_http” (wait 4-seconds)>
Service: Officejet Pro 8500 A910 [D0220F]._http._tcp.local
Service: Xerox Phaser 6280DN (ac:a8:0d)._http._tcp.local
PASS:<Service Browse>
```

The second pass explicitly adds the `MDNS_DISCOVERY_ADDITIONAL` flag to have the callback supplied with matching ADDITIONAL records, and not just the matching ANSWER record. For example, this now results in the server name, port and TXT records being recorded if they are supplied:

```
INFO:<Browsing for “_http” ADDITIONAL (wait 4-seconds)>
INFO:<Callback flags now 03>
Service: Officejet Pro 8500 A910 [D0220F]._http._tcp.local
  http://hp8500.local:80/
  TXT-0 contains 0 entries
Service: Xerox Phaser 6280DN (ac:a8:0d)._http._tcp.local
  http://XRX0000AAACA80D.local:80/
  TXT-0 contains 9 entries
    [0]=“qtotal=5”
    [1]=“ty=Xerox Phaser 6280DN”
    [2]=“product=(Phaser 6280DN)”
    [3]=“pdl=application/postscript”
    [4]=“adminurl=http://.local./”
    [5]=“usb_MFG=Xerox”
    [6]=“usb_MDL=Xerox Phaser 6280DN”
    [7]=“Binary=T”
    [8]=“TBCP=T”
PASS:<Service Browse>
```

The third pass adds the `MDNS_DISCOVERY_ALL` flag requesting that the callback is supplied all ADDITIONAL records in the response if an ANSWER was previously passed for the response being processed. This can be used to allow the callback to gather network address information (from `A` or `AAAA` records as appropriate) so that the callback can supply complete service information in a single result. As can be seen from the following example output, the IPv4 and IPv6 addresses are now provided in the result passed through the mailbox from the callback processing to the foreground client code:

```
INFO:<Browsing for “_http” ALL ADDITIONAL (wait 4-seconds)>
INFO:<Callback flags now 07>
Service: Officejet Pro 8500 A910 [D0220F]._http._tcp.local
  http://hp8500.local:80/
  IPv4 192.168.7.6
  IPv6 2001:4D48:AD00:5A17:6AB5:99FF:FED0:220F
  IPv6 FE80::6AB5:99FF:FED0:220F
  TXT-0 contains 0 entries
Service: Xerox Phaser 6280DN (ac:a8:0d)._http._tcp.local
  http://XRX0000AAACA80D.local:80/
  IPv4 192.168.7.25
  TXT-0 contains 9 entries
    [0]=“qtotal=5”
    [1]=“ty=Xerox Phaser 6280DN”
    [2]=“product=(Phaser 6280DN)”
    [3]=“pdl=application/postscript”
    [4]=“adminurl=http://.local./”
    [5]=“usb_MFG=Xerox”
    [6]=“usb_MDL=Xerox Phaser 6280DN”
    [7]=“Binary=T”
    [8]=“TBCP=T”
PASS:<Service Browse>
```

As mentioned, the example will then perform the same queries for the “_ipp” service name outputting suitable results for the discovered service (if any are actually present on the connected network).

Finally the example performs some explicit queries as would be performed by a simple application that needs to enumerate a (previously discovered, or hardwired) service. The code uses the first valid “_http” service response, and directs specific queries to this service name (unlike the “wildcard” `PTR` queries used to discover services). It uses the `example_query()` to request the `SRV`, and then the `A`/`AAAA` address records as appropriate. The output would be similar to the following.

```
INFO:<Request specific records>
Instance: Officejet Pro 8500 A910 [D0220F]
Service: Officejet Pro 8500 A910 [D0220F]._http._tcp.local
  Host: hp8500.local (Port 80)
  IPv6 2001:4D48:AD00:5A17:6AB5:99FF:FED0:220F
  IPv6 FE80::6AB5:99FF:FED0:220F
  IPv4 192.168.7.6
```

Part LV. NTP Client Support

Name

eCosPro Support for NTP — Overview

Description

The ntpclient package provides an NTP (Network Time Protocol) client for use with the TCP/IP stack in eCos. The client supports Version 3 of the NTP specification as defined in RFC 1305 and supports only unicast messages. It is currently restricted to IPv4 networks only and can only use the BSD IP stack.

The client is based on OpenNTPD 4.6 and implements a subset of the NTP protocol Version 3, sufficient to synchronize the eCos system clock with one or more NTP servers.

The client has been tested against the default Linux **ntpd** server on local hosts and public servers from the UK NTP pool.

Name

NTP Client API — NTP client API and configuration

NTP Client API

The NTP client provides two main entry points:

```
#include <ntpclient.h>

 __externC void cyg_ntpclient_start( cyg_ntpclient_config *config );

 __externC void cyg_ntpclient_stop( void );

__externC void cyg_ntpclient_status( cyg_ntpclient_status *status );

__externC void cyg_ntpclient_log( int log_mask );
```

The `cyg_ntpclient_start()` function starts the NTP client running using parameters defined in the *`config`* argument. The `cyg_ntpclient_stop()` stops the NTP client, after which it can be restarted with a new configuration.

The *`config`* argument contains the following fields that the user may set to control the behaviour of the NTP client. The config structure and its contents must remain accessible until `cyg_ntpclient_stop()` is called, as the NTP client stores a reference to it rather than copying it. Therefore it is likely it will want to be allocated persistently, typically as a static or global.

`const char *server[]`

A list of server IP addresses to access. These are specified as strings in standard dot notation. Unused slots should be set to NULL. The size of this array is controlled by `CYGNUM_NET_NTPCLIENT_SERVER_MAX`.

`int settime`

If non-zero this will force the NTP client to set the system time immediately from the NTP servers. Otherwise it will wait for the time to stabilize before doing this. This is useful on targets that do not have a built-in battery-backed real time clock. Note that by taking the first server time supplied, the initial time set may be less accurate than one would normally expect, it will be adjusted to track the server times more closely as the client accumulates samples.

If this option is set, then the call to `cyg_ntpclient_start()` will not return until the initial time has been set.

The `cyg_ntpclient_status_get()` function returns some information on the state of the client and the NTP peers it is communicating with. On return the *`status`* argument is filled in with the following information.

`double *last_adjustment`

This records the last adjustment made to the clock expressed as a whole number plus fraction of seconds.

`peer[]`

A structure for each peer. The size of this array is controlled by `CYGNUM_NET_NTPCLIENT_SERVER_MAX`. Each entry contains the following fields:

`const char *addr`

The IP address of the peer expressed as a string. This is actually just a pointer to the string passed to `cyg_ntpclient_start()` in the configuration. If the value is NULL then this peer table entry is invalid.

`int trustlevel`

The level of trust the client has in this peer. This is a value between 0 and 10. For a good peer, this will be near the upper end of this range.

`int lasterror`

If non-zero this records the last communication error raised by this peer.

`double delay` The most recent smallest communication delay to the peer.

`double offset` The offset between the local and the remote clock for the same transaction as the *`delay`* field.

The `cyg_ntpclient_log()` function sets some logging option in the NTP client. These are output on the standard output using printf(). The *`log_mask`* is a bitwise OR of the following values: `CYG_NTPCLIENT_LOG_DEBUG`, `CYG_NTPCLIENT_LOG_INFO` and `CYG_NTPCLIENT_LOG_CRIT` which generate debug, information and critical error messages respectively. The value `CYG_NTPCLIENT_LOG_ALL` enables all flags. All flags can be cleared by calling this function with a zero parameter.

The normal idiom for use of the NTP client is to define the cyg_ntpclient structure statically and to then call `cyg_ntpclient_start()` with a pointer to that. The following is a somewhat contrived example:

```
#include <ntpclient.h>

//==========================================================================

static cyg_ntpclient_config ntp_config =
{
    .server             = { "10.0.1.1", "149.255.102.233", NULL },
    .settime            = 1,
};

//==========================================================================

void start_ntp(void)
{
    cyg_ntpclient_start( &ntp_config );
}
```

Configuration Options

The NTP client is enabled by including the `CYGPKG_NET_NTPCLIENT` together with the `CYGPKG_CLOCK_COMMON` package. It also needs the FreeBSD network stack, POSIX and the C library; therefore, it is best to start from the `net` template. The NTP client package contains a number of configuration options:

`CYGNUM_NET_NTPCLIENT_PORT`

This option defines the port number to which the client will direct NTP packets. This should normally be the standard NTP port number of 123, but for testing it may be useful to change it to a non-standard value.

`CYGNUM_NET_NTPCLIENT_STACK_SIZE`

This option defines the size of the stack used by the NTP client thread. This value defines how much more stack than the value given by CYGNUM_HAL_STACK_SIZE_TYPICAL is used. The default of 4KiB should be sufficient for most purposes.

`CYGNUM_NET_NTPCLIENT_THREAD_PRIORITY`

This option defines the priority at which the NTP thread will run. If it is important that the client maintains synchronization with the servers, then this value should be small, giving the client thread a high priority. The default value is 2, making the thread one of the higher priority threads in the system.

`CYGNUM_NET_NTPCLIENT_SERVER_MAX`

This option defines the maximum number of NTP servers the client can access. This controls the number that can be passed in the configuration and the size and number of various data structures allocated statically in the code.

Name

NTP Client Test Programs — Describe the test programs and their host-side support

Test Programs

There are four test programs in the `tests` subdirectory of this package that exercise the NTP client and do duty as examples of its use. These are supported by a test NTP server in the `hosts` directory. There is a `README.TXT` file in that directory describing how to build and run the test server.

ntp_basic

The `ntp_basic.c` test performs no actual tests. Instead it just starts the NTP client and then prints the current time, in the form of seconds since the UNIX epoch, every 10s for some number of iterations before shutting down. The main purpose of this test is to check the basic functionality of the client. With access to a suitable host clock, this can also serve as a manual verification that the client is correctly tracking the server.

ntp_accuracy

The `ntp_accuracy.c` test attempts to check the accuracy of the local clock against the test server. After starting the NTP client and allowing it to synchronize it queries the time on the test server via its control interface and compares the result with local time. It takes a number of samples and if more than 80% of these are within 10ms then the test is considered a pass. It repeats this test a number of times at 10s intervals to look for any clock drift.

ntp_date

The `ntp_date.c` test attempts to check that the client can handle crossing various pathological dates. These include handling leap seconds, the UNIX 32 bit time_t sign overflow in 2038, the UNIX 32 bit unsigned overflow in 2106 and the NTP timestamp era ends in 2036 and 2172. It does this by using the test server's control interface to change its idea of the current data and then watching the local clock as the NTP client resynchronizes to the new date.

ntp_adjtime

The `ntp_adjtime.c` test attempts to check that the underlying clock adjustment subsystem correctly handles various time adjustments. It does this by changing the test server's idea of time by some amount and watching the NTP client resyncronize to the new time. It has the option for each test of either delaying until the adjustment is made, or polling the current time, checking that time continues to increse monotonically.

Part LVI. Simple Network Time Protocol Client

The SNTP package provides implementation of a client for RFC 2030, the Simple Network Time Protocol (SNTP). The client listens for broadcasts or IPv6 multicasts from an NTP server and uses the information received to set the system's time of day clock. This will be either the POSIX CLOCK_REALTIME clock or the wallclock device, or both, depending on the configuration. It can also be configured to send SNTP time requests to specific NTP servers using SNTP's unicast mode.

Note that this package predates the existence of the Common Clock Services and has not yet been adapted to use that package's services. Also this implementation is wholly separate to that of the NTP Client.

Table of Contents

Chapter 192. The SNTP Client

Starting the SNTP client

The sntp client is implemented as a thread which listens for NTP broadcasts and IPv6 multicasts, and optionally sends SNTP unicast requests to specific NTP servers. This thread may be automatically started by the system if it receives a list of (S)NTP servers from the DHCP server and unicast mode is enabled. Otherwise it must be started by the user application. The header file `cyg/sntp/sntp.h` declares the function to be called. The thread is then started by calling the function:

```
void cyg_sntp_start(void);
```

It is safe to call this function multiple times. Once started, the thread will run forever.

What it does

The SNTP client listens for NTP IPv4 broadcasts from any NTP servers, or IPv6 multicasts using the address fe0x:0X::101, where X can be 2 (Link Local), 5 (Site-Local) or 0xe (Global). Such packets contain a timestamp indicating the current time. The packet also contains information about where the server is in the hierarchy of time servers. A server at the root of the time server tree normally has an atomic clock. Such a server is said to be at stratum 0. A time server which is synchronised to a stratum 0 server is said to be at stratum 1 etc. The client will accept any NTP packets from servers using version 3 or 4 of the protocol. When receiving packets from multiple servers, it will use the packets from the server with the lowest stratum. However, if there are no packets from this server for 10 minutes and another server is sending packets, the client will change servers.

If SNTP unicast mode is enabled via the CYGPKG_NET_SNTP_UNICAST option, the SNTP client can additionally be configured with a list of specific NTP servers to query. The general algorithm is as follows: if the system clock has not yet been set via an NTP time update, then the client will send out NTP requests every 30 seconds to all configured NTP servers. Once an NTP time update has been received, the client will send out additional NTP requests every 30 minutes in order to update the system clock. These requests are resent every 30 seconds until a response is received.

The system clock in eCos is accurate to 1 second. The SNTP client will change the system clock when the time difference with the received timestamp is greater than 2 seconds. The change is made as a step.

Configuring the unicast list of NTP servers

If SNTP unicast mode is enabled via the CYGPKG_NET_SNTP_UNICAST option, the SNTP client can be configured with a list of NTP servers to contact for time updates.

By default, this list is configured with NTP server information received from DHCP. The number of NTP servers that are extracted from DHCP can be configured with the CYGOPT_NET_SNTP_UNICAST_MAXDHCP option. This option can also be used to disable DHCP usage entirely.

The list of NTP servers can be manually configured with the following API function. Note that manual configuration will override any servers that were automatically configured by DHCP. But later reconfigurations by DHCP will override manual configurations. Hence it is not recommended to manually configure servers when CYGOPT_NET_SNTP_UNICAST is enabled.

```
#include <cyg/sntp/sntp.h>

void cyg_sntp_set_servers(struct sockaddr *server_list, cyg_uint32 num_servers);
```

This function takes an array of sockaddr structures specifying the IP address and UDP port of each NTP server to query. Currently, both IPv4 and IPv6 sockaddr structures are supported. The num_servers argument specifies how many sockaddr's are contained in the array. The server_list array must be maintained by the caller. Once the array is registered with this function, it must not be modified by the caller until it is replaced or unregistered by another call to this function.

Calling this function with a server_list of NULL and a num_servers value of 0 unregisters any previously configured server_list array.

Finally, note that if this function is called with a non-empty server list, it will implicitly start the SNTP client if it has not already been started (i.e. it will call cyg_sntp_start()).

Warning: timestamp wrap around

The timestamp in the NTP packet is a 32bit integer which represents the number of seconds after 00:00 01/01/1970. This 32bit number will wrap around at 06:28:16 Feb 7 2036. At this point in time, the eCos time will jump back to around 00:00:00 Jan 1 1970 when the next NTP packet is received.

YOU HAVE BEEN WARNED!

The SNTP test program

The SNTP package contains a simple test program. Testing an SNTP client is not easy, so the test program should be considered as more a proof of concept. It shows that an NTP packet has been received, and is accurate to within a few days.

The test program starts the network interfaces using the standard call. It then starts the SNTP thread. A loop is then entered printing the current system time every second for two minutes. When the client receives an NTP packet the time will jump from 1970 to hopefully the present day. Once the two minutes have expired, two simple tests are made. If the time is still less than 5 minutes since 00:00:00 01/01/1970 the test fails. This indicates no NTP messages have been received. Check that the server is actually sending packet, using the correct port (123), correct IPv6 multicast address, and at a sufficiently frequent rate that the target has a chance to receive a message within the 2 minute interval. If all this is correct, assume the target is broken.

The second test is that the current system time is compared with the build time as reported by the CPP macro __DATE__. If the build date is in the future relative to the system time, the test fails. If the build date is more than 90 days in the past relative to the system time the test also fails. If such failures are seen, use wallclock time to verify the time printed during the test. If this seems correct check the build date for the test. This is printed at startup. If all else fails check that the computer used to build the test has the correct time.

If SNTP unicast mode is enabled, the above tests are run twice. The first time, the SNTP client is configured with NTP server addresses from DHCP. The second time, unicast mode is disabled and only multicasts are listened for. Note that the unicast test is partially bogus in the sense that any multicast packet received will also make the unicast test pass. To reduce the chance of this happening the test will wait for a shorter time for replies. This is not ideal, but it is the best that can be done with an automated test.

Part LVII. WLAN

Table of Contents

Chapter 193. WLAN overview

Important

This eCosPro-WiFi Middleware package is STRICTLY LICENSED FOR NON-COMMERCIAL PURPOSES ONLY. It may not be used for Commercial purposes in full or in part in any format, including source code, binary code and object code format.

A Commercial eCosPro License version 3 (or above) which explicity includes this Middleware Package is required for Commercial use.

Introduction

The CYGPKG_NET_WLAN package defines the common eCos Wireless LAN API. This allows eCos applications to operate with a variety of different WiFi driver implementations.

The WLAN package relies on the presence of the CYGPKG_NET_LWIP and the CYGPKG_IO_ETH_DRIVERS packages.

Chapter 194. Configuration

This chapter shows how to include the WLAN support into an eCos configuration, and to configure it once installed.

Configuration Overview

The common eCos WLAN layer is contained in the package `CYGPKG_NET_WLAN`. However, it depends on the services of a collection of other packages for complete functionality. Currently the WLAN implementation is tightly bound with the lwIP TCP/IP networking stack provided by the `CYGPKG_NET_LWIP` package.

Incorporating the WLAN support into your application is straightforward. The essential starting point is to incorporate the WLAN eCos package (`CYGPKG_NET_WLAN`) into your configuration.

This may be achieved directly using **ecosconfig add** on the command line, or the **Build->Packages…** menu item within the eCos Configuration Tool.

Configuration Options

Once added to the eCos configuration, the WLAN package has a number of configuration options.

The CDL configuration provides default settings for the use by wireless network interfaces. The application documented in the section called “wlan_switch” provides an example implementation of using the run-time interface configuration, where appropriate, to associate with different access-points.

`CYGDAT_NET_WLAN_COUNTRY`

> This option is used to define the regulatory domain for the radio.

`CYGPKG_NET_WLAN_STA`*`x`*

> If infrastructure (station) interfaces are provided by the underlying wireless device driver package then a number of these STA*x* options become available.
>
> `CYGPKG_NET_WLAN_STAx_SSID`
>
> > This option specifies the SSID (network name) that the interface should associate with.
>
> `CYGPKG_NET_WLAN_STAx_SEC`
>
> > This option is used to select the security encoding required to associate with the named SSID access-point.
>
> `CYGPKG_NET_WLAN_STAx_PASS`
>
> > For pre-shared key security encodings this option supplies the passphrase associated with the SSID access-point.

`CYGPKG_NET_WLAN_AP`*`x`*

> If master (access-point) or adhoc (p2p) capable interfaces are provided by the underlying wireless device driver package then a number of these AP*x* options become available.
>
> `CYGPKG_NET_WLAN_APx_SSID`
>
> > For access-point or P2P interfaces this option supplies the network name used for the connection.

`CYGPKG_NET_WLAN_APx_SEC`

This option specifies the security encoding exposed by the interface.

`CYGPKG_NET_WLAN_APx_PASS`

For pre-shared key security encoding this option defines the passphrase to be used.

`CYGPKG_NET_WLAN_APx_TYPE`

This option allows selection between the “Infrastructure”(master, access-point) and “Adhoc” (P2P) interface modes.

`CYGPKG_NET_WLAN_APx_CHANNEL`

The radio channel to be used for the interface.

Chapter 195. WLAN API

The WLAN API is primarily via the device driver get/set config (`ioctl()`-alike) interface, implemented via the driver supplied interface function:

`int` **`cyg_lwip_eth_ioctl`** `(`*`netif`*`,` *`key`*`,` *`data`*`,` *`data_length`*`);`

See Chapter 16, *User API* for an overview of the general I/O config interface.

SET operations are used to change the active wireless configuration.

`WLAN_DRV_SET_COMMIT`

> This option is used to activate any pending wireless configuration setting changes. This can result in disassociating from the active network configuration, and establishing a new session with the cached (pending) configuration. This information can include the MODE, ESSID, PASSPHRASE and SECTYPE settings.

`WLAN_DRV_SET_MODE`

> This option is used to configure the pending interface wireless mode. It is not used for the active configuration until a `WLAN_DRV_SET_COMMIT` operation is performed.
>
> ### Note
>
> Not all available WLAN mode definitions will necessarily be supported by the underlying wireless hardware driver. For example the `CYGPKG_NET_WIFI_BROADCOM_WWD` package only supports the modes `WLAN_MODE_INFRA`, `WLAN_MODE_MASTER` and `WLAN_MODE_ADHOC`.
>
> The supplied (`wlan_drv_mode_t`) parameter value can be one of:
>
> `WLAN_MODE_INFRA`
>
> > Infrastructure (multi-cell, roaming) network station. The interface will connect to an access-point, which will act as a bridge to other network segments.
>
> `WLAN_MODE_MASTER`
>
> > Access-point (synchronisation master).
>
> `WLAN_MODE_ADHOC`
>
> > Adhoc (P2P) single-cell group network. Such a network has no structure, and no access-point is involved.
>
> `WLAN_MODE_REPEAT`
>
> > Wireless repeater. Packet forwarder.
>
> `WLAN_MODE_SECOND`
>
> > Secondary master/repeater.
>
> `WLAN_MODE_MONITOR`
>
> > Passive monitor. This is used to listen on a network interface to capture packets, without transmitting to the network.
>
> `WLAN_MODE_MESH`
>
> > IEEE 802.11s mesh network.

`WLAN_MODE_AUTO`

This is a special option that is used to allow the driver to select which mode should be used. Normally this would not be appropriate, but some implementations may be limited to specific hardware implementions.

`WLAN_DRV_SET_ESSID`

This option is used to configure the pending network name (SSID). It is not used for the active configuration until a `WLAN_DRV_SET_COMMIT` operation is performed.

The passed (`wlan_drv_essid_t`) structure provides the SSID vector (upto `WLAN_ESSID_MAX_SIZE` bytes in length).

`WLAN_DRV_SET_SECTYPE`

This option is used to configure the pending network security encoding. It is not used for the active configuration until a `WLAN_DRV_SET_COMMIT` operation is performed.

The (`wlan_drv_sectype_t`) parameter is a bitfield encoding of the desired security features. The `WLAN_SECTYPE_TYPE_MASK` can be used to mask out the main security type encoding:

`WLAN_SECURITY_TYPE_WPA2`

Wi-Fi Protected Access 2.

`WLAN_SECURITY_TYPE_WPA`

Wi-Fi Protected Access.

`WLAN_SECURITY_TYPE_WEP`

Wired Equivalent Privacy.

Note

Deprecated. Due to security weaknesses it is not recommended to use WEP.

`WLAN_SECURITY_TYPE_NONE`

Open network, with no security required.

Along with the basic security type encoding the passed parameter can contain a set of authentication and cipher flags.

The authentication type flags:

`WLAN_SECURITY_AUTH_PSK`

Pre-Shared-Key (passphrase). The PSK mode is sometimes referred to as “Personal”.

`WLAN_SECURITY_AUTH_1X`

IEEE 802.1x (enterprise).

The support cipher flags:

`WLAN_CIPHER_TYPE_WEP`

Wired Equivalent Privacy.

`WLAN_CIPHER_TYPE_TKIP`

Temporal Key Integrity Protocol.

`WLAN_CIPHER_TYPE_CCMP`

Counter Mode with CBC-MAC (CCM) mode Protocol. Sometimes referred to as AES (Advanced Encryption Standard) mode.

`WLAN_DRV_SET_PASSPHRASE`

This option is used to configure the pending network security pre-shared key value. It is not used for the active configuration until a `WLAN_DRV_SET_COMMIT` operation is performed.

The passed (`wlan_drv_passphrase_t`) structure provides the key vector (upto `WLAN_SECURITY_PASSPHRASE_LEN` bytes in length).

Note

For security reasons there is no equivalent `GET` operation to read the active passphrase value.

`WLAN_DRV_SET_ENCODE`

This option is used to configure the pending network security WEP key set. It is not used for the active configuration until a `WLAN_DRV_SET_COMMIT` operation is performed.

The passed (`wlan_drv_wepkeys_t`) structure contains the keys to be used for WEP authentication. The passed *`keys[]`* vector contains the binary keys when the supplied *`length`* field is either `WLAN_SECURITY_WEP104_LEN` or `WLAN_SECURITY_WEP40_LEN`. If hexadecimal ASCII representations are being supplied in the *`keys[]`* vector then the *`length`* is twice the binary key length.

`WLAN_DRV_SET_CHANNEL`

The passed (`wlan_drv_channel_t`) value is used to configure the radio channel to be used for the interface. This is used for `WLAN_MODE_ADHOC` and `WLAN_MODE_MASTER` configurations.

Note

For `WLAN_MODE_INFRA` configurations the WiFi device will scan for the required Access Point on all the allowed channels, and so the channel set via this operation may not actually be used.

`WLAN_DRV_SET_FREQ`

This option is an alternative method for setting the WiFi channel as a frequency (specified in MHz).

`WLAN_DRV_SET_SCAN_START`

This key is used to initiate a wireless network scan. The underlying wireless device driver defines for how long a scan is actually active.

The passed (`wlan_drv_scan_t`) structure is used to define the callback context for the application supplied handler executed for each network found during the scanning process. A handler function pointer, with the prototype as defined in `wlan.h`:

```
typedef void (*wlan_drv_scan_callback_fn)(void *private,wlan_drv_scan_info_t *info);
```

is required, along with a private context pointer.

For each network found the handler is called with a suitable (`wlan_drv_scan_info_t`) *`info`* pointer. If the scan is terminated then a NULL *`info`* parameter is passed to the handler, and the wireless device driver scan support will no longer reference the registered callback.

The eCos scan support is designed to allow the WLAN driver to have a minimal memory footprint, and so the application is responsible for storing/caching any network information seen.

Note

To support this common eCos WLAN layer (allowing eCos applications to use WLAN functionality without having to have special handling for different chipset/driver combinations) when scanning there is a slight overhead in converting the driver specific scan results into the common eCos structure.

The application documented in the section called “wlan_scan” provides an example implementation using the eCos wireless network scan interface.

`WLAN_DRV_SET_SCAN_ABORT`

This key can be used to abort (terminate) an active scan.

`WLAN_DRV_SET_TXPOWER`

The passed (`wlan_drv_dbm_t`) dBm value is used to configure the interface transmit power.

`WLAN_DRV_SET_RTS`

The passsed (`wlan_drv_rts_t`) threshold value specifies the number of bytes in a network packet as the point where the RTS/CTS mechanism is initiated.

GET operations are used to acquire the current, active, wireless configuration.

`WLAN_DRV_GET_INIT`

A non-error return for this operation indicates a WLAN capable driver.

Note

The `CYGPKG_NET_WLAN` package requires that all eCos WLAN capable drivers provide support for a successful result from the `WLAN_DRV_GET_INIT` config key since it is used as an indicator that a specific Ethernet driver instance is a wireless interface. If `WLAN_DRV_GET_INIT` returns an error then the interface does *NOT* support the eCos wireless features.

`WLAN_DRV_GET_CHANNEL`

Returns the active (`wlan_drv_channel_t`) channel number.

`WLAN_DRV_GET_MODE`

Returns the interface (`wlan_drv_mode_t`) mode.

`WLAN_DRV_GET_ESSID`

This key returns the current SSID configuration in the referenced (`wlan_drv_essid_t`) structure.

`WLAN_DRV_GET_SECTYPE`

This key obtains the active (`wlan_drv_sectype_t`) security encoding setting.

`WLAN_DRV_GET_RATE`

Obtains the current (`cyg_drv_rate_t`) bps communication rate value.

`WLAN_DRV_GET_TXPOWER`

Obtains the current (`wlan_drv_dbm_t`) dBm transmission power setting.

`WLAN_DRV_VENDOR`

The interpretation of this key is *NOT* defined. It is provided as an extension hook for any driver (hardware) specific control operations that may be needed.

API

Name

wlan_diag_dump_ascii — Output human-readable string

Synopsis

#include <wlan.h>

`cyg_uint32` **`wlan_diag_dump_ascii`** `(`*`tag`*`, `*`addr`*`, `*`amount`*`);`

Description

Since ESSID and passphrase values may contain non-printable characters this helper function is just a wrapper to `diag_printf()` to display an arbitrary buffer in a human-readable form. It is purely provided for test and diagnostic use.

The *`tag`* parameter is optional and can be NULL. If non-NULL then the simple NUL-terminated string is output before decoding the supplied *`amount`* bytes of data from the *`addr`* memory reference.

Return value

The number of characters output.

Chapter 196. Testing

The default configuration provides a set of basic tests that can be built.

wlan_scan

The wlan_scan application provides an example of using the `WLAN_DRV_SET_SCAN_START` and `WLAN_DRV_SET_SCAN_ABORT` keys. It starts multiple scans (if required) over a configured time period reporting on wireless networks seen.

wlan_switch

The wlan_switch application provides an example of run-time access-point configuration and switching between access points. It shows the use of the `cyg_lwip_eth_ioctl()` interface to set various wireless configuration options and `WLAN_DRV_SET_COMMIT` to update the active setting.

Part LVIII. Cypress WWD WLAN

Table of Contents

Chapter 197. Cypress WWD overview

Introduction

The `CYGPKG_NET_WIFI_BROADCOM_WWD` package implements the eCos specific support for the Cypress (previously Broadcom) WICED Wireless Driver (WWD) sources present in (3rd-party) Cypress WICED-SDK releases. The WWD package is for use with the generic eCos WLAN (wireless networking) support layer to present the common eCos wireless API. The WICED-SDK is now owned and developer by Cypress., but was originally developed by Broadcom, hence the historic use of `BROADCOM` in names provided by this package.

The WWD package relies on the presence of the `CYGPKG_NET_WLAN`, `CYGPKG_NET_LWIP` and the `CYGPKG_IO_ETH_DRIVERS` networking packages. It also requires the eCos Kernel C API (`CYGFUN_KERNEL_API_C`) to provide the required thread support.

Note

The BSD network stacks `CYGPKG_NET_FREEBSD_STACK` and `CYGPKG_NET_OPENBSD_STACK` are not supported by this Cypress WWD support package.

Warning

Due to licensing restrictions the WICED-SDK package *cannot* be distributed as part of an eCos release. The developer is responsible for obtaining a supported WICED-SDK version via the normal Cypress channels. e.g. the WICED Studio developer website.

Currently only WICED-SDK-3.5.2 and WICED-Studio-6.4.0.61 are officially supported. Adding CDL support for future WICED-SDK releases is an easy process assuming no fundamental restructuring of the SDK is undertaken by Cypress.

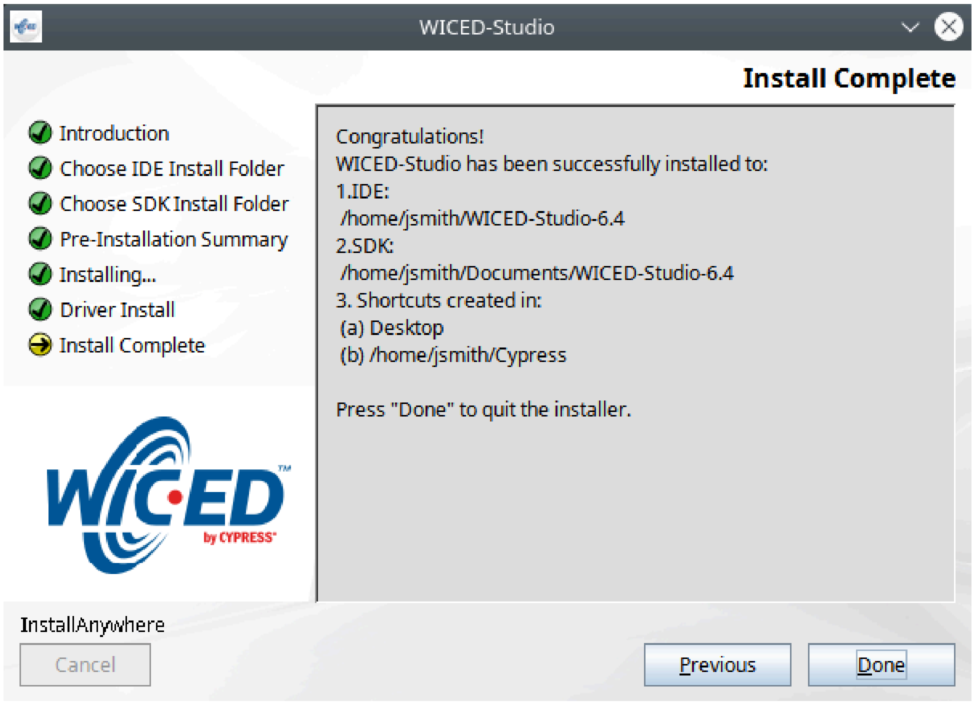
See the section called “WICED-SDK Installation” for a description of how to install the WICED-SDK.

NOTE: The WICED-SDK source tree must be installed into the correct eCosPro tree location prior to any target configuration via ecosconfig or configtool.

WICED-SDK Installation

After acquiring a supported Cypress WICED-Studio release package the SDK needs to be extracted into the correct location within your eCosPro release source tree. The WiFi SDK is written to your filesystem as part of the executable WICED Studio installer run. On completion of the installation process you should be presented with a window similar to Figure 197.1, “Example WICED-Studio installation complete” where the filesystem path location of the SDK should be noted. A copy of the installed SDK tree needs to be copied into the eCosPro tree.

Figure 197.1. Example WICED-Studio installation complete



Congratulations!
WICED-Studio has been successfully installed to:
1.IDE:
/home/jsmith/WICED-Studio-6.4
2.SDK:
/home/jsmith/Documents/WICED-Studio-6.4
3. Shortcuts created in:
(a) Desktop
(b) /home/jsmith/Cypress

Press "Done" to quit the installer.

The source path for the eCosPro tree depends on your specific installation, as does the specific eCosPro release `vsn` number. Similarly, the installer executable created WiFi SDK sub-directory is copied and renamed to conform to the package version naming scheme to allow identification of specific SDKs when multiple WICED SDK trees are installed into an eCosPro tree.

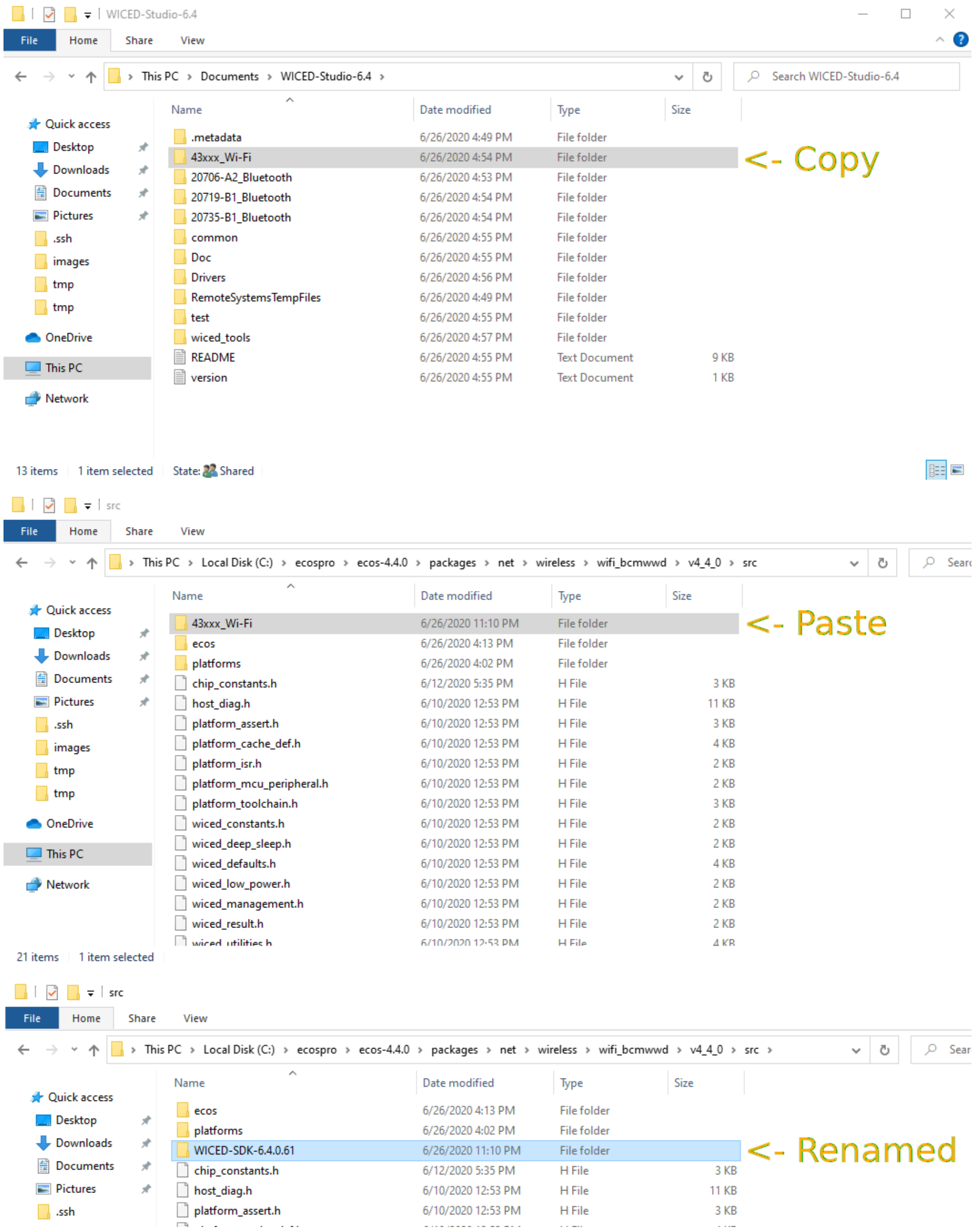
For Linux hosted development this can be achieved as follows:

```
$ cd /home/jsmith/Documents/WICED-Studio-6.4
$ tar cf - 43xxx_Wi-Fi | \
(cd path_to_ecospro_release/packages/net/wireless/wifi_bcmwwd/current/src/; tar xf -)
```

```
$ cd  path_to_ecospro_release/packages/net/wireless/wifi_bcmwwd/current/src
$ mv 43xxx_Wi-Fi WICED-SDK-6.4.0.61
```

For Windows hosted development the `43xxx_Wi-Fi` directory can be copied from its installed SDK location into the eCosPro source tree and renamed accordingly:

Figure 197.2. Example WICED-Studio WiFi directory copy and rename



This PC › Documents › WICED-Studio-6.4 ›

Name	Date modified	Type	Size
.metadata	6/26/2020 4:49 PM	File folder	
43xxx_Wi-Fi	6/26/2020 4:54 PM	File folder	
20706-A2_Bluetooth	6/26/2020 4:53 PM	File folder	
20719-B1_Bluetooth	6/26/2020 4:54 PM	File folder	
20735-B1_Bluetooth	6/26/2020 4:54 PM	File folder	
common	6/26/2020 4:55 PM	File folder	
Doc	6/26/2020 4:55 PM	File folder	
Drivers	6/26/2020 4:56 PM	File folder	
RemoteSystemsTempFiles	6/26/2020 4:49 PM	File folder	
test	6/26/2020 4:55 PM	File folder	
wiced_tools	6/26/2020 4:57 PM	File folder	
README	6/26/2020 4:55 PM	Text Document	9 KB
version	6/26/2020 4:55 PM	Text Document	1 KB

<- Copy

13 items 1 item selected State: Shared

This PC › Local Disk (C:) › ecospro › ecos-4.4.0 › packages › net › wireless › wifi_bcmwwd › v4_4_0 › src

Name	Date modified	Type	Size
43xxx_Wi-Fi	6/26/2020 11:10 PM	File folder	
ecos	6/26/2020 4:13 PM	File folder	
platforms	6/26/2020 4:02 PM	File folder	
chip_constants.h	6/12/2020 5:35 PM	H File	3 KB
host_diag.h	6/10/2020 12:53 PM	H File	11 KB
platform_assert.h	6/10/2020 12:53 PM	H File	3 KB
platform_cache_def.h	6/10/2020 12:53 PM	H File	4 KB
platform_isr.h	6/10/2020 12:53 PM	H File	2 KB
platform_mcu_peripheral.h	6/10/2020 12:53 PM	H File	2 KB
platform_toolchain.h	6/10/2020 12:53 PM	H File	3 KB
wiced_constants.h	6/10/2020 12:53 PM	H File	2 KB
wiced_deep_sleep.h	6/10/2020 12:53 PM	H File	2 KB
wiced_defaults.h	6/10/2020 12:53 PM	H File	4 KB
wiced_low_power.h	6/10/2020 12:53 PM	H File	2 KB
wiced_management.h	6/10/2020 12:53 PM	H File	2 KB
wiced_result.h	6/10/2020 12:53 PM	H File	2 KB
wiced_utilities.h	6/10/2020 12:53 PM	H File	4 KB

<- Paste

21 items 1 item selected

This PC › Local Disk (C:) › ecospro › ecos-4.4.0 › packages › net › wireless › wifi_bcmwwd › v4_4_0 › src ›

Name	Date modified	Type	Size
ecos	6/26/2020 4:13 PM	File folder	
platforms	6/26/2020 4:02 PM	File folder	
WICED-SDK-6.4.0.61	6/26/2020 11:10 PM	File folder	
chip_constants.h	6/12/2020 5:35 PM	H File	3 KB
host_diag.h	6/10/2020 12:53 PM	H File	11 KB
platform_assert.h	6/10/2020 12:53 PM	H File	3 KB

<- Renamed

Note

The WICED Studio installed WiFi SDK directory `43xxx_Wi-Fi` is renamed above to reflect the version number of the WICED Studio installer executable used.

With the SDK tree copied and renamed as above it is now available for eCos configuration.

Previously the WICED-SDK archives were distributed by Broadcom as `.7z` packages (e.g. version 3.5.2).

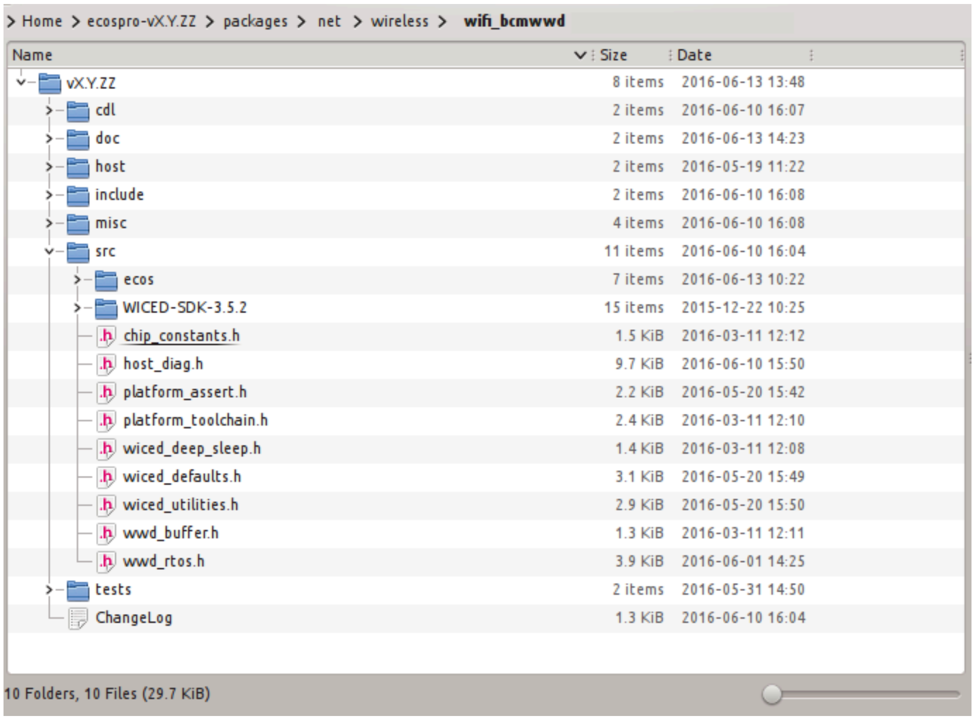
The WICED-SDK sources need to be installed into the `src` sub-directory of the `CYGPKG_NET_WIFI_BROADCOM_WWD` package within the eCosPro release tree.

Using the `WICED-SDK-3.5.2.7z` package as an example, the following is a command-line example of extracting into the eCosPro tree. The source path for the eCosPro tree depends on your specific installation, as does the specific eCosPro release *`vsn`* number:

```
$ cd /path_to_ecospro_release/packages/net/wireless/wifi_bcmwwd/current/src
$ 7z x ~/Downloads/WICED-SDK-3.5.2.7z

7-Zip [64] 9.20  Copyright (c) 1999-2010 Igor Pavlov  2010-11-18
p7zip Version 9.20 (locale=en_GB.UTF-8,Utf16=on,HugeFiles=on,8 CPUs)

Processing archive: /home/user/Downloads/WICED-SDK-3.5.2.7z

Extracting  WICED-SDK-3.5.2/tools/common/OSX/aes_cbc_128
Extracting  WICED-SDK-3.5.2/tools/common/Linux32/aes_cbc_128
[ … 7z output elided … ]
Extracting  WICED-SDK-3.5.2/apps
Extracting  WICED-SDK-3.5.2

Everything is Ok

Folders: 1055
Files: 6954
Size:       878793443
Compressed: 121342379
```

Assuming an example eCosPro release with version number `vX.Y.ZZ`, and using release 3.5.2 of the WICED-SDK, you should end up with a file hierarchy similar to the following figure:

Figure 197.3. Example WICED-SDK installation



> Home > ecospro-vX.Y.ZZ > packages > net > wireless > **wifi_bcmwwd**

Name	Size	Date
vX.Y.ZZ	8 items	2016-06-13 13:48
cdl	2 items	2016-06-10 16:07
doc	2 items	2016-06-13 14:23
host	2 items	2016-05-19 11:22
include	2 items	2016-06-10 16:08
misc	4 items	2016-06-10 16:08
src	11 items	2016-06-10 16:04
ecos	7 items	2016-06-13 10:22
WICED-SDK-3.5.2	15 items	2015-12-22 10:25
chip_constants.h	1.5 KiB	2016-03-11 12:12
host_diag.h	9.7 KiB	2016-06-10 15:50
platform_assert.h	2.2 KiB	2016-05-20 15:42
platform_toolchain.h	2.4 KiB	2016-03-11 12:10
wiced_deep_sleep.h	1.4 KiB	2016-03-11 12:08
wiced_defaults.h	3.1 KiB	2016-05-20 15:49
wiced_utilities.h	2.9 KiB	2016-05-20 15:50
wwd_buffer.h	1.3 KiB	2016-03-11 12:11
wwd_rtos.h	3.9 KiB	2016-06-01 14:25
tests	2 items	2016-05-31 14:50
ChangeLog	1.3 KiB	2016-06-10 16:04

10 Folders, 10 Files (29.7 KiB)

Chapter 198. Configuration

This chapter shows how to include the WWD support into an eCos configuration, and to configure it once installed.

Configuration Overview

The WWD driver is contained in the package `CYGPKG_NET_WIFI_BROADCOM_WWD`. However, it depends on the services of a collection of other packages for complete functionality. Currently the WWD implementation is tightly bound with the lwIP TCP/IP networking stack provided by the `CYGPKG_NET_LWIP` package.

Incorporating the WWD support into your application is straightforward. The essential starting point is to incorporate the WWD eCos package (`CYGPKG_NET_WIFI_BROADCOM_WWD`) into your configuration.

This may be achieved directly using **ecosconfig add** on the command line, or the **Build->Packages…** menu item within the eCos Configuration Tool.

Note

Prior to the first configuration attempt the 3rd-party WICED-SDK sources need to be installed into the eCosPro source tree as per the section called “WICED-SDK Installation”.

The package also relies on specific WWD functionality being provided by the platform (or variant) HAL for supported targets. See Chapter 199, *Platform/Variant HAL* for an overview of the HAL support required.

Chipset Firmware

The Cypress/Broadcom WICED WiFi chipsets require a binary firmware image to be downloaded to the WiFi chip for correct operation. For some platforms the required binary firmware image is available as part of the installed WICED SDK. However, the WICED SDK does not have support all platforms and the configuration world allows for the firmware binary images to be located outside the user installed WICED SDK tree.

For example the Raspberry Pi platforms use binary images included as part of the `CYGPKG_NET_WIFI_BROADCOM_WWD` package. For reference these images are located in individual sub-directories in the eCosPro `packages/net/wireless/wifi_bcmwwd/`*`version`*`/src/platforms` directory.

pi43430 This directory holds the 3rd party binary firmware image for the Raspberry Pi 3B and and Raspberry Pi 0W platforms.

pi43455 This directory holds the 3rd party binary firmware image for the Raspberry Pi 3B+ platform.

A `README.txt` is present in each sub-directory describing from where the 3rd-party binary blobs were acquired.

When porting the WWD driver to a new platform it may be necessary to acquire a suitable binary-blob to match the Cypress chipset being used and the hardware I/O configuration of the platform.

Configuration Options

Once added to the eCos configuration, the WWD package has a number of configuration options. As well as the core CDL options for the common WWD support, a WICED-SDK version specific set of CDL options are also available.

`CYGFUN_NET_WIFI_BROADCOM_WWD_RESOURCES_INDIRECT`

Most Cypress/Broadcom WICED WiFi chipsets require a binary firmware image to be downloaded to the WiFi chip.

Where the platform provides the necessary support, this option can be enabled to configure the application to access non-memory-mapped resource images. If disabled then the resource binaries are built into the application binary. For example, with platforms where limited amounts of (on-chip) direct memory are available the size of the required firmware binary blob could severely limit the usable application code size.

Please read the relevant platform specific documentation for details of any requirements, or processes, involved in initialising the non-memory-mapped firmware storage.

`CYGPKG_NET_WIFI_BROADCOM_WWD_MODULE`

This option allows the selection of the base WICED module to be targetted. Normally the configured eCos target (platform HAL) configuration will force a suitable value, and the developer should not need to manually configure this setting.

Similarly, depending on the configured module, the `CYGPKG_NET_WIFI_BROADCOM_WWD_CHIPSET` option defines the Cypress/Broadcom WiFi chipset to be configured.

`CYGIMP_NET_WIFI_BROADCOM_WWD_ACCESS`

This option selects the transport bus used between the WICED module host CPU and the WiFi controller Normally the configured eCos target (platform HAL) configuration will force a suitable value, and the developer should not need to manually configure this setting.

`CYGPKG_NET_WIFI_BROADCOM_WWD_WICED_SDK`

This is the main option for selecting the supported WICED-SDK version to be used. As mentioned in the section called “WICED-SDK Installation” the relevant WICED-SDK source should be present in the package `src` sub-directory.

As appropriate, WICED-SDK version specific CDL options are then made available to the system. These are not normally user-editable since they are primarily hooks for the config and build systems.

`CYGDBG_WIFI_BROADCOM_WWD_DEBUG`

This option, and its associated sub-options, are primarily for development of the driver internals. It is not normally expected that the application developer making use of the WWD package will ever need to enable the debug features.

If enabled this option provides access to a set of individually controllable diagnostic options. These can be used to provide low-level `diag_printf()` diagnostic output. It should be noted that some diagnostics if enabled will adversely affect the performance of the system such that incorrect behaviour can result.

As well as the WWD package specific options, the lwIP configuration will affect how the WWD driver is configured. For example the calculated `CYGNUM_NET_WIFI_BROADCOM_WWD_MTU` value for the MTU of the WWD layer is limited to what will fit within a single lwIP buffer, and so the setting is based on the relevant lwIP package configuration. The developer may need to tune the lwIP memory footprint to reflect both the expected usage of the system and the available memory at run-time.

Chapter 199. Platform/Variant HAL

To correctly operate the WWD driver needs to have some platform specific support that identifies the WICED-SDK features needed to support a specific Cypress/Broadcom chipset configuration. This chapter is primarily aimed at developers that need to provide WWD support on platforms without existing WWD support.

Some manifests control the features that are provided by the platform, and may be required depending on the specific WICED module being supported.

Table 199.1. WICED options

Manifest	Description
`WICED_CPU_CLOCK_HZ`	Provides the eCos run-time clock frequency to the WICED system.
`WICED_WIFI_USE_GPIO_FOR_BOOTSTRAP_0`	Specifies that a GPIO pin is provided for the `BOOTSTRAP_0` startup configuration of the WiFi chip.
`WICED_WIFI_USE_GPIO_FOR_BOOTSTRAP_1`	Specifies that a GPIO pin is provided for the `BOOTSTRAP_1` startup configuration of the WiFi chip.
`WICED_WIFI_OOB_IRQ_GPIO_PIN`	Out-Of-Band (asynchronous) interrupt pin provided.
`WICED_USE_WIFI_POWER_PIN`	WiFi chip power control pin provided.
`WICED_USE_WIFI_POWER_PIN_ACTIVE_HIGH`	Selects active-HIGH polarity for the power control pin, otherwise is active-LOW.
`WICED_USE_WIFI_RESET_PIN`	WiFi chip reset control pin provided.
`WICED_USE_WIFI_32K_CLOCK_MCO`	32kHz clock provided.

As well as the WICED feature configuration, the platform needs to define the set of hardware features as present on the targetted platform. Only the necessary subset of these will be defined, as dictated by how the WiFi chipset signals are wired on the target platform.

Table 199.2. Hardware manifests

Manifest	Description
`CYGHWR_HAL_STM32_WWD_MCO1`	Pin descriptor for clock source.
`CYGHWR_HAL_PLF_CLOCK_MCO_32K`	Platform specific function used to configure the 32kHz clock source.
`CYGHWR_HAL_PLF_WWD_PIN_POWER`	GPIO pin descriptor for power control.
`CYGHWR_HAL_PLF_WWD_PIN_RESET`	GPIO pin descriptor for reset control.
`CYGHWR_HAL_PLF_WWD_PIN_32K_CLK`	Pin descriptor for 32kHz clock source.
`CYGHWR_HAL_PLF_WWD_PIN_32K_GPIO`	GPIO pin descriptor for 32kHz clock source.
`CYGHWR_HAL_PLF_WWD_PIN_BOOTSTRAP_0`	GPIO pin descriptor for bootstrap 0.
`CYGHWR_HAL_PLF_WWD_PIN_BOOTSTRAP_1`	GPIO pin descriptor for bootstrap 1.
`CYGHWR_HAL_PLF_WWD_PIN_SPI_IRQ`	GPIO pin descriptor for SPI transport bus interrupt.
`CYGHWR_HAL_PLF_WWD_PIN_SPI_CS`	GPIO pin descriptor for SPI transport bus chip-select.
`CYGHWR_HAL_PLF_WWD_PIN_SDIO_OOB_IRQ`	GPIO pin descriptor for SDIO transport bus interrupt.
`CYGHWR_HAL_PLF_WWD_SDIO_ALIGN4`	Manifest to control word-aligned transport bus transfers. This manifest may be needed for some platforms if the underlying

Manifest	Description
	platform transport bus implementation requires word-aligned memory buffers (e.g. due to DMA limitations).
`CYGHWR_HAL_PLF_WWD_SDIO_SPEED`	Manifest defining the platform SDIO clock rate.

If the platform can support “out of application” storage of the WiFi firmware then the CDL option `CYGFUN_NET_WIFI_BROADCOM_WWD_RESOURCES_INDIRECT` will be configured, and the platform is then expected to provide manifests that map to the low-level firmware storage support. Normally such a large binary firmware image will be held in off-chip memory; but for certain architectures it may just be held in a specific region of some on-chip flash memory.

Table 199.3. Indirect firmware access

Manifest	Description
`HAL_PLF_WIFI_BROADCOM_WWD_INDIRECT_FW_READ`	The platform specific function used to read an amount of raw firmware data from a logical byte offset.
`HAL_PLF_WIFI_BROADCOM_WWD_LL_INIT`	If required the function to be called to perform low-level memory system initialisation prior to the WWD layer performing indirect firmware reads.

Part LIX. Common Clock Services

Table of Contents

Chapter 200. Overview

Introduction

The Common Clock package (`CYGPKG_CLOCK_COMMON`) provides a centralized management interface for system time, also known as calendar time, which corresponds to the real time of day. System time is distinct from other clock-related concepts such as the kernel "real time clock" which drives the kernel scheduler, or hardware timers.

The purpose of this package is primarily to keep system time management and related functionality focused in a central location to ensure a consistent and coherent approach to time management across all eCos packages, and to avoid duplication. It is used by the C library and by the POSIX compatibility layer for underlying time support, and also provides interfaces to manipulate time which may be used by services such as NTP, as well as to the user. It encapsulates the underlying eCos wallclock driver layer.

At the present time, it only operates when using the eCos kernel package, as it uses a dedicated thread to perform time management functions.

It has been verified safe for use past year 2038 - the so-called Year 2038 problem refers to the overflow of signed 32-bit representations of time in seconds since 1970-01-01 00:00:00 UTC.

Functionality

The package provides an API which presents functions to the user or higher layers to:

- Get the system time;
- Set the system time;
- Adjust the system time by a small amount, manage such adjustments, and potentially manage other aspects of clock discipline;
- Receive notifications of changes to system time;
- Instruct the package to update a wallclock (RTC) device configured into eCos with the current system time;
- Provide a kernel C API clock handle linked to the clock which is driving system time (which may or may not be the same as the real time clock driving the kernel scheduler). This can in turn be used to set alarms associated with ticks of that clock;
- Convert between system time clock ticks and calendar time, both in the form of relative time offsets, or absolute timestamps.

Concepts and structure

The main focus of the architecture of this package is the clock management thread. This thread acts as a central processing point in the system to: asynchronously initialize the clocks at system startup time; update the wallclock time either if directed by the user, or if required to by configuration settings (at regular periods, or whenever time is updated); and notify registered modules when the system time has been set or updated.

This thread should be a high priority thread as it is inherently time-sensitive, and the behaviour may not be as expected if pre-empted. For example, after the time has been changed, registered modules will be notified, but a higher priority thread could pre-empt and run before the notification has taken place, with the potential for different parts of the system to have different understandings of what the current time is. Although be aware there is a chance this can happen anyway as a potential consequence of the notification process, as obviously only one module can be notified at a time, meaning there can be a lag between the time change and notification.

The API functions of this package represent time using the POSIX standard defined structure struct timespec. This offers the most flexible and high resolution representation. The type is defined in the header file `<time.h>` and the standard mandates its definition as follows:

```
struct timespec {
    time_t tv_sec;    /* Seconds */
    long   tv_nsec;   /* Nanoseconds */
}
```

All uses of absolute time values are expressed using time as a struct timespec with an "epoch" (reference start date) of 1970-01-01 00:00:00 UTC. All time is based in the UTC time frame - no support for timezones is provided, which is instead left to higher layers.

Overall, the system time is maintained by reading a high resolution HAL clock, and maintaining a fixed relationship between that clock value and the set time. Global variables are used within the common clock package to allow for this, and are protected from simultaneous access, thus making the package thread safe. However no functions in this package can be considered safe to call from either an ISR or a DSR. At the beginning of system operation, interlocks are used to ensure that it is not possible to retrieve the system time until the common clock package has been initialized.

Time can be adjusted by a small amount - the maximum allowable range being a parameter set in the package configuration. When this happens, the clock is not set to the new time immediately ("stepped") but instead it gradually converges towards the new time base. It will still increase monotonically. Making changes to time in this way can be much less disruptive to applications, however it is not appropriate for large changes, as it would take either take too long to converge, or spend a significant duration with the time being significantly inaccurate. Time can be adjusted both forwards and backwards.

Conversion functions are available to convert between real time and tick counts for the kernel clock object associated with the clock which is driving system time. These use the clock converter functions provided in the eCos kernel.

Chapter 201. Dependencies

HAL

This package relies on the HAL clock API. It is strongly recommended that the HAL clocks are implemented using the full newer version of the HAL clock API, and not the older backward compatibility API (`HAL_CLOCK_INITIALIZE()`, `HAL_CLOCK_READ()`, `HAL_CLOCK_RESET()`). The package will still operate with the older form of API implementation, but it is likely to be less accurate, and possibly slightly slower.

The HAL clock API provides a high-resolution interface to an appropriate hardware clock, allowing fine-grained access with subtick access. Naturally, the resolution and accuracy that this package is able to provide is subject to what the hardware and HAL clock implementation provides. It is desirable, if possible, for the HAL clock to be configured with auto-reloading on interrupt. This reduces clock drift.

Kernel

At the present time, the common clock package relies on kernel facilities. It is hoped in future that this dependency will be removed.

However at the present time, a clock management thread is used to provide an asynchronous route for managing clock operations. This can be important if the package is required to update the wallclock, since with some wallclock drivers, it can take over one second to update the wallclock time in hardware, and it would be inappropriate for applications to stall and wait for such operations to complete.

It also allows modules to be notified that time has changed, either by a step change or by gradual fine adjustment. It can either use a user-supplied callback function, or broadcast a nominated kernel condition variable as the mechanism for notification. When it does so, it passes both the newly set time, and the cumulative time offset since the last received notification.

If a callback function has been provided, it should return as soon as possible, as no further notifications or other processing by the clock management thread can proceed until control is returned to it.

At present, the only underlying HAL clock supported must be the same clock as that used by the kernel, however the API and implementation have been designed to accommodate alternative clocks, and very little is required to complete such an implementation, once an appropriate platform which requires it has been selected, so that it can be verified. Users of this package should certainly not assume that the clock being used to control system time will continue to be the same as the kernel's clock.

The package also makes use of the kernel clock conversion facilities in order to translate clock ticks to or from real time in a scalable way when calling this package's tick conversion functions. The HAL clock API also provides a mechanism to convert ticks to and from real time, but this is not designed to scale beyond nanosecond times, nor tick values greater than 32-bits in width.

It is strongly recommended that users use the time conversion functions provided by this package, not only to avoid duplication, but also because in the case of conversion of absolute time values, the values returned can be affected by any fine adjustment in progress at that time.

Wallclock (RTC)

This package can optionally be configured to use a wallclock (RTC) device driver if configured in the system (the `CYGPKG_CLOCK_COMMON_USE_WALLCLOCK` CDL configuration option). If enabled, time will be retrieved from the wallclock device when the system starts, which is then used to initialize system time.

If no wallclock device is present in the configuration, then time will be initialized to a default date of 2012-01-01 00:00:00 UTC, although obviously it can then be subsequently set to the correct time by the user, or by services such as a network time client.

The user is free to trigger an update of the wallclock from the system time with the API function `cyg_clock_sync_wallclock()`. This will cause the clock management thread to wake up and set the wallclock time. An argument to the function indicates whether to do so synchronously (i.e. to wait for completion) or asynchronously (complete in the background).

Alternatively, CDL configuration options are provided to force the wallclock to be updated whenever time is set. However this can be expensive if time is updated frequently, which can sometimes happen, for example, with Network Time Protocol (NTP) clients.

A second alternative is also provided to periodically wake the clock management thread up regularly after a configured number of kernel ticks.

C library and POSIX layers

The C library time functions, and POSIX layer clock and timer functions will always use this package if present in the configuration. Obviously there can only be one notional system time, so when this package is present, no other means of providing a system time is used.

It also avoids significant duplication. In particular, similarly implemented clock conversion functions had existed in multiple locations prior to this package being used.

Chapter 202. Configuration

The following properties may be configured in this package using the configuration tool:

Management thread priority (CYGNUM_CLOCK_COMMON_THREAD_PRIO)

> This specifies the thread priority of the clock management thread. It is strongly recommended that this be a high priority (smaller number) as this thread is intrinsically sensitive to time delays.

Management thread stack size (CYGNUM_CLOCK_COMMON_THREAD_STACK_SIZE)

> This sets the stack size for the clock management thread. It must be large enough to accommodate calls into a wallclock driver (potentially in turn using bus drivers such as I²C), as well as any potential registered user callback functions. Note the value of this option is text, not a number.

Fine clock adjustment (CYGFUN_CLOCK_COMMON_ADJTIME)

> This component is used to allow fine adjustment of clock times, using the `cyg_clock_adjust_systime()` API function. This allows system time to be gradually adjusted to a new time, rather than there being a step change by just setting the time. Some extra overhead is incurred when getting and setting time if this is enabled, so it is disabled by default.

Maximum adjustment range (CYGNUM_CLOCK_COMMON_ADJTIME_MAX_RANGE)

> If fine clock adjustment is enabled, this option specifies the limit to how far the clock is allowed to be fine adjusted, measured in milliseconds. Requests to adjust beyond this range are rejected. Setting the range too large results in the clock remaining quite inaccurate over a longer time, and setting the time directly would be more relevant. The default is +/- 128ms which corresponds to the norm for NTPv3 (RFC1305) compliance.

Adjustment period as shift (CYGNUM_CLOCK_COMMON_ADJTIME_PERIOD_SHIFT)

> This option is used to set the period of time over which the time should be adjusted. After this time has elapsed, the clock adjustment will be complete. The value of this option is used as a shift value to give a power of two in nanoseconds. A value of 1 corresponds to 2ns, a value of 2 corresponds to 4ns, a value of 3 corresponds to 8ns, and so on. As a convenience, a calculated CDL variable CYGNUM_CLOCK_COMMON_ADJTIME_PERIOD_SECS is provided alongside to see the effect of different settings of this variable. Having to use a power of two may seem restrictive, but it avoids some particularly expensive calculations which would be detrimental to efficiency and accuracy.

Adjustment period as secs (CYGNUM_CLOCK_COMMON_ADJTIME_PERIOD_SECS)

> This configuration point is not actually used by the package, and exists purely as a convenience to the user. It uses the setting of CYGNUM_CLOCK_COMMON_ADJTIME_PERIOD_SHIFT to report an approximation of the corresponding period of time in seconds over which the time should be adjusted, to allow inspection of the effects of different shift values. The value displayed here is rounded down to the nearest second, although the actual value would not be.

Use wallclock (CYGPKG_CLOCK_COMMON_USE_WALLCLOCK)

> Leave this option enabled if a wallclock (RTC) device driver is to be used to maintain time persistently. Time will be initialized using the wallclock driver. If this option is disabled, then a default date of 2012-01-01 00:00:00 is used when starting up, but the time can be set by the application (for example based on user input, or via an external process such as a network time client), and can then be maintained, even though the time will be lost on power off.

Wallclock updates (CYGPKG_CLOCK_COMMON_WALLCLOCK_UPDATES)

> When this component is enabled, the wallclock can be updated by this package based on the system time held in this package. The user can request an update using the `cyg_clock_sync_wallclock()` API function, along with other automatic methods which can be enabled in the following options.

Always update when set (`CYGSEM_CLOCK_COMMON_ALWAYS_UPDATE_WALLCLOCK`)

> This option defaults to disabled, but if it is enabled, whenever the system time is set it always results in the wallclock also being updated immediately afterwards. This could result in unnecessary load if the time is updated often, or delays if parts of the system need to wait for the update to complete.

Regular update interval (`CYGNUM_CLOCK_COMMON_WALLCLOCK_UPDATE_PERIOD`)

> When this option is enabled, the wallclock will be updated periodically. The value of this option gives the period between updates, measured in kernel ticks. The default value of 720000 would correspond to 2 hours if the kernel is using a 100Hz clock.

Tick conversions (`CYGFUN_CLOCK_COMMON_TICK_CONVERTERS`)

> Various subsystems, and users, may need to convert between system clock ticks and real time. This option enables the provision of functionality to do so. These functions also take account of any fine time adjustments made (if `CYGFUN_CLOCK_COMMON_ADJTIME` is enabled). However this functionality is not enabled by default as there is overhead in having and initialising the converters even if unused. Note that the system clock driving system time may not be the same clock as the kernel clock.

Chapter 203. API reference

Name

cyg_clock_get_systime() — Retrieve the current system time

Synopsis

#include <cyg/clock/api.h>

```
Cyg_ErrNo cyg_clock_get_systime(ts);
```

Description

Retrieves the current system time as a struct timespec and stores it in *ts*. This time is represented as a UTC offset from the Epoch, defined as 1970-01-01 00:00:00 UTC for consistency with POSIX.

Return value

This function returns a standard error code, as defined in `<errno.h>`, or ENOERR on success.

Name

cyg_clock_get_systime_res — Obtain the resolution of the system clock

Synopsis

#include <cyg/clock/api.h>

```
Cyg_ErrNo cyg_clock_get_systime_res(ts);
```

Description

This function can be called to obtain the resolution of the system clock. The resolution is expressed as the time difference (quantum) corresponding to the smallest increment the clock may make, in nanoseconds, and returned in the struct timespec pointed to by *ts*.

Due to rounding errors in calculations, and the possibly of a fine adjustment being in progress (if `CYGFUN_CLOCK_COMMON_ADJTIME` is enabled), it is not guaranteed that the difference between two sequential reads of system time will never be smaller than the resolution returned here.

Return value

This function returns a standard error code, as defined in `<errno.h>`, or ENOERR on success.

Name

cyg_clock_set_systime() — Sets the system clock to the supplied time

Synopsis

```
#include <cyg/clock/api.h>
```

```
Cyg_ErrNo cyg_clock_set_systime(ts);
```

Description

Sets the system time to the time represented in the struct timespec at *ts*. The time is treated as a UTC offset from the Epoch, defined as 1970-01-01 00:00:00 UTC for consistency with POSIX.

Upon setting, the clock management thread will wake up and notify any modules which registered for notifications of time changes. If `CYGSEM_CLOCK_COMMON_ALWAYS_UPDATE_WALLCLOCK` is enabled, it will also update the wallclock device.

Note

If another thread attempts to set the time before the previous time setting has been processed by the clock management thread (including notifications and updating wallclock), it will be forced to block and wait for the previous set operation to complete. This may also occur if the clock management thread is busy updating the wallclock in response to a periodic request as configured by `CYGNUM_CLOCK_COMMON_WALLCLOCK_UPDATE_PERIOD`.

Return value

This function returns a standard error code, as defined in `<errno.h>`, or ENOERR on success.

Name

cyg_clock_sync_wallclock() — Force the wallclock to be updated from system time

Synopsis

#include <cyg/clock/api.h>

```
Cyg_ErrNo cyg_clock_sync_wallclock(wait_for_completion);
```

Description

Calling this function instructs the clock management thread to update the time stored in the wallclock (RTC) device from the current system time.

Depending on the underlying wallclock hardware and driver implementation, this may block for an extended period, possibly over one second. If *wait_for_completion* is true, the function will not return until the wallclock has been updated. If *wait_for_completion* is false, it should not block, unless the eCos kernel is not in use. If there was a failure while updating the wallclock, this may only be reported if *wait_for_completion* is set.

Return value

This function returns a standard error code, as defined in `<errno.h>`, or ENOERR on success.

Name

cyg_clock_adjust_systime() — Adjust the system time

Synopsis

#include <cyg/clock/api.h>

`Cyg_ErrNo` **`cyg_clock_adjust_systime`**`(`*`adj`*`);`

Description

We supply a function to allow fine adjustment of the time. Unlike `cyg_clock_set_systime()`, these adjustments will not cause an abrupt step change but allow gradual adjustment, while still preserving the principle of the clock monotonically increasing.

The cyg_clock_time_adj_t type is defined by including `<cyg/clock/api.h>`, and its contents are as follows:

```
typedef struct cyg_clock_time_adj_s {
    struct {
        unsigned int set_offset : 1;
        unsigned int get_adjremaining : 1;
    } flags;
    struct timespec  offset;       // Time offset
    struct timespec  adjremaining; // Remaining adjustment (can be negative)
} cyg_clock_time_adj_t;
```

Note

Do not rely on the ordering of members of this structure remaining the same in future. Similarly, additional structure members are likely to be added in future.

To adjust to a fixed offset from the current time, the adjustment is set with the *`offset`* member of the struct. The *`set_offset`* flag is also set, to indicate the vality of the *`offset`* member. Time will be gradually adjusted until this offset is reached. The time over which the offset is reached is derived from the value of `CYGNUM_CLOCK_COMMON_ADJTIME_PERIOD_SHIFT`. The offset may be positive or negative (represented by a positive or negative *`tv_sec`* field for the struct timespec).

Setting the offset will immediately cause any modules registered for time change notifications to be notified of the new adjustment, with an indication that an adjustment, rather than a step change occurred.

Setting the *`get_adjremaining`* flag will cause this function to fill in the *`adjremaining`* field with the total adjustment remaining to be enacted on the system clock. In other words, an adjustment was requested, but not fully complete because the adjustment period has not been reached, and the field is set with the adjustment remaining.

If no adjustment remains, the contents of *`adjremaining`* will be set to 0.

If both *`get_adjremaining`* and *`set_offset`* flags are set, then the *`adjremaining`* field is filled in with the adjustment remaining after the adjustment specified by the *`offset`* field has been applied, which will always simply be identical to the requested offset.

To allow for forward compatibility, a cyg_clock_time_adj_t must be initialized with the macro `CYG_CLOCK_INIT_ADJ_T(`*`adjt_p`*`)` where *`adjt_p`* is of type cyg_clock_time_adj_t *.

Example

```
{
  Cyg_ErrNo             err;
  cyg_clock_time_adj_t  adj;
```

```
struct timespec       timediff;

get_time_difference(&timediff);

CYG_CLOCK_INIT_ADJ_T( &adj );
adj.flags.set_offset = 1;
adj.offset = timediff;

err = cyg_clock_adjust_systime( &timediff );
if ( ENOERR != err )
{
  etc. …
```

Return value

This function returns a standard error code, as defined in `<errno.h>`, or ENOERR on success. Notably, ERANGE will be returned if the offset exceeds the limit configured with `CYGNUM_CLOCK_COMMON_ADJTIME_MAX_RANGE`.

Name

Time change notification — Registering and deregistering for notification of changes to system time

Synopsis

#include <cyg/clock/api.h>

`Cyg_ErrNo` **`cyg_clock_timechange_register`**`(`*`info`*`);`

`Cyg_ErrNo` **`cyg_clock_timechange_deregister`**`(`*`info`*`);`

Description

These functions are used to allow users of this package to register and deregister for notifications of changes to system time. This may help allow the user to know when to update any timers or alarms.

Parameters which affect the method of delivering notifications are passed in within the *`info`* function argument. Other fields in that same structure are then used to pass information about time updates back to the user when a time change event occurs, which happens as a consequence of calls to either `cyg_clock_set_systime()` or to `cyg_clock_adjust_systime()`.

There are two primary mechanisms for notification: the user may supply a callback function; or the user can supply a condition variable, which they may then wait on to be signalled.

The cyg_clock_timechange_info structure is defined by including `<cyg/clock/api.h>` and has the following contents relevant to the user:

```
/* Forward definition to avoid circular dependency */
struct cyg_clock_timechange_info_s;

typedef struct cyg_clock_timechange_info_s cyg_clock_timechange_info;

typedef void (cyg_clock_timechange_cb_fn_t)( cyg_clock_timechange_info *changeinfo );

struct cyg_clock_timechange_info_s {
    CYG_ADDRWORD                  userdata;
    cyg_drv_mutex_t               *mutex;
    cyg_drv_cond_t                *cv;
    cyg_clock_timechange_cb_fn_t  *cb;

    /* VALUES ABOVE ARE SET BY USER BEFORE REGISTRATION
     * VALUES BELOW ARE SET BY THE CLOCK PACKAGE ON TIME CHANGE EVENTS
     */

    cyg_bool                      adjtime;
    struct timespec               newtime;
    struct timespec               offset;
};
```

Other members of this structure exist, but are private to the common clock package's implementation and do not form part of the API. They must not be modified by the user.

The cyg_clock_timechange_info structure passed in must be persistent while the registration is in effect, as it will be used by the common clock package to maintain the registration. The address of the same structure object must be passed in on deregistration.

cyg_clock_timechange_info details

Here is a description of the purpose of each of the cyg_clock_timechange_info structure members:

`userdata`

This is user-supplied data, which the user is free to set to any value if it may help uniquely identify the registration.

This value must be set prior to registration.

`mutex`

If non-NULL, a mutex protecting this structure's contents. Once registered with the clock package, it must be locked before reading/writing the cyg_clock_timechange_info structure to avoid any chance of simultaneous access. It is not mandatory in case users can guarantee non-simultaneous access by some other means.

This value must be set prior to registration.

`cv`

If non-NULL, a condition variable which should be signalled when time is updated. It must be associated with the above mutex. It should not be used if setting a callback function.

This value must be set prior to registration.

`cb`

If non-NULL, a user-supplied callback function to be called on time changes. It should not be used if the *`cv`* condition variable has been set to be signalled.

This value must be set prior to registration.

`adjtime`

The value of this boolean field is set by the common clock package on time change events. It will be `true` if is this is a fine adjustment (from `cyg_clock_adjust_systime()`), or false if the time has been set with `cyg_clock_set_systime()`.

If reading this field, but before unlocking the mutex (if applicable) it must be reset to `true`. This is because the clock package will only ever set it from `true` to `false`, never from `false` to `true`. This is so that if multiple time change events occur before the user has processed any of the events, an indication that the time was "set" will override any indication of fine adjustment.

If the time update was due to a fine adjustment (*`adjtime`* is true) then the time change will not necessarily have yet been reflected in what can be read from the system clock - the effect on the system clock happens over a period of time as described earlier.

This field does not need to be initialized prior to registration.

`newtime`

The new time at the point the time update happened. Note that the relevance of this depends on the real-time properties of the program - if the system has been busy performing other operations, including consequences of notifying other users, it may be quite a long time in the past by the time the user can process the event.

`offset`

The offset (difference) from the previous time. If it is positive, the time moved forwards, if negative, the time moved backwards.

It is a mandatory part of the API contract that after reading this value, but before unlocking the mutex (if applicable) it must be reset to 0. This is required so that if multiple time changes occur before any of the events are processed by the API user, the effect on *`offset`* can be cumulative - the clock package can simply modify the existing offset, thus guaranteeing that when the user reads the offset, it will be the offset since it was last read by the user.

Using a callback function

Some special care is required if using the callback function method for notifications.

If a callback function is used, it must be brief, as further users of this package who have registered a callback will not be notified until the callback returns. For this reason, and to improve determinism, and to avoid delaying clock package operations, the condition variable approach should be preferred.

The callback function method is still available however as it is realized that sometimes there may be no alternative.

Users should not call back into any time-keeping functions (in this package or others) from a callback function, in order to avoid re-entrancy issues. An exception is for the time conversion functions.

Even with a callback function, it is recommended to still use a mutex lock to prevent the timechange info structure being updated while it is in the process of being read; and if such a mutex is provided with the 'mutex' member, it will be locked before the callback is called, and unlocked after.

Return value

These functions return a standard error code, as defined in `<errno.h>`, or ENOERR on success.

Name

cyg_clock_sysclock_handle() — Return a handle to the system clock

Synopsis

#include <cyg/kernel/kapi.h> #include <cyg/clock/api.h>

```
cyg_handle_t cyg_clock_sysclock_handle();
```

Description

This function returns a handle to the clock object associated with the hardware clock driving the system time. This clock handle is usable with the kernel C API functions such as `cyg_clock_to_counter()`, and thereby with other kernel functions which use kernel counters such as kernel alarms.

However to be clear, this clock may or may not be the same clock as used for the kernel real-time clock. Users must avoid operations which could interfere with system operation, such as setting the clock resolution or deleting the clock.

Note that this function may be implemented as a macro, and therefore taking the address of this function is not supported.

This function is only supplied when the tick conversion functionality is enabled.

Return value

This function returns a cyg_handle_t which can be used as a handle for kernel C API clock functions. No errors are reported.

Name

Time conversions — Converting between clock ticks and calendar time

Synopsis

#include <cyg/clock/api.h>

Cyg_ErrNo **cyg_clock_ticks_to_time**(*ticks*, *ts*);

Cyg_ErrNo **cyg_clock_time_to_ticks**(*ts*, *ticks*, *roundup*);

void **cyg_clock_ticks_to_reltime**(*ticks*, *ts*);

Cyg_ErrNo **cyg_clock_reltime_to_ticks**(*ts*, *ticks*, *roundup*);

Description

These functions allow conversions between the 'ticks' of the clock which is driving system time, and calendar time values. These are intended to be used in conjunction with the clock identified by the clock handle returned by `cyg_clock_sysclock_handle()`.

`cyg_clock_ticks_to_time()` returns what the calendar time (which is defined as relative to the Epoch 1970-01-01 00:00:00) will be when the system time clock reaches the supplied tick count. If a fine adjustment is in progress, it will be taken into account.

`cyg_clock_time_to_ticks()` returns what the system time clock's tick value will be when the supplied system time (which is defined as relative to the epoch 1970-01-01 00:00:00) is reached. If *roundup* is `true`, *ticks* will be rounded up to the next tick; if *roundup* is `false`, it will be rounded to the nearest tick. If a fine adjustment is in progress, it will be taken into account.

`cyg_clock_ticks_to_reltime()` returns the relative time interval (as a struct timespec) corresponding to the supplied number of ticks. Note this does not take into account any effect of fine clock adjustment.

`cyg_clock_reltime_to_ticks()` returns the number of ticks corresponding to the relative time interval specified in the struct timespec *ts*. If *roundup* is `true`, *ticks* will be rounded up to the next tick; if *roundup* is `false`, it will be rounded to the nearest tick. Note this does not take into account any effect of fine clock adjustment.

These functions are only provided if the configuration option `CYGFUN_CLOCK_COMMON_TICK_CONVERTERS` is enabled.

Return value

These functions return a standard error code, as defined in `<errno.h>`, or `ENOERR` on success. Notably `cyg_clock_reltime_to_ticks()` may return ERANGE for values which cannot be converted.

Part LX. Object Loader

Name

CYGPKG_OBJLOADER — eCos Support for Dynamic Module Loading

Synopsis

#include <cyg/objloader/objload.h>

void ***cyg_ldr_open***(*open_stream*, *data*);

void **cyg_ldr_close**(*handle*);

char ***cyg_ldr_error***();

void ***cyg_ldr_find_symbol***(*handle*, *symbol*);

Description

Note

The Object Loader package does not support all processor architectures at present.

The Object Loader package provides support for dynamically loading executable modules into an eCos system. Modules may be loaded into memory from a variety of sources, linked in to the running system and entry points invoked to execute the code of the module. When the module is no longer required, it may be unloaded and the memory reused for other purposes or other modules.

This system is modelled most closely on the Linux kernel module mechanism, rather than Windows DLLs or Unix shared objects. As a result, it has a number of restrictions:

- Only modules written in C are supported. The Object Loader does not currently provide support for invoking static constructors and destructors, C++ exceptions, RTTI and other parts of the C++ runtime system.
- Automatic symbol resolution only works for references from a module into the main executable. References between modules are not supported, and resolution of unresolved symbols in the main executable to module symbols is not supported.
- Loaded modules need to be built using the same, or similar, configuration to the main system.
- Loaded modules should be built with the same or compatible compiler flags as the main system. There is one important exception. Some architectures including MIPS and Nios II implement a global pointer register. Small global variables are placed in an area of memory up to 64K. The gp register points at this area of memory, allowing the variables to be accessed directly using a single instruction instead of the two or more instructions that would otherwise be required. This technique cannot be used for a dynamically loaded module. Hence the use of gp-relative addressing must be suppressed with a compiler flag, typically -G0.

Creating Loadable Modules

Modules can be just object files as generated by the compiler. In a Makefile including the $(INSTALL_DIR)/include/pkgconf/ecos.mak definitions file, the entry to build module.o might be:

```
module.o: module.c
        $(ECOS_COMMAND_PREFIX)gcc -c -I$(INSTALL_DIR)/include $(ECOS_GLOBAL_CFLAGS) -o $@ $<
        $(ECOS_COMMAND_PREFIX)strip -g $@
```

The compile line generates a .o file. The -I option allows includes to be fetched from the eCos installation. The command prefix and global flags are stored in the ecos.mak file by the eCos build process. If the compile flags include -g or some other debug option then to save memory and maybe load time it is useful to pass the finished module through **strip** to limit the file contents to just the loadable ELF sections.

It is possible to create a module out of several object files by using the linker's ability to perform a partial link:

```
module.o : file1.o file2.o file3.o
        $(ECOS_COMMAND_PREFIX)gcc $(subst --gc-sections,-r,$(ECOS_GLOBAL_LDFLAGS)) -L$(PREFIX)/lib \
            -Tmodule.ld -o $@ $^
        $(ECOS_COMMAND_PREFIX)strip -g $@
```

The `module.ld` linker script is defined by the Object Loader package and is copied out to the install lib directory. It should be used when combining multiple files, or when advanced features such as HAL tables are used in a single object file.

If the module makes use of float, double, long long and some long arithmetic operations, then it should be partially linked against `libgcc` before loading. This can be done with the following makefile fragments:

```
# Single source file module…
module.o: module.c
        $(ECOS_COMMAND_PREFIX)gcc -c -I$(INSTALL_DIR)/include $(ECOS_GLOBAL_CFLAGS) -o $@.tmp $<
        $(ECOS_COMMAND_PREFIX)gcc $(subst --gc-sections,-r,$(ECOS_GLOBAL_LDFLAGS)) \
            -L`dirname \`$(ECOS_COMMAND_PREFIX)gcc $(ECOS_GLOBAL_CFLAGS) \
            -print-libgcc-file-name\`` -L$(PREFIX)/lib -Tmodule.ld -o $@ $@.tmp -lgcc
        $(ECOS_COMMAND_PREFIX)strip -g $@

# Combine multiple object files…
module.o : file1.o file2.o file3.o
        $(ECOS_COMMAND_PREFIX)gcc $(subst --gc-sections,-r,$(ECOS_GLOBAL_LDFLAGS)) \
            -L`dirname \`$(ECOS_COMMAND_PREFIX)gcc $(ECOS_GLOBAL_CFLAGS) \
            -print-libgcc-file-name\`` -L$(PREFIX)/lib -Tmodule.ld -o $@ $^ -lgcc
        $(ECOS_COMMAND_PREFIX)strip -g $@
```

Target Specific Considerations

There are a number of special considerations for particuar target architectures:

- Modules compiled for Thumb may be loaded into targets compiled for either ARM32 or Thumb. Thumb builds of eCos that use the object loader should have the "-mlong-calls" compiler option set. ARM32 builds should have thumb interworking enabled if thumb modules are to be loaded (the object loader module does this automatically). Thumb modules should be compiled with "-mthumb -mthumb-interwork -mlong-calls" compiler options. However, some later ARM variants do not need the "-mthumb-interwork" option since this is implicit in the architecture. For such targets this option need not be given.

- Modules compiled for ARM, Thumb or Thumb 2 may require the "-mlong-calls" compiler option if the module to be loaded will occupy a different region of the address space to the rest of the program. The most frequent scenario causing this to arise is if the main program runs from FLASH memory, but with the module loaded into RAM. If in doubt, use the option as it is always safe, and the only downside is a small code size and runtime execution penalty on function calls.

- Modules compiled for the MIPS16 instruction set may be loaded into a MIPS target, so long as the processor supports the instruction set. To compile and link such a module, the "-mips16" compiler option must be substituted for "-mips32", along with "-fwritable-strings".

- Modules compiled for NIOS II processors must be compiled with the "-G0" compiler option. This ensures that loaded modules do not make assumptions about the accessibility of small initialised data (".sdata") or small zero-initialised data (".sbss") relative to the address it was loaded at.

Loading Modules

The function `cyg_ldr_open()` is used to load a module into memory. It takes two arguments. The first argument defines a module loader, while the second argument is a generic `data` item whose value depends on the loader. If the load is successful, then a non-NULL handle will be returned. A NULL pointer will be returned on failure.

If there is an error in the loading process, then the function `cyg_ldr_error()` will return a string describing the last error that occurred. Note that this is not thread-safe since there is only a single last error recorded for all load operations.

At present the following loaders are implemented:

CYG_LDR_FILESYSTEM

This loader uses FILEIO operations to read an ELF file from a named file in a filesystem. For example, to read a module from the file "/lib/modules/module.o":

```
mod_handle = cyg_ldr_open( CYG_LDR_FILESYSTEM,
                           (CYG_ADDRWORD)"/lib/modules/module.o");
```

This loader is included by default if the `CYGPKG_IO_FILEIO` package is included, although it can be omitted by disabling `CYGPKG_OBJLOADER_LOADER_FS`.

CYG_LDR_MEMORY

This loader uses memory access primitives to read an ELF file from any addressable memory such as ROM, FLASH or RAM. For example to read a module from the location `module_base`:

```
mod_handle = cyg_ldr_open( CYG_LDR_MEMORY,
                           (CYG_ADDRWORD)&module_base );
```

This loader is included by default, although it can be omitted by disabling `CYGPKG_OBJLOADER_LOADER_MEM`.

Loaders `CYG_LDR_FTP`, `CYG_LDR_TFTP`, `CYG_LDR_HTTP` and `CYG_LDR_FLASH` are defined, but not currently implemented.

Unloading Modules

A module may be unloaded by calling `cyg_ldr_close()`, passing it the handle returned from `cyg_ldr_open()`. This will cause the memory occupied by the loader to be released. Any pointers into the code or data of the module will be rendered invalid and should not be used.

Referencing Module Symbols

When a module is loaded, a symbol table listing all the external symbols that it defines is loaded with it. The function `cyg_ldr_find_symbol()` searches this table and returns a pointer to the location defined by a symbol. For example, to create a thread running from a function in a module:

```
cyg_thread_entry_t *thread_entry;

thread_entry = cyg_ldr_find_symbol( handle, "thread_entry");

cyg_thread_create(THREAD_PRIORITY,
                  thread_entry,
                  0,
                  "Module Thread",
                  (void *)thread_stack,
                  THREAD_STACK_SIZE,
                  &thread_handle,
                  &thread_object);
```

Both functions and variables may be accessed in this way.

There is no mechanism for resolving dangling references in the main eCos application, or other modules, to symbols in a newly loaded module. The main eCos application must have all references resolved at link time. However, it is possible to simulate the effect of dynamic resolution by using function pointers. For example define a global function pointer to an initial dummy function:

```
typedef int module_fn_t(int a, int b);

module_fn_t dummy_fn;

module_fn_t *module_fn = dummy_fn;
```

```
int dummy_fn( int a, int b )
{
    return -1;
}
```

When the module is loaded the function pointer can be pointed at the function within the module, and pointed back to the dummy function when it is unloaded:

```
void *mod_handle;

void load_module(void)
{
    mod_handle = cyg_ldr_open( CYG_LDR_FILESYSTEM, (CYG_ADDRWORD)"/lib/modules/module.o");

    module_fn = cyg_ldr_find_symbol( mod_handle, "module_fn");
}

void unload_module(void)
{
    cyg_ldr_close( mod_handle );

    module_fn = dummy_fn;
}
```

One could even implement a form of demand loading by combining `dummy_fn` and `load_module`:

```
int dummy_fn( int a, int b )
{
    mod_handle = cyg_ldr_open( CYG_LDR_FILESYSTEM, (CYG_ADDRWORD)"/lib/modules/module.o");

    module_fn = cyg_ldr_find_symbol( mod_handle, "module_fn");

    return module_fn( a, b );
}
```

Module Open and Close Functions

When a module is loaded the Object Loader will look for a symbol with the name "module_open" and if found will call it with the following prototype:

```
void module_open( void );
```

Similarly, when `cyg_ldr_close()` is called, the Object Loader will look for a symbol named "module_close" and call it, with the same prototype.

External References

When a module is loaded, the Object Loader package performs any relocations it requires and resolves any unresolved symbol references it contains. The load only succeeds if all of these can be completed. For these symbols to be resolved it is necessary for the main eCos executable to contain a symbol table defining the symbols to be resolved. Normal eCos executables do not contain such a symbol table since it would occupy an unreasonably large amount of memory. There is also no mechanism to persuade the linker to include a loadable symbol table into the executable. Hence it is necessary for the application to explicitly define a symbol table that maps symbol names to addresses.

The loader provides an empty table; the user can then define additional entries required by any loadable modules. In order to keep the size of the table to a minimum, the user can selectively include only those functions that are expected to be used by the loader to resolve all references. There are several macros defined in `objload.h` for defining table entries:

`CYG_LDR_TABLE_FUN( name )`

This macro defines a table entry for a function with the given `name`.

`CYG_LDR_TABLE_VAR( name )`

This macro defines a table entry for a data variable with the given `name`.

`CYG_LDR_TABLE_ENTRY( entry_name, symbol_name, address )`

This is a low level macro that allow all aspects of a symbol table entry to be controlled. The *`entry_name`* argument defines the table entry object name (a C language requirement since anonymous objects are not permitted). The *`symbol_name`* argument is a string giving the symbol that will be matched by the loader. The *`address`* argument gives the memory location to which this symbol will resolve.

The `objload.h` file contains a number of macros that collect together groups of functions as a convenient way to include blocks of Kernel, C Library and FILEIO functionality. These include the following:

```
CYG_LDR_TABLE_KAPI_ALARM()
CYG_LDR_TABLE_KAPI_CLOCK()
CYG_LDR_TABLE_KAPI_COND()
CYG_LDR_TABLE_KAPI_COUNTER()
CYG_LDR_TABLE_KAPI_EXCEPTIONS()
CYG_LDR_TABLE_KAPI_FLAG()
CYG_LDR_TABLE_KAPI_INTERRUPTS()
CYG_LDR_TABLE_KAPI_MBOX()
CYG_LDR_TABLE_KAPI_MEMPOOL_FIX()
CYG_LDR_TABLE_KAPI_MEMPOOL_VAR()
CYG_LDR_TABLE_KAPI_MUTEX()
CYG_LDR_TABLE_KAPI_SCHEDULER()
CYG_LDR_TABLE_KAPI_SEMAPHORE()
CYG_LDR_TABLE_KAPI_THREAD()
CYG_LDR_TABLE_STRING()
CYG_LDR_TABLE_STDIO()
CYG_LDR_TABLE_INFRA_DIAG()
CYG_LDR_TABLE_FILEIO()
CYG_LDR_TABLE_NET()
```

Name

CYGPKG_OBJLOADER — Extending the Object Loader

Description

The Object Loader package has a number of features that allow it to be extended. To support a new CPU architecture a new relocator needs to be written. If ELF files are to be read from a source that differs from those currently supported, then a new loader needs to be written. Finally, the mechanism by which the loader allocates the memory used to store loaded sections can be redirected by the application.

Adding New Relocators

When the loader loads a new module some locations in it must be adjusted to account for the address at which is it loaded. References to external symbols must also be installed. The location and nature of these modifications are described by one or more sections in the ELF file which contain a sequence of relocation records. The exact meaning of the relocations that these records define is architecture specific and is usually described as part of the ABI for that CPU type.

To define a new relocator for a CPU, it is necessary to add an extra definition to the `objloader.cdl` file, and add a header and source file to the package. To support the XYZ CPU the following must be added to the `CYGPKG_OBJLOADER_ARCHITECTURE` component:

```
cdl_option CYGBLD_OBJLOADER_ARCHITECTURE_XYZ {
  display       "Support loading on XYZ processors"
  calculated    CYGPKG_HAL_XYZ
  implements    CYGINT_OBJLOADER_RELOCATOR
  define_proc {
     puts $::cdl_header "#include <cyg/objloader/relocate_xyz.h>"
  }
  compile relocate_xyz.c
}
```

The `relocate_xyz.h` file needs to define some macros to customize the loader:

`ELF_ARCH_MACHINE_TYPE`

This defines the value that the *e_machine* machine type field of the ELF header. If this does not match, the module load will fail.

`ELF_ARCH_ENDIANNESS`

This defines the value that the `EI_DATA` byte of the *e_ident* field of the ELF header. It should be either `ELFDATA2LSB` or `ELFDATA2MSB`. If this does not match, the module load will fail. Architectures that are bi-endian need to test either a compiler or a HAL definition to select the correct endianness for the current build.

`CYG_LDR_MAKE_LOCAL_ADDRESS( addr, sym )`

This macro is used to combine the section address of a symbol with information from its symbol table entry. The *addr* argument is the base address of the section in which the symbol is defined. The *sym* is the symbol table entry; this is not a pointer, so fields should be accessed using the "." operator, not the "->" operator. The return value should be of type void *. This is an optional macro, if it is not defined here then a default definition will be used which simply adds the *st_value* field of the symbol table entry to the address.

In addition to these, it may also contain definitions that are useful to the relocator. Typically the relocation record types and any support macros may be defined here.

The `relocate_xyz.c` file contains two functions:

`void cyg_ldr_flush_cache(void)`

This function is called to perform any cache flushing needed. The loader modifies code in memory that will subsequently be executed. It does this using data accesses, so it is essential that these updates are flushed from the data cache, and that stale entries are flushed from the instruction cache. This function must call appropriate HAL cache operations to ensure that this is done.

`cyg_int32 cyg_ldr_relocate(cyg_int32 rel_type, cyg_uint32 flags, cyg_uint32 mem, cyg_int32 sym_value)`

The loader will call this function for each relocation record in each relocation section found in the module.

The *rel_type* argument defines the relocation record type and will be one of the relocation types defined for the architecture. Most architecture ABIs define a large number of relocations, not all of which will be relevant to the use that eCos makes of the object file format. In general only a small subset of relocation types need to be handled which can usually be determined by inspecting the object files generated by compiling eCos.

The *flags* argument contains flags that provide additional information about the relocation record. At present only one flag is defined: `CYG_LDR_FLAG_RELA` which is set when the relocation is from a RELA record, otherwise it comes from a REL record.

The *mem* argument contains the address of the location in memory to be relocated. It is constructed from the base address of the segment targeted by the relocation section, plus the *r_offset* field from the relocation record.

The *sym_value* argument contains the address of any symbol associated with the relocation record. If it was a RELA record, then the contents of the *r_addend* field will have been added.

Adding new Loaders

The Object Loader package needs to fetch a module from some source to load it into memory. This is the job of a loader. A loader consists of an open function plus read, seek and close functions.

The loader open function is supplied as the first parameter to `cyg_ldr_open()`. It is called with the second argument as a parameter. On success it returns a pointer to a CYG_LDR_ELF_OBJECT object. On failure it returns NULL.

The open function has a number of duties, best described by an annotated example for the ABC loader:

```
__externC CYG_LDR_ELF_OBJECT *cyg_ldr_open_abc(CYG_ADDRWORD arg)
{
    // Allocate a CYG_LDR_ELF_OBJECT
    CYG_LDR_ELF_OBJECT * obj = (CYG_LDR_ELF_OBJECT *)cyg_ldr_malloc(sizeof(CYG_LDR_ELF_OBJECT));

    // Allocate a private data descriptor. Depending on the nature of
    // the loader this may not be necessary.
    struct abc_desc *desc = cyg_ldr_malloc(sizeof(struct abc_desc));

    // Check that the memory allocations worked
    if( obj == NULL || desc == NULL )
    {
        if( obj != NULL ) free(obj);
        if( desc != NULL ) free(desc);
        cyg_ldr_last_error = "ERROR IN MALLOC";
        return NULL;
    }

    // Perform any operations to enable access to the ELF file and
    // fill in the descriptor. If this fails then free both descriptor
    // and ELF object, set cyg_ldr_last_error and return NULL.

    // Clear the CYG_LDR_ELF_OBJECT
    memset( obj, 0, sizeof(CYG_LDR_ELF_OBJECT));

    // Install private data pointer
    obj->ptr    = (CYG_ADDRWORD)desc;

    // Install pointers to read, seek and close functions
    obj->read   = cyg_ldr_abc_read;
    obj->seek   = cyg_ldr_abc_seek;
    obj->close  = cyg_ldr_abc_close;

    // Return completed object
    return obj;
}
```

The read function will be called via the pointer in the ELF object whenever the Object Loader needs to read data from the file. It has the following definition:

```
static size_t cyg_ldr_abc_read(struct CYG_LDR_ELF_OBJECT* obj, size_t size, void* buf)
```

The *obj* argument is the ELF object returned from the open function. The *size* argument gives the number of bytes to be read and *buf* points to a location to store them. The function returns the number of bytes read.

The seek function will be called via the pointer in the ELF object whenever the Object Loader needs to reposition the point in the file at which the next read will occur. It has the following definition:

```
static cyg_int32 cyg_ldr_abc_seek(struct CYG_LDR_ELF_OBJECT* obj, cyg_uint32 offset)
```

The *obj* argument is the ELF object returned from the open function. The *offset* argument gives the number of bytes from the start of the file to which the read point should be moved. The function returns the new read offset. Some sources may not be able to reposition the read pointer backwards, and may only be capable of advancing it. If the reposition fails then this function should return -1.

The close function will be called via the pointer in the ELF object when the Object Loader has finished with the file. It has the following definition:

```
static cyg_int32 cyg_ldr_abc_close(struct CYG_LDR_ELF_OBJECT* obj)
```

The *obj* argument is the ELF object returned from the open function. This function should close down access to the file, free the private data descriptor if necessary and set the `obj->ptr` field to zero. It should not free the ELF object itself, the Object Loader will do this itself later. If the close succeeds then this function should return zero, and -1 if it fails.

Redirecting Memory Allocation

All memory allocation in the Object Loader is made via the `cyg_ldr_malloc()` function and it is freed via the `cyg_ldr_free()` function. These have the following prototypes:

```
__externC void *cyg_ldr_malloc(size_t) CYGBLD_ATTRIB_WEAK;

__externC void cyg_ldr_free(void *) CYGBLD_ATTRIB_WEAK;
```

These functions by default simply call the standard `malloc()` and `free()` heap functions. However, they are defined with the `weak` linker attribute. This means that the application can redefine these functions to provide an alternative allocation and free mechanism if, for example, the standard heap support has been omitted.

Part LXI. CPU load measurements

The cpuload package provides a way to estimate the cpuload. It gives an estimated percentage load for the last 100 milliseconds, 1 second and 10 seconds.

Table of Contents

Chapter 204. CPU Load Measurements

CPU Load API

The package allows the CPU load to be estimated. The measurement code must first be calibrated to the target it is running on. Once this has been performed the measurement process can be started. This is a continuous process, so always providing the most up to data measurements. The process can be stopped at any time if required. Once the process is active, the results can be retrieved.

Note that if the target/processor performs any power saving actions, such as reducing the clock speed, or halting until the next interrupt etc, these will interfere with the CPU load measurement. Under these conditions the measurement results are undefined. The synthetic target is one such system. See the implementation details at the foot of this page for further information.

SMP systems are supported and are described later.

The API for load measuring functions can be found in the file `cyg/cpuload/cpuload.h`.

cyg_cpuload_calibrate

This function is used to calibrate the cpu load measurement code. It makes a measurement to determine the CPU properties while idle.

```
void cyg_cpuload_calibrate(cyg_uint32  *calibration);
```

The function returns the calibration value at the location pointed to by *`calibration`*.

This function is quite unusual. For it to work correctly a few conditions must be met. The function makes use of the two highest thread priorities. No other threads must be using these priorities while the function is being used. The kernel scheduler must be started and not disabled. The function takes 100ms to complete during which time no other threads will be run.

cyg_cpuload_create

This function starts the CPU load measurments.

```
void cyg_cpuload_create(cyg_cpuload_t *cpuload,
                        cyg_uint32 calibrate,
                        cyg_handle_t *handle);
```

The measurement process is started and a handle to it is returned in *`*handle`*. This handle is used to access the results and the stop the measurement process.

cyg_cpuload_delete

This function stops the measurement process.

```
void cyg_cpuload_delete(cyg_handle_t handle);
```

`handle` should be the value returned by the create function.

cyg_cpuload_get

This function returns the latest measurements.

```
void cyg_cpuload_get(cyg_handle_t handle,
         cyg_uint32 *average_point1s,
         cyg_uint32 *average_1s,
         cyg_uint32 *average_10s);
```

`handle` should be the value returned by the create function. The load measurements for the last 100ms, 1s and 10s are returned in *`*average_point1s`*,*`*average_1s`* and *`*average_10s`* respectively.

Implementation details

This section gives a few details of how the measurements are made. This should help to understand what the results mean.

When there are no other threads runnable, eCos will execute the idle thread. This thread is always runnable and uses the lowest thread priority. The idle thread does little. It is an endless loop which increments the variable, `idle_thread_loops` and executes the macro `HAL_IDLE_THREAD_ACTION`. The cpu load measurement code makes use of the variable. It periodically examines the value of the variable and sees how much it has changed. The idler the system, the more it will have incremented. From this it is simple to determine the load of the system.

The function `cyg_cpuload_calibrate` executes the idle thread for 100ms to determine how much `idle_thread_loops` is incremented on a system idle for 100ms. `cyg_cpuload_create` starts an alarm which every 100ms calls an alarm function. This function looks at the difference in `idle_thread_loops` since the last invocation of the alarm function and so calculated how idle or busy the system has been. The structure `cyg_cpuload` is updated during the alarm functions with the new results. The 100ms result is simply the result from the last measurement period. A simple filter is used to average the load over a period of time, namely 1s and 10s. Due to rounding errors, the 1s and 10s value will probably never reach 100% on a fully loaded system, but 99% is often seen.

As stated above, clever power management code will interfere with these measurements. The basic assumption is that the idle thread will be executed un-hindered and under the same conditions as when the calibration function was executed. If the CPU clock rate is reduced, the idle thread counter will be incremented less and so the CPU load measurements will give values too high. If the CPU is halted entirely, 100% cpu load will be measured.

SMP Support

This section described how CPU load is measured on SMP systems. SMP support has been introduced without changing or extending the existing API. To achieve this, both `cyg_cpuload_calibrate()` and `cyg_cpuload_create()` query the CPU that they are running on and bind the cyg_cpuload_t object to that CPU. The remaining API calls, `cyg_cpuload_delete()` and `cyg_cpuload_get()`, can be called from any CPU.

To gather load information for each CPU in the system, a separate cyg_cpuload_t object must be allocated, calibrated and created for each CPU. The following example shows how this might be done during application initialization:

```

static cyg_uint32 calibration[HAL_SMP_CPU_COUNT];
static cyg_cpuload_t cpuload[HAL_SMP_CPU_COUNT];
static cyg_handle_t handle[HAL_SMP_CPU_COUNT];

static void create_cpuload( void )
{
    HAL_SMP_CPU_TYPE cpu;
    HAL_SMP_CPU_MASK old_affinity;

    // Get current thread affinity.
    cyg_thread_get_affinity( cyg_thread_self(), &old_affinity );

    for( cpu = 0; cpu < HAL_SMP_CPU_COUNT; cpu++ )
    {
        // Set this thread's affinity to single CPU.
        cyg_thread_set_affinity( cyg_thread_self(), 1<<cpu );

        // Calibrate and create cpuload object.
        cyg_cpuload_calibrate(&calibration[cpu]);
        cyg_cpuload_create(&cpuload[cpu],calibration[cpu],&handle[cpu]);
    }

    // Restore thread's original affinity.
    cyg_thread_set_affinity( cyg_thread_self(), old_affinity );
}
```

Following this, the following call, from any CPU, will return the load for the given CPU:

```
cyg_cpuload_get(handle[cpu],&average_point1s,&average_1s,&average_10s);
```

Part LXII. gprof Profiling Support

Name

`CYGPKG_PROFILE_GPROF` — eCos Support for the gprof profiling tool

Description

The GNU gprof tool provides profiling support. After a test run it can be used to find where the application spent most of its time, and that information can then be used to guide optimization effort. Typical gprof output will look something like this:

```
Each sample counts as 0.003003 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 14.15       1.45     1.45   120000    12.05    12.05  Proc_7
 11.55       2.63     1.18   120000     9.84     9.84  Func_1
  8.04       3.45     0.82                             main
  7.60       4.22     0.78    40000    19.41    86.75  Proc_1
  6.89       4.93     0.70    40000    17.60    28.99  Proc_6
  6.77       5.62     0.69    40000    17.31    27.14  Func_2
  6.62       6.30     0.68    40000    16.92    16.92  Proc_8
  5.94       6.90     0.61                             strcmp
  5.58       7.47     0.57    40000    14.26    26.31  Proc_3
  5.01       7.99     0.51    40000    12.79    12.79  Proc_4
  4.46       8.44     0.46    40000    11.39    11.39  Func_3
  3.68       8.82     0.38    40000     9.40     9.40  Proc_5
  3.32       9.16     0.34    40000     8.48     8.48  Proc_2
…
```

This output is known as the flat profile. The data is obtained by having a hardware timer generate regular interrupts. The interrupt handler stores the program counter of the interrupted code. gprof performs a statistical analysis of the resulting data and works out where the time was spent.

gprof can also provide information about the call graph, for example:

```
index % time    self  children    called     name
…
                0.78    2.69   40000/40000       main [1]
[2]     34.0    0.78    2.69   40000         Proc_1 [2]
                0.70    0.46   40000/40000       Proc_6 [5]
                0.57    0.48   40000/40000       Proc_3 [7]
                0.48    0.00   40000/120000      Proc_7 [3]
```

This shows that function `Proc_1` was called only from `main`, and `Proc_1` in turn called three other functions. Callgraph information is obtained only if the application code is compiled with the `-pg` option. This causes the compiler to insert extra code into each compiled function, specifically a call to `mcount`, and the implementation of `mcount` stores away the data for subsequent processing by gprof.

Caution

There are a number of reasons why the output will not be 100% accurate. Collecting the flat profile typically involves timer interrupts so any code that runs with interrupts disabled will not appear. The current host-side gprof implementation maps program counter values onto symbols using a bin mechanism. When a bin spans the end of one function and the start of the next gprof may report the wrong function. This is especially likely on architectures with single-byte instructions such as an x86. When examining gprof output it may prove useful to look at a linker map or program disassembly.

The eCos profiling package requires some additional support from the HAL packages, and this may not be available on all platforms:

1. There must be an implementation of the profiling timer. Typically this is provided by the variant or platform HAL using one of the hardware timers. If there is no implementation then the configuration tools will report an unresolved conflict related to `CYGINT_PROFILE_HAL_TIMER` and profiling is not possible. Some implementations overload the system clock, which means that profiling is only possible in configurations containing the eCos kernel and `CYGVAR_KERNEL_COUNTERS_CLOCK`.

2. There should be a hardware-specific implementation of `mcount`, which in turn will call the generic functionality provided by this package. It is still possible to do some profiling without `mcount` but the resulting data will be less useful. To check whether or not `mcount` is available, look at the current value of the CDL interface `CYGINT_PROFILE_HAL_MCOUNT` in the graphical configuration tool or in an `ecos.ecc` save file.

3. The current profiling support is only suitable for single-core systems, it is not SMP-aware. Since profiling is driven by interrupts from a timer, samples can only be collected from the CPU to which that interrupt is bound. There is no mechanism for sampling the state of the other CPUs. The array of sample counters is not updated atomically, so updates to the same, or close, entries may result in readings being lost. For these reasons, targets that might support profiling in a single-core configuration will disable it in a multi-core configuration.

This document only describes the eCos profiling support. Full details of gprof functionality and output formats can be found in the gprof documentation. However it should be noted that that documentation describes some functionality which cannot be implemented using current versions of the gcc compiler: the section on annotated source listings describes basic block counting which is not relevant. For basic block counting, the GNU gcov tool should be used instead.

Building Applications for Profiling

To perform application profiling the gprof package `CYGPKG_PROFILE_GPROF` must first be added to the eCos configuration. On the command line this can be achieved using:

```
$ ecosconfig add profile_gprof
$ ecosconfig tree
$ make
```

Alternatively the same steps can be performed using the graphical configuration tool by adding the package "Application profile support" with the `Build->Packages` menu item.

If the HAL packages implement `mcount` for the target platform then usually application code should be compiled with `-pg`. Optionally eCos itself can also be compiled with this option by modifying the configuration option `CYGBLD_GLOBAL_CFLAGS`. Compiling with `-pg` is optional but gives more complete profiling data.

Note

The profiling package itself must not be compiled with `-pg` because that could lead to infinite recursion when doing `mcount` processing. This is handled automatically by the package's CDL.

Profiling does not happen automatically. Instead it must be started explicitly by the application, using a call to `profile_on`. A typical example would be:

```
#include <pkgconf/system.h>
#ifdef CYGPKG_PROFILE_GPROF
# include <cyg/profile/profile.h>
#endif
…
int
main(int argc, char** argv)
{
    …
#ifdef CYGPKG_PROFILE_GPROF
    {
        extern char _stext[], _etext[];
        profile_on(_stext, _etext, 16, 3500);
    }
#endif
    …
}
```

The `profile_on` takes four arguments:

`start address`
`end address`

These specify the range of addresses that will be profiled. Usually profiling should cover the entire application. On most targets the linker script will export symbols `_stext` and `_etext` corresponding to the beginning and end of code, so these can be used as the addresses. It is possible to perform profiling on a subset of the code if that code is located contiguously in memory.

`bucket size`

`profile_on` divides the range of addresses into a number of buckets of this size. It then allocates a single array of 16-bit counters with one entry for each bucket. When the profiling timer interrupts the interrupt handler will examine the program counter of the interrupted code and, assuming it is within the range of valid addresses, find the containing bucket and increment the appropriate counter.

The size of the array counters is determined by the range of addresses being profiled and by the bucket size. For a bucket size of 16, one counter is needed for every 16 bytes of code. For an application with say 512K of code that means dynamically allocating a 64K array. If the target hardware is low on memory then this may be unacceptable, and the requirements can be reduced by increasing the bucket size. However this will affect the accuracy of the results and gprof is more likely to report the wrong function. It also increases the risk of a counter overflow.

For the sake of run-time efficiency the bucket size must be a power of 2, and it will be adjusted if necessary.

`time interval`

The final argument specifies the interval between profile timer interrupts, in units of microseconds. Increasing the interrupt frequency gives more accurate profiling results, but at the cost of higher run-time overheads and a greater risk of a counter overflow. The HAL package may modify this interval because of hardware restrictions, and the generated profile data will contain the actual interval that was used. Usually it is a good idea to use an interval that is not a simple fraction of the system clock, typically 10000 microseconds. Otherwise there is a risk that the profiling timer will disproportionally sample code that runs only in response to the system clock.

`profile_on` can be invoked multiple times, and on subsequent invocations, it will delete profiling data and allocate a fresh profiling range.

Profiling can be turned off using the function `profile_off`:

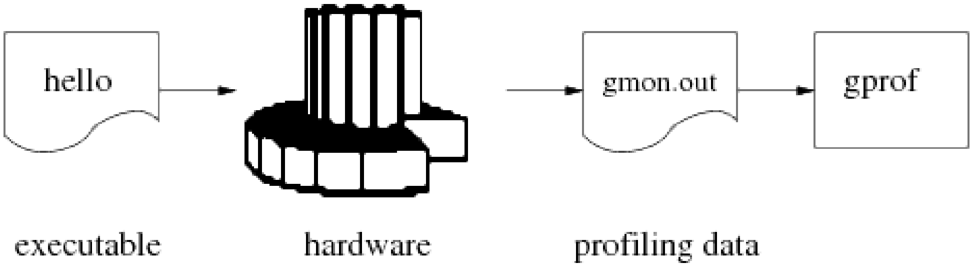
```
void profile_off(void);
```

This will also reset any existing profile data.

If the eCos configuration includes a TCP/IP stack and if a tftp daemon will be used to extract the data from the target then the call to `profile_on` should happen after the network is up. `profile_on` will attempt to start a tftp daemon thread, and this will fail if networking has not yet been enabled.

```
int
main(int argc, char** argv)
{
    …
    init_all_network_interfaces();
    …
#ifdef CYGPKG_PROFILE_GPROF
    {
        extern char _stext[], _etext[];
        profile_on(_stext, _etext, 16, 3000);
    }
#endif
    …
}
```

The application can then be linked and run as usual.



When gprof is used for native development rather than for embedded targets the profiling data will automatically be written out to a file `gmon.out` when the program exits. This is not possible on an embedded target because the code has no direct access to the host's file system. Instead the `gmon.out` file has to be extracted from the target as described below. gprof can then be invoked normally:

```
$ gprof dhrystone
Flat profile:

Each sample counts as 0.003003 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 14.15      1.45     1.45   120000    12.05    12.05  Proc_7
 11.55      2.63     1.18   120000     9.84     9.84  Func_1
  8.04      3.45     0.82                             main
…
```

If `gmon.out` does not contain call graph data, either because `mcount` is not supported or because this functionality was explicitly disabled, then the `-no-graph` must be used.

```
$ gprof --no-graph dhrystone
Flat profile:

Each sample counts as 0.003003 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 14.15      1.45     1.45                             Proc_7
 11.55      2.63     1.18                             Func_1
  8.04      3.45     0.82                             main
…
```

Extracting the Data

By default gprof expects to find the profiling data in a file `gmon.out` in the current directory. This package provides two ways of extracting data: a gdb macro or tftp transfers. Using tftp is faster but requires a TCP/IP stack on the target. It also consumes some additional target-side resources, including an extra tftp daemon thread and its stack. The gdb macro can be used even when the eCos configuration does not include a TCP/IP stack. However it is much slower, typically taking tens of seconds to retrieve all the data for a non-trivial application.

The gdb macro is called **gprof_dump**, and can be found in the file `gprof.gdb` in the `host` subdirectory of this package, and in the `ECOS_INSTALL_DIR/etc` subdirectory. A typical way of using this macro is:

```
(gdb) source <ECOS_INSTALL_DIR>;/etc/gprof.gdb
(gdb) gprof_dump
```

This macro can be used any time after the call to `profile_on`. It will store the profiling data accumulated so far to the file `gmon.out` in the current directory, and then reset all counts. gprof uses only a 16 bit counter for every bucket of code. These counters can easily saturate if the profiling run goes on for a long time, or if the application code spends nearly all its time in just a few tight inner loops. The counters will not actually wrap around back to zero, instead they will stick at 0xFFFF, but this will still affect the accuracy of the gprof output. Hence it is desirable to reset the counters once the profiling data has been extracted.

The file `gprof.gdb` contains two other macros which may prove useful. **gprof_fetch** extracts the profiling data and generates the file `gmon.out`, but does not reset the counters. **gprof_reset** only resets the counters, without extracting the data or overwriting `gmon.out`.

If the configuration includes a TCP/IP stack then the profiling data can be extracted using tftp instead. There are two relevant configuration options. `CYGPKG_PROFILE_TFTP` controls whether or not tftp is supported. It is enabled by default if the configuration includes a TCP/IP stack, but can be disabled to save target-side resources. `CYGNUM_PROFILE_TFTP_PORT` controls the UDP port which will be used. This port cannot be shared with other tftp daemons. If neither application code nor any other package (for example the gcov test coverage package) provides a tftp service then the default port can be used. Otherwise it will be necessary to assign unique ports to each daemon.

If enabled the tftp daemon will be started automatically by `profile_on`. This should only happen once the network is up and running, typically after the call to `init_all_network_interfaces`.

The data can then be retrieved using a standard tftp client. There are a number of such clients available with very different interfaces, but a typical session might look something like this:

```
$ tftp
tftp> connect 10.1.1.134
tftp> binary
tftp> get gmon.out
Received 64712 bytes in 0.9 seconds
tftp> quit
```

The address `10.1.1.134` should be replaced with the target's IP address. Extracting the profiling data by tftp will automatically reset the counters.

Configuration Options

This package contains a number of configuration options. Two of these, `CYGPKG_PROFILE_TFTP` and `CYGNUM_PROFILE_TFTP_PORT`, related to support for tftp transfers and have already been described.

Support for collecting the call graph data via `mcount` is optional and can be controlled via `CYGPKG_PROFILE_CALLGRAPH`. This option will only be active if the HAL provides the underlying `mcount` support and implements `CYGINT_PROFILE_HAL_MCOUNT`. The call graph data allows gprof to produce more useful output, but at the cost of extra run-time and memory overheads. If this option is disabled then the `-pg` compiler flag should not be used.

If `CYGPKG_PROFILE_CALLGRAPH` is enabled then there are two further options which can be used to control memory requirements. Collecting the data requires two blocks of memory, a simple hash table and an array of arc records. The `mcount` code uses the program counter address to index into the hash table, giving the first element of a singly linked list. The array of arc records contains the various linked lists for each hash slot. The required number of arc records depends on the number of function calls in the application. For example if a function `Proc_7` is called from three different places in the application then three arc records will be needed.

`CYGNUM_PROFILE_CALLGRAPH_HASH_SHIFT` controls the size of the hash table. The default value of 8 means that the program counter is shifted right by eight places to give a hash table index. Hence each hash table slot corresponds to 256 bytes of code, and for an application with say 512K of code `profile_on` will dynamically allocate an 8K hash table. Increasing the shift size reduces the memory requirement, but means that each hash table slot will correspond to more code and hence `mcount` will need to traverse a longer linked list of arc records.

`CYGNUM_PROFILE_CALLGRAPH_ARC_PERCENTAGE` controls how much memory `profile_on` will allocate for the arc records. This uses a simple heuristic, a percentage of the overall code size. By default the amount of arc record space allocated will be 5% of the code size, so for a 512K executable that requires approximately 26K. This default should suffice for most applications. In exceptional cases it may be insufficient and a diagnostic will be generated when the profiling data is extracted.

Implementing the HAL Support

The profiling package requires HAL support: A function `hal_enable_profile_timer` and an implementation of `mcount`. The profile timer is required. Typically it will be implemented by the variant or platform HAL using a spare hardware timer, and that HAL package will also implement the CDL interface `CYGINT_PROFILE_HAL_TIMER`. Support for `mcount` is optional but very desirable. Typically it will be implemented by the architectural HAL, which will also implement the CDL interface `CYGINT_PROFILE_HAL_MCOUNT`.

```
#include <pkgconf/system.h>
#ifdef CYGPKG_PROFILE_GPROF
# include <cyg/profile/profile.h>
#endif

int
hal_enable_profile_timer(int resolution)
{
    …
    return actual_resolution;
}
```

This function takes a single argument, a time interval in microseconds. It should arrange for a timer interrupt to go off after every interval. The timer VSR or ISR should then determine the program counter of the interrupted code and register this with the profiling package:

```
    …
    __profile_hit(interrupted_pc);
    …
```

The exact details of how this is achieved, especially obtaining the interrupted PC, are left to the HAL implementor. The HAL is allowed to modify the requested time interval because of hardware constraints, and should return the interval that is actually used.

`mcount` can be more difficult. The calls to `mcount` are generated internally by the compiler and the details depend on the target architecture. In fact `mcount` may not use the standard calling conventions at all. Typically implementing `mcount` requires looking at the code that is actually generated, and possibly at the sources of the appropriate compiler back end.

The HAL `mcount` function should call into the profiling package using standard calling conventions:

```
    …
    __profile_mcount((CYG_ADDRWORD) caller_pc, (CYG_ADDRWORD) callee_pc);
    …
```

If `mcount` was invoked because `main` called `Proc_1` then the caller pc should be an address inside `main`, typically corresponding to the return location, and the callee pc should be an address inside `Proc_1`, usually near the start of the function.

For some targets the compiler does additional work, for example automatically allocating a per-function word of memory to eliminate the need for the hash table. This is too target-specific and hence cannot easily be used by the generic profiling package.

Part LXIII. gcov Test Coverage Support

Name

CYGPKG_GCOV — eCos Support for the gcov test coverage tool

Description

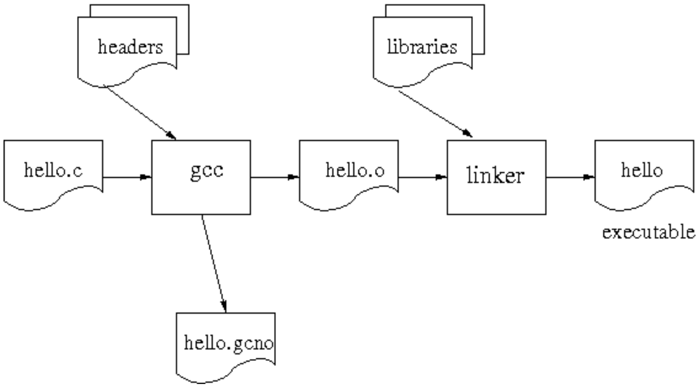
The GNU gcov tool provides test coverage support. After a test run it can be used to find code that was never actually executed. The testing conditions can then be adjusted for another test run to ensure that all the code really has been tested. The tool can also be used to find out how often each line of code was executed. That information can help application developers to determine where cpu time is being spent, and optimization effort can be focussed on critical parts of the code.

A typical fragment of gcov output looks something like this:

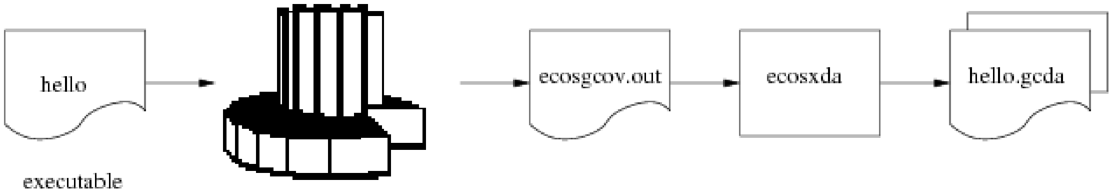
```
       80002:   60:  for (Run_Index = 1; Run_Index <= Number_Of_Runs; ++Run_Index)
       -: 61:  {
       -: 62:
       80000:   63:    Proc_5();
       80000:   64:    while (Int_1_Loc < Int_2_Loc)  /* loop body executed once */
       -: 65:    {
       80000:   66:      Int_3_Loc = 5 * Int_1_Loc - Int_2_Loc;
       80000:   67:      Proc_7 (Int_1_Loc, Int_2_Loc, &Int_3_Loc);
       80000:   68:      Int_1_Loc += 1;
       -: 69:    } /* while */
      240000:   70:    for (Ch_Index = 'A'; Ch_Index <= Ch_2_Glob; ++Ch_Index)
       -:   71:                          /* loop body executed twice */
       -: 72:    {
      160000:   73:      if (Enum_Loc == Func_1 (Ch_Index, 'C'))
       -: 74:       /* then, not executed */
       -: 75:        {
      ######:   76:        Proc_6 (Ident_1, &Enum_Loc);
      ######:   77:        strcpy (Str_2_Loc, "DHRYSTONE PROGRAM, 3'RD STRING");
      ######:   78:        Int_2_Loc = Run_Index;
      ######:   79:        Int_Glob = Run_Index;
       -: 80:        }
       -: 81:    }
           -:   82:    …
           -:   83:  }
```

Each line show the execution count and line number. An execution count of `-:` means that there is no executable code at that line. In this example the main loop is executed 80000 times. The body of the inner `for` loop is executed more often, but the `if` condition never triggers so four lines of code have not been tested.

The gcov tool works in conjunction with the gcc compiler. Application code should be built with two additional compiler flags `-fprofile-arcs` and `-ftest-coverage`. The first option causes the compiler to generate additional code which counts the number of times each basic block is executed. The second option results in additional files with `.gcno` suffixes which allow gcov to map these basic blocks onto lines of source code. Older versions of the compiler used to generate files with `.bb` and `.bbg` suffixes instead.



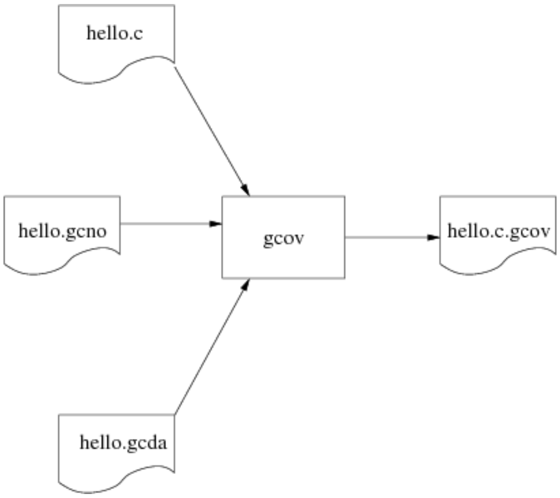
The resulting executable can be run on the target hardware as usual. The basic block counting will initialize automatically and the counts will accumulate. If gcov is used for native development rather than for embedded targets then these counts will be written out to `.gcda` data files automatically when the program exits (older versions of the compiler used to generate files with `.da` suffixes). A typical embedded target will not have access to the host file system so a different approach must be used. The counts can be extracted from the target using either a gdb macro or by a tftp transfer, giving a single `ecosgcov.out` file with counts for the entire application. This file should then be processed with the ecosxda script to give count files for each application source file.



It is now possible to run gcov on each source file. The exact format of the various files varies with the compiler version so it is important to use the version of gcov that comes with the compiler.

```
$ m68k-elf-gcov dhrystone.c
 89.25% of 214 source lines executed in file dhrystone.c
Creating dhrystone.c.gcov.
#
```

gcov will read in the basic block counts from the generated `.gcda` file. These basic blocks are mapped onto the source code using the information in the `.gcno` files.



gcov provides various options, for example it can output summaries for each function. Full details of the available functionality can be found in the gcov section of the gcc documentation.

Building Applications for Test Coverage

To perform application test coverage the gcov package `CYGPKG_GCOV` must first be added to the eCos configuration. On the command line this can be achieved using:

```
$ ecosconfig add gcov
$ ecosconfig tree
$ make
```

Alternatively the same steps can be performed using the graphical configuration tool. The package only has two configuration options related to tftp transfers, described below.

In addition application code should be compiled with two additional options, `-fprofile-arcs` and `-ftest-coverage`. The first option causes the compiler to insert additional code for basic block counting, plus an initialization call to `__gcov_init_func()` which is provided by the eCos gcov package. The second option results in additional `.gcno` output files which gcov will need later. The target-side memory needed to store the basic block counts is allocated statically.

When code is compiled with optimization the compiler may rearrange some of the code, if that leads to better performance. Sometimes this causes the gcov output to be rather confusing. Compiling with `-O0`, thus disabling optimization, can help.

Extracting the Data

The basic block counts must be extracted from the target and saved to a file `ecosgcov.out` on the host. This package provides two ways of doing this: a gdb macro or tftp transfers. Using tftp is faster but requires a TCP/IP stack on the target. It also consumes some additional target-side resources, including an extra tftp daemon thread and its stack. The gdb macro can be used even when the eCos configuration does not include a TCP/IP stack. However it is much slower, typically taking several minutes to retrieve all the counts for a non-trivial application.

The gdb macro is called `gcov_dump`, and can be found in the file `gcov.gdb` in the `host` subdirectory of this package, and in the `ECOS_INSTALL_DIR/etc` subdirectory. A typical way of using this macro is:

```
(gdb) source <ECOS_INSTALL_DIR>/etc/gcov.gdb
(gdb) gcov_dump
```

This macro can be used any time after the application has initialized, and will store the counts accumulated so far to the file `ecosgcov.out` in the current directory. The counts are not reset.

If the configuration includes a TCP/IP stack then the data can be extracted using tftp instead. There are two relevant configuration options. `CYGPKG_GCOV_TFTPD` controls whether or not tftp is supported. It is enabled by default if the configuration includes a TCP/IP stack, but can be disabled to save target-side resources. `CYGNUM_GCOV_TFTPD_PORT` controls the UDP port which will be used. This port cannot be shared with other tftp daemons. If neither application code nor any other package (for example the gprof profiling package) provides a tftp service then the default port can be used. Otherwise it will be necessary to assign unique ports to each daemon.

Using tftp requires some additional code in the application. Specifically the daemon cannot be started until the network is up and running, and that usually happens at the behest of application code rather than automatically. The following code fragment illustrates what is required:

```
#include <pkgconf/system.h>
#include <network.h>
#ifdef CYGPKG_GCOV
# include <pkgconf/gcov.h>
# include <cyg/profile/gcov.h>
#endif
…
int
main(int argc, char** argv)
{
    …
    init_all_network_interfaces();
#ifdef CYGPKG_GCOV_TFTPD
    gcov_start_tftpd();
#endif
    …
}
```

The data can then be retrieved using a standard tftp client. There are a number of such clients available with very different interfaces, but a typical session might look something like this:

```
$ tftp
tftp> connect 10.1.1.134
tftp> binary
tftp> get ecosgcov.out
Received 138740 bytes in 1.7 seconds
tftp> quit
```

The address `10.1.1.134` should be replaced with the target's IP address.

ecosxda

gcov expects separate `.gcda` files for each application source file compiled with `-fprofile-arcs`. However it would be inconvenient to extract each `.gcda` file via tftp or a gdb macro. Instead the data is first written to a single file `ecosgcov.out`. The ecosxda utility script should then be used to process `ecosgcov.out` and generate the `.gcda` files.

The ecosxda script can be found in the `host` subdirectory of this package. Since it is a simple Tcl script it does not need to be built or installed. If desired it can be copied to a suitable location on the user's `PATH`. Alternatively the subdirectory contains suitable `configure` and `Makefile.in` files, allowing the script to be installed automatically as part of the generic eCos host-side build system. The toplevel file `README.host` contains more information about this.

Typically ecosxda will be invoked with no arguments.

```
$ ecosxda
```

It will read in an `ecosgcov.out` file from the current directory and output or update the `.gcda` files appropriate for the application. If a given `.gcda` file already exists then by default ecosxda will read it in and merge the old and new counts, rather than write a new set. This allows data from several test runs to accumulate, giving more comprehensive test coverage. Merging the counts is only possible if the source file has not been recompiled, otherwise the old counts will be discarded to avoid contaminated results.

ecosxda takes a number of command line options.

-h
--help
Provide brief usage information.

-V
--version
Display the version of the ecosxda script being used.

-v
--verbose
Provide additional diagnostic output. Repeated uses increase the level of verbosity.

-n
--no-output
Do not actually create or modify any `.gcda` files. Typically this is used to find out whether any files would be replaced rather than merged, and it can also be used to validate the `ecosgcov.out` file.

-r
--replace
This forces ecosxda to ignore any existing counts in the `.gcda` files, rather than try to merge the existing and new counts. Typically it is used to discard the results from previous test runs.

In addition it is possible to specify the file containing the new counts, instead of the default `ecosgcov.out`. This may prove useful if several sets of results are extracted to different files during a single test run, to determine what code gets run at various stages. For example:

```
$ ecosxda -r stage2.out
```

Directories and eCos Test Coverage

In a simple build environment the source code, the `.gcno` files generated by the compiler, and the `.gcda` files output by ecosxda, will all reside in the same directory. That makes it easy for gcov to find the various files it needs. gcov will also generate its `.gcov` files in the same directory.

In more complicated build environments the source code may be kept completely separate from the build tree. eCos itself provides an example of this: the source code is held in a clean component repository, and builds happen in separate build trees. To use gcov in such an environment it is necessary to understand what files will be created where:

1. The compiler will output the `.gcno` file in the same directory as the object file. For an eCos build tree this will be below the version directory of each package. For example, if the kernel source file `sync/mutex.cxx` is built with `-ftest-coverage` then the `kernel/current/src/sync` subdirectory in the build tree will contain the `mutex.gcno` files.

2. The `.gcda` files will end up in the same directory as the `.gcno` files. The compiler puts the full path name in each object file, and this path is copied into the `ecosgcov.out` file and used by ecosxda. It is assumed that `ecosgcov.out` will be processed on the same machine that was used to compile the code.

3. When gcov is invoked it can be given a full pathname for the source file. By default it assumes that the other files will be in the current directory, but a `-o` command line option can be used to override this.

4. gcov will output its `.gcov` files in the current directory.

To perform test coverage of eCos itself, in addition to or instead of the application, it is necessary to rebuild eCos with the appropriate flags. This involves changing the configuration option `CYGBLD_GLOBAL_CFLAGS` to include `-ftest-coverage` and `-fprofile-arcs`, then performing a clean and a full make. The basic block counts can be extracted and processed with ecosxda as before, and the `.gcno` and `.gcda` files will all end up in the build tree. This test coverage data can then be processed in the build tree using, for example:

```
$ cd <build>
$ cd kernel/<version>
$ m68k-elf-gcov -o . <repo>/kernel/<version>/sync/mutex.cxx
…
 58.50% of 147 source lines executed in file <repo>/kernel/current/src/sync/mutex.cxx
Creating mutex.cxx.gcov.
```

Where <build> is the location of the build tree and <repo> is the location of the eCos component repository.

Additional Target-side Functions

The eCos gcov package provides a small number of additional target-side functions. Prototypes for these are provided in the header file `<cyg/profile/gcov.h>`.

```
…
int
main(int argc, char** argv)
{
    …
    init_all_network_interfaces();
#ifdef CYGPKG_GCOV_TFTPD
    gcov_start_tftpd();
#endif
    …
}
```

If the eCos configuration includes a TCP/IP stack and if a target-side tftp daemon will be used to extract the data from the target to the host then application code should call `gcov_start_tftpd` once the network is up. This cannot be done automatically by the gcov package itself since that package has no simple way of detecting when the network is ready.

```
extern void gcov_reset(void);
```

This function can be used to reset all basic block counts. If the application operates in a number of distinct stages then it may be useful to get coverage data for each stage, rather than a single set of results for the whole test run. It can also be used to get test coverage for a specific sequence of external inputs.

64-bit arithmetic is used for the basic block counts. Hence it should not be necessary to perform occasional resets to avoid counters overflowing.

To operate properly `gcov_reset` needs to disable interrupts for a while, so it should not be used in situations which require hard real-time performance.

```
extern void gcov_dump(void);
```

This is a utility routine which outputs some of the basic block information via `diag_printf` calls. It is intended primarily to help with debugging the gcov code itself.

In addition the `<cyg/profile/gcov.h>` header exports the data type gcov_module and a variable `gcov_head` which acts as the head of a linked list of gcov_module structures. This allows application code to access and manipulate the basic block data directly, if desired.

Part LXIV. CRC Algorithms

The CRC package provides implementation of CRC algorithms. This includes the POSIX CRC calculation which produces the same result as the cksum command on Linux, another 32 bit CRC by Gary S. Brown and a 16bit CRC. The CRC used for Ethernet FCS is also implemented.

Table of Contents

Chapter 205. CRC Functions

CRC API

The package implements a number of CRC functions as described below. The API to these functions is in the include file `cyg/crc/crc.h`.

cyg_posix_crc32

This function implements a 32 bit CRC which is compliant to the POSIX 1008.2 Standard. This is the same as the Linux cksum program.

```
cyg_uint32 cyg_posix_crc32(unsigned char * s, int len);
```

The CRC calculation is run over the data pointed to by *s*, of length *len*. The CRC is returned as an unsigned long.

cyg_crc32

These functions implement a 32 bit CRC by Gary S. Brown. They use the polynomial X^32+X^26+X^23+X^22+X^16+X^12+X^11+X^10+X^8+X^7+X^5+X^4+X^2+X^1+X^0.

```
cyg_uint32 cyg_crc32(unsigned char * s, int len);
cyg_uint32 cyg_crc32_accumulate(cyg_uint32 crc, unsigned char * s, int len);
```

The CRC calculation is run over the data pointed to by *s*, of length *len*. The CRC is returned as an unsigned long.

The CRC can be calculated over data separated into multiple buffers by using the function `cyg_crc32_accumulate()`. The parameter *crc* should be the result from the previous CRC calculation.

cyg_ether_crc32

These functions implement the 32 bit CRC used by the Ethernet FCS word.

```
cyg_uint32 cyg_ether_crc32(unsigned char * s, int len);
cyg_uint32 cyg_ether_crc32_accumulate(cyg_uint32 crc, unsigned char * s, int len);
```

The CRC calculation is run over the data pointed to by *s*, of length *len*. The CRC is returned as an unsigned long.

The CRC can be calculated over data separated into multiple buffers by using the function `cyg_ether_crc32_accumulate()`. The parameter *crc* should be the result from the previous CRC calculation.

cyg_crc16

This function implements a 16 bit CRC. It uses the polynomial x^16+x^12+x^5+1.

```
cyg_uint16 cyg_crc16(unsigned char * s, int len);
```

The CRC calculation is run over the data pointed to by *s*, of length *len*. The CRC is returned as an unsigned short.

Part LXV. CryptoAuthLib

Table of Contents

Chapter 206. CryptoAuthLib overview

Introduction

The `CYGPKG_CRYPTOAUTHLIB` package provides a standard CryptoAuthLib library implementation to eCos applications.

This package is covered by an “AS IS” license as distributed in the original CryptoAuthLib package:

Example 206.1. “AS IS” License

```
(c) 2015-2021 Microchip Technology Inc. and its subsidiaries.

Subject to your compliance with these terms, you may use the Microchip Software
and any derivatives exclusively with Microchip products. It is your
responsibility to comply with third party license terms applicable to your
use of third party software (including open source software) that may
accompany Microchip Software.

Redistribution of this Microchip Software in source or binary form is allowed
and must include the above terms of use and the following disclaimer with the
distribution and accompanying materials.

THIS SOFTWARE IS SUPPLIED BY MICROCHIP "AS IS". NO WARRANTIES, WHETHER
EXPRESS, IMPLIED OR STATUTORY, APPLY TO THIS SOFTWARE, INCLUDING ANY IMPLIED
WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A
PARTICULAR PURPOSE. IN NO EVENT WILL MICROCHIP BE LIABLE FOR ANY INDIRECT,
SPECIAL, PUNITIVE, INCIDENTAL OR CONSEQUENTIAL LOSS, DAMAGE, COST OR EXPENSE
OF ANY KIND WHATSOEVER RELATED TO THE SOFTWARE, HOWEVER CAUSED, EVEN IF
MICROCHIP HAS BEEN ADVISED OF THE POSSIBILITY OR THE DAMAGES ARE FORESEEABLE.
TO THE FULLEST EXTENT ALLOWED BY LAW, MICROCHIP'S TOTAL LIABILITY ON ALL
CLAIMS IN ANY WAY RELATED TO THIS SOFTWARE WILL NOT EXCEED THE AMOUNT OF
FEES, IF ANY, THAT YOU HAVE PAID DIRECTLY TO MICROCHIP FOR THIS SOFTWARE.
```

For definitive CryptoAuthLib documentation please refer to the main Microchip Trust Platform website. We do not duplicate that documentation here.

Chapter 207. Configuration

This chapter shows how to incorporate the CryptoAuthLib support into an eCos configuration, and how to configure it once included.

Configuration Overview

The CryptoAuthLib support is contained in a single eCos package `CYGPKG_CRYPTOAUTHLIB`. However, some functionality is dependant on other eCos features. e.g. the eCos I²C support.

Quick Start

Incorporating the CryptoAuthLib support into your application is straightforward. The essential starting point is to incorporate the CryptoAuthLib eCos package (`CYGPKG_CRYPTOAUTHLIB`) into your configuration.

This may be achieved directly using **ecosconfig add** on the command line, or the **Build->Packages…** menu item within the eCos Configuration Tool.

Depending on the CryptoAuthLib package configuration other packages may be required (e.g. I²C support). The package requires that the `CYGPKG_INFRA` and `CYGPKG_MEMALLOC` packages are included in the eCos application configuration.

Chapter 208. eCos port

Overview

The goal for the CYGPKG_CRYPTOAUTHLIB package is to avoid where possible having to have any core CryptoAuthLib source file changes made specifically for eCos. This is to ensure that re-imports of newer versions of the library sources involve minimal effort. The files are as provided in the official CryptoAuthLib release package as imported, with the following exceptions:

1. Files have been moved, unmodified, to create a standard eCos package tree structure to integrate with the eCosPro build environment

   Only **relevant** files from the original project have been included in the eCos package.

2. The file include/hal/atca_hal.h has a different hal_delay_us() prototype to match the underlying eCos run-time function actually referenced due to the naming clash.

The current CryptoAuthLib version provided by the eCos package is the github tagged release 20220614, which is a v3.3.3 release (8th October 2021) with bug fixes applied up to 14th June 2022.

The original project homepage can be found on github: cryptoauthlib

The release package was downloaded from the github project page: cryptoauthlib/releases/tag/20220614

The following table highlights the files taken from the CryptoAuthLib package and their new location within the eCos CYGPKG_CRYPTOAUTHLIB package:

Original github	eCos package
include/atca_basic.h	include/atca_basic.h
lib/atca_bool.h	include/atca_bool.h
lib/atca_cfgs.h	include/atca_cfgs.h
lib/atca_compiler.h	include/atca_compiler.h
lib/atca_device.h	include/atca_device.h
lib/atca_devtypes.h	include/atca_devtypes.h
lib/atca_helpers.h	include/atca_helpers.h
lib/atca_iface.h	include/atca_iface.h
lib/atca_status.h	include/atca_status.h
lib/atca_version.h	include/atca_version.h
lib/cryptoauthlib.h	include/cryptoauthlib.h
lib/atcacert/atcacert_client.h	include/atcacert/atcacert_client.h
lib/atcacert/atcacert_date.h	include/atcacert/atcacert_date.h
lib/atcacert/atcacert_def.h	include/atcacert/atcacert_def.h
lib/atcacert/atcacert_der.h	include/atcacert/atcacert_der.h
lib/atcacert/atcacert.h	include/atcacert/atcacert.h
lib/atcacert/atcacert_host_hw.h	include/atcacert/atcacert_host_hw.h
lib/atcacert/atcacert_host_sw.h	include/atcacert/atcacert_host_sw.h
lib/atcacert/atcacert_pem.h	include/atcacert/atcacert_pem.h

Original github	eCos package
lib/calib/calib_aes_gcm.h	include/calib/calib_aes_gcm.h
lib/calib/calib_basic.h	include/calib/calib_basic.h
lib/calib/calib_command.h	include/calib/calib_command.h
lib/calib/calib_execution.h	include/calib/calib_execution.h
lib/crypto/atca_crypto_hw_aes.h	include/crypto/atca_crypto_hw_aes.h
lib/crypto/atca_crypto_sw_ecdsa.h	include/crypto/atca_crypto_sw_ecdsa.h
lib/crypto/atca_crypto_sw.h	include/crypto/atca_crypto_sw.h
lib/crypto/atca_crypto_sw_rand.h	include/crypto/atca_crypto_sw_rand.h
lib/crypto/atca_crypto_sw_sha1.h	include/crypto/atca_crypto_sw_sha1.h
lib/crypto/atca_crypto_sw_sha2.h	include/crypto/atca_crypto_sw_sha2.h
lib/crypto/hashes/sha1_routines.h	include/crypto/hashes/sha1_routines.h
lib/crypto/hashes/sha2_routines.	include/crypto/hashes/sha2_routines.
lib/hal/atca_hal.h	include/hal/atca_hal.h
lib/host/atca_host.h	include/host/atca_host.h
lib/jwt/atca_jwt.h	include/jwt/atca_jwt.h
lib/mbedtls/atca_mbedtls_wrap.h	include/mbedtls/atca_mbedtls_wrap.h
third_party/atca_mbedtls_patch.h	include/third_party/atca_mbedtls_patch.h
lib/pkcs11/cryptoki.h	include/pkcs11/cryptoki.h
lib/pkcs11/pkcs11_attrib.h	include/pkcs11/pkcs11_attrib.h
lib/pkcs11/pkcs11_cert.h	include/pkcs11/pkcs11_cert.h
lib/pkcs11/pkcs11_digest.h	include/pkcs11/pkcs11_digest.h
lib/pkcs11/pkcs11_encrypt.h	include/pkcs11/pkcs11_encrypt.h
lib/pkcs11/pkcs11f.h	include/pkcs11/pkcs11f.h
lib/pkcs11/pkcs11_find.h	include/pkcs11/pkcs11_find.h
lib/pkcs11/pkcs11.h	include/pkcs11/pkcs11.h
lib/pkcs11/pkcs11_info.h	include/pkcs11/pkcs11_info.h
lib/pkcs11/pkcs11_init.h	include/pkcs11/pkcs11_init.h
lib/pkcs11/pkcs11_key.h	include/pkcs11/pkcs11_key.h
lib/pkcs11/pkcs11_mech.h	include/pkcs11/pkcs11_mech.h
lib/pkcs11/pkcs11_object.h	include/pkcs11/pkcs11_object.h
lib/pkcs11/pkcs11_os.h	include/pkcs11/pkcs11_os.h
lib/pkcs11/pkcs11_session.h	include/pkcs11/pkcs11_session.h
lib/pkcs11/pkcs11_signature.h	include/pkcs11/pkcs11_signature.h
lib/pkcs11/pkcs11_slot.h	include/pkcs11/pkcs11_slot.h
lib/pkcs11/pkcs11t.h	include/pkcs11/pkcs11t.h
lib/pkcs11/pkcs11_token.h	include/pkcs11/pkcs11_token.h
lib/pkcs11/pkcs11_util.h	include/pkcs11/pkcs11_util.h
third_party/atca_mbedtls_patch.h	include/atca_mbedtls_patch.h

Original github	**eCos package**
lib/atca_basic.c	src/lib/atca_basic.c
lib/atca_cfgs.c	src/lib/atca_cfgs.c
lib/atca_device.c	src/lib/atca_device.c
lib/atca_helpers.c	src/lib/atca_helpers.c
lib/atca_iface.c	src/lib/atca_iface.c
lib/atca_utils_sizes.c	src/lib/atca_utils_sizes.c
lib/atcacert/atcacert_client.c	src/lib/atcacert/atcacert_client.c
lib/atcacert/atcacert_date.c	src/lib/atcacert/atcacert_date.c
lib/atcacert/atcacert_def.c	src/lib/atcacert/atcacert_def.c
lib/atcacert/atcacert_der.c	src/lib/atcacert/atcacert_der.c
lib/atcacert/atcacert_host_hw.c	src/lib/atcacert/atcacert_host_hw.c
lib/atcacert/atcacert_host_sw.c	src/lib/atcacert/atcacert_host_sw.c
lib/atcacert/atcacert_pem.c	src/lib/atcacert/atcacert_pem.c
lib/calib/calib_aes.c	src/lib/calib/calib_aes.c
lib/calib/calib_checkmac.c	src/lib/calib/calib_checkmac.c
lib/calib/calib_derivekey.c	src/lib/calib/calib_derivekey.c
lib/calib/calib_gendig.c	src/lib/calib/calib_gendig.c
lib/calib/calib_hmac.c	src/lib/calib/calib_hmac.c
lib/calib/calib_lock.c	src/lib/calib/calib_lock.c
lib/calib/calib_privwrite.c	src/lib/calib/calib_privwrite.c
lib/calib/calib_secureboot.c	src/lib/calib/calib_secureboot.c
lib/calib/calib_sign.c	src/lib/calib/calib_sign.c
lib/calib/calib_write.c	src/lib/calib/calib_write.c
lib/calib/calib_aes_gcm.c	src/lib/calib/calib_aes_gcm.c
lib/calib/calib_command.c	src/lib/calib/calib_command.c
lib/calib/calib_ecdh.c	src/lib/calib/calib_ecdh.c
lib/calib/calib_genkey.c	src/lib/calib/calib_genkey.c
lib/calib/calib_info.c	src/lib/calib/calib_info.c
lib/calib/calib_mac.c	src/lib/calib/calib_mac.c
lib/calib/calib_random.c	src/lib/calib/calib_random.c
lib/calib/calib_selftest.c	src/lib/calib/calib_selftest.c
lib/calib/calib_updateextra.c	src/lib/calib/calib_updateextra.c
lib/calib/calib_basic.c	src/lib/calib/calib_basic.c
lib/calib/calib_counter.c	src/lib/calib/calib_counter.c
lib/calib/calib_execution.c	src/lib/calib/calib_execution.c
lib/calib/calib_helpers.c	src/lib/calib/calib_helpers.c
lib/calib/calib_kdf.c	src/lib/calib/calib_kdf.c
lib/calib/calib_nonce.c	src/lib/calib/calib_nonce.c

Original github	eCos package
lib/calib/calib_read.c	src/lib/calib/calib_read.c
lib/calib/calib_sha.c	src/lib/calib/calib_sha.c
lib/calib/calib_verify.c	src/lib/calib/calib_verify.c
lib/crypto/atca_crypto_hw_aes_cbc.c	src/lib/crypto/atca_crypto_hw_aes_cbc.c
lib/crypto/atca_crypto_hw_aes_ccm.c	src/lib/crypto/atca_crypto_hw_aes_ccm.c
lib/crypto/atca_crypto_hw_aes_ctr.c	src/lib/crypto/atca_crypto_hw_aes_ctr.c
lib/crypto/atca_crypto_sw_ecdsa.c	src/lib/crypto/atca_crypto_sw_ecdsa.c
lib/crypto/atca_crypto_sw_sha1.c	src/lib/crypto/atca_crypto_sw_sha1.c
lib/crypto/atca_crypto_hw_aes_cbcmac.c	src/lib/crypto/atca_crypto_hw_aes_cbcmac.c
lib/crypto/atca_crypto_hw_aes_cmac.c	src/lib/crypto/atca_crypto_hw_aes_cmac.c
lib/crypto/atca_crypto_pbkdf2.c	src/lib/crypto/atca_crypto_pbkdf2.c
lib/crypto/atca_crypto_sw_rand.c	src/lib/crypto/atca_crypto_sw_rand.c
lib/crypto/atca_crypto_sw_sha2.c	src/lib/crypto/atca_crypto_sw_sha2.c
lib/crypto/hashes/sha1_routines.c	src/lib/crypto/hashes/sha1_routines.c
lib/crypto/hashes/sha2_routines.c	src/lib/crypto/hashes/sha2_routines.c
lib/hal/atca_hal.c	src/lib/hal/atca_hal.c
lib/host/atca_host.c	src/lib/host/atca_host.c
lib/jwt/atca_jwt.c	src/lib/jwt/atca_jwt.c
lib/mbedtls/atca_mbedtls_ecdh.c	src/lib/mbedtls/atca_mbedtls_ecdh.c
lib/mbedtls/atca_mbedtls_ecdsa.c	src/lib/mbedtls/atca_mbedtls_ecdsa.c
lib/mbedtls/atca_mbedtls_wrap.c	src/lib/mbedtls/atca_mbedtls_wrap.c
lib/mbedtls/atca_mbedtls_wrap.h	src/lib/mbedtls/atca_mbedtls_wrap.h
lib/mbedtls/README.md	src/lib/mbedtls/README.md
lib/openssl/atca_openssl_interface.c	src/lib/openssl/atca_openssl_interface.c
lib/openssl/README.md	src/lib/openssl/README.md
lib/pkcs11/pkcs11_attrib.c	src/lib/pkcs11/pkcs11_attrib.c
lib/pkcs11/pkcs11_config.c	src/lib/pkcs11/pkcs11_config.c
lib/pkcs11/pkcs11_digest.c	src/lib/pkcs11/pkcs11_digest.c
lib/pkcs11/pkcs11_find.c	src/lib/pkcs11/pkcs11_find.c
lib/pkcs11/pkcs11_init.c	src/lib/pkcs11/pkcs11_init.c
lib/pkcs11/pkcs11_main.c	src/lib/pkcs11/pkcs11_main.c
lib/pkcs11/pkcs11_object.c	src/lib/pkcs11/pkcs11_object.c
lib/pkcs11/pkcs11_session.c	src/lib/pkcs11/pkcs11_session.c
lib/pkcs11/pkcs11_slot.c	src/lib/pkcs11/pkcs11_slot.c
lib/pkcs11/pkcs11_util.c	src/lib/pkcs11/pkcs11_util.c
lib/pkcs11/pkcs11_cert.c	src/lib/pkcs11/pkcs11_cert.c
lib/pkcs11/pkcs11_debug.c	src/lib/pkcs11/pkcs11_debug.c
lib/pkcs11/pkcs11_encrypt.c	src/lib/pkcs11/pkcs11_encrypt.c

Original github	eCos package
lib/pkcs11/pkcs11_info.c	src/lib/pkcs11/pkcs11_info.c
lib/pkcs11/pkcs11_key.c	src/lib/pkcs11/pkcs11_key.c
lib/pkcs11/pkcs11_mech.c	src/lib/pkcs11/pkcs11_mech.c
lib/pkcs11/pkcs11_os.c	src/lib/pkcs11/pkcs11_os.c
lib/pkcs11/pkcs11_signature.c	src/lib/pkcs11/pkcs11_signature.c
lib/pkcs11/pkcs11_token.c	src/lib/pkcs11/pkcs11_token.c
third_party/atca_mbedtls_patch.c	src/third_party/atca_mbedtls_patch.c
app/tng/tng_atca.h	include/app/tng/tng_atca.h

Chapter 209. Test Programs

Test Programs

Some CryptoAuthLib specific tests are built and can be used to verify correct operation of the library.

1. `basiccheck`

   This test executes some basic device sanity checks to exercise the I2C transport layer, and to verify target device operation. It does not use the CryptoAuthLib test environment, put purely calls the CryptoAuthLib library functionality.

2. `basicinfo`

   This test executes some basic device sanity checks to exercise the I2C transport layer, and to verify target device operation. It uses the CryptoAuthLib test environment and provides similar diagnostic output as the direct `basiccheck` test.

3. `cryptoauth_test`

   This test is only built when `CYGBLD_CRYPTOAUTHLIB_TESTS_MANUAL` is enabled since it relies on the `/dev/haldiag` diagnostic terminal connection for interactive command input.

   The test is a build of the “command-line” test application allowing interaction with devices. On startup the application will display an inital prompt and wait for user input:

   ```
   INFO:<code from 0x20209008 -> 0x202420c4, CRC a43d^gt;
   INFO:<Using haldiag for interactive test>
   INFO:<Enter "help\n" to display help menu>
   $
   ```

   Depending on the eCos configuration, and the specific options for this CryptoAuthLib package, the features of the test may vary. The following is purely an example. The `help` command should list all of the supported commands. Normally a device needs to be selected prior to use. e.g.:

   ```
   $ help
   Usage:
   help - Display Menu
   ecc608 - Set Target Device to ATECC608
   info - Get the Chip Revision
   sernum - Get the Chip Serial Number
   rand - Generate Some Random Numbers
   readcfg - Read the Config Zone
   lockstat - Zone Lock Status
   tng - Run unit tests on TNG type part.
   basic - Run Basic Test on Selected Device
   util - Run Helper Function Tests
   clkdivm0 - Set ATECC608 to ClockDivider M0(0x00)
   clkdivm1 - Set ATECC608 to ClockDivider M1(0x05)
   clkdivm2 - Set ATECC608 to ClockDivider M2(0x0D)
   cd - Run Unit Tests on Cert Data
   cio - Run Unit Test on Cert I/O
   crypto - Run Unit Tests for Software Crypto Functions
   pbkdf2 - Run pbkdf2 tests

   $ ecc608
   Device Selected.

   $ info
   revision:
   00 00 60 01
   ```

```
$ sernum
serial number:
01 23 C4 71 E0 E7 45 37 EE

$
```

Note

For some devices, e.g. ATECC608 family, some operations will not operate, or will return errors, if the device is not locked or configured suitably. Knowledge of the device, and its features, being used should taken into account when running commands and interpreting the results.

Part LXVI. LibTomCrypt Cryptography Library

Important

The LibTomCrypt Cryptography Library package for eCos is distributed as an optional eCos add-on package that may not be included in your release of eCosPro. If this package is not listed in either the graphical or command line eCos Configuration tool, please contact eCosCentric for availability and pricing.

Name

`CYGPKG_CRYPT_LIBTOMCRYPT` — Cryptography

Description

`CYGPKG_CRYPT_LIBTOMCRYPT` is a port to eCos of the Tom St Denis LibTomCrypt cryptography library. Full documentation on this library can be found in the file `crypt.pdf` in the package's `doc` subdirectory.

The port to eCos has involved only very minor changes to the generic LibTomCrypt sources. Appropriate CDL has been added to turn the library into an eCos package. The package's `src` subdirectory has essentially the same contents as a LibTomCrypt tarball. The exported headers have been moved from the `src/headers` subdirectory to the package's `include` subdirectory in accordance with eCos conventions. Some of these headers have had minor modifications to allow the package to be built and used within an eCos configuration. The prebuilt documentation file `crypt.pdf` has been moved to the `doc` subdirectory.

The package has no architectural dependencies so can be added to any eCos configuration. However it does depend on dynamic memory allocation support from `CYGPKG_MEMALLOC`, on standard C library support from `CYGPKG_LIBC_STDLIB`, `CYGPKG_LIBC_I18N`, `CYGPKG_LIBC_STRING`, `CYGPKG_LIBC_TIME`, and on the multi-precision arithmetic support provided by `CYGPKG_LIBTOMMATH`.

The package provides three CDL configuration options. `CYGIMP_CRYPT_LIBTOMCRYPT_SMALL_CODE` corresponds to the libtomcrypt `LTC_SMALL_CODE` option and selects for smaller but slower code. Similarly `CYGIMP_CRYPT_LIBTOMCRYPT_NO_TABLES` corresponds to `LTC_NO_TABLES` and also selects for smaller but slower code. `CYGDBG_CRYPT_LIBTOMCRYPT_ARGCHK` determines what argument checking gets performed. By default in a debug build (`CYGPKG_INFRA_DEBUG` enabled) invalid arguments result in an assertion failure, and in a normal build argument checking is disabled.

Caution

In some jurisdictions use of this library may be subject to patent and trademark restrictions. More information on this can be found in section 1.4 of the `crypt.pdf` document. It is the application developer's responsibility to consider the legal issues before using this library.

Part LXVII. LibTomMath Multi-Precision Math Package

Important

The LibTomMath Multi-Precision Math package for eCos is distributed as an optional eCos add-on package that may not be included in your release of eCosPro. If this package is not listed in either the graphical or command line eCos Configuration tool, please contact eCosCentric for availability and pricing.

Name

CYGPKG_MATH_LIBTOMMATH — Multi-Precision Maths

Description

CYGPKG_MATH_LIBTOMMATH is a port to eCos of the Tom St Denis LibTomMath Multi-Precision Math package. Full documentation on this package can be found in the files bn.pdf, tommath.pdf, and poster.pdf in the package's doc subdirectory.

The port to eCos has involved only very minor changes to the generic LibTomMath sources, to allow the code to adapt to eCos configurations lacking standard I/O functionality. The package's src subdirectory has essentially the same contents as a LibTomMath tarball. The exported headers tommath.h, tommath_class.h and tommath_superclass.h have been moved to the package's include subdirectory, The prebuilt documentation files have been moved to the package's doc subdirectory. Appropriate CDL has been added so that the package can be built as part of an eCos configuration.

The package has no architectural dependencies so can be added to any eCos configuration. However it does depend on dynamic memory allocation support from CYGPKG_MEMALLOC and on standard C library support from CYGPKG_LIBC_STDLIB, CYGPKG_LIBC_I18N, and CYGPKG_LIBC_STRING.

Part LXVIII. BootUp ROM loader

Table of Contents

Chapter 210. BootUp overview

Introduction

eCosPro-BootUp is eCosCentric's commercial name for the CYGPKG_BOOTUP package. The package is not included as standard in eCosPro Developer's Kit releases. Review the board specific documentation to determine if support has been included for your target platform.

The CYGPKG_BOOTUP package implements a lightweight bootROM that is intended to be easily portable and customizable to support the requirements of a given platform. It is purposely designed to provide an uncomplicated and straightforward boot mechanism, but does however provide a framework that the target platform code can use to easily extend its basic functionality. This can be used to incorporate more advanced features such as secure boot capabilities and application or system updates.

The BootUp code does NOT support any debug ROM monitor features. It is targeted at deployment of SoC-based designs that have limited memory resources and where hardware debugging (JTAG, SWD, BDM, etc.) will be used. It does not incorporate a debug agent such as GDB stubs, a CLI or any other debug monitor features. If you require these kinds of features then the RedBoot bootloader and debug firmware will be a more appropriate vehicle.

The BootUp package provides the generic, logical, framework for optional updating and execution of application images. Most of the heavy-lifting implementation for this is provided by architecture, variant or platform specific code that is incorporated when CYGPKG_BOOTUP is built. This is primarily because the location and format of the relevant application images is platform specific. This provides the greatest flexibility to developers with regards to how applications are stored and what extra information may be embedded to support updating and any other custom bootROM features.

Note

Normally it is envisaged that the BootUp ROM image will be loaded onto devices once, and then very rarely (if ever) updated. The BootUp world is designed to be very simple to minimise the chances of errors in the implementation that would stop the system from booting, or from allowing a different main application to be loaded into the device. The simpler implementations make no use of specific run-time configuration, and all implementations do not impose any footprint on the target after it has started the main application.

BootUp has been utilised by various example target platform implementations to support different mechanisms for safe application updates. These examples can be used as the basis of an update mechanism for your target platform. For example:

- *A simple implementation with no update support.* The on-chip BootUp loader is simply used to validate, load and execute an off-chip NVM based, RAM loaded, application.

  The Atmel SAMA5D3x platform provides an example BootUp implementation for starting RAM based applications. This platform also, optionally, supports secure booting an encrypted applications.

- *A basic update mechanism that is intended to support booting and update of on-chip flash resident applications.* In this case, each time the target is reset BootUp checks if a different, valid, application is present in a separate non volatile memory (NVM) area. If an update is found it will be installed into the on-chip flash, prior to the on-chip flash application being executed. The term “different” is used when distinguishing main application images since BootUp does not interpret the binary signature, which need not be a version number. This allows the update system to be used to revert to an earlier version of the application. It will simply update the on-chip application if an application with a different signature is found in the NVM.

  The application itself is responsible for identifying, acquiring, verifying and storing any update into the NVM. BootUp is responsible solely for installation and execution of the application held in on-chip flash.

  The design of this mechanism avoids halving the available on-chip memory space in order to store images of both the existing and updated application. It also avoids possible target complications resulting from executing out of the same memory space that will need to be erased and written to. However, if the on-chip flash is large enough, and the final application small enough, it is possible for the main and alternative application images to be held on the same on-chip flash.

The robustness (safe update) of an application via this BootUp mechanism relies on the code only attempting to perform an update when a different valid application image is available. If the system loses power, or suffers a reset, during an update the original alternative image will still be available. When the system recovers from the reset state it will just re-start the update (regardless of whether the on-chip application image is still valid, since even if it is, it will still be different from the alternative image that started the original update).

Specific examples of this style of BootUp implementation can be found in the ST STM324x9I-EVAL and ST STM32F7xx-EVAL platforms' `ROMAPP` startup type.

- *A more sophisticated update solution that supports a “bundle” of application specific files.* In this case BootUp is responsible for identifying, loading and executing the main application from within a bundle held in NVM. The application is loaded into and executed in external RAM memory.

  The application itself is generally responsible for identifying, acquiring, verifying and installing any available bundle updates into NVM. BootUp is responsible solely for loading and executing an updated “main” application from within the bundle. Optionally, as a fall-back, if no valid bundle can be located in NVM, then BootUp can locate, verify and install a complete bundle from external media such as an SD card.

  The design of this update mechanism enables, in addition to pure application updates, the update of other target resident code and data. These files might include FPGA bitfiles, DSP firmware, application data and so forth. The bundle update system includes support for decompression and checksumming of the bundle contents. See the Bundle image support chapter for further details.

  Examples of this BootUp approach are available in the ST STM324x9I-EVAL and ST STM32F7xx-EVAL platforms' bundle support.

Architectures, variants or platforms that support the BootUp loader will implement `CYGINT_BOOTUP_APPSTART`. This feature allows BootUp to start the main application and is an essential requirement for the BootUp world to be used.

If BootUp starts when there is no valid on-chip application, and no valid alternative application is available, then if the platform provides the macro `CYG_HAL_BOOTUP_BADAPP` then that is used to call platform specific code. This provides a hook to allow target specific feedback indicating the lack of an application.

Configuration

If `CYGINT_BOOTUP_UPDATE` is not defined then the BootUp loader will simply just execute the “main” application if it is present and valid. If the platform defines `CYG_HAL_BOOTUP_COPYAPP` then platform supplied code to copy an application to its executation location is called (e.g. for RAM loaded applications). In reality if neither `CYGINT_BOOT_UPDATE` and `CYG_HAL_BOOTUP_COPYAPP` are provided by the target then there is no real need for the BootUp to be used, and a normal simple `ROM` application configuration would suffice.

When `CYGINT_BOOTUP_UPDATE` is defined by a platform it indicates that the platform provides the mechanism for identifying if a different alternative application image is available, and that the alternative image should replace the current main (normally on-chip) application when different. It is the responsibility of the platform supplied routines to implement how such image validation is performed.

The following configuration options control the main features of the BootUp bootROM world:

`CYGBLD_BUILD_BOOTUP` controls whether the BootUp image is created, and will normally be defined for all BootUp configurations.

`CYGIMP_BOOTUP_UPDATE` controls whether the BootUp bootROM supports replacing the main application image with a different image from the alternative storage location. If not defined then BootUp will simply start the main application.

`CYG_HAL_BOOTUP_COPYAPP` controls whether the platform provides code to copy an application from NVM to RAM for execution.

Platform Support

A platform supporting the BootUp application update and start features provides the necessary support via specifically named functions. When defined by the platform the BootUp code will call the named macro to perform the necessary action. In the list below the function prototypes are shown using example names. Normally the platform will just map the macro name to a specific named function provided by the platform HAL.

- `CYG_HAL_BOOTUP_VALID_MAIN`

  `cyg_bool` *`plf_bootup_valid_main`*`(void);`

  The function referenced by this macro returns boolean `true` if the main application image is valid (this is the application image that BootUp starts). It will return boolean `false` if no valid main application is found.

- `CYG_HAL_BOOTUP_VALID_ALT`

  `cyg_bool` *`plf_bootup_valid_alt`*`(void);`

  The function referenced by this macro returns boolean `true` if the alternative (pending update) application image is valid *and* different from the current main application. This is the application image that BootUp will *automatically* replace the main application with when it is *different* from the current main application. It will return boolean `false` if no valid alternative application is available, or the alternative image is the same as the current main application.

- `CYG_HAL_BOOTUP_VALID_UPDATE`

  `cyg_bool` *`plf_bootup_validate_update`*`(void);`

  The function referenced by this macro is only called if `CYG_HAL_BOOTUP_VALID_MAIN` has returned `false` to indicate that the main application image is missing or invalid. This macro returns boolean `true` if the alternative (pending update) application image is valid and the main application has been updated successfully. This macro will return boolean `false` if no valid alternative image is available or the update process has failed.

  ## Note

  The `CYG_HAL_BOOTUP_VALID_ALT` and `CYG_HAL_BOOTUP_VALID_UPDATE` macros are mutually exclusive, since they implement slightly different “update” logic based on the presence of a valid main application. A platform should only define one of these macros depending on the style of automatic application update required.

- `CYG_HAL_BOOTUP_UPDATE`

  `cyg_bool` *`plf_bootup_update`*`(void);`

  The function referenced by this macro will replace the main application image with the alternative image. It is only called after the respective images have been validated and only when the alternative image is different as ascertained by a `CYG_HAL_BOOTUP_VALID_ALT` call. It will return boolean `true` if it successfully updates the main application, otherwise boolean `false` is returned to indicate failure.

- `CYG_HAL_BOOTUP_COPYAPP`

  `cyg_bool` *`plf_bootup_copyapp`*`(void);`

  The function referenced by this macro returns boolean `true` if the main application image has been successfully copied to its execution location. It will return boolean `false` if no valid main application is available.

- `CYG_HAL_BOOTUP_BADAPP`

  `void plf_bootup_badapp(void);`

  The function referenced by this macro is called when no valid main application exists. This can be used to provide platform specific feedback to the user that the system cannot boot.

If the `CYGSEM_BOOTUP_PLF_STARTUP` configuration option is enabled then the following C function is called:

`void` **`cyg_plf_bootup_startup`**`();`

This function can be used to provide platform specific optional initialisation prior to the normal BootUp operation being started. This may be performing specific hardware initialisation, or some level of Power On Self Test support.

Application Identity

The identification of valid application images is the responsibility of the platform specific code. Depending on the BootUp model implemented by the target platform, it may be the case that a simple binary value will be enough to distinguish different applications, e.g. a monotonically increasing version number, or the `UTC` timestamp when the binary was produced.

If an application signature needs to be embedded inside the actual application binary then platforms that support the use of BootUp will normally provide a mechanism to provide space within the produced eCos executable. For example the Cortex-M architecture provides the `CYGNUM_HAL_CORTEXM_VSR_TABLE_PAD` configuration option that allows space for an application specific header to be provided at a fixed offset from the binary start. Similarly the ARM architecture allows for the `PLATFORM_PREAMBLE` macro to be define to allow for a header to be installed to hold, for example, the BootUp signature, CRC, or whatever information is needed.

The header/block being used to hold such a “signature” need not just contain information for supporting BootUp. For example, it may also need to hold information for the mechanism used to install the pending update application image into the alternative location.

Building BootUp

Platforms that support a BootUp configuration will normally provide a suitable `.ecm` configuration file to allow a minimal BootUp application to be constructed. The specific platform documentation will provide information regarding BootUp configuration and use, and any specific build sequences. Such documentation should be read in conjunction with this generic BootUp information.

This package provides an example import file in the `$ECOS_REPOSITORY/bootup/<version>/misc/bootup_ROM.ecm.example` file. When porting BootUp support to a new platform this example file can be used as the basis for the target specific `.ecm` fragment.

Should it prove necessary to rebuild a BootUp binary it is done most conveniently at the command line. The steps needed to rebuild a ROM version would be similar to:

```
$ ecosconfig new <target> minimal
[ … ecosconfig output elided … ]
$ ecosconfig import $ECOS_REPOSITORY/hal/TARGETPATH/VERSION/misc/bootup_ROM.ecm
[ … ecosconfig output elided … ]
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

At the end of the build the `install/bin` subdirectory will contain the file `bootup.bin`. This may be programmed into the target platform as specified for the system being targeted.

Applications using `VALID_ALT`

The following section is only relevant for target platforms that define `CYG_HAL_BOOTUP_UPDATE` and `CYG_HAL_BOOTUP_VALID_ALT` where the BootUp world is designed to allow for robust in-field updates of the firmware on a device.

For such systems the application started by BootUp must implement some of the support needed with regard to providing the “pending update” application image. When a new firmware update has been downloaded by the main application it will normally force a system restart, which will cause BootUp to perform the necessary update prior to starting the new main application.

The main application, when loading a pending update application into the alternative memory/device (e.g. off-chip SPI), should ensure that it does not set (write) the "valid signature" until it has validated the correct storage of the rest of the image. This requirement ensures that if a system reset occurs before the signature has been written that the image will NOT be interpreted by BootUp as being valid. This allows the BootUp loader to be simpler since it can treat a valid, complete, signature as an indication that the application was written correctly.

At a minimum (and critically) the main application needs to ensure that it invalidates the alternative image signature prior to starting the process of storing a new application image into the alternative area. This ensures a partial image is not incorrectly identified as a valid, pending, update. If CRC checking is also being used by the platform HAL then this likelihood is minimised.

Note

This requirement that the alternative storage area must have a complete valid image, whenever there is a complete signature present, ensures that in the case where the implementation does not use a CRC (or similar check) for the BootUp code to perform validation of a complete image, that any previous previous valid signature is not incorrectly accessed and treated as “valid”. For example, if the writing of a subsequent pending update is interrupted (CPU reset, etc.) before it “overwrites” the signature of the previously stored image. If a pending update is downloaded to the holding area, validated and marked as such *BUT* the system is not rebooted (so the pending update is not applied). Subsequently another pending update starts to download a different image to the holding area but the process is interrupted before the previous image “valid” signature is over-written. The BootUp code could then incorrectly interpret the invalid (partially overwritten) image as being valid if no other validity checks are performed. This is why the *rule* exists that the code performing the pending update storage should first invalidate the signature.

It is the responsibility of the main application storing the pending update to ensure that the complete image is valid *PRIOR* to finalising the platform specific signature. To reiterate, the BootUp package, in combination with the platform HAL support, only requires a simple “signature validity check” for robustness *WHEN* the code storing the pending update ensures that if there is a valid signature in the alternative area then the rest of the bytes are also valid (i.e. a complete valid image is in place).

Since the BootUp will only perform an update when the valid alternative image is different, there is no strict need to erase the signature/image until the main application needs to provide a new update. However, doing so during its normal startup will avoid BootUp having to spend CPU cycles validating and checking the alternative image prior to checking the signature and deciding not to update.

Notes

1. To reiterate, the main application (not BootUp) is responsible for invalidating (erasing/removing) the alternative image, or at least the signature. This allows BootUp to be simpler since it only needs to be able to read from the alternative image location.

2. The signature can either be embedded in the application (assuming the caveat of it being the last data written), or it can be a totally separate distinct section maintained in conjunction with the update application image. Such design decisions are left to the specific platform HAL support. The use of other validation mechanisms, e.g. a CRC, will

only be required if the medium that is used to store the alternative (pending) image can suffer from data corruption, or transient read errors. Such a situation will be extremely unlikely for the vast majority of implementations.

Supported Platform HALs and targets

BootUp is supported by the following platform HALs:

- NXP i.MX RT10XX Variant HAL
- SAMA5D3x-MB Platform HAL
- SAMA5D3 Xplained Platform HAL
- ST STM324x9I-EVAL
- ST STM32F7xx-EVAL
- STM32F746G-DISCO Platform HAL
- STM32H735-DISCO Platform HAL
- STM32L4R9-DISCO Platform HAL
- STM32L476-DISCO Platform HAL
- **Note**

  This is not a complete list. Refer to your platform documentation to determine if it supports `CYGPKG_BOOTUP`.

Part LXIX. Bundle image support

Table of Contents

Chapter 211. Bundle overview

Introduction

The CYGPKG_BUNDLE package implements support for a simple, compact, multi-element, binary distribution format, referred to as a "bundle". The bundle format is primarily designed to be used for in-field system updates in conjunction with the BootUp lightweight BootROM package.

The format can be easily parsed and processed by deeply embedded systems, and is used to encapsulate multiple discrete binary data blobs and optional metadata. Bundles can therefore incorporate all the elements that may be required for a system update, including application executables, FPGA bitfiles, DSP firmware, application data and so forth.

The bundle package implements an API to parse and extract data from a bundle and an associated host tool used to create and manage bundle images.

The STM324X9i-Eval platform includes an example implementation of the BootUp bootROM package that uses the bundle package format as the distribution format underpinning its system update mechanism.

A bundle is normally expected to be used as a matched collection of binaries, and is treated as a whole with regards to production and in-field updates. It is *NOT* expected that in-field operations will ever split and re-combine elements of a deployment package in the field. As such the format is read-only for eCos applications.

Although the bundle format is designed to be flexible and allow modification and extensions, it is vital that for a given target platform that the host-based creation tool and runtime code share a common format and set of expected features. For example the use of MD5 as a "hash" and `zlib` as a compressor.

Configuration

This section shows how to include the bundle support into an eCos configuration, and how to configure it once installed.

The bundle support is contained in a single eCos package `CYGPKG_BUNDLE`. However, it depends on the services of a collection of other packages for complete functionality.

Incorporating the bundle support into your application is straightforward. The essential starting point is to incorporate the bundle eCos package (`CYGPKG_BUNDLE`) into your configuration.

This may be achieved directly using **ecosconfig add** on the command line, or the **Build->Packages…** menu item within the eCos Configuration Tool.

`CYGFUN_BUNDLE_COMPRESS`

> If the eCos `zlib` package `CYGPKG_COMPRESS_ZLIB` is configured then, when enabled, this option provides support for decompressing items. If the `CYGPKG_COMPRESS_ZLIB` package is not available, or this option is disabled, then only uncompressed items within a bundle can be read.

`CYGIMP_BUNDLE_HASH_MD5`

> This option can be enabled to include support for the `RFC 1321` MD5 Message-Digest Algorithm as a valid hash used to verify data integrity.

`CYGIMP_BUNDLE_HASH_SHA256`

> This option can be enabled to include support for the FIPS PUB 180-2 SHA-256 hash as a valid hash used to verify data integrity.

`CYGIMP_BUNDLE_HASH_CRC32`

This option when enabled implements support for the standard IEEE 802.3 (Ethernet) CRC-32 as a valid hash used to verify data integrity.

`CYGNUM_BUNDLE_BUFSIZE`

This option defines the size of the internal buffer used for decompressing data. Currently it makes use of a dynamic memory allocation and so may need to be tuned appropriately for target systems with a small dynamic heap.

`CYGBLD_BUNDLE_BUNDLE`

This option allows the host-side `bundle` tool to be automatically built on suitably capable systems.

Note

This option is disabled by default, since normally only a Linux system with a standard `zlib` library installation would automatically succeed. The package supplied `host/Makefile` provides an example of cross-building (under Linux) the host tool for Windows.

`CYGDBG_BUNDLE_DEBUG`

If this option is enabled then it provides access to individually controlled CDL debug options for various sub-systems or package features. This allows the detail and amount of debug information to be controlled. Normally such diagnostic output would only need to be enabled for developers working on the internals of the bundle processing.

`CYGTST_BUNDLE_BUILD_TESTS`

This option enables the building of any bundle run-time verification tests included in the package.

Chapter 212. Bundle format

Introduction

A “bundle” file has a custom format, designed to minimise the code overhead of supporting the bundle on the target platform.

Note

An advantage of a custom binary format over, for example, the use of a tar-file, is that the binary image held on an SD card for update delivery would be *IDENTICAL* to the version installed in the target's SPI flash.

With the tar-file format there is an overhead associated with individual entries (e.g. 512-byte header, per-item padding) that would mean for (the limited space) target SPI flash storage the processing code would need to extract the individual items for efficient storage in the SPI flash. This process can be avoided by a simple binary structure that is copied unchanged.

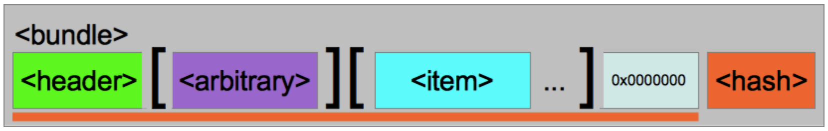
A host-based command-line tool is provided to create a correctly formatted bundle binary image. The binary image can then be used in manufacturing and in-field update processes to distribute the main target application and all associated target data.

Internal Structure

A bundle is a contiguous binary image containing a set of uniquely tagged items. Individual items may be compressed (using the standard `zlib` compression library). The bundle, and each item within a bundle, can optionally have “arbitrary” metadata attached. An overall hash object is used as a verification of the data integrity of the complete bundle image, but individual items also have the option of hashes covering their specific item data, and in the case of compressed items, a hash that can be used to validate the decompressed data.

The following figure highlights the underlying bundle format. The <header> block provides the basic bundle identification and description information. As previously discussed the bundle <arbitrary> block is optional and may not be present in all bundle images. It is perfectly valid for a bundle to contain no items, though a valid bundle always contains the <item> terminator `0x00000000` marker. The bundle is completed with a <hash> object providing the verification code for the complete bundle up to (but not including) the <hash> block.

Figure 212.1. <bundle> image

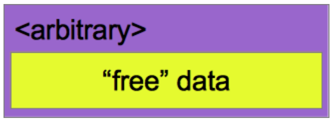


The *optional* <arbitrary> data that may be attached to the complete <bundle>, or to individual <item>s, is just a contiguous, uncompressed (as far as the bundle processing code is concerned), block of metadata. This metadata is not interpreted by the host-tool or bundle API code, but may be used by the target application as required. For example it could be used to hold a set of “key[=value];” pairs to provide “Identification” data for human-readable or programmatic access as required. e.g. it could encode tags like:

```
Version=1.02_B02;Required-HW-Version=rev1b;Development;
```

This arbitrary metadata could also be used as a mechanism for “Manifest” style information, which may be needed to support an organization's release process requirements.

Figure 212.2. <arbitrary> chunk



A <hash> object is used to hold a validation hash which can be used to ensure data integrity. Support is provided for avoiding the overhead of data verification by the special `CYG_BUNDLE_HASH_NONE` signature. However, in most situations (especially when the bundle format is being used as an in-field update delivery mechanism for example) some level of hash verification should be considered.

Figure 212.3. <hash> chunk

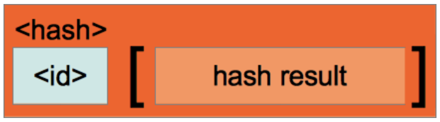


Table 212.1. HASH signatures

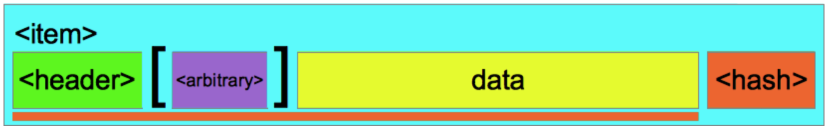
HASH Signature	Description
`CYG_BUNDLE_HASH_NONE`	This is used when no data validation hash support is required. The <hash> object does not contain a “hash result”.
`CYG_BUNDLE_HASH_CRC32`	A standard 4-byte IEEE 802.3 (Ethernet) CRC-32 value.
`CYG_BUNDLE_HASH_MD5`	A 16-byte `RFC 1321` MD5 Message-Digest Algorithm value is held. Though the use of MD5 is deprecated for cryptographic security purposes, its use as a <hash> for “data validity” checks is perfectly acceptable.
`CYG_BUNDLE_HASH_SHA256`	A 32-byte FIPS PUB 180-2 SHA-256 hash value is held.

A <bundle> comprises a set of items, each of which has a simple fixed “enum”-style tag rather than using fixed “ASCII filenames” to identify its specific purpose or use. This tag “namespace” is managed by the platform/customer and is not interpreted by the tools or bundle package API. For example, the customer may have the following mappings: `0x0001`==HostMCU, `0x0401`==MotorDSP, `0x0812`==LineDSP, `0x8003`==MotorTable, etc. The tag is used as both the name and filetype as required by the target application. This approach saves some binary space in the bundle format, plus the run-time overhead of string parsing when dealing with bundle items.

Data items are held either uncompressed or compressed, and may contain an optional <arbitrary> metadata block.

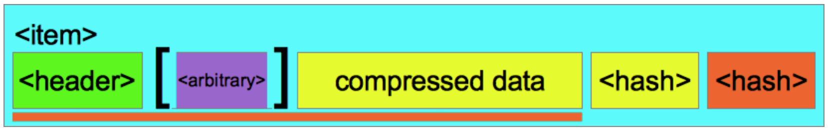
For uncompressed items a single <hash> object is appended covering the validation of the complete <item> chunk.

Figure 212.4. Uncompressed <item>



For compressed items a seperate <hash> object is maintained provided the hash value for the original, uncompressed, source data.

Figure 212.5. Compressed <item>



All <bundle> and <item> descriptor fields are held in the binary image in the target native endianess, and it is the responsibility of the host tool used to create the bundle to ensure the correct target endianness is selected. This ensures that the deeply embedded target does not need to swap bytes, as might be the case if a single fixed endianness was chosen for the bundle format. The bundle header 32-bit signature is designed to allow the endianness of the bundle to be detected at run-time, providing a simple validity check on the target that the bundle is valid.

Note

The host tool creating the bundle may need to be explicitly told the target endianness, which may be different from the build host native order. The provided example host `bundle` application provides the `--little-endian` and `--big-endian` options which can be used to override the host default selection if needed.

As discussed, a bundle is identified by a fixed header 32-bit signature. This signature is used to identify the binary as containing a bundle. At its simplest this signature is used to identify the platform/hardware that will accept the bundle (since the platform based applications will only recognise a valid matching signature value). The system support allows for a MASK/VALUE pair to be defined to allow (if required) a subset of the 32-bit signature to be used for “bundle is valid for this run-time” acceptance.

Notes

1. The “customer specific” field (currently 8-bits) can be used to provide a mechanism to restrict <bundle> images to specific platforms (or revisions of a platform) as required.

2. The `bundle` host executable provided within your eCosPro installation is required to construct, and examine, “bundle” binaries. This executable must be on your path, and will normally be copied into eCosPro host tools bin directory. Under Windows this is typically `C:\eCosPro\ecoshosttools\bin` and on Linux hosts `/opt/ecospro/ecoshosttools/bin`. The above bin directory will be on your path if you use the eCos GUI configuration tool or the eCos CLI Shell environment.

3. The bundle <header> and item <header> descriptors contain a `flags` field that can be extended as required to have any extension settings as needed to add extra information blocks into a bundle or item respectively. This allows for future extensions to the format to be added without affecting software (e.g. the BootUp firmware or the BootUp loaded main application) already in the field, since they will not interpret or be affected by any extra data that may be provided.

Chapter 213. Bundle API

The bundle API provides routines to extract data from a binary bundle image.

For eCos applications the approach used is that a suitable “access” strategy is chosen depending on where the bundle image is held. The access initialisation returns an opaque bundle descriptor reference that is used by an application to find a specific item. Similarly the opaque item descriptor reference is subsequently used to read the required data. The descriptor references need to be released when no longer required to ensure any held resources are returned to the system.

API

Name

cyg_bundle_access_direct — Initialise “direct” bundle context

Synopsis

#include <cyg/bundle/bundle_api.h>

```
cyg_bundle_context_t *cyg_bundle_access_direct(bundle_mem, blen);
```

Description

Initialises context for accessing contiguous directly-accessable (memory-mapped) based bundle image. This access strategy would be used for RAM or memory-mapped flash/ROM based bundles.

Return value

Pointer to object or NULL if unable to create context.

Name

cyg_bundle_access_file — Initialise “file” bundle context

Synopsis

#include <cyg/bundle/bundle_api.h>

```
cyg_bundle_context_t *cyg_bundle_access_file(fd);
```

Description

Initialises context for accessing a file based bundle image. This function requires the caller to have initialised and accessed the relevant file system, and to pass a valid “readable” file descriptor of an opened bundle image file. This functionality is only available if `CYGPKG_IO_FILEIO` is configured.

Return value

Pointer to object or NULL if unable to create context.

Name

cyg_bundle_access_flash — Initialise “flash” bundle context

Synopsis

#include <cyg/bundle/bundle_api.h>

```
cyg_bundle_context_t *cyg_bundle_access_flash(bundle_addr, limit_addr);
```

Description

Initialises context for accessing non-memory-mapped flash based bundle images via the standard eCos flash API. This “flash” access mechanism is primarily for indirectly-accessed flash memory (e.g. SPI devices), though nothing prohibits the use of the flash API from accessing memory-mapped flash images. This functionality is only available if `CYGPKG_IO_FLASH` is configured.

Note

If the target flash memory *is* accessible as a readable memory-mapped area then it is highly recommended to use the “direct” access mechanism for performance and dynamic memory footprint reasons.

Return value

Pointer to object or NULL if unable to create context.

Name

cyg_bundle_access_init — Common bundle context initialisation

Synopsis

#include <cyg/bundle/bundle_api.h>

```
cyg_bundle_context_t *cyg_bundle_access_init(am, aminitctx);
```

Description

The application developer should only need to use this function if providing an alternative access method. This may be needed if none of the default access strategies provide the necessary support. e.g. image is held on a memory using a non-standard bus connection.

Return value

Pointer to object or NULL if unable to create context.

Name

cyg_bundle_access_release — Release bundle context

Synopsis

#include <cyg/bundle/bundle_api.h>

```
void cyg_bundle_access_release(bc);
```

Description

Releases a previously allocated bundle context. To avoid resource leaks the application should always release a previously initialized access bundle context when it is no longer required.

Name

cyg_bundle_verify — Verify bundle and initialise context references

Synopsis

#include <cyg/bundle/bundle_api.h>

`int` **`cyg_bundle_verify`**`(`*`bc`*`, `*`valid_signature_mask`*`, `*`valid_signature`*`);`

Description

The *`bc`* parameter should be a bundle descriptor reference as returned by a suitable `cyg_bundle_access_`…`()` call.

After obtaining a context for the chosen “access” strategy this function **must** be called to verify that the bundle is valid prior to any subsequent data access API calls, since this function also initialises context data-structures required by the item access API functions.

Currently the *`valid_signature_mask`* and *`valid_signature`* parameters are **not** used. The code may be extended in the future to allow bundle validity to be selected based on the 8-bits of platform/application ID specific signature.

The following pseudo code is an example of expected, normal, use case of this function after selecting an access method:

```
#include <pkgconf/system.h>
#include <pkgconf/bundle.h>

#include <cyg/bundle/bundle.h>
#include <cyg/bundle/bundle_api.h>

void do_some_work(int fd)
{
    cyg_bundle_context_t *bc;

    bc = cyg_bundle_access_file(fd);
    if (bc) {
        if (CYG_BUNDLE_VALID == cyg_bundle_verify(bc,CYG_BUNDLE_SIG_MASK,CYG_BUNDLE_SIG_BASE)) {

            … perform item, or bundle arbitrary data, operations …

        } else {
            …report bundle validity error…
        }

        cyg_bundle_access_release(bc);
    } else {
        …report file access error…
    }

    return;
}
```

Return value

If the bundle is valid then the value `CYG_BUNDLE_VALID` is returned, and the *`bc`* context will be valid for subsequent item API calls. If the bundle data is invalid, or cannot be read, then `CYG_BUNDLE_INVALID` is returned.

Name

cyg_bundle_item_find — Provide handle onto bundle item

Synopsis

#include <cyg/bundle/bundle_api.h>

`cyg_bundle_object_t *`**`cyg_bundle_item_find`**`(`*`bc`*`, `*`entag`*`);`

Description

This function is used to generate an item access descriptor that can be subsequently used to extract data from a bundle.

The *`bc`* parameter should be a bundle descriptor reference as returned by a suitable `cyg_bundle_access_...()` call.

The *`entag`* parameter identifies the information to be accessed. The simplest form consists of passing a simple 16-bit (non-zero) `tag` identifier, used when the application requires access to the item data.

The `CYG_BUNDLE_ARBITRARY` flag can be OR-ed with the `tag` value to request access to any “arbitrary” data that may be associated with the requested `tag` item. When using `CYG_BUNDLE_ARBITRARY` then a `0x0000` tag value is acceptable, and is used to reference the parent bundle “arbitrary” data.

Return value

Pointer to the required object descriptor, or NULL if unable to find the data described by the *`entag`* or if an invalid *`entag`* value was used.

Name

cyg_bundle_item_release — Release reference to specific bundle item

Synopsis

#include <cyg/bundle/bundle_api.h>

```
void cyg_bundle_item_release(ic);
```

Description

Release handle onto bundle item. This releases any state held regarding the referenced item. The application should always release the item descriptor when it is no longer required to avoid resource leaks.

Name

cyg_bundle_enumerate — Enumerate bundle contents

Synopsis

#include <cyg/bundle/bundle_api.h>

```
int cyg_bundle_enumerate(bc, cb, private);
```

Description

This function is passed a callback function *cb* that is called once for each item present in the referenced *bc* bundle. The *private* parameter is an application specific context passed to the callback routine, and may reference any data required.

The callback function is passed the specific item identifying *tag*, along with the *length* of the item data. The *arblength* parameter provides the length of any arbitrary data associated with the item, or `0x00000000` if the item has no arbitrary data attached.

The callback function is passed the special *tag* value of `0x0000` to indicate “no more items”.

Return value

If at least 1 item exists in the bundle then `CYG_BUNDLE_VALID` is returned. If the passed *bc* parameter is invalid, or the bundle contains no items then the function will return `CYG_BUNDLE_INVALID`.

Name

cyg_bundle_info — Length of raw item

Synopsis

#include <cyg/bundle/bundle_api.h>

```
uint32_t cyg_bundle_info(ic);
```

Description

This function returns the length of the referenced object data. For uncompressed items (or “arbitrary” attachments) this is the length of the data as held in the bundle image. For compressed items this is the length of the original, raw, data.

Return value

Length of item or 0 if cannot be ascertained.

Name

cyg_bundle_read — Extract data from a bundle item

Synopsis

#include <cyg/bundle/bundle_api.h>

```
uint32_t cyg_bundle_read(ic, dst, offset, amount);
```

Description

This function copies a chunk of data from the reference bundle object to the supplied *dst* address. The *offset* parameter is the byte-offset within the data object where reading should start, with the *amount* specifying the number of bytes to be read. If the data is held in a compressed format then it will be automatically decompressed before being written to the supplied *dst* address. In this case the *offset* parameter always refers to the offset within the decompressed data.

Return value

Number of bytes read, or 0 on error. The returned number of bytes read may be less than the specified *amount* if an attempt is made to read more data than is available for the referenced object.

Chapter 214. Host tool

Introduction

The host-based command-line `bundle` tool is used to create bundle format images, as well as add and delete items, list and verify the contents, extract data, and other relevant commands.

```
Usage: bundle [option(s)] <bundlefilename> [cmds]
```

For a full list of the `bundle` tool's options and commands use the `--help` option:

```
$ bundle --help
```

The <bundlefilename> should always be supplied. The default behaviour if no operations are specified is to `verify` the <bundlefilename> and `list` its contents.

Explicit parameter options are case-insensitive. e.g. When providing a *`hash`* value either `MD5` or `md5` are acceptable and identical in their result.

The command-line bundle parameters are processed left to right, with parameters for operations following the command. The following examples are identical in the result produced:

```
$ bundle testbundle_minimum_md5.bin create hash md5
```

```
$ bundle testbundle_minimum_md5.bin hash md5 create
```

Both examples would create a bundle file `testbundle_minimum_md5.bin` that incorporates an md5 hash.

The following example creates a bundle called `egbundle.bin`. The bundle has a signature of `0x1` and includes bundle metadata from the `manifest.txt`. The entire bundle's contents will be covered by an sha256-based hash. The bundle would contain a single compressed item identified by a tag of `0x2`. The item's data comes from `short.bin` and its metadata from `itemarb.txt`. The item's contents including the tag, metadata, and compressed data will be covered by an md5 hash. Note that as the item includes compressed data an additional md5 hash for the uncompressed data would also be incorporated.

```
$ bundle egbundle.bin create hash sha256 signature 0x01 arbitrary manifest.txt add \
        0x0002:short.bin:compress:md5:itemarb.txt
```

The `bundle` tool's *`add`* command is quite flexible in its usage. It has a single parameter describing the item to be added, with multiple fields separated by ':' characters. Only the *`tag`* and *`srcfile`* are required, with other optional fields specifying whether the source file should be compressed, whether or not a hash should be added and finally what metadata, if any, is to be included with the item.

```
add tag:srcfile[:[compress][:hash[:arbitrarymetadata]]]
```

The *`tag`* value of `0x0000` is reserved for the system, with all other tag values being available for application use. This gives a maximum possible bundle item count of `65635` items, which is unlikely to be a limitation for embedded targets. Most bundles will contain a small (<`10`) number of items, e.g. the main target application, FPGA initialisation data, device initialisation tables, etc.

`add` command examples:

Uncompressed, no hash

```
add 0x0001:datafile
```

Compressed, no hash

```
add 0x2:dfile:compress
```

Compressed, MD5 hash

```
add 11:dfile:c:md5
```

Uncompressed, CRC-32 hash with arbitrary data

```
add 0xF:dfile::crc32:textfile
```

Compressed, no hash with arbitrary data

```
add 2:dfile:C::textfile
```

The `--commands` option allows for a fixed set of bundle operations to be held in a file, rather than being passed individually as command line options to the tool. Individual operations (e.g. command name and its required arguments) should *NOT* be split across lines, but multiple lines are allowed. Lines that start with '#' as the first non-whitespace character are treated as comments, and ignored.

Note

No item metadata add commands are included since we can use “extract” to get data and then “delete” and re-“add” with the required arbitrary file if required.

Warning

Filenames in commands cannot contain ':' since it is used as a field delimiter in the <item> description.

The bundle package includes the complete sources for the host tool. eCosPro releases include a pre-built version of the tool for either Linux or Windows as appropriate. If needed the tool can be built from the supplied source, with the only non-standard-library dependency being the “zlib” library. Linux systems will normally provide “zlib” as standard, though it is easily built if required for other host systems.

Chapter 215. Bundle tests

bundle1

The bundle1 test application provides a set of pre-built valid and invalid bundle images, and verifies the bundle run-time support against these “known” images. It provides a method of verifying that any changes to the eCos bundle access sources remain backwards compatible and have not had an adverse effect.

Part LXX. RTT

Table of Contents

Chapter 216. RTT overview

Introduction

The `CYGPKG_RTT` package provides a standard Segger RTT (Real Time Transfer) implementation to eCos applications.

Note

Host access to RTT channels is not limited to the Segger debug tools. Access is supported by 3rd-party tools, including, for example, Ronetix PEEDI and OpenOCD. It is not a feature limited to use with J-Link/J-TracePro H/W debug setups.

This package is covered by an “AS IS” license as distributed in the original RTT package:

Example 216.1. “AS IS” License

```
                SEGGER Microcontroller GmbH
                   The Embedded Experts

       (c) 1995 - 2021 SEGGER Microcontroller GmbH
      www.segger.com     Support: support@segger.com

    SEGGER RTT  Real Time Transfer for embedded targets


All rights reserved.

SEGGER strongly recommends to not make any changes
to or modify the source code of this software in order to stay
compatible with the RTT protocol and J-Link.

Redistribution and use in source and binary forms, with or
without modification, are permitted provided that the following
condition is met:

- Redistributions of source code must retain the above copyright
 notice, this condition and the following disclaimer.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES,
INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
DISCLAIMED. IN NO EVENT SHALL SEGGER Microcontroller BE LIABLE FOR
ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT
OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS;
OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE
USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH
DAMAGE.
```

For definitive RTT documentation please refer to the main Segger J-Link RTT website. We do not duplicate that documentation here.

Chapter 217. Configuration

This chapter shows how to incorporate the RTT support into an eCos configuration, and how to configure it once included.

Configuration Overview

The RTT support is contained in a single eCos package `CYGPKG_RTT`. However, some functionality may be dependant on other eCos features. e.g. the Cortex-M HAL.

Quick Start

Incorporating the RTT support into your application is straightforward. The essential starting point is to incorporate the RTT eCos package (`CYGPKG_RTT`) into your configuration.

This may be achieved directly using **ecosconfig add** on the command line, or the **Build->Packages…** menu item within the eCos Configuration Tool.

Depending on the RTT package configuration, and the other packages present in the configuration, further package specific configuration **may** be required. For example, the `CYGPKG_HAL_CORTEXM` packages allows for the use of RTT for HAL diagnostics, but the option needs to be explicitly selected.

Options

Various package specific configuration settings define how the RTT world is present.

`CYGNUM_RTT_MAX_UP_BUFFERS`

This option configures the number of UP buffers (Target->Host) available. This value should be at least `2` if also using `CYGPKG_SYSTEMVIEW` for event tracing.

`CYGNUM_RTT_BUFFER_SIZE_UP`

This option configures the size (in bytes) of the buffer used for terminal output (UP channel 0) from the Target to the Host.

`CYGNUM_RTT_MAX_DOWN_BUFFER`

This option configures the number of DOWN buffers (Host->Target) available.

`CYGNUM_RTT_BUFFER_SIZE_DOWN`

This option configures the size (in bytes) of the buffer for terminal input from the Host to the Target. For exampe this could be used for keyboard input. Normally the terminal DOWN buffer can be significantly smaller than the corresponding terminal UP buffer since external tool interaction is going to be less than diagnostics generated.

`CYGNUM_RTT_BUFFER_SIZE_PRINTF`

This option configures the size of the internal buffer used for RTT bulk-send of `printf` characters via RTT.

`CYG_RTT_MODE`

This option sets the default, initial, operation mode for Target->Host buffers. Use `NO_BLOCK_SKIP` for non-blocking TX where data not yet uploaded is overwritten. Use `NO_BLOCK_TRIM` for non-blocking TX where further data is dropped if the upload buffer is full. Use `BLOCK_IF_FIFO_FULL` when the RTT calls should block until space is available in the upload buffer, so no data is lost. The default setting can be over-ridden at run-time by an application if required, via the `SEGGER_RTT_ConfigUpBuffer()` function.

`CYGNUM_RTT_MEMCPY_BYTELOOP`

If required this option replaces the standard `memcpy()` usage with a byte loop. The byte loop can have a lower overhead than `memcpy()` if small amounts of data are being copied, and may be required for some architectures where memory access restrictions are in place.

`CYGNUM_RTT_MAX_INTR_PRI`

The priority for the RTT lock.

`CYGDAT_RTT_SECTION`

This option defines a string with the name of the section to be used for the RTT data objects. If relevent for a target configuration this will allow the explicit placing of the RTT structures into a specific memory area (e.g. SRAM). It is the responsibility of the developer to choose a section name supported by the target architecture linker script. For example, for most target architectures this can be a section named with the prefix "`.sram`", e.g. "`.sram.rtt`". If the option is not enabled then the RTT data structures are linked to the normal data section for the target configuration.

If this option is enabled then the further option `CYGDAT_RTT_BUFFER_SECTION` is available, which can be set to allow the explicit placement of RTT buffers. The `CYGDAT_RTT_BUFFER_SECTION` option defines the string with the name of the section to be used for the RTT up and down "Terminal" buffers. If this option is not enabled, but `CYGDAT_RTT_SECTION` is, then the buffers will be placed in the named `CYGDAT_RTT_SECTION`.

`CYGBLD_RTT_PRINTF`

The `SEGGER_RTT_printf()` implementation is not normally required, since the standard eCos diag_printf() provides similar functionality. This option can be enabled if required by the application.

Chapter 218. eCos port

Overview

The goal for the CYGPKG_RTT package is to avoid, where possible, having to have any core RTT source file changes made specifically for eCos. This is to ensure that re-imports of newer versions of the Segger sources involve minimal effort. The files are as provided in the official JLinkRTT release package as imported, with the following exceptions:

1. Files have been moved, unmodified, to create a standard eCos package tree structure to integrate with the eCosPro build environment

   Only **relevant** files from the original project have been included in the eCos package.

2. The file include/Config/SEGGER_RTT_Conf.h is a wrapper that includes the original, **unmodified**, version of the Segger supplied header. This is done to allow for configuration of the underlying RTT system without having to change the original header source file.

3. Portions of the test source files found in the original Examples directory have been used, but wrapped as eCos test applications. This has been done to avoid inclusion of unnecessary run-time code.

The current RTT version provided by the eCos package is the SEGGER_RTT_V788m release contained within the JLink_Linux_V788m_x86_64.tgz Linux package.

The following table highlights the files taken from the RTT package and their new location within the eCos CYGPKG_RTT package:

Original	eCos package
Config/SEGGER_RTT_Conf.h	include/Config/base/SEGGER_RTT_Conf.h
RTT/SEGGER_RTT.h	include/RTT/SEGGER_RTT.h
RTT/SEGGER_RTT.c	src/SEGGER_RTT.c
RTT/SEGGER_RTT_ASM_ARMv7M.S	src/SEGGER_RTT_ASM_ARMv7M.S
RTT/SEGGER_RTT_printf.c	src/SEGGER_RTT_printf.c
LICENSE.md	doc/LICENSE.md
README.md	doc/README.md

The include structure clunkiness is a side-effect of Segger embedding relative pathnames in their original sources, which we have maintined for the sake of forward maintenance.

Chapter 219. Test Programs

Test Programs

Some RTT specific tests are built and can be used to verify correct operation of the support. The test cases built depend on the configuration options.

1. `rtt_printf`

   This test is built when `CYGBLD_RTT_PRINTF` is enabled. It provides an example of `SEGGER_RTT_printf()` use.

Note

`CYGBLD_RTT_PRINTF` is disabled by default since the standard eCos `diag_printf()` et-al provides for vararg based diagnostics, and so including `SEGGER_RTT_printf()` will just be an increase in the application footprint, since the eCos diagnostic routines are likely to be present from use by other packages.

Part LXXI. eCos Support for Segger SystemView tracing

Table of Contents

Chapter 220. SystemView overview

Introduction

Current version based on `SystemView_Src_V350a.zip` downloaded from the Segger website.

Some of the files in this package are covered by the Segger AS-IS license.

The RTT sample code can also be found in the JLink package. e.g. `JLink_Linux_V695a_x86_64/Samples/RTT/SEGGER_RTT_V695a.tgz`

The following table highlights the files taken from the original `SystemView_Src` release, and their new location within the eCos `CYGPKG_SYSTEMVIEW` package:

Original	**eCos package**
`Config/SEGGER_SYSVIEW_Conf.h`	`include/Config/SEGGER_SYSVIEW_Conf.h`
`Config/Global.h`	`include/Global.h`
`SEGGER/SEGGER.h`	`include/SEGGER.h`
`SEGGER/SEGGER_SYSVIEW_ConfDefaults.h`	`include/SEGGER_SYSVIEW_ConfDefaults.h`
`SEGGER/SEGGER_SYSVIEW.h`	`include/segger_SYSVIEW.h`
`SEGGER/SEGGER_SYSVIEW_Int.h`	`src/3rd_party/SEGGER/SEGGER_SYSVIEW_Int.h`
`SEGGER/SEGGER_SYSVIEW.c`	`src/3rd_party/SEGGER/SEGGER_SYSVIEW.c`

Currently these Segger supplied sources are **unmodified**, which should allow for easier updating to newer releases from Segger if/when required.

The SystemView host application has support for receiving telemetry from: J-Link/RTT H/W debug connection, serial line (UART) or a TCP/IP network connection.

The Segger SystemView documentation (Segger document: UM08027) provides an overview of using the host application, as well as technical details of the target implementation API, of which this eCos package is an example. If required, the developer can embed suitable target side calls into their code as needed (e.g. `SEGGER_SYSVIEW_MarkStart()`) when relying on the support from this package.

Note

In this documentation we refer to the eCos target side implementation as `CYGPKG_SYSTEMVIEW` (the target package), and use SystemView to refer to the host side Segger application.

Chapter 221. SystemView Recording

There are two forms of continuous recording supported. The use of a H/W debugger or a dedicated Recorder communication channel.

H/W debugger

This mechanism records data from an active H/W debugger RTT session, where the H/W debugger directly accesses the target memory to extract the SystemView events. There is no requirement for any (other than the basic core `CYGPKG_SYSTEMVIEW` setup) run-time code on the target being sampled, and normally would provide for the lowest overhead, highest throughput, mechanism for obtaining instrumentation data.

As with all tracing involving target side software (in this case the writing of the event information into the RTT memory buffers), there will be a cost to the performance of the system not experienced when `CYGPKG_SYSTEMVIEW` is not enabled.

Support for receiving RTT recorded events into the host SystemView application can take two forms. Direct J-Link/J-Trace support or a socket based TCP/IP proxy helper service.

J-Link/J-Trace H/W debugger

This direct H/W debugger access is selected by the SystemView application `J-Link` Recorder Configuration option.

svproxy

When using a H/W debugger interface that supports exposing RTT channels via TCP/IP network streams then a proxy helper application can be used to obtain the H/W debugger accessed RTT buffer data for SystemView from the debugger, and present a SystemView application `IP` Recorder channel to the viewer application.

The `host/svproxy.c` source implements a simple BSD socket interface Linux example implementation. It provides a connection between the Segger SystemView application `IP` Recorder support and a non-Segger H/W debugger which presents the data from RTT channel 1 "SystemView" as a network stream (for example Ronetix PEEDI, or OpenOCD).

The proxy provides a service on a specified port allowing the SystemView application to connect and wait for event data to be supplied. The proxy does not need to execute on the same host as the SystemView application.

The target H/W debugger supplied stream should be referenced using its network address and port number as presented by the H/W debugger session. NOTE: The examples below assume that the target H/W debug session has been started, and that the eCos application executed until after the `SEGGER_SYSVIEW_Conf()` function has completed (to initialise the data structure searched for by the H/W debugger). This can be achieved by setting a breakpoint on the function `cyg_hal_invoke_constructors()` as a holding place whilst the H/W debugger network server is started and the `svproxy` tool executed.

For example, assuming a PEEDI configured to expose the RTT channel 1 on port 19112 as follows:

```
rt1064evk_swd> rtt setup 0x80030000 0x10000
rt1064evk_swd> rtt start
++ info: control block found at 0x8003AD1C
rt1064evk_swd> rtt list
target -> host (max 2):
    0: 'Terminal', size 1024
    1: 'SysView', size 8192
host -> target (max 2):
    0: 'Terminal', size 16
    1: 'SysView', size 8
rt1064evk_swd> rtt server_start 19112 1
```

Using OpenOCD is very similar, where via the telnet server exposed via OpenOCD or using the GDB `monitor` command support, a RTT server can be started supplied channel 1 on a network connection. The only difference being that the `rtt setup` requires the Segger identification string when searching for the memory based descriptor. e.g.

```
rtt setup 0x24000000 0x40000 “SEGGER RTT”
rtt start
rtt server start 19112 1
```

I/O Communication

The SystemView application Recorder Configuration allows for `UART` (serial) or `IP` (TCP/IP socket) to be selected as sources for obtaining instrumentation events from the target device.

Note

Currently this eCos package does not yet support the direct provision of the `IP` mechanism. See the proxy support for an `IP` Recorder access to a H/W debugger connection.

These I/O channel based mechanisms, whilst not requiring a H/W debugger to be attached, do require further resources on the target system. Support code is required to provide the necessary I/O transport. Two **mutually exclusive** solutions are provided.

`CYGOPT_SYSTEMVIEW_RECORDER_HAL`

> If the target platform/variant supports the feature then this option can be configured. It is a very simple, low-level, blocking-write transport implementation that will not by itself generate Kernel instrumentation events. This makes it usable if high-frequency Kernel instrumentation is enabled (e.g. detailed ISR, thread and synchronisation object activity).
>
> This approach assumes a UART-style interface is used. The platform supplies an initialisation routine and a pair of routines to read and write data to the host SystemView application. This means that this option is only available on targets where the HAL provides the relevant support (e.g. `mimxrt1xxx_evk`, `stm32f746g_disco`, ...).

`CYGOPT_SYSTEMVIEW_RECORDER`

> When configured this enables a higher-level Recorder approach, where a (normally low-priority) helper thread provides the periodic upload of event data to the host SystemView application.
>
> The `UART` configuration requires a dedicated eCos serial I/O (UART port or USB CDC-ACM) connection. There is a greater run-time overhead to supporting this transport option over and above the J-Link H/W debugger approach or `CYGOPT_SYSTEMVIEW_RECORDER_HAL` approach. At least one helper thread is created, and the eCos I/O API is used. If Kernel instrumentation is enabled then the helper code will also be instrumented, which will greatly increase the instrumentation load.
>
> If Kernel instrumentation is not enabled then this solution is sufficient for tracing the standard eCos thread, ISR and operation activity, along with any application generated SystemView events (e.g. markers).

Support for the Recorder feature can be manually enabled in the package configuration if required.

Performance and Analysis

When performing analysis of data captured by the SystemView application, where a run-time I/O mechanism is in use, the user needs to be aware that the act of recording measurement events **will** affect the overall performance of the system. This needs to be taken into account when checking reported performance.

Note

The following timing measurements were made against a MIMXRT1064-EVK platform (Cortex-M7 clocked at 432MHz) with a simple GPIO driven signal on J23#1 measured using a Saleae LogicPro16. The GPIO line was set and cleared as the first and last operations respectively of the `Cyg_RealTimeClock::isr` which is the ISR handler for the `SysTick` event.

For the following table the `SysTick` ISR was chosen as an example, since it is a high-frequency, common event that is likely to be used when analysing scheduling and thread interactions. On most systems it is configured with a 10ms period.

In the following table, the acronyms used in the headers are:

KIM

Build with Kernel Instrumentation to Memory buffer configured, with interrupt events instrumented. This form of instrumentation has a reasonably low overhead, but is limited to the size of buffer that can be set aside on the target.

KIS

Build with Kernel Instrumentation to SystemView events (RTT memory buffer), with interrupt events instrumented. This approach allows for the possibility of continuous recording over long execution runs depending on the RTT access tools used.

SVHR

SystemView events delivered to the host application via an eCos SystemView HAL Recorder (low-level direct I/O) with the UART using a baud rate of `1000000` (1MHz).

SVRTT

SystemView events delivered to the host application via a J-Trace PRO H/W debugger directly accessing RTT memory buffers, via a 4MHz SWD connection.

Table 221.1. Example Instrumentation “cost”

`Cyg_RealTimeClock::isr()`	**`-O2`**	**`-O0`**	**`-O2`** **KIM**	**`-O0`** **KIM**	**`-O0`** **SVRTT**	**`-O0`** **KIS** **SVRTT**	**`-O0`** **KIS** **SVHR**
Handler function execution time captured as the sampled GPIO SET/CLR period	0.7us	~3us	~8us	~18us	~3us	~4us	~202us
SystemView v3.52a reported `SysTick` execution time (includes ISR entry/exit code)	-	-	-	-	~3.8us	~8.9us	~370us

As can be seen there is a **huge** difference in the time taken to execute that single, relatively simple, ISR function depending on the compiler optimisation level, whether or not memory-buffer based Kernel-instrumentation is enabled, or whether the kernel-instrumentation is directed to a SystemView Recorder (the HAL recorder in the example above) where there is the cost of transmitting the bytes over a UART channel (where the configured baud rate of that channel will also affect the timing reported) or is accessed via H/W debugger background memory accesses. The table highlights that any software based instrumentation solution has a hit on the system throughput.

So tuning of the SystemView captures, and the speed of the underlying transport channel need to be considered when performing anaylsis based on how the data was captured for SystemView.

Overflows

Since the normal target operating mode is to perform non-blocking writes of events into the RTT capture buffer, it is possible (and expected in some cases) that the host read side (whether J-Link or communication channel based Recorder) will detect an `*** Overflow ***`.

This can occur if the volume/rate of events being generated is overloading the available I/O channel, in which case the target configuration can be updated to reduce the quantity of events (by disabling tracking of certain events), or by increasing the buffering available to the target-side `CYGPKG_SYSTEMVIEW` support.

It is also possible for overflow to occur as a side-effect of the host SystemView application not actually recording (i.e. “Start Recording” not being triggered). If the `CYGPKG_SYSTEMVIEW` configuration is using `CYGFUN_SYSTEMVIEW_START_ON_INIT` then the eCos code will be recording events from startup, which may be before the host SystemView application can actually connect to the device when using a communication channel Recorder implemented by the eCos run-time; since before SystemView can attach the eCos target has to initialise and provide the drivers/software-stack needed for the I/O access. e.g. Presenting the `UART` Recorder via a target USB CDC-ACM device.

When configured for `CYGOPT_SYSTEMVIEW_RECORDER_HAL` as the event communication channel, the option `CYGOPT_SYSTEMVIEW_RECORDER_HAL_WAIT` is made available. This option is normally disabled, since when enabled it will cause the run-time to disable interrupts and wait in a **busy** poll waiting to receive the host SystemView application “connect and start a recording” commands. This feature can be used to ensure that all captured events are made available to the SystemView application; which may be critical when instrumenting/analysising application initialisation. The caveat is that the target will appear to be “hung” until it receives a valid hello message from the SystemView application.

Chapter 222. Events

When the `CYGBLD_SYSTEMVIEW_ENABLED` option is enabled the eCos run-time will **always** generate SystemView thread events. This allows basic application flow of control between threads to be analysed.

SystemView Events

The `CYGPKG_SYSTEMVIEW` package also provides for generation of records for some other system events.

The following options are enabled by default, but can be disabled to reduce the instrumentation capture load if overflows are encountered.

`CYGIMP_SYSTEMVIEW_TRACE_ISR`

Used to track ISR (Interrupt Service Routine) entry and exit.

`CYGIMP_SYSTEMVIEW_TRACE_TIMERS`

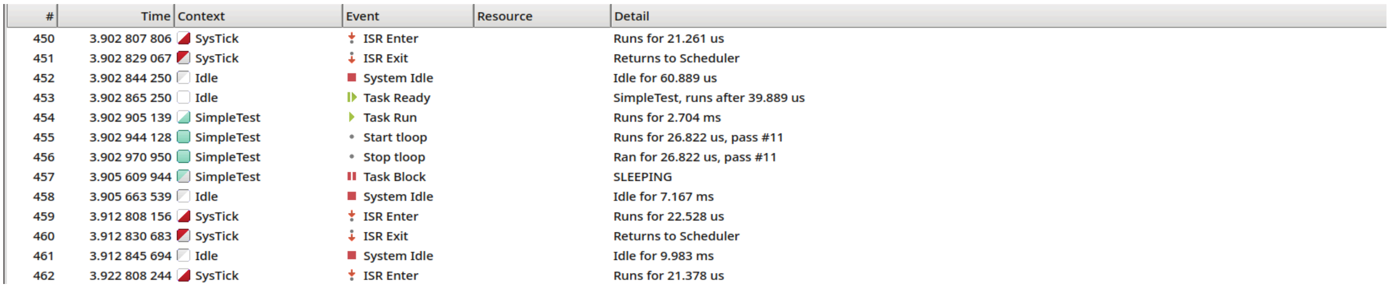
Used to track Kernel alarm function calls. The events wrap the alarm calls within the Kernel `Cyg_Counter::tick()` function.

`CYGIMP_SYSTEMVIEW_TRACE_HEAP`

Used to track `CYGPKG_MEMALLOC` heap creation, allocation and free events. These events can be used to track dynamic memory allocation footprints of an application.

The following image is a screenshot from the SystemView "System" event window highlighting the basic thread and ISR information recorded. It shows a thread named "SimpleTest" being woken, executing for 2.704ms before sleeping and the idle thread executing again.

Figure 222.1. Example from application with `SEGGER_SYSVIEW_Mark()` use



#	Time	Context	Event	Resource	Detail
450	3.902 807 806	SysTick	ISR Enter		Runs for 21.261 us
451	3.902 829 067	SysTick	ISR Exit		Returns to Scheduler
452	3.902 844 250	Idle	System Idle		Idle for 60.889 us
453	3.902 865 250	Idle	Task Ready		SimpleTest, runs after 39.889 us
454	3.902 905 139	SimpleTest	Task Run		Runs for 2.704 ms
455	3.902 944 128	SimpleTest	Start tloop		Runs for 26.822 us, pass #11
456	3.902 970 950	SimpleTest	Stop tloop		Ran for 26.822 us, pass #11
457	3.905 609 944	SimpleTest	Task Block		SLEEPING
458	3.905 663 539	Idle	System Idle		Idle for 7.167 ms
459	3.912 808 156	SysTick	ISR Enter		Runs for 22.528 us
460	3.912 830 683	SysTick	ISR Exit		Returns to Scheduler
461	3.912 845 694	Idle	System Idle		Idle for 9.983 ms
462	3.922 808 244	SysTick	ISR Enter		Runs for 21.378 us

Aside: The above example also highlights the use of an application generated SystemView "marker". The events #`455` and #`456` show the application marked block with the name `tloop` took `26.822us` for the 11th execution of the code covered by the marker.

Kernel Instrumentation

If the eCos Kernel instrumentation (`CYGPKG_KERNEL_INSTRUMENT`) feature is enabled along with the `CYGBLD_SYSTEMVIEW_KERNEL_INSTRUMENT` option then `CYGPKG_SYSTEMVIEW` provides a wrapper header (`CYGBLD_KERNEL_INSTRUMENT_WRAPPER_H`) to replace the Kernel package implementation with code to generate SystemView events.

Note

In this case the normal Kernel instrumentation options `CYGPKG_KERNEL_INSTRUMENT_TIMESTAMPS` and `CYGPKG_KERNEL_INSTRUMENT_BUFFER` are ignored, since the `CYGPKG_SYSTEMVIEW` support is used to record instrumentation. However the `CYGPKG_KERNEL_INSTRUMENT_BUFFER` option should be disabled.

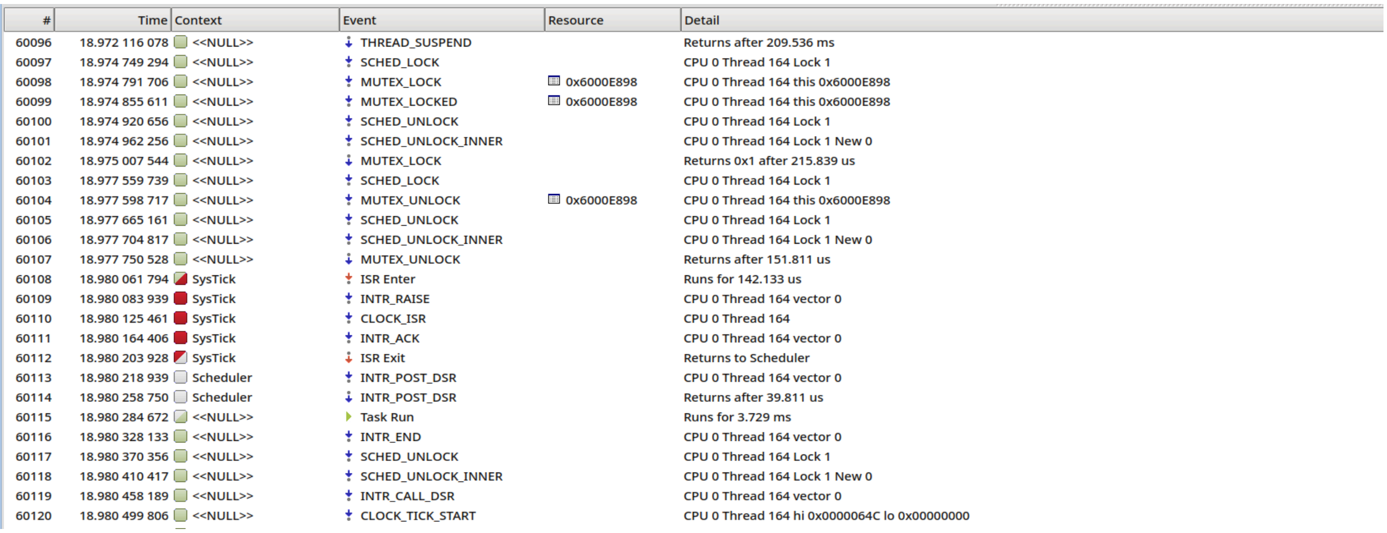
The specific `CYGPKG_KERNEL` instrumentation types can then be manually enabled/disabled as required for the SystemView analysis being performed. e.g. `CYGPKG_KERNEL_SCHED` instrumenting low-level scheduler events can be disabled to save on bandwith; if tracking thread and synchronisation object events is sufficient for the analysis.

Similarly, most packages that make use of Kernel instrumentation do so as eCos configuration conditionals.

These low-level kernel instrumentation events can provide very low-level detail of the inner processing of eCos, and the application use of kernel scheduling objects, but it can generate vast amounts of trace data.

The following image is a screenshot from the SystemView “System” event window which highlights the level of detail available when Kernel instrumentation is enabled. Entry #`60098` is the start of a mutex lock operation, with the lock call completing after `215.839us` as shown by entry #`60102`.

Figure 222.2. Example from application with Kernel instrumentation enabled



#	Time	Context	Event	Resource	Detail
60096	18.972 116 078	<<NULL>>	THREAD_SUSPEND		Returns after 209.536 ms
60097	18.974 749 294	<<NULL>>	SCHED_LOCK		CPU 0 Thread 164 Lock 1
60098	18.974 791 706	<<NULL>>	MUTEX_LOCK	0x6000E898	CPU 0 Thread 164 this 0x6000E898
60099	18.974 855 611	<<NULL>>	MUTEX_LOCKED	0x6000E898	CPU 0 Thread 164 this 0x6000E898
60100	18.974 920 656	<<NULL>>	SCHED_UNLOCK		CPU 0 Thread 164 Lock 1
60101	18.974 962 256	<<NULL>>	SCHED_UNLOCK_INNER		CPU 0 Thread 164 Lock 1 New 0
60102	18.975 007 544	<<NULL>>	MUTEX_LOCK		Returns 0x1 after 215.839 us
60103	18.977 559 739	<<NULL>>	SCHED_LOCK		CPU 0 Thread 164 Lock 1
60104	18.977 598 717	<<NULL>>	MUTEX_UNLOCK	0x6000E898	CPU 0 Thread 164 this 0x6000E898
60105	18.977 665 161	<<NULL>>	SCHED_UNLOCK		CPU 0 Thread 164 Lock 1
60106	18.977 704 817	<<NULL>>	SCHED_UNLOCK_INNER		CPU 0 Thread 164 Lock 1 New 0
60107	18.977 750 528	<<NULL>>	MUTEX_UNLOCK		Returns after 151.811 us
60108	18.980 061 794	SysTick	ISR Enter		Runs for 142.133 us
60109	18.980 083 939	SysTick	INTR_RAISE		CPU 0 Thread 164 vector 0
60110	18.980 125 461	SysTick	CLOCK_ISR		CPU 0 Thread 164
60111	18.980 164 406	SysTick	INTR_ACK		CPU 0 Thread 164 vector 0
60112	18.980 203 928	SysTick	ISR Exit		Returns to Scheduler
60113	18.980 218 939	Scheduler	INTR_POST_DSR		CPU 0 Thread 164 vector 0
60114	18.980 258 750	Scheduler	INTR_POST_DSR		Returns after 39.811 us
60115	18.980 284 672	<<NULL>>	Task Run		Runs for 3.729 ms
60116	18.980 328 133	<<NULL>>	INTR_END		CPU 0 Thread 164 vector 0
60117	18.980 370 356	<<NULL>>	SCHED_UNLOCK		CPU 0 Thread 164 Lock 1
60118	18.980 410 417	<<NULL>>	SCHED_UNLOCK_INNER		CPU 0 Thread 164 Lock 1 New 0
60119	18.980 458 189	<<NULL>>	INTR_CALL_DSR		CPU 0 Thread 164 vector 0
60120	18.980 499 806	<<NULL>>	CLOCK_TICK_START		CPU 0 Thread 164 hi 0x0000064C lo 0x00000000

The above screenshot is from an execution of the `CYGPKG_KERNEL` test `mutex3`, where the created thread has not been given a name, hence the empty thread name being displayed as `<<NULL>>`. This example was captured on the eCos `stm32f746g_disco` target platform (executing at 168MHz and using the UART HAL Recorder with a baud rate of 2MHz).

Warning

When using eCos itself to support the `CYGOPT_SYSTEMVIEW_RECORDER` channel this means that the act of providing support to SystemView will itself add more low-level events, further compounded if the USB CDC-ACM or network stacks are used for the Recorder support, since they will have their own threads and kernel synchronisation objects that will further increase the load. It is more likely that RTT overflows will occur with such a configuration.

It is recommended to use, when available for the selected target, the lower-level `CYGOPT_SYSTEMVIEW_RECORDER_HAL` option when using Kernel instrumentation **if** the use of H/W RTT access is not available.

Infra Trace

In extremis, the Infrastructure (`CYGPKG_INFRA`) package option `CYGDBG_USE_TRACING` can be enabled in conjunction with selecting `CYGDBG_INFRA_DEBUG_TRACE_ASSERT_SYSTEMVIEW` as the trace destination output module.

This feature will provide a trace of the execution of suitably annotated functions. This will mostly be similar in nature to the information that can be obtained from enabling (the lighter weight) Kernel instrumentation, but can be useful for developers wishing to understand the location of the source for the code being executed.

Note

The use of this feature is not normally **recommended**, but can be useful for some investigation/analysis cases.

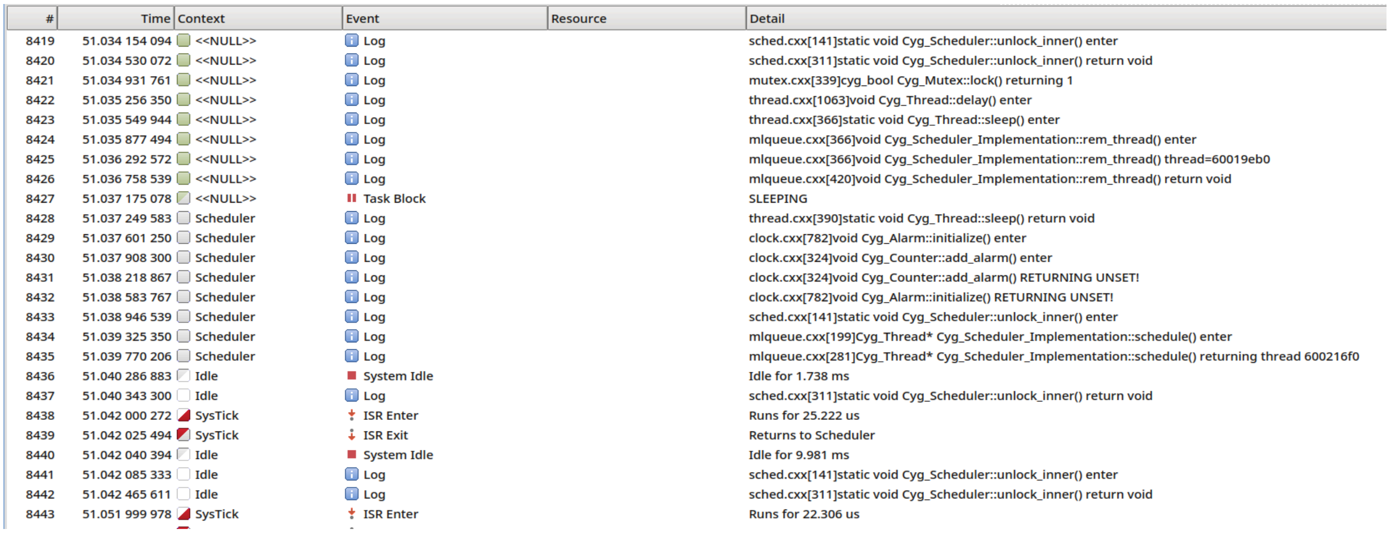
The recording of long strings via the SystemView `Log` mechanism will affect the performance of the application under test, due to the formatting and transfer overhead of long strings.

If detailed eCos operation is required then normally Kernel instrumentation would be sufficient, and whilst it can also generate significant amounts of trace data it will be less than the overhead of the `printf`-style output from the Infrastructure Tracing support.

The following image highlights the style of Tracing output generated.

The `Detail` entries are prefixed with the <*`source-filename`*>[*`source-line#`*]<*`function-name`*> fields, which are then followed by the Tracing specific message text.

Figure 222.3. Example from application using INFRA trace



#	Time	Context	Event	Resource	Detail
8419	51.034 154 094	<<NULL>>	Log		sched.cxx[141]static void Cyg_Scheduler::unlock_inner() enter
8420	51.034 530 072	<<NULL>>	Log		sched.cxx[311]static void Cyg_Scheduler::unlock_inner() return void
8421	51.034 931 761	<<NULL>>	Log		mutex.cxx[339]cyg_bool Cyg_Mutex::lock() returning 1
8422	51.035 256 350	<<NULL>>	Log		thread.cxx[1063]void Cyg_Thread::delay() enter
8423	51.035 549 944	<<NULL>>	Log		thread.cxx[366]static void Cyg_Thread::sleep() enter
8424	51.035 877 494	<<NULL>>	Log		mlqueue.cxx[366]void Cyg_Scheduler_Implementation::rem_thread() enter
8425	51.036 292 572	<<NULL>>	Log		mlqueue.cxx[366]void Cyg_Scheduler_Implementation::rem_thread() thread=60019eb0
8426	51.036 758 539	<<NULL>>	Log		mlqueue.cxx[420]void Cyg_Scheduler_Implementation::rem_thread() return void
8427	51.037 175 078	<<NULL>>	Task Block		SLEEPING
8428	51.037 249 583	Scheduler	Log		thread.cxx[390]static void Cyg_Thread::sleep() return void
8429	51.037 601 250	Scheduler	Log		clock.cxx[782]void Cyg_Alarm::initialize() enter
8430	51.037 908 300	Scheduler	Log		clock.cxx[324]void Cyg_Counter::add_alarm() enter
8431	51.038 218 867	Scheduler	Log		clock.cxx[324]void Cyg_Counter::add_alarm() RETURNING UNSET!
8432	51.038 583 767	Scheduler	Log		clock.cxx[782]void Cyg_Alarm::initialize() RETURNING UNSET!
8433	51.038 946 539	Scheduler	Log		sched.cxx[141]static void Cyg_Scheduler::unlock_inner() enter
8434	51.039 325 350	Scheduler	Log		mlqueue.cxx[199]Cyg_Thread* Cyg_Scheduler_Implementation::schedule() enter
8435	51.039 770 206	Scheduler	Log		mlqueue.cxx[281]Cyg_Thread* Cyg_Scheduler_Implementation::schedule() returning thread 600216f0
8436	51.040 286 883	Idle	System Idle		Idle for 1.738 ms
8437	51.040 343 300	Idle	Log		sched.cxx[311]static void Cyg_Scheduler::unlock_inner() return void
8438	51.042 000 272	SysTick	ISR Enter		Runs for 25.222 us
8439	51.042 025 494	SysTick	ISR Exit		Returns to Scheduler
8440	51.042 040 394	Idle	System Idle		Idle for 9.981 ms
8441	51.042 085 333	Idle	Log		sched.cxx[141]static void Cyg_Scheduler::unlock_inner() enter
8442	51.042 465 611	Idle	Log		sched.cxx[311]static void Cyg_Scheduler::unlock_inner() return void
8443	51.051 999 978	SysTick	ISR Enter		Runs for 22.306 us

The example above was captured on the eCos `stm32f746g_disco` target platform (executing at 168MHz and using the UART HAL Recorder with a baud rate of 2MHz).

Chapter 223. Configuration

CYGBLD_SYSTEMVIEW_ENABLED

These are the main developer configurable options for controlling `CYGPKG_SYSTEMVIEW` operation when `CYGBLD_SYSTEMVIEW_ENABLED` is enabled.

`CYGBLD_SYSTEMVIEW_KERNEL_INSTRUMENT`

This option allows the Kernel instrumentation to be replaced by SystemView event record generation, without having to explicitly add SystemView support to the existing eCos sources. A wrapper function is used to replace the low-level instrumentation call used by packages configured for Kernel instrumentation.

Note

The Kernel option `CYGPKG_KERNEL_INSTRUMENT_BUFFER` **should** be disabled when using `CYGPKG_SYSTEMVIEW` to capture Kernel instrumentation.

It is also recommended to disable the `CYGDBG_KERNEL_INSTRUMENT_FLAGS` feature, so that instrumentation events are generated unconditionally.

`CYGFUN_SYSTEMVIEW_START_ON_INIT`

If enabled then SystemView tracing is started automatically during application initialisation, so that application initialisation and startup events are captured. When this option is enabled **and** H/W debugging is being used, it is normally useful to have your debug session stop at the entry to the `SEGGER_SYSVIEW_Start()` function so that a host SystemView session can be synchronised to capture all events.

`CYGFUN_SYSTEMVIEW_STOP_ON_EXIT`

If enabled this option provides a `CYG_SYSTRACE` macro that is used by the `cyg_test_exit()` function to stop the SystemView event tracing. Since the normal test application exit implementation does not disable interrupts, it avoids continuing to record events when the main application thread has terminated.

`CYGNUM_SYSTEMVIEW_RTT_BUFFER_SIZE`

This specifies the size in bytes of the SystemView Target->Host buffer. NOTE: Depending on the target CPU frequency, and the enabled trace events a large buffer may be required to avoid capture overflow. The performance of the SystemView tracing to avoid overrun also depends on the other RTT loads on the system (e.g. diagnostics Terminal channel). Some tuning may be required in both the events captured and the RTT buffer sizes to ensure all events are captured in real-time.

`CYGIMP_SYSTEMVIEW_USE_STATIC_BUFFER`

If enabled then a statically allocated buffer for the configured maximum packlet size is used. If disabled then a stack buffer is used, with a lock used to serialise packet writing.

Warning

It is important to ensure that the stack allocations of all threads are sufficient for the extra load of the RTT operations for recording events, and also for the transmission code if a “Recorder” is being used.

`CYGNUM_SYSTEMVIEW_MAX_STRING_LEN`

This option defines the maximum string length that can be passed to the SEGGER_SYSVIEW print and description routines. It also defines the maximum packet size, so will affect stack requirements if `CYGIMP_SYSTEMVIEW_USE_STATIC_BUFFER` is **not** enabled.

`CYGIMP_SYSTEMVIEW_TRACE_ISR`

When enabled this option controls generation of SystemView trace instrumentation for ISR entry/exit events. This is distinct from the extra information recorded when the Kernel instrumentation `INTR` events are being generated.

`CYGIMP_SYSTEMVIEW_TRACE_TIMERS`

When enabled this option controls generation of SystemView trace instrumentation for timer callback function events. This is distinct from the extra information recorded when the Kernel intsrumentation `CLOCK` and `ALARM` events are being generated.

`CYGIMP_SYSTEMVIEW_TRACE_HEAP`

When enabled this option controls generation of SystemView trace instrumentation for memory allocation events from the `CYGPKG_MEMALLOC` package.

`CYGDAT_SYSTEMVIEW_APP_NAME`

This is the string used for the `Name` property recorded for a SystemView trace. It can be used to identify specific applications when working with multiple “SystemView Data” captures.

`CYGDAT_SYSTEMVIEW_CORE_NAME`

This is the string used for the `Core` identificaion property recorded for a SystemView trace. It can identify the hardware architecture used for a capture. Normally this option will have a default value set by the platform architecture support.

`CYGDAT_SYSTEMVIEW_DEVICE_NAME`

This is the string used for the `Device` identificaion property recorded for a SystemView trace. It can identify the hardware device used for a capture. If this option is disabled then the `HAL_PLATFORM_BOARD` manifest is used.

CYGOPT_SYSTEMVIEW_RECORDER_HAL

The following options are specific to configurations where the simple, platform supplied, I/O channel support is used for direct communication with the SystemView application `UART` Recorder support. The eCos `HAL` Recorder implementation allows for Kernel instrumentation (and Infra tracing) to be generated without the Recorder itself increasing the instrumentation load.

`CYGNUM_SYSTEMVIEW_RECORDER_HAL_UART_BAUD`

This option defines the communication rate for the platform supplied low-level communication channel. The value configured here should match the value set for the host SystemView application `Target->Recorder Configuration->UART` selection.

`CYGNUM_SYSTEMVIEW_RECORDER_HAL_TXBUF`

This option defines the size of the temporary holding buffer used when transmitting data from the RTT buffer to the remote host SystemView application. This option is a compromise between the data footprint and the amount of data written by SystemView Record events.

This holding buffer can be smaller than the SystemView RTT buffer, and never needs to be larger. The buffer minimises the number of `SEGGER_RTT_ReadUpBufferNoLock()` calls and provides for a tighter TX loop transmitting bytes to the host SystemView application, which may be important since for the HAL Recorder world the performance/latency of the event transmission **will** affect the application performance and the timing of the system.

Also, due to the ordering of the initialisation code it is possible for a number of Kernel instrumentation events to be generated (and recorded in the SystemView RTT buffer) prior to the HAL communication channel being initialised and pending RTT data forwarded to the host SystemView application.

`CYGOPT_SYSTEMVIEW_RECORDER_HAL_WAIT`

When enabled this option will block the system in the HAL Recorder callback until an active SystemView connection is started. This can be used to ensure no event overflow occurs, allowing for a complete application startup to be instrumented. This option should only be enabled when explicitly required by the developer, since the target will be unresponsive until a SystemView connection is established.

CYGOPT_SYSTEMVIEW_RECORDER

As an alternative to the H/W debugger J-Link RTT direct-memory-access method of obtaining continuous events, the remote SystemView application can utilise a network connection or a UART connection to communicate with a target application supplied continuous recorder instance. This option, when enabled, provides the run-time support for such recorders. This uses a helper thread to periodically poll the SystemView buffer and forward data using the appropriate eCos I/O world.

Note

Unlike the alternative `CYGOPT_SYSTEMVIEW_RECORDER_HAL` approach, this Recorder does not rely on target platform specific support, but purely on normal eCos support. However, since the standard eCos thread and device driver interfaces are used, it means that **if** Kernel instrumentation is enabled then the Recorder helper threads and drivers **will** record events as well as the application threads/drivers being analysed.

This means that this helper thread Recorder approach is **not** recommended for Kernel instrumented configuration.

The following options are common to the Recorder support, irrespective of the configured transport options.

`CYGNUM_SYSTEMVIEW_RECORDER_PRI`

The Recorder thread priority is normally low to minimise the instrusiveness of the SystemView support (defaults to 1 higher than “idle”). The priority can be raised if overflows are detected due to the thread not being scheduled often enough, though the first approach (if possible) may just be to increase the SystemView buffer size (`CYGNUM_SYSTEMVIEW_RTT_BUFFER_SIZE`).

Note

It should be noted that since the recorder is a normal eCos thread, the operation of the recorder thread will cause SystemView thread events to be generated, and other events depending on the configuration options selected.

`CYGNUM_SYSTEMVIEW_RECORDER_TX_IDLE`

This option specifies the delay in milliseconds between checks for pending data from the remote SystemView application when the TX side is idle. When TX data is available the channel is also checked for pending RX.

`CYGNUM_SYSTEMVIEW_RECORDER_TX_POLL`

This option specifies the number of milliseconds the Recorder thread will sleep between polled checks for pending data for the remote SystemView application. The actual delay depends on the period of the scheduler tick, which is defined by the target configuration, but is normally ~10ms. So the delay implemented by this option will be rounded to the next higher scheduler tick boundary.

This setting in conjunction with the Recorder thread priority, the idle delay, the size of the RTT buffer and the corresponding TX threshold, can be tuned to avoid loss of instrumentation (RTT overflow) and the timeliness of TX to the SystemView application when matching the bandwidth of the Recorder channel with the frequency of SystemView events being recorded.

`CYGNUM_SYSTEMVIEW_RECORDER_TX_WATERMARK`

This option sets the threshold for the SystemView RTT buffer fill used to trigger a transmission to the application over the communication channel. This option can be tuned to affect the frequency of the Recorder write operations used to flush the RTT event buffer.

As well as the common Recorder options above, the configured transport channels may provide further options for the helper thread based Recorder configuration.

CYGBLD_SYSTEMVIEW_RECORDER_UART

This option enables the UART based continuous recording support to be available to the `CYGOPT_SYSTEMVIEW_RECORDER` world.

Note

This serial support is distinct from, and mutually exclusive to, the separate direct-UART access support provided when using the `CYGOPT_SYSTEMVIEW_RECORDER_HAL` feature.

This presents a serial I/O connection that can be accessed by the remote SystemView application to obtain event records. If the relevant packages are configured this can be a U(S)ART connection or a CDC-ACM (USB) connection. The SystemView event delivery is via the threaded `CYGOPT_SYSTEMVIEW_RECORDER` support.

`CYGDAT_SYSTEMVIEW_RECORDER_UART_DEVICE`

This option specifies the name of the serial peripheral device to use for the SystemView UART Recorder channel. This needs to be configured appropriately for the target, to an available device that will **not** be used by the normal application code. The serial channel is dedicated to SystemView event transport.

`CYGNUM_SYSTEMVIEW_RECORDER_UART_BAUD`

This option specifies the communication rate to be used for the Recorder serial channel. It should match the setting configured for the host SystemView application `UART` Recorder.

Part LXXII. RedBoot User's Guide

Table of Contents

Chapter 224. Getting Started with RedBoot

RedBoot™ is an acronym for "Red Hat Embedded Debug and Bootstrap", and is the standard embedded system debug/bootstrap environment from eCosCentric, replacing the previous generation of debug firmware: CygMon and GDB stubs. It provides a complete bootstrap environment for a range of embedded operating systems, such as embedded Linux® and eCos®, and includes facilities such as network downloading and debugging. It also provides a simple flash file system for boot images.

Note

Red Hat no longer maintain nor support RedBoot and have contributed both RedBoot and eCos to the stewardship of the Free Software Foundation (FSF). eCosCentric are now the sole commercial maintainers of both eCos and RedBoot.

RedBoot provides a wide set of tools for downloading and executing programs on embedded target systems, as well as tools for manipulating the target system's environment. It can be used for both product development (debug support) and for end product deployment (flash and network booting).

Here are some highlights of RedBoot,s capabilities:

- Boot scripting support
- Simple command line interface for RedBoot configuration and management, accessible via serial (terminal) or Ethernet (telnet)
- Integrated GDB stubs for connection to a host-based debugger via serial or ethernet. (Ethernet connectivity is limited to local network only)
- Attribute Configuration - user control of aspects such as system time and date (if applicable), default Flash image to boot from, default failsafe image, static IP address, etc.
- Configurable and extensible, specifically adapted to the target environment
- Network bootstrap support including setup and download, via BOOTP, DHCP and TFTP
- X/Y/Z Modem support for image download via serial
- Power On Self Test

Although RedBoot is derived from eCos, it may be used as a generalized system debug and bootstrap control software for any embedded system and any operating system. For example, with appropriate additions, RedBoot could replace the commonly used BIOS of PC (and certain other) architectures. Red Hat is currently installing RedBoot on all embedded platforms as a standard practice, and RedBoot is now generally included as part of all eCosCentric eCos ports. Users who specifically wish to use RedBoot with the eCos operating system should refer to the *Getting Started with eCos* document, which provides information about the portability and extendability of RedBoot in an eCos environment.

More information about RedBoot on the web

The eCosCentric RedBoot Documentation contains updated documentation for all public and commercial versions of RedBoot, including new features and capabilities.

The RedBoot Net Distribution web site contains downloadable sources and documentation for all publically released targets.

Installing RedBoot

To install the RedBoot package on target hardware, follow the procedures detailed in the documentation accompanying the release. This is normally found in the [eCos and eCosPro Reference Manual] or may also be found in the respective target hardware documentation in eCosCentric's eCos and eCosPro Reference Manual for publically available target hardware.

Although there are other possible configurations, RedBoot is usually run from the target platform‚s flash boot sector or boot ROM, and is designed to run when your system is initially powered on. The method used to install the RedBoot image into non-volatile storage varies from platform to platform. In general, it requires that the image be programmed into flash in situ or programmed into the flash or ROM using a device programmer. In some cases this will be done at manufacturing time; the platform being delivered with RedBoot already in place. In other cases, you will have to program RedBoot into the appropriate device(s) yourself. Installing to flash in situ may require special cabling or interface devices and software provided by the board manufacturer. The details of this installation process for a given platform will be found in Installation and Testing. Once installed, user- specific configuration options may be applied, using the **fconfig** command, providing that persistent data storage in flash is present in the relevant RedBoot version. See the section called “Configuring the RedBoot Environment” for details.

User Interface

RedBoot provides a command line user interface (CLI). At the minimum, this interface is normally available on a serial port on the platform. If more than one serial interface is available, RedBoot is normally configured to try to use any one of the ports for the CLI. Once command input has been received on one port, that port is used exclusively until the board is reset or the channel is manually changed by the user. If the platform has networking capabilities, the RedBoot CLI is also accessible using the `telnet` access protocol. By default, RedBoot runs `telnet` on port TCP/9000, but this is configurable and/or settable by the user.

RedBoot also contains a set of GDB “stubs”, consisting of code which supports the GDB remote protocol. GDB stub mode is automatically invoked when the '$' character appears anywhere on a command line unless escaped using the '\' character. The platform will remain in GDB stub mode until explicitly disconnected (via the GDB protocol). The GDB stub mode is available regardless of the connection method; either serial or network. Note that if a GDB connection is made via the network, then special care must be taken to preserve that connection when running user code. eCos contains special network sharing code to allow for this situation, and can be used as a model if this methodology is required in other OS environments.

RedBoot Editing Commands

RedBoot uses the following line editing commands.

- Delete (0x7F) or Backspace (0x08) erases the character to the left of the cursor.
- ^A or HOME moves the cursor (insertion point) to the beginning of the line.
- ^K erases all characters on the line from the cursor to the end.
- ^E or END positions the cursor to the end of the line.
- ^D or DELETE erases the character under the cursor.
- ^F or RIGHT-ARROW moves the cursor one character to the right.
- ^B or LEFT-ARROW moves the cursor one character to the left.
- ^P or UP-ARROW replaces the current line by a previous line from the history buffer. A small number of lines can be kept as history. Using ^P (and ^N), the current line can be replaced by any one of the previously typed lines.
- ^N or DOWN-ARROW replaces the current line by the next line from the history buffer.

Note

In this description, ^A means the character formed by typing the letter “A” while holding down the control key.

In the case of the **fconfig** command, additional editing commands are possible. As data are entered for this command, the current/previous value will be displayed and the cursor placed at the end of that data. The user may use the editing keys (above) to move around in the data to modify it as appropriate. Additionally, when certain characters are entered at the end of the current value, i.e. entered separately, certain behavior is elicited.

- ^ (caret) switch to editing the previous item in the **fconfig** list. If fconfig edits item A, followed by item B, pressing ^ when changing item B, allows you to change item A. This is similar to the up arrow. Note: ^P and ^N do not have the same meaning while editing **fconfig** data and should not be used.

- . (period) stop editing any further items. This does not change the current item.

- Return leaves the value for this item unchanged. Currently it is not possible to step through the value for the start-up script; it must always be retyped.

RedBoot Command History

RedBoot provides support for listing and repeating previously entered commands. A list of previously entered commands may be obtained by typing **history** at the command line:

```
RedBoot> history
  0 fis list
  1 fconfig -l
  2 load -m ymodem
  3 history
```

The following history expansions may be used to execute commands in the history list:

- !! repeats last command.

- !**n** repeats command **n**.

- !**string** repeats most recent command starting with **string**.

RedBoot Startup Mode

RedBoot can normally be configured to run in a number of startup modes (or just "modes" for short), determining its location of residence and execution:

ROM mode
ROMINT mode
EEPROM mode

In this mode, RedBoot both resides and executes from ROM memory (ROMINT:internal flash, ROM:external flash, EPROM: EEPROM memory). This mode is used when there are limited RAM resources. The flash commands cannot update the region of flash where the RedBoot image resides. In order to update the RedBoot image in flash, it is necessary to run a RAM mode instance of RedBoot. The exact location is defined by the port and provided in the target hardware documentation section of the eCos and eCosPro Reference Manual.

ROMRAM mode

In this mode, RedBoot resides in ROM memory (flash or EPROM), but is copied to RAM memory before it starts executing. The RAM footprint is larger than for ROM mode, but there are two advantages to make up for this: it normally runs faster (relevant only on slower boards) and it is able to update the flash region where the image resides.The exact location is defined by the port and provided in the target hardware documentation section of the eCos and eCosPro Reference Manual.

RAM mode
SRAM mode
JTAG mode

In this mode, RedBoot both resides and executes from RAM or SRAM memory. The memory may be on-chip RAM or SRAM or external RAM or SRAM. The exact location is defined by the port and provided in the target hardware documentation section of the eCos and eCosPro Reference Manual. This is used for updating a primary ROM mode image in situ and sometimes as part of the RedBoot installation on the board when there's already an existing (non-RedBoot) boot monitor available.

You can only use ROM, ROMRAM and EEPROM mode images for booting a board - a RAM mode image cannot run unless loaded by another ROM monitor. There is no need for this startup mode if a RedBoot ROMRAM mode image is the primary boot monitor. When this startup mode is programmed into flash (as a convenience as it's fast to load from flash) it will generally be named as "RedBoot[RAM]" in the FIS directory.

The chosen mode has influence on flash and RAM resource usage (see the section called “RedBoot Resource Usage”) and the procedure of an in situ update of RedBoot in flash (see Chapter 227, *Updating RedBoot*).

The startup mode is controlled by the option CYG_HAL_STARTUP which resides in the platform HAL. Some platforms provide only some of the RAM, ROM, ROMRAM, etc. modes, others provide additional modes.

To see mode of a currently executing RedBoot, issue the **version** command, which prints the RedBoot banner, including the startup mode (here ROM):

```
RedBoot>version

RedBoot(tm) bootstrap and debug environment [ROM]
Non-certified release, version UNKNOWN - built 13:31:57, May 17 2002
```

RedBoot Resource Usage

RedBoot takes up both flash and RAM resources depending on its startup mode and number of enabled features. There are also other resources used by RedBoot, such as timers. Platform-specific resources used by RedBoot are listed in the platform specific parts of this manual.

Both flash and RAM resources used by RedBoot depend to some degree on the features enabled in the RedBoot configuration. It is possible to reduce in particular the RAM resources used by RedBoot by removing features that are not needed. Flash resources can also be reduced, but due to the granularity of the flash (the block sizes), reductions in feature size do not always result in flash resource savings.

Flash Resources

On many platforms, a ROM mode RedBoot image resides in the first flash sectors, working as the board's primary boot monitor. On these platforms, it is also normal to reserve a similar amount of flash for a secondary RAM mode image, which is used when updating the primary ROM mode image.

On other platforms, a ROMRAM mode RedBoot image is used as the primary boot monitor. On these platforms there is not normally reserved space for a RAM mode RedBoot image, since the ROMRAM mode RedBoot is capable of updating the primary boot monitor image.

Most platforms also contain a FIS directory (keeping track of available flash space) and a RedBoot config block (containing RedBoot board configuration data).

To see the amount of reserved flash memory, run the **fis list** command:

```
RedBoot> fis list
Name              FLASH addr  Mem addr    Length      Entry point
RedBoot           0x00000000  0x00000000  0x00020000  0x00000000
```

```
RedBoot[RAM]       0x00020000  0x06020000  0x00020000  0x060213C0
RedBoot config     0x0007F000  0x0007F000  0x00001000  0x00000000
FIS directory      0x00070000  0x00070000  0x0000F000  0x00000000
```

To save flash resources, use a ROMRAM mode RedBoot, or if using a ROM mode RedBoot, avoid reserving space for the RedBoot[RAM] image (this is done by changing the RedBoot configuration) and download the RAM mode RedBoot whenever it is needed. If the RedBoot image takes up a fraction of an extra flash block, it may be possible to reduce the image size enough to free this block by removing some features.

RAM Resources

RedBoot reserves RAM space for its run-time data, and such things as CPU exception/interrupt tables. It normally does so at the bottom of the memory map. It may also reserve space at the top of the memory map for configurable RedBoot features such as the net stack and zlib decompression support.

To see the actual amount of reserved space, issue the **version** command, which prints the RedBoot banner, including the RAM usage:

```
RedBoot> version

RedBoot(tm) bootstrap and debug environment [ROM]
Non-certified release, version UNKNOWN - built 13:31:57, May 17 2002

Platform: FooBar (SH 7615)
Copyright (C) 2000, 2001, 2002, Free Software Foundation, Inc.

RAM: 0x06000000-0x06080000, 0x06012498-0x06061000 available
FLASH: 0x00000000 - 0x00080000, 8 blocks of 0x00010000 bytes each.
```

To simplify operations that temporarily need data in free memory, the limits of free RAM are also available as aliases (aligned to the nearest kilo-byte limit). These are named FREEMEMLO and FREEMEMHI, and can be used in commands like any user defined alias:

```
RedBoot> load -r -b %{FREEMEMLO} file
Raw file loaded 0x06012800-0x06013e53, assumed entry at 0x06012800
```

```
RedBoot> x -b %{FREEMEMHI}
06061000: 86 F5 EB D8 3D 11 51 F2  96 F4 B2 DC 76 76 8F 77  |….=.Q…..vv.w|
06061010: E6 55 DD DB F3 75 5D 15  E0 F3 FC D9 C8 73 1D DA  |.U…u]…...s..|
```

To reduce RedBoot's RAM resource usage, use a ROM mode RedBoot. The RedBoot features that use most RAM are the net stack, the flash support and the gunzip support. These, and other features, can be disabled to reduce the RAM footprint, but obviously at the cost of lost functionality.

Configuring the RedBoot Environment

Once installed, RedBoot will operate fairly generically. However, there are some features that can be configured for a particular installation. These depend primarily on whether flash and/or networking support are available. The remainder of this discussion assumes that support for both of these options is included in RedBoot.

Target Network Configuration

Each node in a networked system needs to have a unique address. Since the network support in RedBoot is based on TCP/IP, this address is an IP (Internet Protocol) address. There are two ways for a system to “know” its IP address. First, it can be stored locally on the platform. This is known as having a static IP address. Second, the system can use the network itself to discover its IP address. This is known as a dynamic IP address. RedBoot supports this dynamic IP address mode by use of either the BOOTP protocol, or a subset of the DHCP protocol. In this case, RedBoot will ask the network (actually some generic server on the network) for the IP address to use.

Note

Currently, RedBoot only supports BOOTP and a very simple form of DHCP which is limited to additional data items, not lease-based address allocation. If you wish to use this subset of DHCP, then ensure you the configuration option `CYGSEM_REDBOOT_NETWORKING_DHCP` is enabled in RedBoot's configuration.

If you are intending to use RedBoot for network debugging, and are also intending to use BOOTP or DHCP in the loaded application, then you should not also use BOOTP/DHCP with RedBoot. Otherwise the DHCP server is likely to give both RedBoot's and the application's network stacks the same IP address which will result in problems.

The choice of IP address type is made via the **fconfig** command. Once a selection is made, it will be stored in flash memory. RedBoot only queries the flash configuration information at reset, so any changes will require restarting the platform.

Here is an example of the RedBoot **fconfig** command, showing network addressing:

```
RedBoot> fconfig -l
Run script at boot: false
Use BOOTP for network configuration: false
Local IP address: 192.168.1.29
Default server IP address: 192.168.1.101
DNS server IP address: 192.168.1.1
GDB connection port: 9000
Network debug at boot time: false
```

In this case, the board has been configured with a static IP address listed as the Local IP address. The default server IP address specifies which network node to communicate with for TFTP service. This address can be overridden directly in the TFTP commands.

The `DNS server IP address` option controls where RedBoot should make DNS lookups. A setting of 0.0.0.0 will disable DNS lookups. The DNS server IP address can also be set at runtime.

If the selection for `Use BOOTP for network configuration` had been `true`, these IP addresses would be determined at boot time, via the BOOTP protocol. The final number which needs to be configured, regardless of IP address selection mode, is the `GDB connection port`. RedBoot allows for incoming commands on either the available serial ports or via the network. This port number is the TCP port that RedBoot will use to accept incoming connections.

These connections can be used for GDB sessions, but they can also be used for generic RedBoot commands. In particular, it is possible to communicate with RedBoot via the telnet protocol. For example, on Linux®:

```
% telnet redboot_board 9000
Connected to redboot_board
Escape character is ’^],.
RedBoot>
```

Host Network Configuration

RedBoot may require three different classes of service from a network host:

- dynamic IP address allocation, using BOOTP
- TFTP service for file downloading
- DNS server for hostname lookups

Depending on the host system, these services may or may not be available or enabled by default.

In particular, on most Linux distributions, none of these services will be configured out of the box. As the method for configuring and enabling these services varies widely between Linux distributions and even between versions of the same distribution, the reader is advised to refer to the system documentation of your chosen Linux distribution and version for more details.

The Windows Desktop operating system does not provide these services. Instead these services are included with Windows Server. Please refer to your system administrator, or Microsoft TechNet, for details on how to configure these services on your Windows Server. Alternatively, if you do not have access to a Windows Server or Linux or do not wish to use either, you can use freely available MaraDNS, currently available from http://www.maradns.org/, to provide a DNS server and the Open TFTP Server, currently available from https://sourceforge.net/projects/tftp-server/, to provide a TFTP service on your Windows Desktop.

BOOTP/DHCP server settings for most Linux distributions

First, ensure that you have the proper package, `dhcp` (not `dhcpd`) installed. The DHCP server provides Dynamic Host Configuration, that is, IP address and other data to hosts on a network. It does this in different ways. Next, there can be a fixed relationship between a certain node and the data, based on that node,s unique Ethernet Station Address (ESA, sometimes called a MAC address). The other possibility is simply to assign addresses that are free. The sample DHCP configuration file shown does both. Refer to the DHCP documentation for more details.

Example 224.1. Sample DHCP configuration file

```
--------------- /etc/dhcpd.conf -----------------------------
default-lease-time 600;
max-lease-time 7200;
option subnet-mask 255.255.255.0;
option broadcast-address 192.168.1.255;
option domain-name-servers 198.41.0.4, 128.9.0.107;
option domain-name “bogus.com”;
allow bootp;
shared-network BOGUS {
  subnet 192.168.1.0 netmask 255.255.255.0
    {
       option routers 192.168.1.101;
       range 192.168.1.1 192.168.1.254;
    }
}
host mbx {
   hardware ethernet 08:00:3E:28:79:B8;
   fixed-address 192.168.1.20;
   filename “/tftpboot/192.168.1.21/zImage”;
   default-lease-time -1;
   server-name “srvr.bugus.com”;
   server-identifier 192.168.1.101;
   option host-name “mbx”;
}
```

Once the DHCP package has been installed and the configuration file set up, type:

```
# service dhcpd start
```

DNS server for most Linux distributions

First, ensure that you have the proper package installed. Then change the configuration (in `/etc/named.conf`) so that the `forwarders` point to the primary nameservers for your machine, normally using the nameservers listed in `/etc/resolv.conf`.

Example 224.2. Sample `/etc/named.conf` for most Linux distributions

```
--------------- /etc/named.conf -----------------------------
// generated by named-bootconf.pl

options {
  directory "/var/named";
  /*
   * If there is a firewall between you and nameservers you want
   * to talk to, you might need to uncomment the query-source
   * directive below.  Previous versions of BIND always asked
   * questions using port 53, but BIND 8.1 uses an unprivileged
   * port by default.
   */
  // query-source address * port 53;


  forward first;
  forwarders {
    212.242.40.3;
    212.242.40.51;
  };
};

//
// a caching only nameserver config
//
// Uncomment the following for Red Hat Linux 7.2 or above:
// controls {
//         inet 127.0.0.1 allow { localhost; } keys { rndckey; };
// };
// include "/etc/rndc.key";
zone "." IN {
  type hint;
  file "named.ca";
};

zone "localhost" IN {
  type master;
  file "localhost.zone";
  allow-update { none; };
};

zone "0.0.127.in-addr.arpa" IN {
  type master;
  file "named.local";
  allow-update { none; };
};
```

Make sure the server is started. Older distributions may use the command:

```
# service named start
```

and is started on the next reboot with the command

```
# chkconfig named on
```

Newer Linux distributions may use the command:

```
# systemctl start named
```

and is started on the next reboot with the command

```
# systemctl enable named
```

Finally, if you are developing on Linux and using your host to provide the DNS server, you may wish to change `/etc/resolv.conf` to use `127.0.0.1` as the nameserver for your local machine. Depending on the Linux distribution and its age, you may have to manually set the nameserver in your Network Configuration.

RedBoot network gateway

RedBoot only provides simple routing. If a remote host is not on the same directly connected IP subnet as that associated with the network interface (even if on the same ’wire,), then you will need to configure a gateway which can be used to route packets to the remote host. First you should ensure RedBoot has been built with support for using a gateway using the configuration option `CYGSEM_REDBOOT_NETWORKING_USE_GATEWAY`. Then you can either:-

- Hard-code a gateway into the configuration with the option `CYGDAT_REDBOOT_DEFAULT_GATEWAY_IP_ADDR`;
- Use RedBoot's persistent flash configuration to set the “Gateway IP address” gateway persistently in flash using the **fconfig** command; or
- Configure RedBoot to use obtain an address from a DHCP server (not just BOOTP), and then configure the DHCP server to provide the gateway IP address.

Verification

Once your network setup has been configured, perform simple verification tests as follows:

- Reboot your system, to enable the setup, and then try to ’ping, the target board from a host.
- Once communication has been established, try to ping a host using the RedBoot ping command - both by IP address and hostname.
- Try using the RedBoot load command to download a file from a host.

Chapter 225. RedBoot Commands and Examples

Introduction

RedBoot provides three basic classes of commands:

- Program loading and execution
- Flash image and configuration management
- Miscellaneous commands

Given the extensible and configurable nature of eCos and RedBoot, there may be extended or enhanced sets of commands available.

The basic format for commands is:

```
RedBoot> COMMAND [-S]... [-s val]... operand
```

Commands may require additional information beyond the basic command name. In most cases this additional information is optional, with suitable default values provided if they are not present.

Format	Description	Example
-S	A boolean switch; the behavior of the command will differ, depending on the presence of the switch. In this example, the **`-f`** switch indicates that a complete initialization of the FIS data should be performed. There may be many such switches available for any given command and any or all of them may be present, in any order.	`RedBoot>` **`fis init -f`**
-s *`val`*	A qualified value; the letter "s" introduces the value, qualifying it's meaning. In the example, **`-b 0x100000`** specifies where the memory dump should begin. There may be many such switches available for any given command and any or all of them may be present, in any order.	`RedBoot>` **`dump -b 0x100000 -l 0x20`**
`operand`	A simple value; some commands require a single parameter for which an additional **`-X`** switch would be redundant. In the example, **`JFFS2`** is the name of a flash image. The image name is always required, thus is no need to qualify it with a switch. Note that any un-qualified operand must always appear at the end of the command.	`RedBoot>` **`fis delete JFFS2`**

The list of available commands, and their syntax, can be obtained by typing **help** at the command line:

```
RedBoot> help
Manage aliases kept in FLASH memory
      alias name [value]
Set/Query the system console baud rate
      baudrate [-b <rate>]
Manage machine caches
```

```
      cache [ON | OFF]
Display/switch console channel
      channel [-1|<channel number>]
Display disk partitions
      disks
Display (hex dump) a range of memory
      dump -b <location> [-l <length>] [-s]
Manage flash images
      fis {cmds}
Manage configuration kept in FLASH memory
      fconfig [-i] [-l] [-n] [-f] [-d] | [-d] nickname [value]
Execute code at a location
      go [-w <timeout>] [-c] [-n] [entry]
Help about help?
      help [<topic>]
Set/change IP addresses
      ip_address [-l <local_ip_address>[/<mask_length>]] [-h <server_address>]
Load a file
      load [-r] [-v] [-d] [-c <channel>] [-h <host>] [-m {TFTP | HTTP | {x|y|z}MODEM | disk}]
      [-b <base_address>] <file_name>
Network connectivity test
      ping [-v] [-n <count>] [-t <timeout>] [-i <IP_addr]
      -h <host>
Reset the system
      reset
Display RedBoot version information
      version
Display (hex dump) a range of memory
      x -b <location> [-l <length>] [-s]
```

Commands can be abbreviated to their shortest unique string. Thus in the list above, **d,du,dum** and dump are all valid for the **dump** command. The **fconfig** command can be abbreviated **fc**, but **f** would be ambiguous with **fis**.

There is one additional, special command. When RedBoot detects '$' or '+' (unless escaped via '\') in a command, it switches to GDB protocol mode. At this point, the eCos GDB stubs take over, allowing connections from a GDB host. The only way to get back to RedBoot from GDB mode is to restart the platform.

Note

Multiple commands may be entered on a single line, separated by the semi-colon “;” character.

The standard RedBoot command set is structured around the bootstrap environment. These commands are designed to be simple to use and remember, while still providing sufficient power and flexibility to be useful. No attempt has been made to render RedBoot as the end-all product. As such, things such as the debug environment are left to other modules, such as GDB stubs, which are typically included in RedBoot.

The command set may be also be extended on a platform basis.

Common Commands

Name

alias — Manipulate command line aliases

Synopsis

`alias { name } [ value ]`

Arguments

Name	Type	Description	Default
name	Name	The name for this alias.	*none*
value	String	Replacement value for the alias.	*none*

Description

The **alias** command is used to maintain simple command line aliases. These aliases are shorthand for longer expressions. When the pattern %{name} appears in a command line, including in a script, the corresponding value will be substituted. Aliases may be nested.

If no value is provided, then the current value of the alias is displayed.

If the system supports non-volatile configuration data via the **fconfig** command (see the section called “Persistent State Flash-based Configuration and Control ”), then the value will be saved and used when the system is reset.

Examples

Set an alias.

```
RedBoot> alias joe "This is Joe"
Update RedBoot non-volatile configuration - continue (y/n)? n
```

Display an alias.

```
RedBoot> alias joe
'joe' = 'This is Joe'
```

Use an alias. Note: the "=" command simply echoes the command to to console.

```
RedBoot> = %{joe}
This is Joe
```

Aliases can be nested.

```
RedBoot> alias frank "Who are you? %{joe}"
Update RedBoot non-volatile configuration - continue (y/n)? n
RedBoot> = %{frank}
Who are you? This is Joe
```

Notice how the value of %{frank} changes when %{joe} is changed since the value of %{joe} is not evaluated until %{frank} is evaluated.

```
RedBoot> alias joe "This is now Josephine"
Update RedBoot non-volatile configuration - continue (y/n)? n
RedBoot> = %{frank}
Who are you? This is now Josephine
```

Name

baudrate — Set the baud rate for the system serial console

Synopsis

`baudrate` [-b *`rate`*]

Arguments

Name	**Type**	**Description**	**Default**
-b *`rate`*	Number	The baud rate to use for the serial console.	*none*

Description

The **baudrate** command sets the baud rate for the system serial console.

If no value is provided, then the current value of the console baud rate is displayed.

If the system supports non-volatile configuration data via the **fconfig** command (see the section called “Persistent State Flash-based Configuration and Control ”), then the value will be saved and used when the system is reset.

Examples

Show the current baud rate.

```
RedBoot> baudrate
Baud rate = 38400
```

Change the console baud rate. In order to make this operation safer, there will be a slight pause after the first message to give you time to change to the new baud rate. If it doesn't work, or a less than affirmative answer is given to the "continue" prompt, then the baud rate will revert to the current value. Only after the baud rate has been firmly established will *RedBoot* give you an opportunity to save the value in persistent storage.

```
RedBoot> baudrate -b 57600
Baud rate will be changed to 57600 - update your settings
Device baud rate changed at this point
Baud rate changed to 57600 - continue (y/n)? y
Update RedBoot non-volatile configuration - continue (y/n)? n
```

Name

cache — Control hardware caches

Synopsis

`cache` [[on] | [off]]

Arguments

Name	**Type**	**Description**	**Default**
on		Turn the caches on	*none*
off		Turn the caches off	*none*

Description

The **cache** command is used to manipulate the caches on the processor.

With no options, this command specifies the state of the system caches.

When an option is given, the caches are turned off or on appropriately.

Examples

Show the current cache state.

```
RedBoot> cache
Data cache: On, Instruction cache: On
```

Disable the caches.

```
RedBoot> cache off
RedBoot> cache
Data cache: Off, Instruction cache: Off
```

Enable the caches.

```
RedBoot> cache on
RedBoot> cache
Data cache: On, Instruction cache: On
```

Name

channel — Select the system console channel

Synopsis

`channel` [[-1] | [*`channel_number`*]]

Arguments

Name	**Type**	**Description**	**Default**
-1		Reset the console channel	*none*
channel_number	Number	Select a channel	*none*

Description

With no arguments, the **channel** command displays the current console channel number.

When passed an argument of 0 upward, this command switches the console channel to that channel number. The mapping between channel numbers and physical channels is platform specific but will typically be something like channel 0 is the first serial port, channel 1 is the second, etc.

When passed an argument of -1, this command reverts RedBoot to responding to whatever channel receives input first, as happens when RedBoot initially starts execution.

Examples

Show the current channel.

```
RedBoot> channel
Current console channel id: 0
```

Change to an invalid channel.

```
RedBoot> channel 99
**Error: bad channel number '99'
```

Revert to the default channel setting (any console mode).

```
RedBoot> channel -1
```

Name

cksum — Compute POSIX checksums

Synopsis

`cksum` {-b *`location`*} {-l *`length`*}

Arguments

Name	**Type**	**Description**	**Default**
-b *`location`*	Memory address	Location in memory for stat of data.	*none*
-l *`length`*	Number	Length of data	*none*

Description

Computes the POSIX checksum on a range of memory (either RAM or FLASH). The values printed (decimal cksum, decimal length, hexadecimal cksum, hexadecimal length) can be compared with the output from the Linux program 'cksum'.

Examples

Checksum a buffer.

```
RedBoot> cksum -b 0x100000 -l 0x100
POSIX cksum = 3286483632 256 (0xc3e3c2b0 0x00000100)
```

Checksum an area of memory after loading a file. Note that the base address and length parameters are provided by the preceding load command.

```
RedBoot> load -r -b %{FREEMEMLO} redboot.bin
Raw file loaded 0x06012800-0x0602f0a8
RedBoot> cksum
Computing cksum for area 0x06012800-0x0602f0a8
POSIX cksum = 2092197813 116904 (0x7cb467b5 0x0001c8a8)
```

Name

disks — List available disk partitions.

Synopsis

```
disks
```

Arguments

None.

Description

The **disks** command is used to list disk partitions recognized by RedBoot.

Examples

Show what disk partitions are available.

```
RedBoot> disks
hda1     Linux Swap
hda2     Linux
00100000: 00 3E 00 06 00 06 00 06  00 00 00 00 00 00 00 00  |.>..............|
00100010: 00 00 00 78 00 70 00 60  00 60 00 60 00 60 00 60  |...x.p.`.`.`.`.`|
```

Name

dump — Display memory.

Synopsis

`dump` {-b *`location`*} [-l *`length`*] [-s] [[-1] | [-2] | [-4]]

Arguments

Name	Type	Description	Default
-b *`location`*	Memory address	Location in memory for start of data.	*none*
-l *`length`*	Number	Length of data	32
-s	Boolean	Format data using Motorola S-records.	
-1		Access one byte (8 bits) at a time. Only the least significant 8 bits of the pattern will be used.	-1
-2		Access two bytes (16 bits) at a time. Only the least significant 16 bits of the pattern will be used.	-1
-4		Access one word (32 bits) at a time.	-1

Description

Display a range of memory on the system console.

The **x** is a synonym for **dump**.

Note that this command could be detrimental if used on memory mapped hardware registers.

The memory is displayed at most sixteen bytes per line, first as the raw hex value, followed by an ASCII interpretation of the data.

Examples

Display a buffer, one byte at a time.

```
RedBoot> mfill -b 0x100000 -l 0x20 -p 0xDEADFACE
RedBoot> x -b 0x100000
00100000: CE FA AD DE CE FA AD DE  CE FA AD DE CE FA AD DE  |................|
00100010: CE FA AD DE CE FA AD DE  CE FA AD DE CE FA AD DE  |................|
```

Display a buffer, one short (16 bit) word at a time. Note in this case that the ASCII interpretation is suppressed.

```
RedBoot> dump -b 0x100000 -2
00100000: FACE DEAD FACE DEAD  FACE DEAD FACE DEAD
00100010: FACE DEAD FACE DEAD  FACE DEAD FACE DEAD
```

Display a buffer, one word (32 bit) word at a time. Note in this case that the ASCII interpretation is suppressed.

```
RedBoot> dump -b 0x100000 -4
00100000: DEADFACE DEADFACE DEADFACE DEADFACE
00100010: DEADFACE DEADFACE DEADFACE DEADFACE
```

Display the same buffer, using Motorola S-record format.

```
RedBoot> dump -b 0x100000 -s
```

```
S31500100000CEFAADDECEFAADDECEFAADDECEFAADDE8E
S31500100010CEFAADDECEFAADDECEFAADDECEFAADDE7E
```

Display a buffer, with visible ASCII strings.

```
RedBoot> d -b 0xfe00b000 -l 0x80
0xFE00B000: 20 25 70 0A 00 00 00 00  41 74 74 65 6D 70 74 20 | %p.....Attempt |
0xFE00B010: 74 6F 20 6C 6F 61 64 20  53 2D 72 65 63 6F 72 64 |to load S-record|
0xFE00B020: 20 64 61 74 61 20 74 6F  20 61 64 64 72 65 73 73 | data to address|
0xFE00B030: 3A 20 25 70 20 5B 6E 6F  74 20 69 6E 20 52 41 4D |: %p [not in RAM|
0xFE00B040: 5D 0A 00 00 2A 2A 2A 20  57 61 72 6E 69 6E 67 21 |]...*** Warning!|
0xFE00B050: 20 43 68 65 63 6B 73 75  6D 20 66 61 69 6C 75 72 | Checksum failur|
0xFE00B060: 65 20 2D 20 41 64 64 72  3A 20 25 6C 78 2C 20 25 |e - Addr: %lx, %|
0xFE00B070: 30 32 6C 58 20 3C 3E 20  25 30 32 6C 58 0A 00 00 |02lX <> %02lX...|
0xFE00B080: 45 6E 74 72 79 20 70 6F  69 6E 74 3A 20 25 70 2C |Entry point: %p,|
```

Name

help — Display help on available commands

Synopsis

`help[ topic ]`

Arguments

Name	Type	Description	Default
topic	String	Which command to provide help for.	All commands

Description

The **help** command displays information about the available RedBoot commands. If a *topic* is given, then the display is restricted to information about that specific command.

If the command has sub-commands, e.g. **fis**, then the topic specific display will print additional information about the available sub-commands. special (ICMP) packets to a specific host. These packets should be automatically returned by that host. The command will indicate how many of these round-trips were successfully completed.

Examples

Show generic help. Note that the contents of this display will depend on the various configuration options for RedBoot when it was built.

```
RedBoot> help
Manage aliases kept in FLASH memory
   alias name [value]
Manage machine caches
   cache [ON | OFF]
Display/switch console channel
   channel [-1|<channel number>]
Compute a 32bit checksum [POSIX algorithm] for a range of memory
   cksum -b <location> -l <length>
Display (hex dump) a range of memory
   dump -b <location> [-l <length>] [-s] [-1|-2|-4]
Manage FLASH images
   fis {cmds}
Manage configuration kept in FLASH memory
   fconfig [-i] [-l] [-n] [-f] [-d] | [-d] nickname [value]
Execute code at a location
   go [-w <timeout>] [entry]
Uncompress GZIP compressed data
   gunzip -s <location> -d <location>
Help about help?
   help [<topic>]
Read I/O location
   iopeek [-b <location>] [-1|2|4]
Write I/O location
   iopoke [-b <location>] [-1|2|4] -v <value>
Set/change IP addresses
   ip_address [-l <local_ip_address>[/<mask_length>]] [-h <server_address>]
Load a file
   load [-r] [-v] [-d] [-h <host>] [-m {TFTP | HTTP | {x|y}MODEM -c <channel_number>}]
        [-f <flash_address>] [-b <base_address>]  <file_name>
Compare two blocks of memory
   mcmp -s <location> -d <location> -l <length> [-1|-2|-4]
```

```
Fill a block of memory with a pattern
   mfill -b <location> -l <length> -p <pattern>
  [-1|-2|-4]
Network connectivity test
   ping [-v] [-n <count>] [-l <length>] [-t <timeout>] [-r <rate>]
        [-i <IP_addr>] -h <IP_addr>
Reset the system
   reset
Display RedBoot version information
   version
Display (hex dump) a range of memory
   x -b <location> [-l <length>] [-s] [-1|-2|-4]
```

Help about a command with sub-commands.

```
RedBoot> help fis
Manage FLASH images
   fis {cmds}
Create an image
  fis create -b <mem_base> -l <image_length> [-s <data_length>]
      [-f <flash_addr>] [-e <entry_point>] [-r <ram_addr>] [-n] <name>
Display an image from FLASH Image System [FIS]
  fis delete name
Erase FLASH contents
  fis erase -f <flash_addr> -l <length>
Display free [available] locations within FLASH Image System [FIS]
  fis free
Initialize FLASH Image System [FIS]
  fis init [-f]
Display contents of FLASH Image System [FIS]
  fis list [-c] [-d]
Load image from FLASH Image System [FIS] into RAM
  fis load [-d] [-b <memory_load_address>] [-c] name
Write raw data directly to FLASH
  fis write -f <flash_addr> -b <mem_base> -l <image_length>
```

Name

iopeek — Read I/O location

Synopsis

`iopeek` [-b *location*] [[-1] | [-2] | [-4]]

Arguments

Name	Type	Description	Default
-b *location*	I/O address	I/O Location.	*none*
-1		Access a one byte (8 bit) I/O location.	-1
-2		Access a two byte (16 bit) I/O location.	-1
-4		Access a one word (32 bit) I/O location.	-1

Description

Reads a value from the I/O address space.

Examples

Examine 8 bit value at I/O location 0x3F8.

```
RedBoot> iopeek -b 0x3f8
0x03f8 = 0x30
```

Examine 32 bit value at I/O location 0x3f8.

```
RedBoot> iopeek -b 0x3f8 -4
0x03f8 = 0x03c10065
```

Name

iopoke — Write I/O location

Synopsis

`iopoke` [-b *`location`*] [[-1] | [-2] | [-4]] [-v *`value`*]

Arguments

Name	Type	Description	Default
-b *`location`*	I/O address	I/O Location.	*none*
-1		Access a one byte (8 bit) I/O location. Only the 8 least significant bits of value will be used	-1
-2		Access a two byte (16 bit) I/O location. Only the 16 least significant bits of value will be used	-1
-4		Access a one word (32 bit) I/O location.	-1

Description

Writes a value to the I/O address space.

Examples

Write 0x0123 to 16 bit I/O location 0x200.

```
RedBoot> iopoke -b 0x200 -v 0x123 -2
```

Name

gunzip — Uncompress GZIP compressed data

Synopsis

`gunzip` {-s *source*} {-d *destination*}

Arguments

Name	**Type**	**Description**	**Default**
-s *location1*	Memory address	Location of GZIP compressed data to uncompress.	Value set by last **load** or **fis load** command.
-d *location2*	Memory address	Destination to write uncompressed data to.	*none*

Description

Uncompress GZIP compressed data.

Examples

Uncompress data at location 0x100000 to 0x200000.

```
RedBoot> gunzip -s 0x100000 -d 0x200000
Decompressed 38804 bytes
```

Name

ip_address — Set IP addresses

Synopsis

`ip_address` [-b] [-l *local_IP_address* [/ *netmask_length*]] [-h *server_IP_address*] [-d *DNS_server_IP_address*]

Arguments

Name	Type	Description	Default
-b	Boolean	Obtain an IP address using BOOTP or DHCP.	don't use BOOTP/ DHCP
-l *local_IP_address*[*/netmask_length*]	Numeric IP or DNS name	The IP address RedBoot should use, optionally with the network mask length.	*none*
-h *server_IP_address*	Numeric IP or DNS name	The IP address of the default server. Use of this address is implied by other commands, such as **load**.	*none*
-d *DNS_server_IP_address*	Numeric IP or DNS name	The IP address of the DNS server.	*none*

Description

The **ip_address** command is used to show and/or change the basic IP addresses used by RedBoot. IP addresses may be given as numeric values, e.g. 192.168.1.67, or as symbolic names such as www.ecoscentric.com if DNS support is enabled.

The `-b` option is used to cause the target to perform a bootp or dhcp negotiation to get an IP address.

The `-l` option is used to set the IP address used by the target device. The network mask length can also be specified

The `-h` option is used to set the default server address, such as is used by the **load** command.

The `-d` option is used to set the default DNS server address which is used for resolving symbolic network addresses. Note that an address of 0.0.0.0 will disable DNS lookups.

Examples

Display the current network settings.

```
RedBoot> ip_address
IP: 192.168.1.31, Default server: 192.168.1.101, DNS server IP: 0.0.0.0, DNS domain name:
```

Change the DNS server address.

```
RedBoot> ip_address -d 192.168.1.101
IP: 192.168.1.31, Default server: 192.168.1.101, DNS server IP: 192.168.1.101, DNS domain name:
```

Change the DNS domain name.

```
RedBoot> ip_address -D example.com
IP: 192.168.1.31, Default server: 192.168.1.101, DNS server IP: 192.168.1.101, DNS domain name: example.com
```

Change the default server address.

```
RedBoot> ip_address -h 192.168.1.104
IP: 192.168.1.31, Default server: 192.168.1.104, DNS server IP: 192.168.1.101, DNS domain name:
```

Set the IP address to something new, with a 255.255.255.0 netmask

```
RedBoot> ip_address -l 192.168.1.32/24
IP: 192.168.1.32, Default server: 192.168.1.104, DNS server IP: 192.168.1.101, DNS domain name:
```

Name

load — Download programs or data to the RedBoot platform

Synopsis

`load` [-v] [-d] [-r] [-m [{[[xmodem] | [ymodem] | [tftp] | [disk] | [file]]}]] [-h *`IP_address`*] [-f *`location`*] [-b *`location`*] [-c *`channel`*] [*`file_name`*]

Arguments

Name	Type	Description	Default
-v	Boolean	Display a small spinner (indicator) while the download is in progress. This is just for feedback, especially during long loads. Note that the option has no effect when using a serial download method since it would interfere with the protocol.	*quiet*
-d	Boolean	Decompress data stream (gzip data)	*non-compressed data*
-r	Boolean	Raw (or binary) data. -b or -f must be used	*formatted (S-records, ELF image, etc)*
-m tftp		Transfer data via the network using TFTP protocol.	TFTP
-m http		Transfer data via the network using HTTP protocol.	TFTP
-m xmodem		Transfer data using *X-modem* protocol.	TFTP
-m ymodem		Transfer data using *Y-modem* protocol.	TFTP
-m disk		Transfer data from a local disk.	TFTP
-m file		Transfer data from a local filesystem such as JFFS2 or FAT.	TFTP
-m mem		Load data from a memory address. Enabled with the configuration option `CYGFUN_REDBOOT_LOAD_FROM_MEM`. Supply the memory address and length to load from by using a “filename” in the format: *`ADDRESS`*,*`LENGTH`* For example **`0x10000400,0x54350`**	TFTP
-h *`IP_address`*	Numeric IP or DNS name	The IP address of the TFTP or HTTP server.	Value set by **ip_address**
-b *`location`*	Number	Address in memory to load the data. Formatted data streams will have an implied load address which this option may override.	*Depends on data format*
-f *`location`*	Number	Address in flash to load the data. Formatted data streams will have an implied load address which this option may override.	*Depends on data format*
-c *`channel`*	Number	Specify which I/O channel to use for download. This option is only supported when using either xmodem or ymodem protocol.	*Depends on data format*
`file_name`	String	The name of the file on the TFTP or HTTP server or the local disk. Details of how this is specified for TFTP are host- specific. For local disk files, the name must be in *disk*: *filename* format. The disk portion must match one of the disk names listed by the **disks** command.	*None*

Description

The **load** command is used to download data into the target system. Data can be loaded via a network connection, using either the TFTP or HTTP protocols, or the console serial connection using the X/Y modem protocol. Files may also be loaded directly from local filesystems on disk. Files to be downloaded may either be executable images in ELF executable program format, Motorola S-record (SREC) format or raw data.

Note

When downloading an ELF image, RedBoot will forcibly terminate the transfer once all the relevant (loadable) ELF sections have been received. This behaviour reduces download time when using the X/Y modem protocol over a slow serial connection. However, the terminal emulator may report that the transfer is incomplete and has been cancelled. Such messages are normal and may be ignored.

Examples

Download a Motorola S-record (or ELF) image, using TFTP, specifying the base memory address.

```
RedBoot> load redboot.ROM -b 0x8c400000
Address offset = 0x0c400000
Entry point: 0x80000000, address range: 0x80000000-0x8000fe80
```

Download a Motorola S-record (or ELF) image, using HTTP, specifying the host [server] address.

```
RedBoot> load /redboot.ROM -m HTTP -h 192.168.1.104
Address offset = 0x0c400000
Entry point: 0x80000000, address range: 0x80000000-0x8000fe80
```

Load an ELF file from /dev/hda1 which should be an EXT2 partition:

```
RedBoot> load -mode disk hda1:hello.elf
Entry point: 0x00020000, address range: 0x00020000-0x0002fd70
```

Load an ELF file from /jffs2/applications which should be a directory in a JFFS2 filesystem:

```
RedBoot> load -mode file /jffs2/applications/hello.elf
Entry point: 0x00020000, address range: 0x00020000-0x0002fd70
```

Name

mcmp — Compare two segments of memory

Synopsis

`mcmp` {-s *`location1`*} {-d *`location1`*} {-l *`length`*} [[-1] | [-2] | [-4]]

Arguments

Name	**Type**	**Description**	**Default**
-s *`location1`*	Memory address	Location for start of data.	*none*
-d *`location2`*	Memory address	Location for start of data.	*none*
-l *`length`*	Number	Length of data	*none*
-1		Access one byte (8 bits) at a time. Only the least significant 8 bits of the pattern will be used.	-4
-2		Access two bytes (16 bits) at a time. Only the least significant 16 bits of the pattern will be used.	-4
-4		Access one word (32 bits) at a time.	-4

Description

Compares the contents of two ranges of memory (RAM, ROM, FLASH, etc).

Examples

Compare two buffers which match (result is *quiet*).

```
RedBoot> mfill -b 0x100000 -l 0x20 -p 0xDEADFACE
RedBoot> mfill -b 0x200000 -l 0x20 -p 0xDEADFACE
RedBoot> mcmp -s 0x100000 -d 0x200000 -l 0x20
```

Compare two buffers which don't match.

Only the first non-matching element is displayed.

```
RedBoot> mcmp -s 0x100000 -d 0x200000 -l 0x30 -2
Buffers don't match - 0x00100020=0x6000, 0x00200020=0x0000
```

Name

mcopy — Copy memory

Synopsis

`mcopy` {-s *`source`*} {-d *`destination`*} {-l *`length`*} [[-1] | [-2] | [-4]]

Arguments

Name	**Type**	**Description**	**Default**
-s *`location1`*	Memory address	Location of data to copy.	*none*
-d *`location2`*	Memory address	Destination for copied data.	*none*
-l *`length`*	Number	Length of data	*none*
-1		Copy one byte (8 bits) at a time.	-4
-2		Copy two bytes (16 bits) at a time.	-4
-4		Copy one word (32 bits) at a time.	-4

Description

Copies memory (RAM, ROM, FLASH, etc) from one area to another.

Examples

Copy 16 bits at a time.

```
RedBoot> mfill -b 0x100000 -l 0x20 -2 -p 0xDEAD
RedBoot> mfill -b 0x200000 -l 0x20 -2 -p 0x0
RedBoot> dump  -b 0x200000 -l 0x20 -2
00200000: 0000 0000 0000 0000  0000 0000 0000 0000
00200010: 0000 0000 0000 0000  0000 0000 0000 0000
RedBoot> mcopy -s 0x100000 -d 0x200000 -2 -l 0x20
RedBoot> dump  -b 0x200000 -l 0x20 -2
00200000: DEAD DEAD DEAD DEAD  DEAD DEAD DEAD DEAD
00200010: DEAD DEAD DEAD DEAD  DEAD DEAD DEAD DEAD
```

Name

mfill — Fill RAM with a specified pattern

Synopsis

`mfill` {-b *`location`*} {-l *`length`*} {-p *`value`*} {-a *`mask`*} [[-1] | [-2] | [-4]]

Arguments

Name	Type	Description	Default
-b *`location`*	Memory address	Location in memory for start of data.	*none*
-l *`length`*	Number	Length of data	*none*
-p *`pattern`*	Number	Data value to fill with	0
-a *`mask`*	Number	Use location address ANDed with mask as pattern, overrides -p	0
-1		Access one byte (8 bits) at a time. Only the least significant 8 bits of the pattern will be used.	-4
-2		Access two bytes (16 bits) at a time. Only the least significant 16 bits of the pattern will be used.	-4
-4		Access one word (32 bits) at a time.	-4

Description

Fills a range of memory with the given pattern.

Examples

Fill a buffer with zeros.

```
RedBoot> x -b 0x100000 -l 0x20
00100000: 00 3E 00 06 00 06 00 06  00 00 00 00 00 00 00 00  |.>..............|
00100010: 00 00 00 78 00 70 00 60  00 60 00 60 00 60 00 60  |...x.p.`.`.`.`.`|
RedBoot> mfill -b 0x100000 -l 0x20
RedBoot> x -b 0x100000 -l 0x20
00100000: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00100010: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
```

Fill a buffer with a pattern.

```
RedBoot> mfill -b 0x100000 -l 0x20 -p 0xDEADFACE
RedBoot> x -b 0x100000 -l 0x20
00100000: CE FA AD DE CE FA AD DE  CE FA AD DE CE FA AD DE  |................|
00100010: CE FA AD DE CE FA AD DE  CE FA AD DE CE FA AD DE  |................|
```

Fill a buffer with masked address.

```
RedBoot> mfill -b 0x01000000 -l 0x20 -a 0x0FFF -2
RedBoot> x -b 0x01000000 -l 0x20 -2
01000000: 0000 0002 0004 0006  0008 000A 000C 000E
01000010: 0010 0012 0014 0016  0018 001A 001C 001E
```

Name

ping — Verify network connectivity

Synopsis

`ping` [-v] [-i *local_IP_address*] [-l *length*] [-n *count*] [-t *timeout*] [-r *rate*] {-h *server_IP_address*}

Arguments

Name	**Type**	**Description**	**Default**
-v	Boolean	Be verbose, displaying information about each packet sent.	*quiet*
-n *local_IP_address*	Number	Controls the number of packets to be sent.	10
-i *local_IP_address*	Numeric IP or DNS name	The IP address RedBoot should use.	Value set by **ip_address**
-h *server_IP_address*	Numeric IP or DNS name	The IP address of the host to contact.	*none*
-l *length*	Number	The length of the ICMP data payload.	64
-r *length*	Number	How fast to deliver packets, i.e. time between successive sends. A value of 0 sends packets as quickly as possible.	1000ms (1 second)
-t *length*	Number	How long to wait for the round-trip to complete, specified in milliseconds.	1000ms (1 second)

Description

The **ping** command checks the connectivity of the local network by sending special (ICMP) packets to a specific host. These packets should be automatically returned by that host. The command will indicate how many of these round-trips were successfully completed.

Examples

Test connectivity to host 192.168.1.101.

```
RedBoot> ping -h 192.168.1.101
Network PING - from 192.168.1.31 to 192.168.1.101
PING - received 10 of 10 expected
```

Test connectivity to host 192.168.1.101, with verbose reporting.

```
RedBoot> ping -h 192.168.1.101 -v -n 4
Network PING - from 192.168.1.31 to 192.168.1.101
 seq: 1, time: 1 (ticks)
 seq: 2, time: 1 (ticks)
 seq: 3, time: 1 (ticks)
 seq: 4, time: 1 (ticks)
PING - received 10 of 10 expected
```

Test connectivity to a non-existent host (192.168.1.109).

```
RedBoot> ping -h 192.168.1.109 -v -n 4
PING: Cannot reach server '192.168.1.109' (192.168.1.109)
```

Name

reset — Reset the device

Synopsis

```
reset
```

Arguments

None

Description

The **reset** command causes the target platform to be reset. Where possible (hardware support permitting), this will be equivalent to a power-on reset condition.

Examples

Reset the platform.

```
RedBoot> reset
... Resetting.+... Waiting for network card: .
Socket Communications, Inc: Low Power Ethernet CF Revision C 5V/3.3V 08/27/98
Ethernet eth0: MAC address 00:c0:1b:00:ba:28
IP: 192.168.1.29, Default server: 192.168.1.101

RedBoot(tm) bootstrap and debug environment [ROM]
Non-certified release, version UNKNOWN - built 10:41:41, May 14 2002

Platform: Compaq iPAQ Pocket PC (StrongARM 1110)
Copyright (C) 2000, 2001, 2002, Free Software Foundation, Inc.

RAM: 0x00000000-0x01fc0000, 0x00014748-0x01f71000 available
FLASH: 0x50000000 - 0x51000000, 64 blocks of 0x00040000 bytes each.
RedBoot>
```

Name

version — Display RedBoot version information

Synopsis

`version`

Arguments

None

Description

The **version** command simply displays version information about RedBoot.

Examples

Display RedBoot's version.

```
RedBoot> version
RedBoot(tm) debug environment - built 09:12:03, Feb 12 2001
Platform: XYZ (PowerPC 860)
Copyright (C) 2000, 2001, Free Software Foundation, Inc.
RAM: 0x00000000-0x00400000
```

Flash Image System (FIS)

If the platform has flash memory, RedBoot can use this for image storage. Executable images, as well as data, can be stored in flash in a simple file store. The **fis** command (fis is short for Flash Image System) is used to manipulate and maintain flash images.

Name

fis init — Initialize Flash Image System (FIS)

Synopsis

```
fis init[ -f ]
```

Arguments

Name	Type	Description	Default
-f		All blocks of flash memory (except for the boot blocks) will be erased as part of the initialization procedure.	

Description

This command is used to initialize the Flash Image System (FIS). It should normally only be executed once, when RedBoot is first installed on the hardware. If the reserved images or their sizes in the FIS change, due to a different configuration of RedBoot being used, it may be necessary to issue the command again though.

Note

Subsequent executions will cause loss of previously stored information in the FIS.

Examples

Initialize the FIS directory.

```
RedBoot> fis init
About to initialize [format] flash image system - continue (y/n)? y
*** Initialize FLASH Image System
    Warning: device contents not erased, some blocks may not be usable
... Erase from 0x00070000-0x00080000: .
... Program from 0x0606f000-0x0607f000 at 0x00070000: .
```

Initialize the FIS directory and all of flash memory, except for first blocks of the flash where the boot monitor resides.

```
RedBoot> fis init -f
About to initialize [format] flash image system - continue (y/n)? y
*** Initialize FLASH Image System
... Erase from 0x00020000-0x00070000: .....
... Erase from 0x00080000-0x00080000:
... Erase from 0x00070000-0x00080000: .
... Program from 0x0606f000-0x0607f000 at 0x00070000: .
```

Name

fis list — List Flash Image System directory

Synopsis

`fis list` [-*c*] [-*d*]

Arguments

Name	Type	Description	Default
-c		Show image checksum instead of memory address (column `Mem addr` is replaced by `Checksum`).	
-d		Show image data length instead of amount of flash occupied by image (column `Length` is replaced by `Datalen`).	

Description

This command lists the images currently available in the FIS. Certain images used by RedBoot have fixed names and have reserved slots in the FIS (these can be seen after using the **fis init** command). Other images can be manipulated by the user.

Note

The images are listed in the order they appear in the FIS directory, not by name or creation time.

Examples

List the FIS directory.

```
RedBoot> fis list
Name              FLASH addr  Mem addr    Length      Entry point
RedBoot           0x00000000  0x00000000  0x00020000  0x00000000
RedBoot config    0x0007F000  0x0007F000  0x00001000  0x00000000
FIS directory     0x00070000  0x00070000  0x0000F000  0x00000000
```

List the FIS directory, with image checksums substituted for memory addresses.

```
RedBoot> fis list -c
Name              FLASH addr  Checksum    Length      Entry point
RedBoot           0x00000000  0x00000000  0x00020000  0x00000000
RedBoot config    0x0007F000  0x00000000  0x00001000  0x00000000
FIS directory     0x00070000  0x00000000  0x0000F000  0x00000000
```

List the FIS directory with image data lengths substituted for flash block reservation lengths.

```
RedBoot> fis list -d
Name              FLASH addr  Mem addr    Datalen     Entry point
RedBoot           0x00000000  0x00000000  0x00000000  0x00000000
RedBoot config    0x0007F000  0x0007F000  0x00000000  0x00000000
FIS directory     0x00070000  0x00070000  0x00000000  0x00000000
```

Name

fis free — Free flash image

Synopsis

```
fis free
```

Description

This command shows which areas of the flash memory are currently not in use. When a block contains non-erased contents it is considered in use. Since it is possible to force an image to be loaded at a particular flash location, this command can be used to check whether that location is in use by any other image.

Note

There is currently no cross-checking between actual flash contents and the FIS directory, which mans that there could be a segment of flash which is not erased that does not correspond to a named image, or vice-versa.

Examples

Show free flash areas.

```
RedBoot> fis free
        0xA0040000 .. 0xA07C0000
        0xA0840000 .. 0xA0FC0000
```

Name

fis create — Create flash image

Synopsis

`fis create` {-b *`data address`*} {-l *`length`*} [-f *`flash address`*] [-e *`entry`*] [-r *`relocation address`*] [-s *`data length`*] [-n] [*`name`*]

Arguments

Name	Type	Description	Default
-b	Number	Address of data to be written to the flash.	Address of last loaded file. If not set in a load operation, it must be specified.
-l	Number	Length of flash area to occupy. If specified, and the named image already exists, the length must match the value in the FIS directory.	Length of area reserved in FIS directory if the image already exists, or the length of the last loaded file. If neither are set, it must be specified.
-f	Number	Address of flash area to occopy.	The address of an area reserved in the FIS directory for extant images. Otherwise the first free block which is large enough will be used.
-e	Number	Entry address for an executable image, used by the **fis load** command.	The entry address of last loaded file.
-r	Number	Address where the image should be relocated to by the **fis load** command. This is only relevant for images that will be loaded with the **fis load** command.	The load address of the last loaded file.
-s	Number	Actual length of data written to image. This is used to control the range over which the checksum is made.	It defaults to the length of the last loaded file.
-n		When set, no image data will be written to the flash. Only the FIS directory will be updated.	
`name`	String	Name of flash image.	

Description

This command creates an image in the FIS directory. The data for the image must exist in RAM memory before the copy. Typically, you would use the RedBoot **load** command to load file into RAM and then the **fis create** command to write it to a flash image.

Examples

Trying to create an extant image, will require the action to be verified.

```
RedBoot> fis create RedBoot -f 0xa0000000 -b 0x8c400000 -l 0x20000
An image named ’RedBoot, exists - continue (y/n)? n
```

Create a new test image, let the command find a suitable place.

```
RedBoot> fis create junk -b 0x8c400000 -l 0x20000
... Erase from 0xa0040000-0xa0060000: .
... Program from 0x8c400000-0x8c420000 at 0xa0040000: .
... Erase from 0xa0fe0000-0xa1000000: .
... Program from 0x8c7d0000-0x8c7f0000 at 0xa0fe0000: .
```

Update the RedBoot[RAM] image.

```
RedBoot> load redboot_RAM.img
Entry point: 0x060213c0, address range: 0x06020000-0x06036cc0
RedBoot> fis create RedBoot[RAM]
No memory address set.
An image named 'RedBoot[RAM]' exists - continue (y/n)? y
* CAUTION * about to program 'RedBoot[RAM]'
            at 0x00020000..0x00036cbf from 0x06020000 - continue (y/n)? y
... Erase from 0x00020000-0x00040000: ..
... Program from 0x06020000-0x06036cc0 at 0x00020000: ..
... Erase from 0x00070000-0x00080000: .
... Program from 0x0606f000-0x0607f000 at 0x00070000: .
```

Name

fis load — Load flash image

Synopsis

`fis load` [-b *`load address`*] [-c] [-d] [*`name`*]

Arguments

Name	Type	Description	Default
-b	Number	Address the image should be loaded to. Executable images normally load at the location to which the file was linked. This option allows the image to be loaded to a specific memory location, possibly overriding any assumed location.	If not specified, the address associated with the image in the FIS directory will be used.
-c		Compute and print the checksum of the image data after it has been loaded into memory.	
-d		Decompress gzipped image while copying it from flash to RAM.	
`name`	String	The name of the file, as shown in the FIS directory.	

Description

This command is used to transfer an image from flash memory to RAM.

Once the image has been loaded, it may be executed using the **go** command.

Examples

Load and run RedBoot[RAM] image.

```
RedBoot> fis load RedBoot[RAM]
RedBoot> go
```

Name

fis delete — Delete flash image

Synopsis

`fis delete` { *name* }

Arguments

Name	Type	Description	Default
name	Number	Name of image that should be deleted.	

Description

This command removes an image from the FIS. The flash memory will be erased as part of the execution of this command, as well as removal of the name from the FIS directory.

Note

Certain images are reserved by RedBoot and cannot be deleted. RedBoot will issue a warning if this is attempted.

Examples

```
RedBoot> fis list
Name              flash addr   Mem addr    Length    Entry point
RedBoot          0xA0000000   0xA0000000  0x020000  0x80000000
RedBoot config   0xA0FC0000   0xA0FC0000  0x020000  0x00000000
FIS directory    0xA0FE0000   0xA0FE0000  0x020000  0x00000000
junk             0xA0040000   0x8C400000  0x020000  0x80000000
RedBoot> fis delete junk
Delete image ’junk, - continue (y/n)? y
... Erase from 0xa0040000-0xa0060000: .
... Erase from 0xa0fe0000-0xa1000000: .
... Program from 0x8c7d0000-0x8c7f0000 at 0xa0fe0000: .
```

Name

fis lock — Lock flash area

Synopsis

`fis lock` {-f *flash_address*} {-l *length*}

Arguments

Name	Type	Description	Default
flash_address	Number	Address of area to be locked.	
length	Number	Length of area to be locked.	

Description

This command is used to write-protect (lock) a portion of flash memory, to prevent accidental overwriting of images. In order to make any modifications to the flash, a matching **fis unlock** command must be issued. This command is optional and will only be provided on hardware which can support write-protection of the flash space.

Note

Depending on the system, attempting to write to write-protected flash may generate errors or warnings, or be benignly quiet.

Examples

Lock an area of the flash

```
RedBoot> fis lock -f 0xa0040000 -l 0x20000
... Lock from 0xa0040000-0xa0060000: .
```

Name

fis unlock — Unlock flash area

Synopsis

`fis unlock` {-f *flash_address*} {-l *length*}

Arguments

Name	Type	Description	Default
flash_address	Number	Address of area to be unlocked.	
length	Number	Length of area to be unlocked.	

Description

This command is used to unlock a portion of flash memory forcibly, allowing it to be updated. It must be issued for regions which have been locked before the FIS can reuse those portions of flash.

Note

Some flash devices power up in locked state and always need to be manually unlocked before they can be written to.

Examples

Unlock an area of the flash

```
RedBoot> fis unlock -f 0xa0040000 -l 0x20000
... Unlock from 0xa0040000-0xa0060000: .
```

Name

fis erase — Erase flash area

Synopsis

`fis erase` {-f *`flash_address`*} {-l *`length`*}

Arguments

Name	**Type**	**Description**	**Default**
`flash_address`	Number	Address of area to be erased.	
`length`	Number	Length of area to be erased.	

Description

This command is used to erase a portion of flash memory forcibly. There is no cross-checking to ensure that the area being erased does not correspond to an existing image.

Examples

Erase an area of the flash

```
RedBoot> fis erase -f 0xa0040000 -l 0x20000
... Erase from 0xa0040000-0xa0060000: .
```

Name

fis read — Read flash area

Synopsis

`fis read` {-b *mem_address*} {-l *length*} {-f *flash_address*}

Arguments

Name	Type	Description	Default
mem_address	Number	Address data should be written to.	
length	Number	Length of data to be read.	
flash_address	Number	Address in flash to read from.	

Description

This command is used to read data from flash to memory. This command is only present if a flash device does not support direct random access (for example an SPI NOR flash).

Examples

Read an area of flash into memory

```
RedBoot> fis read -b 0x0606f000 -l 0x1000 -f 0x00020000
```

Name

fis write — Write flash area

Synopsis

`fis write` {-b *`mem_address`*} {-l *`length`*} {-f *`flash_address`*}

Arguments

Name	**Type**	**Description**	**Default**
`mem_address`	Number	Address of data to be written to flash.	
`length`	Number	Length of data to be writtem.	
`flash_address`	Number	Address of flash to write to.	

Description

This command is used to write data from memory to flash. There is no cross-checking to ensure that the area being written to does not correspond to an existing image.

Examples

Write an area of data to the flash

```
RedBoot> fis write -b 0x0606f000 -l 0x1000 -f 0x00020000
* CAUTION * about to program FLASH
            at 0x00020000..0x0002ffff from 0x0606f000 - continue (y/n)? y
... Erase from 0x00020000-0x00030000: .
... Program from 0x0606f000-0x0607f000 at 0x00020000: .
```

Filesystem Interface

If the platform has access to secondary storage, then RedBoot may be able to access a filesystem stored on this device. RedBoot can access FAT filesystems stored on IDE disks, CompactFlash devices, MMC and SD cards or USB mass storage devices and can use JFFS2 filesystems stored in FLASH memory. The **fs** command is used to manipulate files on filesystems. Applications may be loaded into memory using the *file* mode of the **load** command.

Name

fs info — Print filesystem information

Synopsis

`fs info`

Arguments

The command takes no arguments.

Description

This command prints information about the filesystems that are available. Three lists are produced. The first is a list of the filsystem implementations available in RedBoot; names from this list may be used in the *`-t`* option to the **fs mount** command. The second list describes the block devices and partitions that are available for mounting a filesystem; names from this list may be used in the *`-d`* option to the **fs mount** command. The last list describes the filesystems that are already mounted.

Examples

```
RedBoot> fs info
Filesystems available:
fatfs

Devices available:
/dev/mmcsd0/0
/dev/mmcsd0/1
/dev/mmcsd0/2
/dev/usbms/0/0
/dev/usbms/0/1

Mounted filesystems:
      Mountpoint           Device Filesystem
            /usb   /dev/usbms/0/1 fatfs:sync=write
           /boot    /dev/mmcsd0/1 fatfs:sync=write
RedBoot>
```

Name

fs mount — Mount a filesystem

Synopsis

`fs mount` [-d *`device`*] [-t *`fstype`*] {mountpoint}

Arguments

Name	Type	Description	Default
`device`	String	Device containing filsystem to mount.	undefined
`fstype`	String	Filesystem type.	fatfs:sync=write
`mountpoint`	String	Pathname for filesystem root.	/

Description

This command is used make a filesystem available for access with the filesystem access commands. Three things need to be defined to do this. First, the name of the device on which the filesystem is stored needs to be given to the *`-d`* option. Secondly, the type of filesystem it is needs to be given to the *`-t`* option. Finally, the pathname by which the new filesystem will be accessed needs to be supplied. Following a successful mount, the root of the filesystem will be accessible at the mountpoint.

The *`-t`* option is optional. If not given a default filesystem type is chosen. At present this is "fatfs:sync=write" which chooses a FAT filesystem that writes data back to the device immediately. See the FAT filesystem documentation for more details of the options available.

RedBoot must have been built with the required filesystem support enabled in its eCos configuration in order to be able to access filesystems of the necessary type. On the majority of platforms, no filesystems are included in RedBoot at all due to the extra memory footprint overhead for RedBoot this would otherwise incur.

Examples

The following example mounts a JFFS2 partition identified by the FIS partition name "jffs2test" at location `/flash` :

```
RedBoot> fs info
Filesystems available:
ramfs
jffs2
fatfs

Devices available:
/dev/flash/
/dev/mmcsd0/

Mounted filesystems:
         Device Filesystem Mounted on
    <undefined>      ramfs /
RedBoot> fs mount -d /dev/flash/fis/jffs2test -t jffs2 /flash
RedBoot> fs info
Filesystems available:
ramfs
jffs2
fatfs

Devices available:
/dev/flash/
```

```
/dev/mmcsd0/

Mounted filesystems:
         Device Filesystem Mounted on
    <undefined>      ramfs /
    /dev/flash/fis/jffs2test      jffs2 /flash
RedBoot>
```

Consult the documentation within the generic Flash driver package on Flash I/O devices for further information on configuration and usage of `/dev/flash/` devices for use with JFFS2.

Further examples of mount commands are:

- Mount a JFFS2 partition located at offset 0x40000 in the first flash device, of length 2Mbytes, at location `/jffs2` :

  ```
  RedBoot> fs mount -d /dev/flash/0/0x40000,0x200000 -t jffs2 /jffs2
  ```

- Mount a FAT partition located on the first partition of an SD card at location `/` :

  ```
  RedBoot> fs mount -d /dev/mmcsd0/1 -t fatfs /
  ```

- Mount a ROMfs partition located at address 0x48000000 in flash at location `/romfs` :

  ```
  RedBoot> fs mount -d 0x48000000 -t romfs /romfs
  ```

  Note that ROMfs uses absolute addresses as the device name, and not `/dev/flash/` Flash I/O devices.

Name

fs umount — Unmount filesystem

Synopsis

`fs umount` { *`mountpoint`* }

Arguments

Name	**Type**	**Description**	**Default**
`mountpoint`	String	Mountpoint of filesystem to unmount.	

Description

This command removes a filesystem from being accessible using the filesystem commands. The single argument needs to be the mountpoint that was used when mounting the filesystem. This command will fail if the current directory is currently within the filesystem to be unmounted.

Examples

Unmount a JFF2 partititon:

```
RedBoot> fs info
Filesystems available:
ramfs
jffs2

Devices available:
/dev/flash1

Mounted filesystems:
         Device Filesystem Mounted on
    <undefined>      ramfs /
    /dev/flash1      jffs2 /flash
RedBoot> fs umount /flash
RedBoot> fs info
Filesystems available:
ramfs
jffs2

Devices available:
/dev/flash1

Mounted filesystems:
         Device Filesystem Mounted on
    <undefined>      ramfs /
RedBoot>
```

Name

fs cd — Change filesystem directory

Synopsis

`fs cd [ directory ]`

Arguments

Name	Type	Description	Default
directory	String	Pathname to directory to change to.	Root directory

Description

This command changes the current filesystem directory. Subsequent filesystem commands will be executed in the new directory. If no argument is given, then the current directory is set back to the root of the filesystem name space.

Examples

Change current directory:

```
RedBoot> fs ls
d---------       128 .
d---------       128 ..
d---------        96 tests
----------      4096 image
RedBoot> fs cd tests
RedBoot> fs ls
d---------        96 .
d---------       128 ..
----------     16384 test1
RedBoot>
```

Name

fs mkdir — Create filesystem directory

Synopsis

`fs mkdir { directory }`

Arguments

Name	**Type**	**Description**	**Default**
directory	String	Pathname to directory to delete.	

Description

This command creates (makes) a directory in the filesystem.

Examples

```
RedBoot> fs ls
d---------       128 .
d---------       128 ..
----------      4096 image
RedBoot> fs mkdir tests
RedBoot> fs ls
d---------       128 .
d---------       128 ..
d---------        64 tests
----------      4096 image
RedBoot>
```

Name

fs rmdir — Delete filesystem directory

Synopsis

`fs rmdir` { *`directory`* }

Arguments

Name	Type	Description	Default
`directory`	String	Pathname to directory to delete.	

Description

This command deletes a directory from the filesystem. If the directory contains files or other directories then this command will fail.

Examples

Delete directory:

```
RedBoot> fs ls
d---------       128 .
d---------       128 ..
d---------        96 tests
----------      4096 image
RedBoot> fs rmdir tests
RedBoot> fs ls
d---------       128 .
d---------       128 ..
----------      4096 image
RedBoot>
```

Name

fs rm — Delete file

Synopsis

`fs rm { file }`

Arguments

Name	Type	Description	Default
file	String	Pathname of file to delete.	

Description

This command deletes a file from the filesystem.

Examples

Change current directory:

```
RedBoot> fs ls tests
d---------        96 .
d---------       128 ..
----------     16384 test1
RedBoot> fs rm tests/test1
RedBoot> fs ls tests
d---------        96 .
d---------       128 ..
RedBoot>
```

Name

fs mv — Move file

Synopsis

`fs mv` { *`source`* } { *`dest`* }

Arguments

Name	Type	Description	Default
source	String	Pathname of file to move.	
dest	String	Pathname to new file location.	

Description

This command moves a file within a filesystem. This command will fail if the destination file already exists, or is in a different filesystem.

Examples

Rename a file:

```
RedBoot> fs ls tests
d---------        96 .
d---------       128 ..
----------     12288 test1
RedBoot> fs mv tests/test1 tests/test2
RedBoot> fs ls tests
d---------       128 .
d---------       128 ..
----------     12288 test2
RedBoot>
```

Name

fs cp — Copy file

Synopsis

`fs cp` { *`source`* } { *`dest`* }

Arguments

Name	Type	Description	Default
`source`	String	Pathname of file to copy.	
`dest`	String	Pathname to new file location.	

Description

This command copies a file from one location to another. Source and destination need not be on the same file system. If the destination file already exists it will be overwritten with the new file contents. This command will fail if the source file does not exist.

Examples

Copy a file:

```
RedBoot> fs ls /usb
drwxrwxrwx         0 System Volume Information
drwxrwxrwx         0 TEST
RedBoot> fs cp /boot/LICENCE.broadcom /usb/LICENCE.broadcom
RedBoot> fs ls /usb
drwxrwxrwx         0 System Volume Information
drwxrwxrwx         0 TEST
-rwxrwxrwx      1494 LICENCE.broadcom
RedBoot>
```

Name

fs cat — Display file

Synopsis

`fs cat` { *`file`* }

Arguments

Name	Type	Description	Default
file	String	Pathname of file to display.	

Description

This command displays the contents of the given file on the console. It is useful for examining configuration, script or other text files. The file is sent to the console with no additional interptetation, so applying this to a binary file may result in chaos on screen. The command will fail if the file does not exist.

Examples

Copy a file:

```
RedBoot> fs cat /boot/redboot.txt
# RedBoot init file
#--------------------------------------------------------------------------------
# Network setup

# Optional command to set a specific IP address
#ip_address -l 10.44.44.44/8

#--------------------------------------------------------------------------------
# JTAG setup

# Default ALT4 set
jtag 22 23 24 25 26 27

# ALT5 set, with TRST on pin 22 ALT4
#jtag 4 5 6 12 13 22

# Alternate mixed set used by some earlier JTAG implementations
# jtag 4 22 24 25 27

# All JTAG pins, useful for testing changes to pin settings.
#jtag 4 5 6 12 13 22 23 24 25 26 27

#--------------------------------------------------------------------------------
# EOF
RedBoot>
```

Name

fs ls — List filesystem directory

Synopsis

fs ls[*directory*]

Arguments

Name	**Type**	**Description**	**Default**
directory	String	Pathname to directory to list.	Current directory

Description

This command prints a list of the contents of the named directory. Each line of the listing starts with a set of UNIX-like access flags, the first character of this will be a ”d“ if this entry is a directory. Following this is the size of the file in bytes and the last item is its name.

Examples

List the current directory:

```
RedBoot> fs ls
d---------       128 .
d---------       128 ..
----------      4096 image
d---------        96 tests
RedBoot>
```

List a subdirectory:

```
RedBoot> fs ls tests
d---------        96 .
d---------       128 ..
----------     16384 test1
RedBoot>
```

Name

fs write — Write to filesystem

Synopsis

`fs write` [-b *mem_address*] [-l *length*] {name}

Arguments

Name	Type	Description	Default
mem_address	Number	Address of data to be written to flash.	Address of last loaded file. If not set by a load operation it must be specified.
length	Number	Length of data to be written.	Length of last loaded file.
name	String	Name of file to create.	

Description

This command is used to write data from memory to a file. If the file does not exist it will be created. If it does exist, then it will be overwritten with the new contents.

Examples

Write an area of data to a file

```
RedBoot> fs write -b 0x0606f000 -l 0x1000 image
RedBoot> fs ls
d---------       128 .
d---------       128 ..
----------      4096 image
d---------        96 tests
RedBoot>
```

Persistent State Flash-based Configuration and Control

RedBoot provides flash management support for storage in the flash memory of multiple executable images and of non-volatile information such as IP addresses and other network information.

RedBoot on platforms that support flash based configuration information will report the following message the first time that RedBoot is booted on the target:

```
flash configuration checksum error or invalid key
```

This error can be ignored if no flash based configuration is desired, or can be silenced by running the **fconfig** command as described below. At this point you may also wish to run the **fis init** command. See other fis commands in the section called “Flash Image System (FIS)”.

Certain control and configuration information used by RedBoot can be stored in flash.

The details of what information is maintained in flash differ, based on the platform and the configuration. However, the basic operation used to maintain this information is the same. Using the **fconfig -l** command, the information may be displayed and/or changed.

If the optional flag `-i` is specified, then the configuration database will be reset to its default state. This is also needed the first time RedBoot is installed on the target, or when updating to a newer RedBoot with different configuration keys.

If the optional flag `-l` is specified, the configuration data is simply listed. Otherwise, each configuration parameter will be displayed and you are given a chance to change it. The entire value must be typed - typing just carriage return will leave a value unchanged. Boolean values may be entered using the first letter (`t` for true, `f` for false). At any time the editing process may be stopped simply by entering a period (.) on the line. Entering the caret (^) moves the editing back to the previous item. See “RedBoot Editing Commands”, the section called “RedBoot Editing Commands”.

If any changes are made in the configuration, then the updated data will be written back to flash after getting acknowledgment from the user.

If the optional flag `-n` is specified (with or without `-l`) then “nicknames” of the entries are used. These are shorter and less descriptive than “full” names. The full name may also be displayed by adding the `-f` flag.

The reason for telling you nicknames is that a quick way to set a single entry is provided, using the format

```
  RedBoot> fconfig nickname value
```

If no value is supplied, the command will list and prompt for only that entry. If a value is supplied, then the entry will be set to that value. You will be prompted whether to write the new information into flash if any change was made. For example

```
  RedBoot> fconfig -l -n
  boot_script: false
  bootp: false
  bootp_my_ip: 10.16.19.176
  bootp_server_ip: 10.16.19.66
  dns_ip: 10.16.19.1
  gdb_port: 9000
  net_debug: false
  RedBoot> fconfig bootp_my_ip 10.16.19.177
  bootp_my_ip: 10.16.19.176 Setting to 10.16.19.177
  Update RedBoot non-volatile configuration - continue (y/n)? y
  ... Unlock from 0x507c0000-0x507e0000: .
  ... Erase from 0x507c0000-0x507e0000: .
  ... Program from 0x0000a8d0-0x0000acd0 at 0x507c0000: .
  ... Lock from 0x507c0000-0x507e0000: .
  RedBoot>
```

Additionally, nicknames can be used like aliases via the format %{nickname}. This allows the values stored by **`fconfig`** to be used directly by scripts and commands.

Depending on how your terminal program is connected and its capabilities, you might find that you are unable to use line-editing to delete the ’old, value when using the default behaviour of **fconfig** ***`nickname`*** or just plain **fconfig**, as shown in this example:

```
RedBoot> fco bootp
bootp: false_
```

The user deletes the word “false;” and enters “true” so the display looks like this:

```
RedBoot> fco bootp
bootp: true
Update RedBoot non-volatile configuration - continue (y/n)? y
... Unlock from ...
RedBoot> _
```

To edit when you cannot backspace, use the optional flag `-d` (for “dumb terminal”) to provide a simpler interface thus:

```
RedBoot> fco -d bootp
bootp: false ? _
```

and you enter the value in the obvious manner thus:

```
RedBoot> fco -d bootp
bootp: false ? true
Update RedBoot non-volatile configuration - continue (y/n)? y
... Unlock from ...
RedBoot> _
```

One item which is always present in the configuration data is the ability to execute a script at boot time. A sequence of RedBoot commands can be entered which will be executed when the system starts up. Optionally, a time-out period can be provided which allows the user to abort the startup script and proceed with normal command processing from the console.

```
RedBoot> fconfig -l
Run script at boot: false
Use BOOTP for network configuration: false
Local IP address: 192.168.1.29
Default server IP address: 192.168.1.101
DNS domain name: example.com
DNS server IP address: 192.168.1.1
GDB connection port: 9000
Network debug at boot time: false
```

The following example sets a boot script and then shows it running.

```
RedBoot> fconfig
Run script at boot: false t
      Boot script:
Enter script, terminate with empty line
>> fi li
    Boot script timeout: 0 10
Use BOOTP for network configuration: false .
Update RedBoot non-volatile configuration - continue (y/n)? y
... Erase from 0xa0fc0000-0xa0fe0000: .
... Program from 0x8c021f60-0x8c022360 at 0xa0fc0000: .
RedBoot>
RedBoot(tm) debug environment - built 08:22:24, Aug 23 2000
Copyright (C) 2000, Free Software Foundation, Inc.


RAM: 0x8c000000-0x8c800000
flash: 0xa0000000 - 0xa1000000, 128 blocks of 0x00020000 bytes ea.
Socket Communications, Inc: Low Power Ethernet CF Revision C \
5V/3.3V 08/27/98 IP: 192.168.1.29, Default server: 192.168.1.101 \
== Executing boot script in 10 seconds - enter ^C to abort
RedBoot> fi li
Name              flash addr   Mem addr    Length    Entry point
RedBoot          0xA0000000   0xA0000000  0x020000  0x80000000
RedBoot config   0xA0FC0000   0xA0FC0000  0x020000  0x00000000
FIS directory    0xA0FE0000   0xA0FE0000  0x020000  0x00000000
RedBoot>
```

Notes

- The bold characters above indicate where something was entered on the console. As you can see, the **fi li** command at the end came from the script, not the console. Once the script is executed, command processing reverts to the console.
- RedBoot supports the notion of a boot script timeout, i.e. a period of time that RedBoot waits before executing the boot time script. This period is primarily to allow the possibility of canceling the script. Since a timeout value of zero (0) seconds would never allow the script to be aborted or canceled, this value is not allowed. If the timeout value is zero, then RedBoot will abort the script execution immediately.

On many targets, RedBoot may be configured to run from ROM or it may be configured to run from RAM. Other configurations are also possible. All RedBoot configurations will execute the boot script, but in certain cases it may be desirable to limit the execution of certain script commands to one RedBoot configuration or the other. This can be accomplished by prepending `{<startup type>}` to the commands which should be executed only by the RedBoot configured for the specified startup type. The following boot script illustrates this concept by having the ROM based RedBoot load and run the RAM based RedBoot. The RAM based RedBoot will then list flash images.

```
RedBoot> fco
Run script at boot: false t
Boot script:
Enter script, terminate with empty line
>> {ROM}fis load RedBoot[RAM]
>> {ROM}go
>> {RAM}fis li
>>
Boot script timeout (1000ms resolution): 2
Use BOOTP for network configuration: false
 ...
Update RedBoot non-volatile configuration - continue (y/n)? y
... Unlock from 0x007c0000-0x007e0000: .
... Erase from 0x007c0000-0x007e0000: .
... Program from 0xa0015030-0xa0016030 at 0x007df000: .
... Lock from 0x007c0000-0x007e0000: .
RedBoot> reset
... Resetting.
+Ethernet eth0: MAC address 00:80:4d:46:01:05
IP: 192.168.1.153, Default server: 192.168.1.10

RedBoot(tm) bootstrap and debug environment [ROM]
eCosCentric certified release, version R1.xx - built 12:27:26, Sep 11 2003

Platform: IQ80310 (XScale)
Copyright (C) 2000, 2001, Free Software Foundation, Inc.

RAM: 0xa0000000-0xa2000000, 0xa001b088-0xa1fdf000 available
FLASH: 0x00000000 - 0x00800000, 64 blocks of 0x00020000 bytes each.
== Executing boot script in 2.000 seconds - enter ^C to abort
RedBoot> fis load RedBoot[RAM]
RedBoot> go
+Ethernet eth0: MAC address 00:80:4d:46:01:05
IP: 192.168.1.153, Default server: 192.168.1.10

RedBoot(tm) bootstrap and debug environment [RAM]
eCosCentric certified release, version R1.xx - built 18:03:27, Sep 11 2003
```

```
Platform: IQ80310 (XScale)
Copyright (C) 2000, 2001, Free Software Foundation, Inc.

RAM: 0xa0000000-0xa2000000, 0xa0057fe8-0xa1fdf000 available
FLASH: 0x00000000 - 0x00800000, 64 blocks of 0x00020000 bytes each.
== Executing boot script in 2.000 seconds - enter ^C to abort
RedBoot> fis li
Name              FLASH addr  Mem addr    Length      Entry point
RedBoot           0x00000000  0x00000000  0x00040000  0x00002000
RedBoot config    0x007DF000  0x007DF000  0x00001000  0x00000000
FIS directory     0x007E0000  0x007E0000  0x00020000  0x00000000
RedBoot>
```

Persistent State in a NAND-based environment

On some boards it is necessary to use NAND flash to store persistent state and configuration. This is normally provided by the config store.

For simplicity, a *bridge driver* has been developed which allows the config store to be used by RedBoot as if it were NOR flash. (The reason for this is that the RedBoot fconfig system has a much lower overhead than the NAND storage system, in order to operate in environments with very limited resources. NAND-based boards have much more flash and RAM available, so the overhead of the extra layers is not an issue.)

To RedBoot, the config store behaves identically to regular (NOR) flash storage. However its use has a number of implications:

- If the NAND is repartitioned or erased, all persistent configuration is likely to be lost.
- The configuration is susceptible to the same wearout and read disturb effects that affect all NAND parts over time. (This is mitigated by allowing multiple NAND blocks for the config store partition, which are used in rotation.)

Even though the bridge driver may be in use, it is still possible to use the config store directly for other options.

Note

Some platform HALs use this mechanism to set the size of the config store partition in such a way that does not depend on RedBoot. Refer to the documentation for those HAL ports for details.

Manipulating persistent state stored on NAND

To modify configuration items managed by RedBoot, use the **`fconfig`** command in the usual way. See the section called “Persistent State Flash-based Configuration and Control ”.

To modify items in the config store which are not managed by RedBoot, use the **`nconfig`** command. See the section called “NAND configuration commands ”. You may also wish to manipulate the NAND array with the **`nand`** command family; see the section called “NAND manipulation commands ”.

Executing Programs from RedBoot

Once an image has been loaded into memory, either via the **load** command or the **fis load** command, execution may be transfered to that image.

Note

The image is assumed to be a stand-alone entity, as RedBoot gives the entire platform over to it. Typical examples would be an eCos application or a Linux kernel.

Name

go — Execute a program

Synopsis

`go` [-w *timeout*] [-c] [-n] [*start_address*]

Arguments

Name	Type	Description	Default
-w *timeout*	Number	How long to wait before starting execution.	0
-c	Boolean	Go with caches enabled.	caches off
-n	Boolean	Go with network interface stopped.	network enabled
start_address	Number	Address in memory to begin execution.	Value set by last **load** or **fis load** command.

Description

The **go** command causes RedBoot to give control of the target platform to another program. This program must execute stand alone, e.g. an eCos application or a Linux kernel.

If the -w option is used, RedBoot will print a message and then wait for a period of time before starting the execution. This is most useful in a script, giving the user a chance to abort executing a program and move on in the script.

Examples

Execute a program - *no explicit output from RedBoot*.

```
RedBoot> go 0x40040
```

Execute a program with a timeout.

```
RedBoot> go -w 10
About to start execution at 0x00000000 - abort with ^C within 10 seconds
^C
RedBoot>
```

Note that the starting address was implied (0x00000000 in this example). The user is prompted that execution will commence in 10 seconds. At anytime within that 10 seconds the user may type Ctrl+C on the console and RedBoot will abort execution and return for the next command, either from a script or the console.

Name

exec — Execute a Linux kernel

Synopsis

`exec` [-w *`timeout`*] [-r *`ramdisk_address`*] [-s *`ramdisk_length`*] [-b *`load_address`* {-l *`load_length`*}] [-c *`kernel_command_line`*] [*`entry_point`*]

Arguments

Name	Type	Description	Default
-w *`timeout`*	Number	Time to wait before starting execution.	0
-r *`ramdisk_address`*	Number	Address in memory of "initrd"-style ramdisk - passed to Linux kernel.	*None*
-s *`ramdisk_length`*	Number	Length of ramdisk image - passed to Linux kernel.	*None*
-b *`load_address`*	Number	Address in memory of the Linux kernel image.	Value set by **load** or **fis load**
-l *`load_length`*	Number	Length of Linux kernel image.	*none*
-c *`kernel_command_line`*	String	Command line to pass to the Linux kernel.	*None*
-x		Boot kernel with endianess opposite of RedBoot endianess.	Boot kernel with same endianess as RedBoot
`entry_address`	Number	Starting address for Linux kernel execution	Implied by architecture

Description

The **exec** command is used to execute a non-eCos application, typically a Linux kernel. Additional information may be passed to the kernel at startup time. This command is quite special (and unique from the **go** command) in that the program being executed may expect certain environmental setups, for example that the MMU is turned off, etc.

The Linux kernel expects to have been loaded to a particular memory location which is architecture dependent(0xC0008000 in the case of the SA1110). Since this memory is used by RedBoot internally, it is not possible to load the kernel to that location directly. Thus the requirement for the "-b" option which tells the command where the kernel has been loaded. When the **exec** command runs, the image will be relocated to the appropriate location before being started. The "-r" and "-s" options are used to pass information to the kernel about where a statically loaded ramdisk (initrd) is located.

The "-c" option can be used to pass textual "command line" information to the kernel. If the command line data contains any punctuation (spaces, etc), then it must be quoted using the double-quote character '"'. If the quote character is required, it should be written as '\"'.

The "-x" option is optionally available on some bi-endian platforms. It is used to boot a kernel built with an endianess opposite of RedBoot.

Examples

Execute a Linux kernel, passing a command line, which needs relocation. The result from RedBoot is normally quiet, with the target platform being passed over to Linux immediately.

```
RedBoot> exec -b 0x100000 -l 0x80000 -c "noinitrd root=/dev/mtdblock3 console=ttySA0"
```

Execute a Linux kernel, default entry address and no relocation required, with a timeout. The *emphasized lines* are output from the loaded kernel.

```
RedBoot> exec -c "console=ttyS0,38400 ip=dhcp nfsroot=/export/elfs-sh" -w 5
Now booting linux kernel:
Base address 0x8c001000 Entry 0x8c210000
Cmdline : console=ttyS0,38400 ip=dhcp nfsroot=/export/elfs-sh
About to start execution at 0x8x210000 - abort with ^C within 5 seconds
Linux version 2.4.10-pre6 (...) (gcc version 3.1-stdsh-010931) #3 Thu Sep 27 11:04:23 BST 2001
```

NAND configuration commands

Note

This section does not cover configuration items managed by RedBoot; for those, use the **`fconfig`** command instead (the section called “Persistent State Flash-based Configuration and Control ”).

The *config store* is the backing store for this persistent information. Data is stored as *(key, value)* pairs; values are *typed*. This section refers only to the RedBoot commands for manipulating the config store.

Note

The config key **`ecos.fakeflash.config`** contains the RedBoot configuration information as managed by the **fconfig** command. Do not edit this key directly; it is automatically managed by RedBoot and the driver stack.

Name

nconfig list — List configuration keys

Synopsis

```
nconfig list
```

Arguments

The command takes no arguments.

Description

This command lists all known configuration keys. The list is output in arbitrary order.

Examples

```
RedBoot> nconfig list
The config store contains:
  nand.partition1.size
  nand.partition1.base
  ecos.fakeflash.config
RedBoot>
```

Note

The config key **`ecos.fakeflash.config`** contains the RedBoot configuration information as managed by the **fconfig** command. Do not edit this key directly; it is automatically managed by RedBoot and the driver stack.

Name

nconfig info — Query metadata for one or more config keys

Synopsis

```
nconfig info <key> [<key>...]
```

Arguments

One or more config keys. Information is returned about each of them in turn.

Description

This command outputs the metadata (type and data size) for the requested configuration key(s) in the form they currently appear in the store.

Note

The type of a data key is not fixed; it may be changed later, either via RedBoot or by an application. Similarly, the reported size of a variable-length type (`string` or `bytes`) only reflects its current size in storage and may change later.

Examples

```
RedBoot> nconfig info nand.partition1.base nand.partition1.size
Key         : nand.partition1.base
  type      : uint
  data size : 4
Key         : nand.partition1.size
  type      : uint
  data size : 4
RedBoot> nconfig info foo
Key foo not found
RedBoot>
```

Name

nconfig types — List all known config data types

Synopsis

```
nconfig types
```

Arguments

This command takes no arguments.

Description

This command lists all data types the config store knows about.

Config store data types

`uint`	Unsigned 32-bit integer
`bool`	Boolean
`string`	A printable, NULL-terminated string of arbitrary length.
`bytes`	A byte array of arbitrary length.

Note

It is possible that data could be present of an unknown type, if support for new types has been added to an application. The config store has been specifically designed to cope with unknown types and not disturb them.

Examples

```
RedBoot> nconfig types
This build supports the following types:
uint, bool, bytes, string
RedBoot>
```

Name

nconfig get — Outputs the current value of one or more config keys

Synopsis

```
nconfig get <key> [<key>...]
```

Arguments

One or more config keys.

Description

This command retrieves and outputs the current values of one or more config keys. Byte arrays are pretty-printed in hex; strings are output without line breaks.

Note

It is possible that data could be present of an unknown type, if support for new types has been added to an application. The **get** logic will not be able to display types it does not know about.

Examples

```
RedBoot> nconfig get nand.partition1.base nand.partition1.size
nand.partition1.base=16
nand.partition1.size=200
RedBoot> nconfig get app1.magic
app1.magic=This is a string example
RedBoot> nconfig get app2.magic
ecos.fakeflash.config: <64 bytes, hex follows>a
 00100000cefaad0b 010c0100626f6f74 5f73637269707400 000000000411010c
 626f6f745f736372 6970745f64617461 00626f6f745f7363 7269707400000000
RedBoot>
```

Name

nconfig put — Writes a config key

Synopsis

```
nconfig put <key> <type> <value>
```

Arguments

The config key, type and the value or data to be written to it.

Description

This command writes a single key to the store. If the key was already present in the store, its existing value is overwritten.

The value of a `bool` variable may be specified as **`true`**, **`True`**, **`t`**, **`T`**, **`1`**; **`false`**, **`False`**, **`f`**, **`F`**, or **`0`**.

Notes

- This interface does not support the `bytes` type yet.
- To write a string containing a space from within RedBoot, it is necessary to surround the string in quote marks.
- This command may be configured out with the `CYGSEM_REDBOOT_CONFIGSTORE_PUT_CMD` option.

Examples

```
RedBoot> nconfig put nand.partition1.size uint 100
Written OK
RedBoot> nconfig put app1.magic string "This is a string"
Written OK
RedBoot> nconfig put baz.qux bool true
Written OK
RedBoot> nconfig get baz.qux
baz.qux=True
RedBoot>
```

Name

nconfig del — Deletes a config key

Synopsis

```
nconfig del <key>
```

Arguments

The config key to delete.

Description

This command deletes a key and its data from the store.

Note

This command may be configured out with the `CYGSEM_REDBOOT_CONFIGSTORE_DEL_CMD` option.

Examples

```
RedBoot> nconfig del foo.bar
Deleted OK
RedBoot> nconfig del foo.bar
Key foo.bar not found
RedBoot>
```

Name

nconfig dump — Diagnostic dump output

Synopsis

`nconfig dump`

Arguments

This command takes no arguments.

Description

This command outputs a diagnostic dump of all the configuration data in the store.

Note

This command is not present by default. It may be configured in, with the `CYGSEM_REDBOOT_CONFIGSTORE_DUMP_CMD` option.

Caution

This command can generate excessive amounts of output, which may take a long time to send over a slow debug channel.

Note

While this command outputs byte arrays in hex, they are not line-wrapped.

Examples

```
RedBoot> nconfig dump
Store serial number 52:
  app1.magic=This is a string
  nand.partition1.size=200
  nand.partition1.base=16
  ecos.fakeflash.config=00100000cefaad0b 010c0100626f6f74 5f73637269707400  626f6f745f736372 6970745f64617461 \
00626f6f745f7363 7269707400000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 \
0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 \
0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000
 (Ends)
RedBoot>
```

NAND manipulation commands

Notes

- These commands are only available when there is at least one NAND driver configured.
- All the **`nand badblocks`** commands may be configured out simultaneously by turning off the `CYGSEM_REDBOOT_NAND_BADBLOCKS_CMD` option.

Name

nand list — List NAND devices

Synopsis

`nand list`

Arguments

The command takes no arguments.

Description

This command lists all known NAND devices for which a driver is loaded in the current eCos configuration. The list is output in arbitrary order.

Examples

```
RedBoot> nand list
NAND devices available:
        onboard
RedBoot>
```

Name

nand info — Output information about a NAND device

Synopsis

```
nand info <dev> [<dev>...]
```

Arguments

Specify the NAND device name(s) to query. If none are given, all available devices will be queried in turn.

Description

This command outputs information about selected NAND devices. The output includes information about the device physical configuration (not including the spare area of each page) as well as the current configured partition table.

Note

For information about how the partition table is defined and controlled on a particular device, refer to the relevant platform HAL documentation. Some devices can be reconfigured dynamically from within RedBoot; others have a static configuration which requires that eCos (and RedBoot) be rebuilt.

Examples

```
RedBoot> nand info
NAND device `onboard':
  2048 bytes/page, 64 pages/block, capacity 2048 blocks x 128 kB = 256 MB
  Partition Start Blocks
       0       6     10
       1      16    200
RedBoot>
```

Name

nand badblocks states — List all current valid BBT states

Synopsis

```
nand badblocks states
```

Arguments

This command takes no arguments.

Description

This command lists the names of the possible states that may be applied to each block within the Bad Block Table.

Examples

```
RedBoot> nand badblocks states
The possible block states are:
        0 : OK
        1 : worn bad
        2 : reserved
        3 : factory bad
RedBoot>
```

Name

nand badblocks summary — Summary of the Bad Block Table

Synopsis

```
nand badblocks summary [-d <device>]
```

Arguments

This command takes an optional device argument. If not specified, and only one device is present, it applies to the single device; if more than one device is present, it must be specified.

Description

This command outputs a human-readable summary of the information held within the Bad Block Table of a device.

Examples

```
RedBoot> nand badblocks summary
Device `onboard' has 2048 blocks, of which 2 are factory bad, 2 reserved. 2044 blocks (99%) are OK.
RedBoot>
```

Name

nand badblocks list — List not-OK blocks in a Bad Block Table

Synopsis

```
nand badblocks list [-d <device>]
```

Arguments

This command takes an optional device argument. If not specified, and only one device is present, it applies to the single device; if more than one device is present, it must be specified.

Description

This command outputs a human-readable list of the numbers of all not-OK blocks in a device, according to its Bad Block Table.

Examples

```
RedBoot> nand badblocks list -d onboard
factory bad: 100, 101. (count: 2 blocks)
worn bad: no blocks
reserved: 2046, 2047. (count: 2 blocks)
RedBoot>
```

Name

nand badblocks mark — Manipulate the Bad Block Table

Synopsis

```
nand badblocks mark [-d <device>] -b <block> -s <state>
```

Arguments

`-b block`

The block number whose state is to be manipulated.

> **Note**
>
> Block numbers to this command are device block numbers, with 0 meaning the first block on the device.

`-s state`

The new state for the given block. This must be a state number, as output by **`nand badblocks states`**.

`-d device`

(Optional.) If not specified, and only one device is present, the command is assumed to apply to the NAND device; if more than one device is present, it must be specified.

Description

This command explicitly manipulates the Bad Block Table.

> **Note**
>
> This command may be configured out with the `CYGSEM_REDBOOT_NAND_BADBLOCKS_MARK_CMD` option.

> **Warning**
>
> Misuse of this command is likely to cause corruption of data you care about!
>
> - Live data in affected blocks can be rendered inaccessible to eCos.
> - Blocks which have been marked as unreliable due to wear can be brought back into circulation but will still be unreliable.

Examples

```
RedBoot> nand badblocks mark -d onboard -b 2000 -s 1
OK
RedBoot> nand badblocks list -d onboard
factory bad: 100, 101. (count: 2 blocks)
worn bad: 2000. (count: 1 blocks)
reserved: 2046, 2047. (count: 2 blocks)
RedBoot> nand badblocks mark -d onboard -b 2000 -s 0
OK
RedBoot> nand badblocks list -d onboard
factory bad: 100, 101. (count: 2 blocks)
worn bad: no blocks
reserved: 2046, 2047. (count: 2 blocks)
RedBoot>
```

Name

nand erase — Erase an entire NAND partition

Synopsis

```
nand erase <device>/<partition>
```

Arguments

This command takes a single argument encoding the device and partition to erase.

Description

This command erases every block of a NAND partition.

Note

This command may be configured out with the `CYGSEM_REDBOOT_NAND_ERASE_CMD` option.

Examples

```
RedBoot> nand info
NAND device `onboard':
  2048 bytes/page, 64 pages/block, capacity 2048 blocks x 128 kB = 256 MB
  Partition Start Blocks
       0       6     10
       1      16    200
RedBoot> nand erase onboard/2
nand erase: partition not found
RedBoot> nand erase onboard/1
Erasing partition blocks 0 to 199...
............................Skipping block 84 (factory bad)
.Skipping block 85 (factory bad)
......................................
Erase complete.
RedBoot>
```

Name

nand eraseblock — Erase a specific NAND block

Synopsis

```
nand eraseblock -d <device> -b <block> [-n <count>]
```

Arguments

-b block The block number to erase (or, if -n is given, the first block).

> ### Note
>
> Block numbers to this command are device block numbers, with 0 meaning the first block on the device.

-n count The number of blocks to erase. This defaults to 1.

-d device The name of the device to affect.

Description

This command erases one or more blocks on a NAND device.

Notes

- This command may be configured out with the `CYGSEM_REDBOOT_NAND_ERASEBLOCK_CMD` option.
- This command ignores the Bad Blocks Table. It will attempt to erase any block on the device, regardless of its state.

Warning

This command is powerful, but potentially dangerous as it will erase data without any safety checks. It is possible to erase RedBoot, other files required for system bootstrap, the Bad Block Table itself, any stored configuration data, etc. Be very sure of the block number before committing!

Examples

```
RedBoot> nand eraseblock -d onboard -b 42 -n 2
Erasing device onboard blocks 42-43...
Erase complete.
RedBoot>
```

Chapter 226. Rebuilding RedBoot

Introduction

Prebuilt images of RedBoot are provided for all target platforms supported by an eCosCentric release, so it is not normally required to rebuild RedBoot in order to be able to begin your software development on eCos/eCosPro and the target platform. However, for later development you may wish to build your own cut down version of RedBoot or custom enhanced version that includes features such as manufacturing tests, initial programming, recovery, security and so on. This chapter describes the process of building, or rebuilding, RedBoot.

RedBoot is built as an application on top of eCos. The makefile rules for building RedBoot are part of the eCos CDL package so building RedBoot may be easily achieved. Typically building involves a command shell and the command line tool ecosconfig, with additional configuration or final build done through the eCos Configuration Tool.

Building RedBoot requires only a few steps: selecting the platform and the RedBoot template, importing a platform specific configuration file, and finally starting the build.

The platform specific configuration file makes sure the settings are correct for building RedBoot for the given target platform. Each target platform supporting RedBoot normally provides at least one configuration file whose name typically indicates the startup mode of RedBoot (see the section called “RedBoot Startup Mode”). For example, `redboot_RAM.ecm` indicates the filename of a RAM mode RedBoot configuration and `redboot_ROM.ecm` or `redboot_ROMRAM.ecm` for a ROM or ROMRAM mode RedBoot configuration filename respectively. There may be additional configuration files according to the requirements of the particular platform. These files are placed into the `install/etc/redboot/<target>` directory of the install tree, or may be found in the `misc` sub-directory of the platform HAL directory for older releases of eCos.

The RedBoot build process results in a number of files in the `install/bin` directory. The ELF file `redboot.elf` is the principal result. Depending on the platform CDL, there will also be generated versions of RedBoot in other file formats, such as `redboot.bin` (binary format, good when doing an update of a primary RedBoot image, see the section called “Update the primary RedBoot flash image”), `redboot.srec` (Motorola S-record format, good when downloading a RAM mode image for execution), and `redboot.img` (stripped ELF format, good when downloading a RAM mode image for execution, smaller than the .srec file). Some platforms may provide additional file formats and also relocate some of these files to a particular address making them more suitable for downloading using a different boot monitor or flash programming tools.

The platform specific information in the relevant platform's HAL documentation must be consulted as there may be other special instructions required to build RedBoot.

Variables

These instructions assume that the *`ECOS_REPOSITORY`* environment variable contains the full pathname of the `packages` subdirectory or directories containing the eCosPro repository or repositories from which RedBoot is to be built.

These instructions also make use of the *`ECOS_TARGET`* and *`REDBOOT_CFG`* environment variables to simplify and provide generic instructions for building RedBoot. Their used is not required and users are free to replace these with their own actual values in the instructions provided.

The instructions provided in this section are for a **bash** shell environment but are also applicable to a Windows **CMD** environment. The only differences between the instructions for either is the appearance of environment variables in the instructions and the different format of directory paths. For example, where you see the use of the *`${REDBOOT_CFG}`* environment variable in the **bash** shell, *`%REDBOOT_CFG%`* must be used in the Windows **CMD** environment.

The setting of an envirnment variable also differs. For example in the **bash** shell a variables may be set as follows:

```
$ export REDBOOT_CFG=redboot_RAM
```

In the windows **CMD** environment a variable is set as follows:

```
C:\users\demo> set REDBOOT_CFG=redboot_RAM
```

The same applies for the *${ECOS_TARGET}* environment variable. In the example below it is set for *at91sam9g45ek* target platform, the AT91SAM9G45-EKES board. For the **bash** shell:

```
$ export ECOS_TARGET=at91sam9g45ek
```

For the Windows **CMD**:

```
C:\users\demo> set ECOS_TARGET=at91sam9g45ek
```

Note

Windows users using eCosPro Developer's Kits may also use the **bash** shell by changing the shell opened from the eCos Configuration Tool using the Tools → Shell menu option by selecting View → Settings (**Ctrl+T**) , selecting the *Viewers/Shell* tab within the resulting dialog and changing the Command Shell to **bash**.

Building RedBoot using ecosconfig

To build RedBoot using the ecosconfig tool in a command line environment:

1. Create a temporary directory for building RedBoot, and change into it. For example:

   ```
   $ mkdir /tmp/${REDBOOT_CFG}
   $ cd /tmp/${REDBOOT_CFG}
   ```

2. Create a partial build tree to instantiate the platform specific configuration files for the chosen platform:

   ```
   $ ecosconfig --noresolve new ${ECOS_TARGET} redboot
   $ ecosconfig --ignore-errors --no-resolve tree
   $ make etc
   ```

   At this point all the relevant RedBoot configuration files should be found in the `install/etc/redboot/`*${ECOS_TARGET}* subdirectory. `install/etc` will become further populated with other configuration files, such as example PEEDI and OpenOCD configuration files, after a complete build of RedBoot.

3. Import the appropriate platform RedBoot configuration file:

   ```
   $ ecosconfig --no-resolve import install/etc/${ECOS_TARGET}/${REDBOOT_CFG}.ecm
   ```

   At this point the eCos configuration in `ecos.ecc` will contain all the unresolved settings required to build a *${REDBOOT_CFG}* for the *${ECOS_TARGET}* platform

4. Resolve any conflicts and create build tree for RedBoot:

   ```
   $ ecosconfig resolve
   $ ecosconfig tree
   ```

5. RedBoot can now be built:

   ```
   $ make
   ```

   The resulting RedBoot files will be in the associated install directory, in this example, `./install/bin`.

Note

Older revisions of eCos or some platforms may not support the `etc` make target. In these instances the RedBoot configuration file will not be found in the `install/etc/`*${ECOS_TARGET}*`/` subdirectory but may be found in the `misc` subdirectory of the platform HAL within *`${ECOS_REPOSITORY}`*. In these instances please refer to the specific platform HAL documentation for instructions on how to build RedBoot using an appropriate configuration file.

To build for another configuration, simply change the *`REDBOOT_CFG`* definition accordingly. For example:

```
export REDBOOT_CFG=redboot_ROM
mkdir /tmp/${REDBOOT_CFG}
cd /tmp/${REDBOOT_CFG}
ecosconfig --no-resolve new ${ECOS_TARGET} redboot
ecosconfig --no-resolve --ignore-errors tree
make etc
ecosconfig --no-resolve import install/etc/${REDBOOT_CFG}.ecm
ecosconfig resolve
ecosconfig tree
make
```

Notes

- The options `--no-resolve` and `--ignore-errors` are essential to certain targets. Their purpose is to delay the libcdl inferrence engine from being applied to a configuration until the required settings for the RedBoot configuration have been imported. This prevents any inferrences from being made during each creation step of the RedBoot configuration which may steer the final outcome to an unresolveable conflict. This is because inferrences cannot currently be implicitly undone within a configuration, and some inferences may need to be reversed by settings imported within *`${REDBOOT_CFG}`*`.ecm`. The final **ecosconfig resolve** command above therefore applies the libcdl inferrence engine to the configuration settings once all the required settings have been made. The options are otherwise harmless for targets which are not sensitive to premature inferrences.

- If the **bash** shell or Windows **CMD** was started from within the eCos Configuration Tool, the environment variable *`ECOS_TARGET`* will be set within the shell to the target of the configuration tool's eCos configuration at the time **bash** or **CMD** was started.

Rebuilding RedBoot from the eCos Configuration Tool

While it is possible to rebuild RedBoot from the eCos Configuration Tool, creating a RedBoot configuration is not always simple due to certain behavioural limitations of the eCos Configuration Tool. Instead, developers are recommended to use the Tools → Shell menu option to create a command shell and follow the instructions in the section called “Building RedBoot using ecosconfig” up to and including the **make etc** step with the inclusion of the `--compat` and `--config=<`*`savefile`*`>.ecc` options to every **ecosconfig** command. For example, assuming a working directory of `/tmp`:

```
cd /tmp
ecosconfig --compat --config=<savefile>.ecc --no-resolve new ${ECOS_TARGET} redboot
ecosconfig --compat --config=<savefile>.ecc --no-resolve --ignore-errors tree
make etc
```

Note

The environment variable *`${ECOS_TARGET}`* will be set by the eCos Configuration Tool to the current configuration's hardware, or profile's default hardware if no configuration has been created. It may therefore not be necessary to set this variable.

At this point the 1st stage RedBoot configuration can be loaded into the eCos Configuration Tool using File → Open to open the file `/tmp/<`*`savefile`*`>.ecc` and the RedBoot settings imported from `/tmp/<`*`savefile`*`>_install/etc/redboot/`*`${ECOS_TARGET}`*`/`*`${REDBOOT_CFG}`*`.ecm`. using File → Import .

Depending on the platform, a number of conflicts may need to be resolved before the build can be started. To resolve conflicts, if any, use the menu item Tools → Resolve Conflicts .

If you wish to switch to a different location or filename, you may now do so using the menu option File → Save As .

Generate a build tree to instantiate the platform specific configuration files for the chosen platform Build → Generate Build Tree .

Then start the build (Build → Library (**F7**)) and wait for it to complete. The resulting RedBoot files will be in the associated install directory, for the example this would be `<savefile>_install/bin`

As noted above, please also refer to the platform's HAL documentation to determine if there are any additional platform specific instructions that must be followed when rebuilding RedBoot.

Chapter 227. Updating RedBoot

Introduction

RedBoot normally resides in internal flash on the CPU or external flash on the board. It is often possible to update RedBoot in situ using a hardware debugger or even Redboot's flash management commands. Occasionally vendor specific software tools such as ATMEL's SAM-BA In- system Programmer may also be required.

The process of updating RedBoot in situ is documented in this section. For this process, it is assumed that the target is connected to a host system and that there is a serial or TCPIP connection giving access to the RedBoot CLI. For platforms with a ROMRAM mode RedBoot, skip to the section called “Update the primary RedBoot flash image”.

Older boards may use EEPROM although this is a lot less common nowadays. In this case of EPROM, updating RedBoot normally necessitates physically removing the part and reprogramming a new RedBoot image into it using prommer hardware.

Note

The addresses and sizes included in the below are examples only, and will differ from those you will see. This is normal and should not cause concern.

Load and start a RedBoot RAM instance

There are a number of choices here. The basic case is where a RAM mode image has been stored in the FIS (flash Image System). To load and execute this image, use the commands:

```
RedBoot> fis load RedBoot[RAM]
RedBoot> go
```

If this image is not available, or does not work, then an alternate RAM mode image can be loaded via one of several other methods:

- TFTP via a network connection:

  ```
  RedBoot> load redboot_RAM.img
  Entry point: 0x060213c0, address range: 0x06020000-0x060369c8
  RedBoot> go
  ```

- X, Y or Z modem: (y modem in this example)

  ```
  RedBoot> load -m y -r -b %{FREEMEMLO}
  ...
  Raw file loaded 0x06046800-0x06062fe8, assumed entry at 0x06046800
  RedBoot> go
  ```

- A filesystem, if available:

  ```
  RedBoot> load hda2:redboot_RAM.img
  Entry point: 0x060213c0, address range: 0x06020000-0x060369c8
  RedBoot> go
  ```

Notes

- Refer to the **load** command for a complete list of options available.
- If you expect to be doing this more than once, it is a good idea to program the RAM mode image into the flash. You do this using the **fis create** command after having downloaded the RAM mode image, but before you start it.

Some platforms support locking (write protecting) certain regions of the flash, while others do not. If your platform does not support locking, simply ignore the **fis unlock** and **fis lock** steps (the commands will not be recognized by RedBoot).

```
RedBoot> fis unlock RedBoot[RAM]
  ... Unlock from 0x00000000-0x00020000: ..
RedBoot> fis create RedBoot[RAM]
An image named 'RedBoot[RAM]' exists - continue (y/n)? y
* CAUTION * about to program 'RedBoot[RAM]'
            at 0x00020000..0x000369c7 from 0x06020000 - continue (y/n)?y
... Erase from 0x00020000-0x00040000: ..
... Program from 0x06020000-0x060369c8 at 0x00020000: ..
... Erase from 0x00070000-0x00080000: .
... Program from 0x0606f000-0x0607f000 at 0x00070000: .
RedBoot> fis lock RedBoot[RAM]
  ... Lock from 0x00000000-0x00020000: ..
```

Update the primary RedBoot flash image

An instance of RedBoot should now be running on the target from RAM. This can be verified by looking for the mode identifier in the banner. It should be either [RAM] or [ROMRAM].

If this is the first time RedBoot is running on the board or if the flash contents has been damaged, initialize the FIS directory:

```
RedBoot> fis init -f
About to initialize [format] FLASH image system - continue (y/n)? y
*** Initialize FLASH Image System
... Erase from 0x00020000-0x00070000: .....
... Erase from 0x00080000-0x00080000:
... Erase from 0x00070000-0x00080000: .
... Program from 0x0606f000-0x0607f000 at 0x00070000: .
```

It is important to understand that the presence of a correctly initialized FIS directory allows RedBoot to automatically determine the flash parameters. Additionally, executing the steps below as stated without loading other data or using other flash commands (than possibly **fis list**) allows RedBoot to automatically determine the image location and size parameters. This greatly reduces the risk of potential critical mistakes due to typographical errors. It is still always possible to explicitly specify parameters, and indeed override these, but it is not advised.

Note

If the new RedBoot image has grown beyond the slot in flash reserved for it, it is necessary to change the RedBoot configuration option CYGBLD_REDBOOT_MIN_IMAGE_SIZE so the FIS is created with adequate space reserved for RedBoot images. In this case, it is necessary to re-initialize the FIS directory as described above, using a RAM mode RedBoot compiled with the updated configuration.

Using the **load** command, download the new flash based image from the host, relocating the image to RAM:

```
RedBoot> load -r -b %{FREEMEMLO} redboot_ROM.bin
Raw file loaded 0x06046800-0x06062fe8, assumed entry at 0x06046800
```

Notes

- This command loads the RedBoot image using the TFTP protocol via a network connection. Other methods of loading are available, refer to the load command for more details.

- The binary version of the image is being downloaded. This is to ensure that the memory after the image is loaded should match the contents of the file on the host. Loading SREC or ELF versions of the image does not guarantee this since these formats may contain holes, leaving bytes in these holes in an unknown state after the load, and thus causing a likely cksum difference. It is possible to use these, but then the step verifying the cksum below may fail.

Once the image is loaded into RAM, it should be checksummed, thus verifying that the image on the target is indeed the image intended to be loaded, and that no corruption of the image has happened. This is done using the cksum command:

```
RedBoot> cksum
Computing cksum for area 0x06046800-0x06062fe8
POSIX cksum = 2535322412 116712 (0x971df32c 0x0001c7e8)
```

Compare the numbers with those for the binary version of the image on the host. If they do not match, try downloading the image again.

Assuming the cksum matches, the next step is programming the image into flash using the FIS commands.

Some platforms support locking (write protecting) certain regions of the flash, while others do not. If your platform does not support locking, simply ignore the **fis unlock** and **fis lock** steps (the commands will not be recognized by RedBoot).

```
RedBoot> fis unlock RedBoot
... Unlock from 0x00000000-0x00020000: ..
RedBoot> fis create RedBoot
An image named 'RedBoot' exists - continue (y/n)? y
* CAUTION * about to program 'RedBoot'
            at 0x00000000..0x0001c7e7 from 0x06046800 - continue (y/n)? y
... Erase from 0x00000000-0x00020000: ..
... Program from 0x06046800-0x06062fe8 at 0x00000000: ..
... Erase from 0x00070000-0x00080000: .
... Program from 0x0606f000-0x0607f000 at 0x00070000: .
RedBoot> fis lock RedBoot
  ... Lock from 0x00000000-0x00020000: ..
```

Reboot; run the new RedBoot image

Once the image has been successfully written into the flash, simply reset the target OR RUN THE **reset** command and the new version of RedBoot should be running.

When installing RedBoot for the first time, or after updating to a newer RedBoot with different configuration keys, it is necessary to update the configuration directory in the flash using the **fconfig** command. See the section called “Persistent State Flash-based Configuration and Control ”.

Chapter 228. Initial Installation

Hardware Installation

Initial installation of RedBoot into the internel or external flash, or EEPROM, of target hardware is described alongside the respective target hardware documentation in the eCos and eCosPro Reference Manual.

Note

The RedBoot installation documentation of outdated hardware and hardware no longer supported has been removed from editions of this document from July 2016 onwards. If the eCos and eCosPro Reference Manual does not contain instructions on installing RedBoot for your hardware, please refer to the [RedBoot User's Guide] or [eCos Reference Manual] accompanying your eCos distribution.

What to Expect

The initial installation of RedBoot into a target's internal or external flash may normally be achieved using a hardware debugger, an application or tools provided by either the target hardware's manufacturer or the chip vendor of the target hardware, or even RedBoot's flash management commands from a revision of RedBoot running on the target hardware. In the latter case, a RAM executable RedBoot would have been loaded into the target hardware's RAM through the use of a hardware debugger from where it would have been started.

For example:

- ATMEL's SAM-BA In-system Programmer is a vendor specific software tool used for, amongst other, programming binary images into internal or external flash of ATMEL components.
- The Ronetix PEEDI is a hardware debugger that may be used to either program the internal or external flash of target hardware, or may be used to load RedBoot into the RAM (internal, external or otherwise) and commence it's execution, thereby allowing RedBoot to program itself into the flash of the target hardware.

Part LXXIII. Robust Boot Loader

Name

CYGPKG_RBL — provide a robust boot service

Description

The Robust Boot Loader (RBL) package provides an alternative to the usual RedBoot facilities for managing flash hardware, the **fis** and **fconfig** commands. It provides the following facilities:

1. An application can be stored in flash memory. This can be loaded and run automatically during RedBoot startup, or it can started manually.
2. The application can be updated in a robust fashion. The RBL code will automatically maintain a backup copy of the previous version of the application in flash. If something goes wrong during the update, for example a power cut, the system can still boot up using the backup.
3. There is also support for a block of persistent data. Applications can save a block of memory to flash and restore it later, possibly after a reboot. Again updates will happen in a robust fashion with a primary and a backup copy.
4. The RBL functionality can be accessed by application code via a suitable API. Some of the functionality is also available via RedBoot commands.

The RBL package is aimed primarily at field deployment of production systems rather than at application development. The code update facility allows the application to be updated when necessary. The persistent data can be used to hold per-unit settings and any state that should be preserved across power failures or anything else that causes a reboot. There is also some control over standard I/O: in a deployed system the serial port may be connected to some other hardware and outputting a RedBoot banner during startup could be confusing.

The package comes in two parts. When building RedBoot for a particular target platform the package detects the presence of CYGPKG_REDBOOT in the configuration and modifies the available functionality. In any other configuration, used for building the application, the package provides a set of library routines that allow access to this functionality.

The RBL code has gone through a number of versions, affecting the protocol used between the RedBoot code and the application. The version is selected via the configuration option CYGNUM_RBL_VERSION. RedBoot and the application must use the same version of RBL: an application linked against a V1 version of the RBL library routines will not work on top of a RedBoot built with the V2 code, and vice versa. By default the latest version of the protocol will be used.

RedBoot Builds

The first step in building RedBoot with RBL support is to create a RedBoot configuration appropriate for the target platform. This is somewhat target specific so the appropriate platform HAL or RedBoot documentation should be consulted for further details. Typically RedBoot should be configured for ROM startup.

Given this initial configuration the RBL package CYGPKG_RBL should now be added to the configuration, using one of the eCos configuration tools. For example with the command line tool this involves using **ecosconfig add rbl**. This involves a conflict related to the configuration option CYGPKG_REDBOOT_FLASH. The RBL code replaces the standard RedBoot flash support, based around the **fis** and **fconfig** commands, so that must be disabled.

It is also possible to enable the configuration option CYGOPT_RBL_FLASH_OVERRIDE to continue to permit use of RedBoot's standard flash management facilities, but use of this option is discouraged as it is possible to use RedBoot's flash management commands in ways that conflict with RBL. In particular RedBoot's FIS will not be aware of the location of the RBL-managed code and data blocks. Also, RBL is not aware of the location of RedBoot's FIS and fconfig data in Flash, so it is strongly recommended to use the CYGDAT_RBL_RESERVED_FLASH_BLOCKS option to reserve the flash area used for FIS and fconfig data, usually the final one or two blocks in flash. As a result of these sorts of complications, caution is strongly advised if considering use of CYGOPT_RBL_FLASH_OVERRIDE.

There are a number of other configuration options which may need to be changed at this point:

`CYGNUM_RBL_VERSION`

This determines the protocol used between the RedBoot code and the application library routines. The default is to use the latest version. If RedBoot should continue to work with existing application binaries using an older version then this configuration option should be updated to match.

`CYGNUM_RBL_FLASH_BASE`

This option is only of interest if the target hardware has multiple banks of flash. The current version of the package requires that all RBL code and data blocks reside in a single bank. By default this will be the same bank of flash that holds RedBoot, either as determined by `CYGNUM_REDBOOT_FLASH_BASE` or the first bank of flash. If `CYGNUM_RBL_FLASH_BASE` is enabled then its value will determine the bank of flash used for the RBL code and data blocks. This may be useful if for example the board has a small NOR flash for holding RedBoot and a much larger serial dataflash for holding the eCos application and its data.

`CYGDAT_RBL_RESERVED_FLASH_BLOCKS`

Some of the flash blocks should not be used by the RBL package to store code or data. On a typical system it will be necessary to reserve at least flash block 0 because that is used to hold RedBoot itself. If a single flash block is too small to hold RedBoot or if there are other blocks which should be reserved then these should be listed in the value of `CYGDAT_RBL_RESERVED_FLASH_BLOCKS`, using a comma-separated list of numbers.

Note that if there are (smaller) boot blocks, RBL will instead consider them merged together as full sized blocks. For example, if the normal block size is 64K, but there are eight 8K boot blocks, those boot blocks will be counted as if they were another single 64K block, and the block numbers used for this option must reflect that.

`CYGNUM_RBL_CODE_BLOCKS`
`CYGNUM_RBL_DATA_BLOCKS`

The RBL code inside RedBoot needs to know how many flash blocks to allocate for the application code and for the persistent data. Because it maintains both primary and backup versions the actual requirements will be double the sum of these two options. These values will be hard-coded into the versions of RedBoot that get deployed in the field and cannot easily be changed, so they should be chosen carefully.

`CYGDAT_REDBOOT_DEFAULT_IP_ADDR`

Because the flash is used for storing RBL blocks there is nowhere for Redboot to store an **fis** directory or any **fconfig** settings. Hence certain settings like the default IP address cannot be managed via **fconfig**. Instead such settings must be configured statically via configuration options. Typically this will not be a problem because RBL will be used primarily in systems deployed in the field and RedBoot is used only to start the application. If application code needs such settings then they can be held in the RBL persistent data.

`CYGDAT_REDBOOT_DEFAULT_BOOT_SCRIPT`
`CYGNUM_REDBOOT_BOOT_SCRIPT_DEFAULT_TIMEOUT`

Usually RedBoot will obtain its boot script, if any, from the **fconfig** data. This is not available when using RBL so instead the boot script should be specified using a configuration option. For a deployed system `CYGDAT_REDBOOT_DEFAULT_BOOT_SCRIPT` should be set to `"rbl boot\n"` (including the double quotes). If the flash contains a valid application then this will be loaded and run automatically.

RedBoot gives users a chance to interrupt the system before running the boot script. Typically this is irrelevant for a deployed system because there will be nothing attached to the terminal port, but it can be useful during development if for example a broken version of the application has been installed by mistake. For a deployed system it may be desirable to reduce the timeout from the default 10 seconds, so that the application restarts more quickly after a power failure.

`CYGGLO_RBL_STDIO`

This provides control over standard I/O behaviour and is described in more detail below.

Once RedBoot has been appropriately configured it can be built and installed as usual for the target platform.

Application Builds

In addition to the RedBoot extensions the RBL package provides a number of functions for use by application code. These functions interact with the main RBL code inside the currently installed RedBoot using the eCos virtual vector mechanism.

The RBL package is not part of any standard eCos configuration so it must be added explicitly to the configuration used for application builds, for example by using **ecosconfig add**. The package's CDL script will detect that `CYGPKG_REDBOOT` is not defined and hence know that the package is being used for an application build rather than for extending RedBoot. The package does not require any special support from other parts of eCos.

The package's `misc` subdirectory contains three example programs that illustrate the use of the RBL API, together with a `README` and various support files.

Standard I/O

The RBL package builds on standard RedBoot functionality such as boot scripts. Some of this functionality is desirable in a development environment but can cause problems for a system deployed in the field. For example during startup RedBoot will usually output a banner message via a serial port, and it will listen on that serial port for an incoming control-C character in case the developer wants to abort the boot script. When a system is deployed that serial port may be connected to other hardware which does not expect the banner message, or which might be sending a stream of data that happens to include the occasional control-C.

When configuring and building RedBoot it is possible to change the default standard I/O behaviour using the configuration option `CYGGLO_RBL_STDIO`. This can take one of three values:

`standard`

RedBoot I/O behaves as usual, so typically the RedBoot banner will be sent out of a serial port and RedBoot will abort the boot script if it detects an incoming control-C. Application standard I/O is also not affected so for example a `printf` call will result in data being sent out of the serial port.

`suppress_redboot`

RedBoot I/O is suppressed. Any RedBoot output such as the banner message will be discarded, and incoming characters will be ignored. When the application is started via **rbl boot** I/O is reset so the application behaves as normal.

`suppress_all`

RedBoot I/O is suppressed as before, but I/O is not reset when the application is started. Hence any `printf` or similar output produced by the application gets discarded as well (at least in the default configuration where output is sent via the HAL diagnostic pseudo-device). The serial port is now available for other purposes, for example it can be accessed via a full serial driver.

Suppressing application I/O in this way will only work if the application uses the HAL virtual vector mechanisms to route I/O activity via RedBoot. If instead the application is configured to ignore the virtual vectors, for example by disabling `CYGSEM_HAL_USE_ROM_MONITOR`, then the RedBoot `CYGGLO_RBL_OUTPUT` setting will have no effect on application I/O.

Manipulating the standard I/O behaviour like this should only be done when the application will be started automatically via an **rbl boot** command in the boot script. If instead applications will be run via a gdb session interacting with RedBoot then suppressing RedBoot I/O will interfere with the gdb traffic.

There is also a common HAL configuration option that application developers should be aware of: `CYGDBG_HAL_DEBUG_GDB_CTRLC_SUPPORT`. By default this will be enabled for a RAM startup application. It causes the system startup code to install an interrupt handler that looks for incoming control-C characters and switch control to the gdb stubs. Usually this is sensible behaviour during development, but the option should be disabled for a deployed system. Note that it is the application configuration that needs to be changed, not the RedBoot configuration.

Name

`rbl` — access RBL functionality via the RedBoot prompt

Synopsis

`rbl info`

`rbl newcode` -b <buffer> -l <length>

`rbl newdata` -b <buffer> -l <length>

`rbl boot`

`rbl condboot`

Description

A RedBoot configured with RBL support will provide a new command **rbl** with various sub-commands. These allow users to access the RBL functionality at the RedBoot prompt.

rbl info can be used to get information about the RBL subsystem, for example how many flash blocks are allocated to each code block. Typical output might look like:

```
RedBoot> rbl info
Code block A : backup
  First flash block 1 (address 0xffe40000)
  Size 52012, sequence number 3
Code block B : primary
  First flash block 3 (address 0xffec0000)
  Size 52028, sequence number 4
Data block A : primary
  First flash block 2 (address 0xffe80000)
  Size 272, sequence number 11
Data block B : backup
  First flash block 5 (address 0xfff40000)
  Size 272, sequence number 10
```

This shows that the code is currently on its fourth revision and is held in flash block 2 at the given address. Here all code and data blocks fit into a single flash block, which will not always be the case. The data is currently on revision 11.

The **rbl newcode** command can be used to install a new revision of the code, although usually this will be done by the application itself via a call to `rbl_update_code`. However the RedBoot command can be used for the initial installation or if the currently installed version is broken somehow. Typically the code will first be loaded into RAM using a RedBoot **load** command, then programmed into flash.

```
RedBoot> load -r -m ymodem -b %{freememlo}
Raw file loaded 0x0000d400-0x00019f3b, assumed entry at 0x0000d400
xyzModem - CRC mode, 409(SOH)/0(STX)/0(CAN) packets, 3 retries
RedBoot> rbl newcode -b %{freememlo} -l 52028
... Erase from 0xffe40000-0xffe80000: .
... Program from 0x0000d400-0x00019f3c at 0xffe40000: .
... Program from 0x00005728-0x0000573c at 0xffe7ffec: .
```

The loaded program should be a stripped ELF executable appropriate for the target platform. The `-b` option specifies the memory location. Usually the RedBoot `%{freememlo}` variable will be used for this. The `-l` option corresponds to the file length. First the appropriate flash block or blocks are erased. Next the code is written to the flash. Finally the RBL code writes a little trailer at the end of the flash block containing a checksum, a sequence number, and similar information.

The **rbl newdata** command provides the same functionality for persistent data. Again the data is first loaded into RAM, then programmed into flash.

```
RedBoot> load -r -m ymodem -b %{freememlo}
Raw file loaded 0x0000d400-0x0000d5ba, assumed entry at 0x0000d400
xyzModem - CRC mode, 6(SOH)/0(STX)/0(CAN) packets, 3 retries
RedBoot> rbl newdata -b %{freememlo} -l 443
... Erase from 0xfff40000-0xfff80000: .
... Program from 0x0000d400-0x0000d5bb at 0xfff40000: .
... Program from 0x00005728-0x0000573c at 0xfff7ffec: .
```

The **rbl boot** command is used to load and run the current primary code block. The block should contain a stripped ELF executable which still contains the required relocation tables and the entry point, so there is no need for additional options. During development this command can be run manually to try out the current version of the application. In a production system RedBoot can be configured to run this command automatically by setting the configuration option `CYGDAT_REDBOOT_DEFAULT_BOOT_SCRIPT`. The command does not return. If it is necessary to get back to a RedBoot prompt then either the target board should be reset or the loaded application should call `rbl_reset`.

rbl condboot is a variant of **rbl boot**, available only on certain platforms. The command checks a platform-specific condition, for example the state of a jumper or a button. Depending on the condition **condboot** will either proceed to load and run the current primary code block in exactly the same way as **rbl boot**, or it will do nothing. Again in a production system RedBoot can be configured to run this command automatically by setting the configuration option `CYGDAT_REDBOOT_DEFAULT_BOOT_SCRIPT`. Following power up or reset RedBoot will normally run the current application, but if the button is held down then it will provide an interactive session instead (unless of course the boot script runs additional commands after **rbl condboot**). The interactive session allows the usual RedBoot and **rbl** commands to be executed, so for example the user can perform a ymodem transfer and then replace the primary code block via **rbl newcode**.

Jumpers and buttons are inherently platform-specific so **rbl condboot** will only be built if the platform HAL provides a suitable macro `HAL_RBL_CONDBOOT`. Typically this macro would be defined in the header file `cyg/hal/plf_io.h` which is automatically #include'd by the RBL code. The macro should take the following form:

```
#define HAL_RBL_CONDBOOT(_do_boot_)     \
    CYG_MACRO_START                     \
    …                                   \
    CYG_MACRO_END
```

`_do_boot_` should be set to 1 if the system should proceed with the bootstrap, i.e. load and run the primary code block. It should be set to 0 if **rbl condboot** should do nothing. Depending on the complexity of the hardware the macro body may involve just a couple of lines of inline code or it may involve a function call into the main platform HAL code, for example:

```
#define HAL_RBL_CONDBOOT(_do_boot_)                            \
    CYG_MACRO_START                                            \
    extern int hal_alaia_rbl_condboot(void);                   \
    _do_boot_ = hal_alaia_rbl_condboot();                      \
    CYG_MACRO_END
```

Name

RBL functions — allow applications to access RBL services

Synopsis

#include <cyg/rbl/rbl.h>

cyg_bool **rbl_get_flash_details**(*details*);

rbl_flash_block_purpose **rbl_get_flash_block_purpose**(*block*);

cyg_bool **rbl_get_block_details**(*which*, *details*);

cyg_bool **rbl_update_code**(*buffer*, *length*);

cyg_bool **rbl_update_data**(*buffer*, *length*);

cyg_bool **rbl_load_data**(*buffer*, *length*);

void **rbl_reset**();

Flash Details

rbl_get_flash_details can be used by application code to get information about the flash hardware and how it is being used by the RBL code inside RedBoot. The function takes a single argument, a pointer to an rbl_flash_details structure.

```
typedef struct rbl_flash_details {
    cyg_uint8*  rbl_flash_base;
    int         rbl_flash_block_size;
    int         rbl_flash_num_blocks;
    int         rbl_code_num_blocks;
    int         rbl_data_num_blocks;
    int         rbl_trailer_size;
} rbl_flash_details;
```

The *rbl_flash_base* field gives the location of the flash in the target's memory map. Application code does not usually need this information since the flash hardware is entirely managed by RedBoot, but it may be useful for debugging purposes.

The *rbl_flash_block_size* and *rbl_flash_num_blocks* provide further information about the flash hardware. Typical sets of values might be 8 blocks of 256K apiece, or 64 blocks of 64K. If the flash chips support smaller boot blocks then the eCos flash management code will usually treat these as a single full-size block.

rbl_code_num_blocks gives the number of flash blocks that will be used for the primary and the backup code blocks. It corresponds to the value of the CYGNUM_RBL_CODE_BLOCKS configuration option used when building RedBoot. *rbl_data_num_blocks* provides the same information for the persistent data blocks, with a value of 0 indicating that the support for persistent data was disabled.

The RBL code uses a small amount of flash memory for management purposes. This amount of memory is given by *rbl_trailer_size*. Application code can determine the maximum size of an executable using:

```
rbl_flash_details details;
if (! rbl_get_flash_details(&details)) {
    fputs("Error: failed to get RBL flash details\n", stderr);
    return false;
 }
size = (details.rbl_flash_block_size * details.rbl_code_num_blocks) -
       details.rbl_trailer_size;
```

The `rbl_get_flash_details` function will return true on success, false on failure. The most likely reason for failure is that the current RedBoot installation does not have RBL support.

Flash Blocks

`rbl_get_flash_block_purpose` can be used to find out how the RBL code inside RedBoot has allocated each flash block. Typically this is used only for debugging purposes. The argument should be a small number between 0 and `details.rbl_flash_num_blocks - 1`. The return value will be one of the following:

`rbl_flash_block_reserved`
: This flash block is reserved, for example it may hold some or all of the RedBoot code. Flash blocks can be reserved using the configuration option `CYGDAT_RBL_RESERVED_FLASH_BLOCKS` when building RedBoot.

`rbl_flash_block_code_A`
`rbl_flash_block_code_B`
`rbl_flash_block_data_A`
`rbl_flash_block_data_B`
: The flash block is used for an RBL code or data block.

`rbl_flash_block_free`
: This flash block is not used by the RBL code. It may be used by application code for other purposes.

`rbl_flash_block_invalid`
: This value will be returned if the argument to `rbl_get_flash_block_purpose` is outside the valid range. It will also be returned if the current RedBoot installation does not have RBL support.

RBL Block Details

`rbl_get_block_details` can be used to get information about a specific RBL block, for example the primary code block. The function takes two arguments. The first identifies the particular RBL block that is of interest:

`rbl_block_code_primary`
`rbl_block_code_backup`
`rbl_block_data_primary`
`rbl_block_data_backup`
: These identify an RBL block by purpose.

`rbl_block_code_A`
`rbl_block_code_B`
`rbl_block_data_A`
`rbl_block_data_B`
: These identify an RBL block by memory location. RedBoot will allocate flash blocks to code and data in the above order.

The second argument should be a pointer to an rbl_block_details structure which will be used for storing the results.

```
typedef struct rbl_block_details {
    cyg_bool    rbl_valid;
    cyg_uint32  rbl_first_flash_block;
    void*       rbl_address;
    cyg_uint32  rbl_size;
    cyg_uint32  rbl_sequence_number;
} rbl_block_details;
```

At any time a particular RBL block may or may not contain valid code or data. For example if the system has an installed application which has not yet been updated then the primary code block will be valid but the backup code block will be invalid. The other fields will only be valid if the *`rbl_valid`* flag is set.

The *`rbl_first_flash_block`* and *`rbl_address`* fields give information about where an RBL block is held in memory. If an RBL block is spread over multiple flash blocks then care has to be taken: there may be one or more reserved flash blocks in the middle of an RBL block.

The *rbl_size* field gives the current size of a code or data block. This is the actual size specified when the block was updated, not the maximum size.

The *rbl_sequence_number* is used by the RBL code to distinguish between primary and backup blocks. It may also prove useful for debugging purposes.

`rbl_get_block_details` returns true on success, false on failure. The function can fail if the current RedBoot installation does not have RedBoot support, or if an invalid argument is passed.

Updating the Code

`rbl_update_code` is used to install a new code image. It takes two arguments, a buffer and a length. The buffer should contain an ELF executable valid for the target platform, usually stripped to remove unnecessary debug information. Filling this buffer is left to application code.

The function returns true on success, false on failure. It can fail if the current RedBoot installation does not have RedBoot support, if an invalid argument is passed, or if the specified size is larger than the flash space available for a code block.

A flash update involves erasing one or more flash blocks and then programming in the new data. It is important that while this is happening no other threads or interrupt handlers access the flash hardware since that would interfere with the update. Hence interrupts will be disabled for some time while the update is happening.

Updating the Data

`rbl_update_data` is used to install a new version of the persistent data. It takes two arguments, a buffer and a length. The RBL code does not care about the contents of the buffer, this is left entirely to application code.

The function returns true on success, false on failure. It can fail if the current RedBoot installation does not have RBL support, if an invalid argument is passed, or if the specified size is larger than the flash space available for a data block. It is also possible that support for persistent data was disabled completely inside RedBoot by setting the configuration option `CYGNUM_RBL_DATA_BLOCKS` to 0.

A flash update involves erasing one or more flash blocks and then programming in the new data. It is important that while this is happening no other threads or interrupt handlers access the flash hardware since that would interfere with the update. Hence interrupts will be disabled for some time while the update is happening.

Loading the Data

`rbl_load_data` is used to load the current version of the persistent data back into memory. It takes two arguments, a buffer and a length. If there is a valid primary data block then the RBL code will transfer that data into the specified buffer. The amount of data transferred will be either the actual block size or the length argument, whichever is smaller.

The function returns true on success, false on failure. It can fail if the current RedBoot installation does not have RBL support, if an invalid argument is passed, or if there is no current primary data block.

Restarting the Hardware

`rbl_reset` can be used to restart the hardware. Typically this is used after a code update. The function takes no arguments and does not return.

Name

`V2 RBL functions` — allow applications to access RBL services

Synopsis

#include <cyg/rbl/rbl.h>

`cyg_bool` **`rbl_update_codeV`**`(`*`count`*`,` *`buffers[]`*`,` *`lengths[]`*`);`

`cyg_bool` **`rbl_update_dataV`**`(`*`count`*`,` *`buffers[]`*`,` *`lengths[]`*`);`

`cyg_bool` **`rbl_load_dataV`**`(`*`count`*`,` *`buffers[]`*`,` *`lengths[]`*`);`

`cyg_bool` **`rbl_update_code_begin`**`();`

`cyg_bool` **`rbl_update_code_block`**`(`*`buffer`*`,` *`length`*`);`

`cyg_bool` **`rbl_update_code_end`**`();`

`cyg_bool` **`rbl_update_code_abort`**`();`

`cyg_bool` **`rbl_update_data_begin`**`();`

`cyg_bool` **`rbl_update_data_block`**`(`*`buffer`*`,` *`length`*`);`

`cyg_bool` **`rbl_update_data_end`**`();`

`cyg_bool` **`rbl_update_data_abort`**`();`

`cyg_bool` **`rbl_load_data_begin`**`();`

`cyg_bool` **`rbl_load_data_block`**`(`*`where`*`,` *`length`*`);`

`cyg_bool` **`rbl_update_data_end`**`();`

Description

The original V1 API required single buffers for all update and load operations. This proved unduly restrictive, especially when installing a new code image obtained over a network, because there would be no guarantee that a single buffer of the required size could be dynamically allocated when required. Hence the API was extended for V2 with vector functions, allowing the code and data to be spread over multiple buffers, and with transaction functions, allowing new code images to be installed a piece at a time.

Vector Functions

The three vector functions `rbl_update_codeV`, `rbl_update_dataV` and `rbl_load_dataV` work in terms of a series of buffers rather than a single buffer. For example `rbl_load_data` is equivalent to:

```
cyg_bool
rbl_load_data(cyg_uint8* where, cyg_uint32 size)
{
    cyg_uint8    dataV[1];
    cyg_uint32   sizesV[1];

    dataV[0]  = where;
    sizesV[0] = size;
    return rbl_load_dataV(1, dataV, sizesV);
}
```

Obviously the vector functions become rather more useful for counts greater than 1. The update functions still require that all of the new images are resident in memory, but they no longer have to be in a single contiguous buffer.

When updating some flash drivers may impose limitations on the sizes. For example if the target hardware has a single 16-bit wide flash device then the flash driver may require that all flash write operations happen in multiples of 2 bytes, and the entries in the `sizesV` array should satisfy this requirement.

Transaction Functions

The transaction functions begin/block/end allow RBL operations to be performed in stages. For example `rbl_load_dataV` is equivalent to:

```
cyg_bool
rbl_load_dataV(cyg_uint32 count, cyg_uint8* whereV[], cyg_uint32 sizesV[])
{
    cyg_uint32 i;

    if (! rbl_load_data_begin()) {
        return false;
    }
    for (i = 0; i < count; i++) {
        if (! rbl_load_data_block(whereV[i], sizesV[i])) {
            return false;
        }
    }
    return rbl_load_data_end();
}
```

The begin function must be called at the start of a function. At any one time there can be only one code update, one data update, and one data load in progress, and the begin function will block if another thread is performing a conflicting RBL operation. Once the transaction is started the application can perform one or more block operations, and the transaction should normally be committed with an end function call. For an update the begin function will erase the appropriate flash blocks, the block function will write data to the flash, and the end function will write trailer data containing size, checksum and sequence number. At that point the image becomes the new primary code or data RBL block.

As an additional restriction, `rbl_update_data_end` will block if some other thread is currently loading data. This avoids confusion since otherwise that thread would end up loading data that is no longer primary, and it also avoids problems if another update operation is started immediately.

Unlike the simple or vector update functions, the transaction functions do not require that all of the new image is present in memory at the same time. Instead it is possible to begin a transaction, fetch the first part of the image over the network and install that, fetch the next part, and so on. If the application is unable to complete an update, for example because the network connection is lost, then there should be a call to the abort function instead of to the end function. When a transaction is aborted no trailer gets written to flash so the new image remains invalid and the old image stays as the primary.

As with the vector operations the flash driver may impose limitations on the size arguments to `rbl_update_code_block` and `rbl_update_block`. For example if the target hardware uses a 16-bit wide flash chip then the size argument may have to be a multiple of two bytes.

Error Conditions

All of the RBL functions return a simple boolean to indicate failure. In reality failures are unlikely, but can be caused by the following:

1. The currently installed RedBoot was built without RBL functionality so there is no code in the system to keep track of RBL images and install new ones.

2. RedBoot uses a different version of the RBL protocol, for example V1 when the application has been built with V2.
3. The RedBoot RBL code was unable to initialize the system. This can happen if the actual hardware does not match the RedBoot configuration, for example if the flash chips actually present are smaller than expected and cannot hold all the code and data blocks specified by the `CYGDAT_RBL_RESERVED_FLASH_BLOCKS`, `CYGNUM_RBL_CODE_BLOCKS` and `CYGNUM_RBL_DATA_BLOCKS` configuration options.
4. An attempt is made to load or update data when RedBoot has been configured with zero RBL data blocks,
5. An attempt is made to load more data than is actually present in the current data image, or to install a new image that does not fit in the number of configured flash blocks.
6. An unexpected error occurs inside the flash driver, for example an attempt to erase a flash block fails.

For the transaction functions, if an error occurs then the transaction is automatically aborted. There is no need for the application to call `rbl_update_code_abort` or `rbl_update_data_abort` explicitly, or to end a load operation.

Part LXXIV. RedBoot Extra Initialization

Name

CYGPKG_RBINIT — provide extra RedBoot initialization

Description

The RedBoot Extra Initialisation (RBINIT) package provides extra default initialization for RedBoot. This may be used to execute a set of initial commands, or to perform any additional platform or system specific initialization. The RBINIT package is aimed primarily at field deployment of production systems rather than at application development.

Building RedBoot

The first step in building RedBoot with RBINIT support is to create a RedBoot configuration appropriate for the target platform. This is somewhat target specific so the appropriate platform HAL or RedBoot documentation should be consulted for further details. Typically RedBoot should be configured for ROM startup.

Given this initial configuration the RBINIT package `CYGPKG_RBINIT` should now be added to the configuration, using one of the eCos configuration tools. For example with the command line tool this involves using **ecosconfig add rbinit**.

There are a number of other configuration options which may be changed at this point:

`CYGGLO_RBINIT_STDIO_DISABLE`

> This option causes all output generated during the extra initialization to be discarded. Turn this option off to get output for debugging purposes.

`CYGPKG_RBINIT_PRI`

> This option selects the priority of the extra initialization routine. The value of this option may either be `RedBoot_INIT_BEFORE_NET` or `RedBoot_INIT_AFTER_NET` which, as the names imply, cause the extra initialization to occur either before or after any network device is initialized.

Once RedBoot has been appropriately configured it can be built and installed as usual for the target platform.

Extra Initialization Function

The purpose of the extra initialization package is to execute the `rbinit_exec()` function. A default version of this function is contained in the `rbinit_exec.c` file within this package in the source repository.

The default content of this function provides support for loading and executing primary or secondary applications from a filesystem. The default function will first attempt to mount a JFFS2 filesystem named “jffs2” in the RedBoot FIS table. If that fails it will attempt to mount the first partition of an IDE hard disk using a FAT filesystem. Finally if that fails it will attempt to mount a RAM filesystem, although it will be empty. If a filesystem has been successfully mounted, it will attempt to load into RAM and run a program named “`app.primary`” from the root of the filesystem, and if that fails, a program named “`app.secondary`”.

A customized version of this function may be provided instead by using one of the following methods:

- Editing the file directly in the source repository.
- Adding an equivalent function in your application build and ensure its object file is linked into your application. This will cause the default implementation of `rbinit_exec()` provided in this package to be overridden. You may wish to use the version in this package as a template to start with.
- By copying the `rbinit_exec.c` file into the configuration's build tree and editing it there.

Part LXXV. Unity

Table of Contents

Chapter 229. Unity overview

Introduction

The `CYGPKG_UNITY` package provides a standard Unity Test framework implementation to eCos applications.

This package is covered by an “MIT” license as distributed in the original Unity package:

Example 229.1. “MIT” License

```
The MIT License (MIT)

Copyright (c) <year> 2007-21 Mike Karlesky, Mark VanderVoord, Greg Williams

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
THE SOFTWARE.
```

For definitive Unity documentation please refer to the main Unit Test website. We do not duplicate that documentation here.

Chapter 230. Configuration

This chapter shows how to incorporate the Unity support into an eCos configuration, and how to configure it once included.

Configuration Overview

The Unity support is contained in a single eCos package `CYGPKG_UNITY`. However, some functionality is dependant on other eCos features. e.g. the eCos dynamic memory allocator.

Quick Start

Incorporating the Unity support into your application is straightforward. The essential starting point is to incorporate the Unity eCos package (`CYGPKG_UNITY`) into your configuration.

This may be achieved directly using **ecosconfig add** on the command line, or the **Build->Packages…** menu item within the eCos Configuration Tool.

Depending on the Unity package configuration other packages may be required. The package requires that the `CYGPKG_INFRA` packages is included in the eCos application configuration.

Chapter 231. eCos port

Overview

The goal for the `CYGPKG_UNITY` package is to avoid where possible having to have any core Unity source file changes made specifically for eCos. This is to ensure that re-imports of newer versions of the library sources involve minimal effort. The files are as provided in the official Unity release package as imported, with the following exceptions:

1. Files have been moved, unmodified, to create a standard eCos package tree structure to integrate with the eCosPro build environment

   Only **relevant** files from the original project have been included in the eCos package.

The current Unity version provided by the eCos package is the github tagged release `v2.5.2` (released 26th January 2021).

The original project homepage can be found on github: Unity

The release package was downloaded from the github project page: releases/tag/v2.5.2

The following table highlights the files taken from the Unity package and their new location within the eCos `CYGPKG_UNITY` package:

Original github	**eCos package**
`src/unity.h`	`include/unity.h`
`src/unity_internals.h`	`include/unity_internals.h`
`extras/memory/src/unity_internals.h`	`include/unity_memory.h`
`extras/fixture/src/unity_fixture.h`	`include/unity_fixture.h`
`extras/fixture/src/unity_fixture_internals.h`	`include/unity_fixure_internals.h`
`src/unity.c`	`src/unity.c`
`extras/memory/src/unity_memory.c`	`src/unity_memory.c`
`extras/fixture/src/unity_fixture.c`	`src/unity_fixture.c`
`test/tests/self_assessment_utils.h`	`tests/tests/self_assessment_utils.h`
`test/tests/test_unity_core.c`	`tests/tests/test_unity_core.c`
`extras/memory/test/unity_memory_Test.c`	`tests/memory/unity_memory_Test.c`
`extras/memory/test/unity_output_Spy.c`	`tests/memory/unity_output_Spy.c`
`extras/memory/test/unity_output_Spy.h`	`tests/memory/unity_output_Spy.h`
`test/tests/test_unity_arrays.c`	`tests/tests/test_unity_arrays.c`
`test/tests/test_unity_doubles.c`	`tests/tests/test_unity_doubles.c`
`test/tests/test_unity_floats.c`	`tests/tests/test_unity_floats.c`
`test/tests/test_unity_integers.c`	`tests/tests/test_unity_integers.c`
`test/tests/test_unity_integers_64.c`	`tests/tests/test_unity_integers_64.c`
`test/tests/test_unity_memory.c`	`tests/tests/test_unity_memory.c`
`test/tests/test_unity_parameterized.c`	`tests/tests/test_unity_parameterized.c`
`test/tests/test_unity_strings.c`	`tests/tests/test_unity_strings.c`

Original github	**eCos package**
examples/example_1/src/ProductionCode.h	tests/example_1/ProductionCode.h
examples/example_1/src/ProductionCode.c	tests/example_1/ProductionCode.c
examples/example_1/test/TestProductionCode.c	tests/example_1/TestProductionCode.c
examples/example_1/src/ProductionCode2.h	tests/example_1/ProductionCode2.h
examples/example_1/src/ProductionCode2.c	tests/example_1/ProductionCode2.c
examples/example_1/test/TestProductionCode2.c	tests/example_1/TestProductionCode2.c

Chapter 232. Test Programs

Test Programs

Some Unity specific tests are built and can be used to verify correct operation of the library.

The original Unity `examples` are all very slight variations of the same underlying example for different build environments, so we only provide a single variant for the eCos tests build.

1. `unity_core`

   This test exercises the core Unity functionality.

2. `unity_core_memory`

   This test exercises the extra memory functionality if CYGPKG_UNITY_MEMORY is enabled.

3. `unity_floats`

   This test exercises the single-precision floating point `(float)` support when `UNITY_EXCLUDE_FLOAT` is **not** defined.

4. `unity_doubles`

   This test exercises the double-precision floating point `(double)` support when `UNITY_EXCLUDE_DOUBLE` is **not** defined.

5. `unity_integers`

   This test exercises the basic integer assert functionality.

6. `unity_integers_64`

   This test exercises the 64-bit support when `UNITY_SUPPORT_64` **is** defined, which is the case when using the default eCos `unity_config.h` configuration header.

7. `unity_memory`

   This test exercises the memory equality support. It is not related to the extra `CYGPKG_UNIT_MEMORY` functionality, but tests the core memory buffer assert support.

8. `unity_strings`

   This test exercises the core string functionality.

9. `example_1_1`

   This test is a simple example of using Unity based on the original Unity package `examples/example_1/test/TestProductionCode.c` source.

10. `example_1_2`

    This test is a simple example of using Unity based on the original Unity package `examples/example_1/test/TestProductionCode2.c` source.

Part LXXVI. Synthetic Target Architecture

Table of Contents

Chapter 233. eCos Synthetic Target

Name

The eCos synthetic target — Overview

Description

Usually eCos runs on either a custom piece of hardware, specially designed to meet the needs of a specific application, or on a development board of some sort that is available before the final hardware. Such boards have a number of things in common:

1. Obviously there has to be at least one processor to do the work. Often this will be a 32-bit processor, but it can be smaller or larger. Processor speed will vary widely, depending on the expected needs of the application. However the exact processor being used tends not to matter very much for most of the development process: the use of languages such as C or C++ means that the compiler will handle those details.

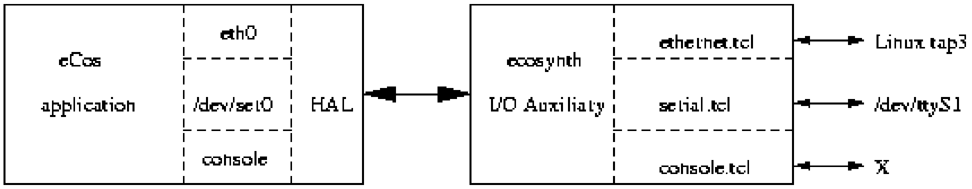
2. There needs to be memory for code and for data. A typical system will have two different types of memory. There will be some non-volatile memory such as flash, EPROM or masked ROM. There will also be some volatile memory such as DRAM or SRAM. Often the code for the final application will reside in the non-volatile memory and all of the RAM will be available for data. However updating non-volatile memory requires a non-trivial amount of effort, so for much of the development process it is more convenient to burn suitable firmware, for example RedBoot, into the non-volatile memory and then use that to load the application being debugged into RAM, alongside the application data and a small area reserved for use by the firmware.

3. The platform must provide certain mimimal I/O facilities. Most eCos configurations require a clock signal of some sort. There must also be some way of outputting diagnostics to the user, often but not always via a serial port. Unless special debug hardware is being used, source level debugging will require bidirectional communication between a host machine and the target hardware, usually via a serial port or an ethernet device.

4. All the above is not actually very useful yet because there is no way for the embedded device to interact with the rest of the world, except by generating diagnostics. Therefore an embedded device will have additional I/O hardware. This may be fairly standard hardware such as an ethernet or USB interface, or special hardware designed specifically for the intended application, or quite often some combination. Standard hardware such as ethernet or USB may be supported by eCos device drivers and protocol stacks, whereas the special hardware will be driven directly by application code.

Much of the above can be emulated on a typical PC running Linux. Instead of running the embedded application being developed on a target board of some sort, it can be run as a Linux process. The processor will be the PC's own processor, for example an x86, and the memory will be the process' address space. Some I/O facilities can be emulated directly through system calls. For example clock hardware can be emulated by setting up a `SIGALRM` signal, which will cause the process to be interrupted at regular intervals. This emulation of real hardware will not be particularly accurate, the number of cpu cycles available to the eCos application between clock ticks will vary widely depending on what else is running on the PC, but for much development work it will be good enough.

Other I/O facilities are provided through an I/O auxiliary process, ecosynth, that gets spawned by the eCos application during startup. When an eCos device driver wants to perform some I/O operation, for example send out an ethernet packet, it sends a request to the I/O auxiliary. That is an ordinary Linux application so it has ready access to all normal Linux I/O facilities. To emulate a device interrupt the I/O auxiliary can raise a `SIGIO` signal within the eCos application. The HAL's interrupt subsystem installs a signal handler for this, which will then invoke the standard eCos ISR/DSR mechanisms. The I/O auxiliary is based around Tcl scripting, making it easy to extend and customize. It should be possible to configure the synthetic target so that its I/O functionality is similar to what will be available on the final target hardware for the application being developed.



A key requirement for synthetic target code is that the embedded application must not be linked with any of the standard Linux libraries such as the GNU C library: that would lead to a confusing situation where both eCos and the Linux libraries attempted to provide functions such as `printf`. Instead the synthetic target support must be implemented directly on top of the Linux kernels' system call interface. For example, the kernel provides a system call for write operations. The actual function `write` is implemented in the system's C library, but all it does is move its arguments on to the stack or into certain registers and then execute a special trap instruction such as `int 0x80`. When this instruction is executed control transfers into the kernel, which will validate the arguments and perform the appropriate operation. Now, a synthetic target application cannot be linked with the system's C library. Instead it contains a function `cyg_hal_sys_write` which, like the C library's `write` function, pushes its arguments on to the stack and executes the trap instruction. The Linux kernel cannot tell the difference, so it will perform the I/O operation requested by the synthetic target. With appropriate knowledge of what system calls are available, this makes it possible to emulate the required I/O facilities. For example, spawning the ecosynth I/O auxiliary involves system calls `cyg_hal_sys_fork` and `cyg_hal_sys_execve`, and sending a request to the auxiliary uses `cyg_hal_sys_write`.

In many ways developing for the synthetic target is no different from developing for real embedded targets. eCos must be configured appropriately: selecting a suitable target such as **`i386linux`** will cause the configuration system to load the appropriate packages for this hardware; this includes an architectural HAL package and a platform-specific package; the architectural package contains generic code applicable to all Linux platforms, whereas the platform package is for specific Linux implementations such as the x86 version and contains any processor-specific code. Selecting this target will also bring in some device driver packages. Other aspects of the configuration such as which API's are supported are determined by the template, by adding and removing packages, and by fine-grained configuration.

In other ways developing for the synthetic target can be much easier than developing for a real embedded target. For example there is no need to worry about building and installing suitable firmware on the target hardware, and then downloading and debugging the actual application over a serial line or a similar connection. Instead an eCos application built for the synthetic target is mostly indistinguishable from an ordinary Linux program. It can be run simply by typing the name of the executable file at a shell prompt. Alternatively you can debug the application using whichever version of gdb is provided by your Linux distribution. There is no need to build or install special toolchains. Essentially using the synthetic target means that the various problems associated with real embedded hardware can be bypassed for much of the development process.

The eCos synthetic target provides emulation, not simulation. It is possible to run eCos in suitable architectural simulators but that involves a rather different approach to software development. For example, when running eCos on the psim PowerPC simulator you need appropriate cross-compilation tools that allow you to build PowerPC executables. These are then loaded into the simulator which interprets every instruction and attempts to simulate what would happen if the application were running on real hardware. This involves a lot of processing overhead, but depending on the functionality provided by the simulator it can give very accurate results. When developing for the synthetic target the executable is compiled for the PC's own processor and will be executed at full speed, with no need for a simulator or special tools. This will be much faster and somewhat simpler than using an architectural simulator, but no attempt is made to accurately match the behaviour of a real embedded target.

Name

Installation — Preparing to use the synthetic target

Host-side Software

To get the full functionality of the synthetic target, users must build and install the I/O auxiliary ecosynth and various support files. It is possible to develop applications for the synthetic target without the auxiliary, but only limited I/O facilities will be available. The relevant code resides in the `host` subdirectory of the synthetic target architectural HAL package, and building it involves the standard **configure**, **make**, and **make install** steps.

There are two main ways of building the host-side software. It is possible to build both the generic host-side software and all package-specific host-side software, including the I/O auxiliary. in a single build tree. This involves using the **configure** script at the toplevel of the eCos repository, which will automatically search the `packages` hierarchy for host-side software. For more information on this, see the `README.host` file at the top of the repository. Note that if you have an existing build tree which does not include the synthetic target architectural HAL package then it will be necessary to rerun the toplevel configure script: the search for appropriate packages happens at configure time.

The alternative is to build just the host-side for this package. This involves creating a suitable build directory and running the **configure** script. Note that building directly in the source tree is not allowed.

```
$ cd <somewhere suitable>
$ mkdir synth_build
$ cd synth_build
$ <repo<>/packages/hal/synth/arch/<version>/host/configure <options>
$ make
$ make install
```

The code makes extensive use of Tcl/TK and requires version 8.3 or later. This is checked by the **configure** script. By default it will use the system's Tcl installation in `/usr`. If a different, more recent Tcl installation should be used then its location can be specified using the options `--with-tcl=<path>`, `--with-tcl-header=<path>` and `--with-tcl-lib=<path>`. For more information on these options see the `README.host` file at the toplevel of the eCos repository.

Some users may also want to specify the install location using a `--prefix=<path>` option. The default install location is `/usr/local`. It is essential that the `bin` subdirectory of the install location is on the user's search `PATH`, otherwise the eCos application will be unable to locate and execute the I/O auxiliary ecosynth.

Because ecosynth is run automatically by an eCos application rather than explicitly by the user, it is not installed in the `bin` subdirectory itself. Instead it is installed below `libexec`, together with various support files such as images. At configure time it is usually possible to specify an alternative location for `libexec` using `--exec-prefix=<path>` or `--libexecdir=<path>`. These options should not be used for this package because the eCos application is built completely separately and does not know how the host-side was configured.

Toolchain

When developing eCos applications for a normal embedded target it is necessary to use a suitable cross-compiler and related tools such as the linker. Developing for the synthetic target is easier because you can just use the standard GNU tools (gcc, g++, ld, …) which were provided with your Linux distribution, or which you used to build your own Linux setup. Any reasonably recent version of the tools, for example gcc 2.96(Red Hat) as shipped with Red Hat Linux 7, should be sufficient.

There is one important limitation when using these tools: current gdb will not support debugging of eCos threads on the synthetic target. As far as gdb is concerned a synthetic target application is indistinguishable from a normal Linux application, so it assumes that any threads will be created by calls to the Linux `pthread_create` function provided by the C library. Obviously this is not the case since the application is never linked with that library. Therefore gdb never notices the eCos thread mechanisms and assumes the application is single-threaded. Fixing this is possible but would involve non-trivial changes to gdb.

Theoretically it is possible to develop synthetic target applications on, for example, a PC running Windows and then run the resulting executables on another machine that runs Linux. This is rarely useful: if a Linux machine is available then usually that machine will also be used for building ecos and the application. However, if for some reason it is necessary or desirable to build on another machine then this requires a suitable cross-compiler and related tools. If the application will be running on a typical PC with an x86 processor then a suitable configure triplet would be **`i686-pc-linux-gnu`**. The installation instructions for the various GNU tools should be consulted for further information.

Hardware Preparation

Preparing a real embedded target for eCos development can be tricky. Often the first step is to install suitable firmware, usually RedBoot. This means creating and building a special configuration for eCos with the RedBoot template, then somehow updating the target's flash chips with the resulting RedBoot image. Typically it will also be necessary to get a working serial connection, and possibly set up ethernet as well. Although usually none of the individual steps are particularly complicated, there are plenty of ways in which things can go wrong and it can be hard to figure out what is actually happening. Of course some board manufacturers make life easier for their developers by shipping hardware with RedBoot preinstalled, but even then it is still necessary to set up communication between host and target.

None of this is applicable to the synthetic target. Instead you can just build a normal eCos configuration, link your application with the resulting libraries, and you end up with an executable that you can run directly on your Linux machine or via gdb. A useful side effect of this is that application development can start before any real embedded hardware is actually available.

Typically the memory map for a synthetic target application will be set up such that there is a read-only ROM region containing all the code and constant data, and a read-write RAM region for the data. The default locations and sizes of these regions depend on the specific platform being used for development. Note that the application always executes out of ROM: on a real embedded target much of the development would involve running RedBoot firmware there, with application code and data loaded into RAM; usually this would change for the final system; the firmware would be replaced by the eCos application itself, configured for ROM bootstrap, and it would perform the appropriate hardware initialization. Therefore the synthetic target actually emulates the behaviour of a final system, not of a development environment. In practice this is rarely significant, although having the code in read-only memory can help catch some problems in application code.

Name

Execution — Arguments and configuration files

Description

The procedure for configuring and building eCos and an application for the synthetic target is the same as for any other eCos target. Once an executable has been built it can be run like any Linux program, for example from a shell prompt,

```
$ ecos_hello <options>
```

or using gdb:

```
$ gdb --nw --quiet --args ecos_hello <options>
(gdb) run
Starting program: ecos_hello <options>
```

By default use of the I/O auxiliary is disabled. If its I/O facilities are required then the option `--io` must be used.

Note

In future the default behaviour may change, with the I/O auxiliary being started by default. The option `--nio` can be used to prevent the auxiliary from being run.

Command-line Arguments

The syntax for running a synthetic target application is:

```
$ <ecos_app> [options] [-- [app_options]]
```

Command line options up to the `--` are passed on to the I/O auxiliary. Subsequent arguments are not passed on to the auxiliary, and hence can be used by the eCos application itself. The full set of arguments can be accessed through the variables `cyg_hal_sys_argc` and `cyg_hal_sys_argv`.

The following options are accepted as standard:

`--io`
This option causes the eCos application to spawn the I/O auxiliary during HAL initialization. Without this option only limited I/O will be available.

`--nio`
This option prevents the eCos application from spawning the I/O auxiliary. In the current version of the software this is the default.

`-nw`, `--no-windows`
The I/O auxiliary can either provide a graphical user interface, or it can run in a text-only mode. The default is to provide the graphical interface, but this can be disabled with `-nw`. Emulation of some devices, for example buttons connected to digital inputs, requires the graphical interface.

`-w`, `--windows`
The `-w` causes the I/O auxiliary to provide a graphical user interface. This is the default.

`-v`, `--version`
The `-v` option can be used to determine the version of the I/O auxiliary being used and where it has been installed. Both the auxiliary and the eCos application will exit immediately.

`-h`, `--help`
`-h` causes the I/O auxiliary to list all accepted command-line arguments. This happens after all devices have been initialized, since the host-side support for some of the devices may extend the list of recognised options. After this both the auxiliary and the eCos application will exit immediately. This option implies `-nw`.

`-k`, `--keep-going`

If an error occurs in the I/O auxiliary while reading in any of the configuration files or initializing devices, by default both the auxiliary and the eCos application will exit. The `-k` option can be used to make the auxiliary continue in spite of errors, although obviously it may not be fully functional.

`-nr`, `--no-rc`

Normally the auxiliary processes two user configuration files during startup: `initrc.tcl` and `mainrc.tcl`. This can be suppressed using the `-nr` option.

`-x`, `--exit`

When providing a graphical user interface the I/O auxiliary will normally continue running even after the eCos application has exited. This allows the user to take actions such as saving the current contents of the main text window. If run with `-x` then the auxiliary will exit as soon the application exits.

`-nx`, `--no-exit`

When the graphical user interface is disabled with `-nw` the I/O auxiliary will normally exit immediately when the eCos application exits. Without the graphical frontend there is usually no way for the user to interact directly with the auxiliary, so there is no point in continuing to run once the eCos application will no longer request any I/O operations. Specifying the `-nx` option causes the auxiliary to continue running even after the application has exited.

`-V`, `--verbose`

This option causes the I/O auxiliary to output some additional information, especially during initialization.

`-l <file>`, `--logfile <file>`

Much of the output of the eCos application and the I/O auxiliary is simple text, for example resulting from eCos `printf` or `diag_printf` calls. When running in graphical mode this output goes to a central text window, and can be saved to a file or edited via menus. The `-l` can be used to automatically generate an additional logfile containing all the text. If graphical mode is disabled then by default all the text just goes to the current standard output. Specifying `-l` causes most of the text to go into a logfile instead, although some messages such as errors generated by the auxiliary itself will still go to stdout as well.

`-t <file>`, `--target <file>`

During initialization the I/O auxiliary reads in a target definition file. This file holds information such as which Linux devices should be used to emulate the various eCos devices. The `-t` option can be used to specify which target definition should be used for the current run, defaulting to `default.tdf`. It is not necessary to include the `.tdf` suffix, this will be appended automatically if necessary.

`-geometry <geometry>`

This option can be used to control the size and position of the main window, as per X conventions.

The I/O auxiliary loads support for the various devices dynamically and some devices may accept additional command line arguments. Details of these can be obtained using the `-h` option or by consulting the device-specific documentation. If an unrecognised command line argument is used then a warning will be issued.

The Target Definition File

The eCos application will want to access devices such as `eth0` or `/dev/ser0`. These need to be mapped on to Linux devices. For example some users may all traffic on the eCos `/dev/ser0` serial device to go via the Linux serial device `/dev/ttyS1`, while ethernet I/O for the eCos `eth0` device should be mapped to the Linux ethertap device `tap3`. Some devices may need additional configuration information, for example to limit the number of packets that should be buffered within the I/O auxiliary. The target definition file provides all this information.

By default the I/O auxiliary will look for a file `default.tdf`. An alternative target definition can be specified on the command line using `-t`, for example:

```
$ bridge_app --io -t twineth
```

A .tdf suffix will be appended automatically if necessary. If a relative pathname is used then the I/O auxiliary will search for the target definition file in the current directory, then in ~/.ecos/synth/, and finally in its install location.

A typical target definition file might look like this:

```
synth_device console {
    # appearance -foreground white -background black
    filter trace {^TRACE:.*} -foreground HotPink1 -hide 1
}

synth_device ethernet {
    eth0 real eth1
    eth1 ethertap tap4 00:01:02:03:FE:06

    ## Maximum number of packets that should be buffered per interface.
    ## Default 16
    #max_buffer 32

    ## Filters for the various recognised protocols.
    ## By default all filters are visible and use standard colours.
    filter ether  -hide 0
    #filter arp    -hide 1
    #filter ipv4   -hide 1
    #filter ipv6   -hide 1
}
```

A target definition file is actually a Tcl script that gets run in the main interpreter of the I/O auxiliary during initialization. This provides a lot of flexibility if necessary. For example the script could open a socket to a resource management server of some sort to determine which hardware facilities are already in use and adapt accordingly. Another possibility is to adapt based on command line arguments. Users who are not familiar with Tcl programming should still be able to edit a simple target definition file without too much difficulty, using a mixture of cut'n'paste, commenting or uncommenting various lines, and making small edits such as changing tap4 to eth2.

Each type of device will have its own entry in the target definition file, taking the form:

```
synth_device <device type> {
    <options>
}
```

The documentaton for each synthetic target device should provide details of the options available for that device, and often a suitable fragment that can be pasted into a target definition file and edited. There is no specific set of options that a given device will always provide. However in practice many devices will use common code exported by the main I/O auxiliary, or their implementation will involve some re-use of code for an existing device. Hence certain types of option are common to many devices.

A good example of this is filters, which control the appearance of text output. The above target definition file defines a filter trace for output from the eCos application. The regular expression will match output from the infrastructure package's tracing facilities when CYGDBG_USE_TRACING and CYGDBG_INFRA_DEBUG_TRACE_ASSERT_SIMPLE are enabled. With the current settings this output will not be visible by default, but can be made visible using the menu item System Filters. If made visible the trace output will appear in an unusual colour, so users can easily distinguish the trace output from other text. All filters accept the following options:

-hide [0|1]

This controls whether or not text matching this filter should be invisible by default or not. At run-time the visibility of each filter can be controlled using the System Filters menu item.

-foreground <colour>

This specifies the foreground colour for all text matching this filter. The colour can be specified using an RGB value such as #F08010, or a symbolic name such as "light steel blue". The X11 utility showrgb can be used to find out about the available colours.

-background <colour>

This specifies the background colour for all text matching the filter. As with -foreground the colour can be specified using a symbolic name or an RGB value.

Some devices may create their own subwindows, for example to monitor ethernet traffic or to provide additional I/O facilities such as emulated LED's or buttons. Usually the target definition file can be used to control the layout of these windows.

The I/O auxiliary will not normally warn about **synth_device** entries in the target definition file for devices that are not actually needed by the current eCos application. This makes it easier to use a single file for several different applications. However it can lead to confusion if an entry is spelled incorrectly and hence does not actually get used. The `-V` command line option can be used to get warnings about unused device entries in the target definition file.

If the body of a **synth_device** command contains an unrecognised option and the relevant device is in use, the I/O auxiliary will always issue a warning about such options.

User Configuration Files

During initialization the I/O auxiliary will execute two user configuration files, `initrc.tcl` and `mainrc.tcl`. It will look for these files in the directory `~/.ecos/synth/`. If that directory does not yet exist it will be created and populated with initial dummy files.

Both of these configuration files are Tcl scripts and will be run in the main interpreter used by the I/O auxiliary itself. This means that they have full access to the internals of the auxiliary including the various Tk widgets, and they can perform file or socket I/O if desired. The section Writing New Devices - host contains information about the facilities available on the host-side for writing new device drivers, and these can also be used in the initialization scripts.

The `initrc.tcl` script is run before the auxiliary has processed any requests from the eCos application, and hence before any devices have been instantiated. At this point the generic command-line arguments has been processed, the target definition file has been read in, and the hooks functionality has been initialized. If running in graphical mode the main window will have been created, but has been withdrawn from the screen to allow new widgets to be added without annoying screen flicker. A typical `initrc.tcl` script could add some menu or toolbar options, or install a hook function that will be run when the eCos application exits.

The `mainrc.tcl` script is run after eCos has performed all its device initialization and after C++ static constructors have run, and just before the call to `cyg_start` which will end up transferring control to the application itself. A typical `mainrc.tcl` script could look at what interrupt vectors have been allocated to which devices and create a little monitor window that shows interrupt activity.

Session Information

When running in graphical mode, the I/O auxiliary will read in a file `~/.ecos/synth/guisession` containing session information. This file should not normally be edited manually, instead it gets updated automatically when the auxiliary exits. The purpose of this file is to hold configuration options that are manipulated via the graphical interface, for example which browser should be used to display online help.

Warning

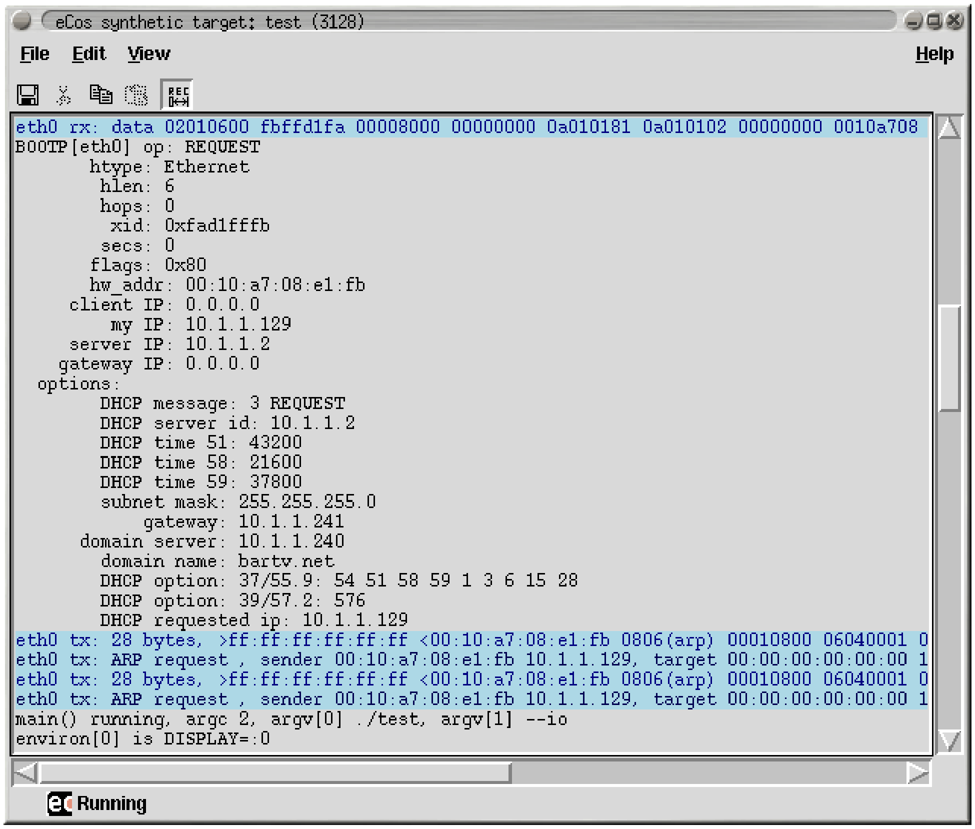
GUI session functionality is not yet available in the current release. When that functionality is fully implemented it is possible that some target definition file options may be removed, to be replaced by graphical editing via a suitable preferences dialog, with the current settings saved in the session file.

Name

User Interface — Controlling the I/O Auxiliary

Description

The synthetic target auxiliary is designed to support both extensions and user customization. Support for the desired devices is dynamically loaded, and each device can extend the user interface. For example it is possible for a device to add menu options, place new buttons on the toolbar, create its own sub-window within the overall layout, or even create entire new toplevel windows. These subwindows or toplevels could show graphs of activity such as interrupts or packets being transferred. They could also allow users to interact with the eCos application, for example by showing a number of buttons which will be mapped on to digital inputs in the eCos application. Different applications will have their own I/O requirements, changing the host-side support files that get loaded and that may modify the user interface. The I/O auxiliary also reads in user configuration scripts which can enhance the interface in the same way. Therefore the exact user interface will depend on the user and on the eCos application being run. However the overall layout is likely to remain the same.



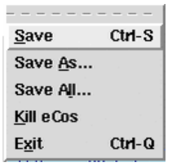
```
eth0 rx: data 02010600 fbffd1fa 00008000 00000000 0a010181 0a010102 00000000 0010a708
BOOTP[eth0] op: REQUEST
       htype: Ethernet
        hlen: 6
        hops: 0
         xid: 0xfad1fffb
        secs: 0
       flags: 0x80
       hw_addr: 00:10:a7:08:e1:fb
     client IP: 0.0.0.0
         my IP: 10.1.1.129
     server IP: 10.1.1.2
    gateway IP: 0.0.0.0
  options:
        DHCP message: 3 REQUEST
        DHCP server id: 10.1.1.2
        DHCP time 51: 43200
        DHCP time 58: 21600
        DHCP time 59: 37800
        subnet mask: 255.255.255.0
            gateway: 10.1.1.241
      domain server: 10.1.1.240
        domain name: bartv.net
        DHCP option: 37/55.9: 54 51 58 59 1 3 6 15 28
        DHCP option: 39/57.2: 576
        DHCP requested ip: 10.1.1.129
eth0 tx: 28 bytes, >ff:ff:ff:ff:ff:ff <00:10:a7:08:e1:fb 0806(arp) 00010800 06040001 0
eth0 tx: ARP request , sender 00:10:a7:08:e1:fb 10.1.1.129, target 00:00:00:00:00:00 1
eth0 tx: 28 bytes, >ff:ff:ff:ff:ff:ff <00:10:a7:08:e1:fb 0806(arp) 00010800 06040001 0
eth0 tx: ARP request , sender 00:10:a7:08:e1:fb 10.1.1.129, target 00:00:00:00:00:00 1
main() running, argc 2, argv[0] ./test, argv[1] --io
environ[0] is DISPLAY=:0
```

The title bar identifies the window as belonging to an eCos synthetic target application and lists both the application name and its process id. The latter is especially useful if the application was started directly from a shell prompt and the user now wants to attach a gdb session. The window has a conventional menu bar with the usual entries, plus a toolbar with buttons for common operations such as cut and paste. Balloon help is supported.

There is a central text window, possibly surrounded by various sub-windows for various devices. For example there could be a row of emulated LED's above the text window, and monitors of ethernet traffic and interrupt activity on the right. At the bottom of the window is a status line, including a small animation that shows whether or not the eCos application is still running.

Menus and the Toolbar

Usually there will be four menus on the menu bar: File , Edit, View and Help.

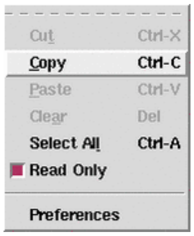


On the File menu there are three entries related to saving the current contents of the central text window. Save is used to save the currently visible contents of the text window. Any text that is hidden because of filters will not be written to the savefile. If there has been a previous Save or Save As operation then the existing savefile will be re-used, otherwise the user will be asked to select a suitable file. Save As also saves just the currently visible contents but will always prompt the user for a filename. Save All can be used to save the full contents of the text window, including any text that is currently hidden. It will always prompt for a new filename, to avoid confusion with partial savefiles.

Usually the eCos application will be run from inside gdb or from a shell prompt. Killing off the application while it is being debugged in a gdb session is not a good idea, it would be better to use gdb's own **kill** command. Alternatively the eCos application itself can use the `CYG_TEST_EXIT` or `cyg_hal_sys_exit` functionality. However it is possible to terminate the application from the I/O auxiliary using Kill eCos . A clean shutdown will be attempted, but that can fail if the application is currently halted inside gdb or if it has crashed completely. As a last resort `SIGKILL` will be used.

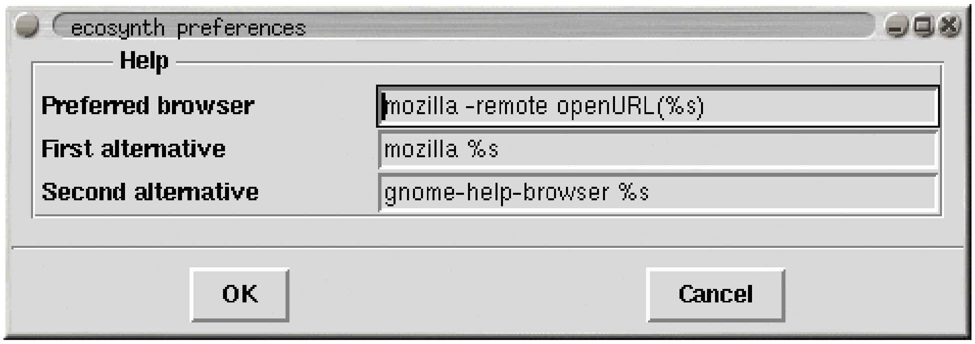
When operating in graphical mode the I/O auxiliary will normally continue to run even after the eCos application has exited. This allows the user to examine the last few lines of output, and perhaps perform actions such as saving the output to a file. The Exit menu item can be used to shut down the auxiliary. Note that this behaviour can be changed with command line arguments `--exit` and `--no-exit`.

If Exit is used while the eCos application is still running then the I/O auxiliary will first attempt to terminate the application cleanly, and then exit.



The Edit menu contains the usual entries for text manipulation: Cut, Copy , Paste, Clear and Select All. These all operate on the central text window. By default this window cannot be edited so the cut, paste and clear operations are disabled. If the user wants to edit the contents of the text window then the Read Only checkbutton should be toggled.

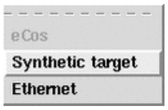
The Preferences menu item brings up a miscellaneous preferences dialog. One of the preferences relates to online help: the I/O auxiliary does not currently have a built-in html viewer; instead it will execute an external browser of some sort. With the example settings shown, the I/O auxiliary will first attempt to interact with an existing mozilla session. If that fails it will try to run a new mozilla instance, or as a last result use the Gnome help viewer.



Preferred browser: mozilla -remote openURL(%s)
First alternative: mozilla %s
Second alternative: gnome-help-browser %s

The View menu contains the System Filters entry, used to edit the settings for the current filters.

The Help menu can be used to activate online help for eCos generally, for the synthetic target as a whole, and for specific devices supported by the generic target. The Preferences dialog can be used to select the browser that will be used.



Note

At the time of writing there is no well-defined toplevel index file for all eCos documentation. Hence the relevant menu item is disabled. Documentation for the synthetic target and the supported devices is stored as part of the package itself so can usually be found fairly easily. It may be necessary to set the `ECOS_REPOSITORY` environment variable.

The Main Text Window

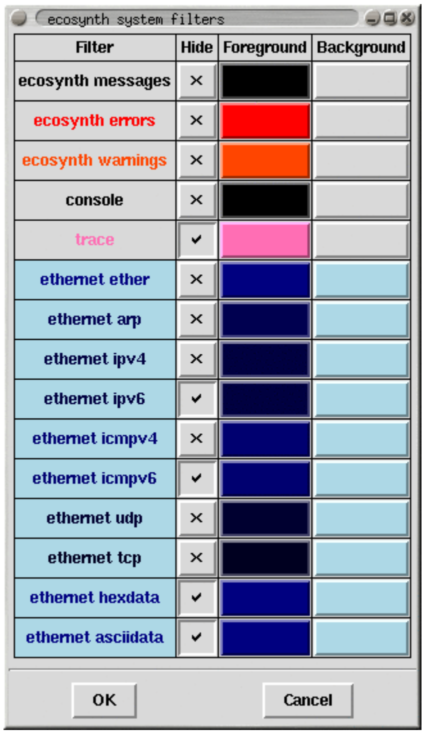
The central text window holds the console output from the eCos application: the screen shot above shows DHCP initialization data from the TCP/IP stack, and some output from the `main` thread at the bottom. Some devices can insert text of their own, for example the ethernet device support can be configured to show details of incoming and outgoing packets. Mixing the output from the eCos application and the various devices can make it easier to understand the order in which events occur.

The appearance of text from different sources can be controlled by means of filters, and it is also possible to hide some of the text. For example, if tracing is enabled in the eCos configuration then the trace output can be given its own colour scheme, making it stand out from the rest of the output. In addition the trace output is generally voluminous so it can be hidden by default, made visible only to find out more about what was happening when a particular problem occurred. Similarly the ethernet device support can output details of the various packets being transferred, and using a different background colour for this output again makes it easier to distinguish from console output.

The default appearance for most filters is controlled via the target definition file. An example entry might be:

```
filter trace {^TRACE:.*} -foreground HotPink1 -hide 1
```

The various colours and the hide flag for each filter can be changed at run-time, using the System Filters item on the View menu. This will bring up a dialog like the following:



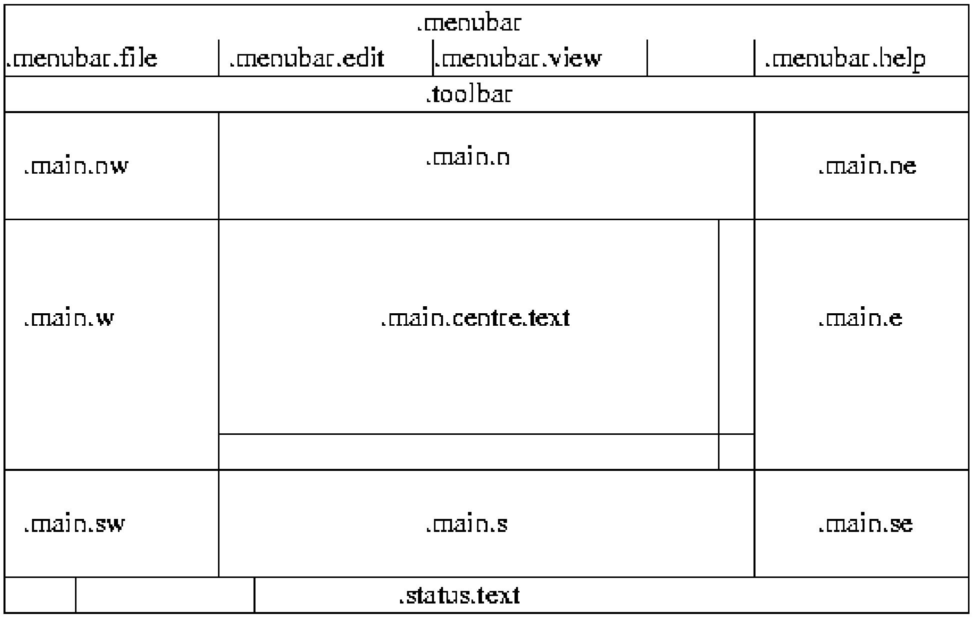
Filter	Hide	Foreground	Background
ecosynth messages	✕	⬛	⬜
ecosynth errors	✕	🟥	⬜
ecosynth warnings	✕	🟧	⬜
console	✕	⬛	⬜
trace	✓	🟪	⬜
ethernet ether	✕	🟦	🟦
ethernet arp	✕	🟦	🟦
ethernet ipv4	✕	⬛	🟦
ethernet ipv6	✓	⬛	🟦
ethernet icmpv4	✕	🟦	🟦
ethernet icmpv6	✓	🟦	🟦
ethernet udp	✕	⬛	🟦
ethernet tcp	✕	⬛	🟦
ethernet hexdata	✓	🟦	🟦
ethernet asciidata	✓	🟦	🟦

It should be noted that the text window is line-oriented, not character-oriented. If an eCos application sends a partial line of text then that will remain buffered until a newline character is received, rather than being displayed immediately. This avoids confusion when there is concurrent output from several sources.

By default the text window is read-only. This means it will not allow cut, paste and clear operations, and keyboard input will be ignored. The Edit menu has a checkbutton Read Only which can be toggled to allow write operations. For example, a user could type in a reminder of what was happening at this time, or paste in part of a gdb session. Such keyboard input does not get forwarded to the eCos application: if the latter requires keyboard input then that should happen via a separate keyboard device.

Positioning Optional Windows

Some devices may create their own subwindows, for example to monitor ethernet traffic or to provide additional I/O facilities such as emulated LED's or buttons. Usually the target definition file can be used to control the layout of these windows. This requires an understanding of the overall layout of the display.



Subwindows are generally packed in one of eight frames surrounding the central text window: `.main.nw`, `.main.n`, `.main.ne`, `.main.w`, `.main.e`, `.main.sw`, `.main.s`, and `.main.se`. To position a row of LED's above the text window and towards the left, a target definition file could contain an entry such as:

```
synth_device led {
    pack -in .main.n -side left
    …
}
```

Similarly, to put a traffic monitor window on the right of the text window would involve something like:

```
    …
    monitor_pack -in .main.e -side bottom
    …
```

Often it will be sufficient to specify a container frame and one of `left`, `right`, `top` or `bottom`. Full control over the positioning requires an understanding of Tcl/Tk and in particular the packing algorithm, and an appropriate reference work should be consulted.

Global Settings

Note

This section still to be written - it should document the interaction between X resources and ecosynth, and how users can control settings such as the main foreground and background colours.

Name

The console device — Show output from the eCos application

Description

The eCos application can generate text output in a variety of ways, including calling `printf` or `diag_printf`. When the I/O auxiliary is enabled the eCos startup code will instantiate a console device to process all such output. If operating in text mode the output will simply go to standard output, or to a logfile if the `-l` command line option is specified. If operating in graphical mode the output will go to the central text window, and optionally to a logfile as well. In addition it is possible to control the appearance of the main text via the target definition file, and to install extra filters for certain types of text.

It should be noted that the console device is line-oriented, not character-oriented. This means that outputting partial lines is not supported, and some functions such as `fflush` and `setvbuf` will not operate as expected. This limitation prevents much possible confusion when using filters to control the appearance of the text window, and has some performance benefits - especially when the eCos application generates a great deal of output such as when tracing is enabled. For most applications this is not a problem, but it is something that developers should be aware of.

The console device is output-only, it does not provide any support for keyboard input. If the application requires keyboard input then that should be handled by a separate eCos device package and matching host-side code.

Installation

The eCos side of the console device is implemented by the architectural HAL itself, in the source file `synth_diag.c`, rather than in a separate device package. Similarly the host-side implementation, `console.tcl`, is part of the architectural HAL's host-side support. It gets installed automatically alongside the I/O auxiliary itself, so no separate installation procedure is required.

Target Definition File

The target definition file can contain a number of entries related to the console device. These are all optional, they only control the appearance of text output. If such control is desired then the relevant options should appear in the body of a **synth_device** entry:

```
synth_device console {
    …
}
```

The first option is **appearance**, used to control the appearance of any text generated by the eCos application that does not match one of the installed filters. This option takes the same argument as any other filter, for example:

```
synth_device console {
    appearance -foreground white -background black
    …
}
```

Any number of additional filters can be created with a **filter** option, for example:

```
synth_device console {
    …
    filter trace {^TRACE:.*} -foreground HotPink1 -hide 1
    …
}
```

The first argument gives the new filter a name which will be used in the filters dialog. Filter names should be unique. The second argument is a Tcl regular expression. The console support will match each line of eCos output against this regular expression, and if a match is found then the filter will be used for this line of text. The above example matches any line of output that begins with `TRACE:`, which corresponds to the eCos infrastructure's tracing facilities. The remaining options control the desired appearance for matched text. If some eCos output matches the regular expressions for several different filters then only the first match will be used.

Target-side Configuration Options

There are no target-side configuration options related to the console device.

Command Line Arguments

The console device does not use any command-line arguments.

Hooks

The console device does not provide any hooks.

Additional Tcl Procedures

The console device does not provide any additional Tcl procedures that can be used by other scripts.

Name

cyg_hal_sys_xyz — Access Linux system facilities

Synopsis

#include <cyg/hal/hal_io.h>

```
int cyg_hal_sys_xyzzy(...);
```

Description

On a real embedded target eCos interacts with the hardware by peeking and poking various registers, manipulating special regions of memory, and so on. The synthetic target does not access hardware directly. Instead I/O and other operations are emulated by making appropriate Linux system calls. The HAL package exports a number of functions which allow other packages, or even application code, to make these same system calls. However this facility must be used with care: any code which calls, for example, `cyg_hal_sys_write` will only ever run on the synthetic target; that functionality is obviously not provided on any real hardware because there is no underlying Linux kernel to implement it.

The synthetic target only provides a subset of the available system calls, specifically those calls which have proved useful to implement I/O emulation. This subset can be extended fairly easily if necessary. All of the available calls, plus associated data structures and macros, are defined in the header file `cyg/hal/hal_io.h`. There is a simple convention: given a Linux system call such as `open`, the synthetic target will prefix `cyg_hal_sys` and provide a function with that name. The second argument to the `open` system call is a set of flags such as `O_RDONLY`, and the header file will define a matching constant `CYG_HAL_SYS_O_RDONLY`. There are also data structures such as cyg_hal_sys_sigset_t, matching the Linux data structure sigset_t.

In most cases the functions provided by the synthetic target behave as per the documentation for the Linux system calls, and section 2 of the Linux man pages can be consulted for more information. There is one important difference: typically the documentation will say that a function returns `-1` to indicate an error, with the actual error code held in `errno`; the actual underlying system call and hence the `cyg_hal_sys_xyz` provided by eCos instead returns a negative number to indicate an error, with the absolute value of that number corresponding to the error code; usually it is the C library which handles this and manipulates errno, but of course synthetic target applications are not linked with that Linux library.

However, there are some exceptions. The Linux kernel has evolved over the years, and some of the original system call interfaces are no longer appropriate. For example the original `select` system call has been superseded by `_newselect`, and that is what the `select` function in the C library actually uses. The old call is still available to preserve binary compatibility but, like the C library, eCos makes use of the new one because it provides the appropriate functionality. In an attempt to reduce confusion the eCos function is called `cyg_hal_sys__newselect`, in other words it matches the official system call naming scheme. The authoritive source of information on such matters is the Linux kernel sources themselves, and especially its header files.

eCos packages and applications should never `#include` Linux header files directly. For example, doing a `#include </usr/include/fcntl.h>` to access additional macros or structure definitions, or alternatively manipulating the header file search path, will lead to problems because the Linux header files are likely to duplicate and clash with definitions in the eCos headers. Instead the appropriate functionality should be extracted from the Linux headers and moved into either `cyg/hal/hal_io.h` or into application code, with suitable renaming to avoid clashes with eCos names. Users should be aware that large-scale copying may involve licensing complications.

Adding more system calls is usually straightforward and involves adding one or more lines to the platform-specific file in the appropriate platform HAL, for example `syscall-i386-linux-1.0.S`. However it is necessary to do some research first about the exact interface implemented by the system call, because of issues such as old system calls that have been superseded. The required information can usually be found fairly easily by searching through the Linux kernel sources and possibly the GNU C library sources.

Name

Writing New Devices — extending the synthetic target, target-side

Synopsis

#include <cyg/hal/hal_io.h>

`int` **`synth_auxiliary_instantiate`**`(`*`package`*`, `*`version`*`, `*`device`*`, `*`instance`*`, `*`data`*`);`

`void` **`synth_auxiliary_xchgmsg`**`(`*`device_id`*`, `*`request`*`, `*`arg1`*`, `*`arg2`*`, `*`txdata`*`, `*`txlen`*`, `*`reply`*`, `*`rxdata`*`, `*`rxlen`*`, `*`max_rxlen`*`);`

Description

In some ways writing a device driver for the synthetic target is very similar to writing one for a real target. Obviously it has to provide the standard interface for that class of device, so for example an ethernet device has to provide `can_send`, `send`, `recv` and similar functions. Many devices will involve interrupts, so the driver contains ISR and DSR functions and will call `cyg_drv_interrupt_create`, `cyg_drv_interrupt_acknowledge`, and related functions.

In other ways writing a device driver for the synthetic target is very different. Usually the driver will not have any direct access to the underlying hardware. In fact for some devices the I/O may not involve real hardware, instead everything is emulated by widgets on the graphical display. Therefore the driver cannot just peek and poke device registers, instead it must interact with host-side code by exchanging message. The synthetic target HAL provides a function `synth_auxiliary_xchgmsg` for this purpose.

Initialization of a synthetic target device driver is also very different. On real targets the device hardware already exists when the driver's initialization routine runs. On the synthetic target it is first necessary to instantiate the device inside the I/O auxiliary, by a call to `synth_auxiliary_instantiate`. That function performs a special message exchange with the I/O auxiliary, causing it to load a Tcl script for the desired type of device and run an instantiation procedure within that script.

Use of the I/O auxiliary is optional: if the user does not specify `--io` on the command line then the auxiliary will not be started and hence most I/O operations will not be possible. Device drivers should allow for this possibility, for example by just discarding any data that gets written. The HAL exports a flag `synth_auxiliary_running` which should be checked.

Instantiating a Device

Device instantiation should happen during the C++ prioritized static constructor phase of system initialization, before control switches to `cyg_user_start` and general application code. This ensures that there is a clearly defined point at which the I/O auxiliary knows that all required devices have been loaded. It can then perform various consistency checks and clean-ups, run the user's `mainrc.tcl` script, and make the main window visible.

For standard devices generic eCos I/O code will call the device initialization routines at the right time, iterating through the `DEVTAB` table in a static constructor. The same holds for network devices and file systems. For more custom devices code like the following can be used:

```
#include <cyg/infra/cyg_type.h>
class mydev_init {
  public:
    mydev_init() {
        …
    }
};
static mydev_init mydev_init_object CYGBLD_ATTRIB_INIT_PRI(CYG_INIT_IO);
```

Some care has to be taken because the object `mydev_init_object` will typically not be referenced by other code, and hence may get eliminated at link-time. If the code is part of an eCos package then problems can be avoided by putting the relevant file in `libextras.a`:

```
cdl_package CYGPKG_DEVS_MINE {
    …
    compile -library=libextras.a init.cxx
}
```

For devices inside application code the same can be achieved by linking the relevant module as a `.o` file rather than putting it in a `.a` library.

In the device initialization routine the main operation is a call to `synth_auxiliary_instantiate`. This takes five arguments, all of which should be strings:

`package`
: For device drivers which are eCos packages this should be a directory path relative to the eCos repository, for example `devs/eth/synth/ecosynth`. This will allow the I/O auxiliary to find the various host-side support files for this package within the install tree. If the device is application-specific and not part of an eCos package then a NULL pointer can be used, causing the I/O auxiliary to search for the support files in the current directory and then in `~/.ecos/synth` instead.

`version`
: For eCos packages this argument should be the version of the package that is being used, for example `current`. A simple way to get this version is to use the `SYNTH_MAKESTRING` macro on the package name. If the device is application-specific then a NULL pointer should be used.

`device`
: This argument specifies the type of device being instantiated, for example `ethernet`. More specifically the I/O auxiliary will append a `.tcl` suffix, giving the name of a Tcl script that will handle all I/O requests for the device. If the application requires several instances of a type of device then the script will only be loaded once, but the script will contain an instantiation procedure that will be called for each device instance.

`instance`
: If it is possible to have multiple instances of a device then this argument identifies the particular instance, for example `eth0` or `eth1`. Otherwise a NULL pointer can be used.

`data`
: This argument can be used to pass additional initialization data from eCos to the host-side support. This is useful for devices where eCos configury must control certain aspects of the device, rather than host-side configury such as the target definition file, because eCos has compile-time dependencies on some or all of the relevant options. An example might be an emulated frame buffer where eCos has been statically configured for a particular screen size, orientation and depth. There is no fixed format for this string, it will be interpreted only by the device-specific host-side Tcl script. However the string length should be limited to a couple of hundred bytes to avoid possible buffer overflow problems.

Typical usage would look like:

```
    if (!synth_auxiliary_running) {
      return;
    }
    id = synth_auxiliary_instantiate("devs/eth/synth/ecosynth",
             SYNTH_MAKESTRING(CYGPKG_DEVS_ETH_ECOSYNTH),
             "ethernet",
             "eth0",
             (const char*) 0);
```

The return value will be a device identifier which can be used for subsequent calls to `synth_auxiliary_xchgmsg`. If the device could not be instantiated then `-1` will be returned. It is the responsibility of the host-side software to issue suitable diagnostics explaining what went wrong, so normally the target-side code should fail silently.

Once the desired device has been instantiated, often it will be necessary to do some additional initialization by a message exchange. For example an ethernet device might need information from the host-side about the MAC address, the interrupt vector, and whether or not multicasting is supported.

Communicating with a Device

Once a device has been instantiated it is possible to perform I/O by sending messages to the appropriate Tcl script running inside the auxiliary, and optionally getting back replies. I/O operations are always initiated by the eCos target-side, it is not possible for the host-side software to initiate data transfers. However the host-side can raise interrupts, and the interrupt handler inside the target can then exchange one or more messages with the host.

There is a single function to perform I/O operations, `synth_auxiliary_xchgmsg`. This takes the following arguments:

`device_id`
: This should be one of the identifiers returned by a previous call to `synth_auxiliary_instantiate`, specifying the particular device which should perform some I/O.

`request`
: Request are just signed 32-bit integers that identify the particular I/O operation being requested. There is no fixed set of codes, instead each type of device can define its own.

`arg1`
`arg2`
: For some requests it is convenient to pass one or two additional parameters alongside the request code. For example an ethernet device could define a multicast-all request, with `arg1` controlling whether this mode should be enabled or disabled. Both `arg1` and `arg2` should be signed 32-bit integers, and their values are interpreted only by the device-specific Tcl script.

`txdata`
`txlen`
: Some I/O operations may involve sending additional data, for example an ethernet packet. Alternatively a control operation may require many more parameters than can easily be encoded in `arg1` and `arg2`, so those parameters have to be placed in a suitable buffer and extracted at the other end. `txdata` is an arbitrary buffer of `txlen` bytes that should be sent to the host-side. There is no specific upper bound on the number of bytes that can be sent, but usually it is a good idea to allocate the transmit buffer statically and keep transfers down to at most several kilobytes.

`reply`
: If the host-side is expected to send a reply message then `reply` should be a pointer to an integer variable and will be updated with a reply code, a simple 32-bit integer. The synthetic target HAL code assumes that the host-side and target-side agree on the protocol being used: if the host-side will not send a reply to this message then the `reply` argument should be a NULL pointer; otherwise the host-side must always send a reply code and the `reply` argument must be valid.

`rxdata`
`rxlen`
: Some operations may involve additional data coming from the host-side, for example an incoming ethernet packet. `rxdata` should be a suitably-sized buffer, and `rxlen` a pointer to an integer variable that will end up containing the number of bytes that were actually received. These arguments will only be used if the host-side is expected to send a reply and hence the `reply` argument was not NULL.

`max_rxlen`
: If a reply to this message is expected and that reply may involve additional data, `max_rxlen` limits the size of that reply. In other words, it corresponds to the size of the `rxdata` buffer.

Most I/O operations involve only some of the arguments. For example transmitting an ethernet packet would use the `request`, `txdata` and `txlen` fields (in addition to `device_id` which is always required), but would not involve `arg1` or `arg2` and no reply would be expected. Receiving an ethernet packet would involve `request`, `rxdata`, `rxlen` and `max_rxlen`; in addition `reply` is needed to get any reply from the host-side at all, and could be used to indicate whether or not any more packets are buffered up. A control operation such as enabling multicast mode would involve `request` and `arg1`, but none of the remaining arguments.

Interrupt Handling

Interrupt handling in the synthetic target is much the same as on a real target. An interrupt object is created using cyg_drv_interrupt_create, attached, and unmasked. The emulated device - in other words the Tcl script running inside the I/O auxiliary - can raise an interrupt. Subject to interrupts being disabled and the appropriate vector being masked, the system will invoke the specified ISR function. The synthetic target HAL implementation does have some limitations: there is no support for nested interrupts, interrupt priorities, or a separate interrupt stack. Supporting those might be appropriate when targetting a simulator that attempts to model real hardware accurately, but not for the simple emulation provided by the synthetic target.

Of course the actual implementation of the ISR and DSR functions will be rather different for a synthetic target device driver. For real hardware the device driver will interact with the device by reading and writing device registers, managing DMA engines, and the like. A synthetic target driver will instead call synth_auxiliary_xchgmsg to perform the I/O operations.

There is one other significant difference between interrupt handling on the synthetic target and on real hardware. Usually the eCos code will know which interrupt vectors are used for which devices. That information is fixed when the target hardware is designed. With the synthetic target interrupt vectors are assigned to devices on the host side, either via the target definition file or dynamically when the device is instantiated. Therefore the initialization code for a target-side device driver will need to request interrupt vector information from the host-side, via a message exchange. Such interrupt vectors will be in the range 1 to 31 inclusive, with interrupt 0 being reserved for the real-time clock.

Name

Writing New Devices — extending the synthetic target, host-side

Description

On the host-side adding a new device means writing a Tcl/Tk script that will handle instantiation and subsequent requests from the target-side. These scripts all run in the same full interpreter, extended with various commands provided by the main I/O auxiliary code, and running in an overall GUI framework. Some knowledge of programming with Tcl/Tk is required to implement host-side device support.

Some devices can be implemented entirely using a Tcl/Tk script. For example, if the final system will have some buttons then those can be emulated in the synthetic target using a few Tk widgets. A simple emulation could just have the right number of buttons in a row. A more advanced emulation could organize the buttons with the right layout, perhaps even matching the colour scheme, the shapes, and the relative sizes. With other devices it may be necessary for the Tcl script to interact with an external program, because the required functionality cannot easily be accessed from a Tcl script. For example interacting with a raw ethernet device involves some `ioctl` calls, which is easier to do in a C program. Therefore the `ethernet.tcl` script which implements the host-side ethernet support spawns a separate program `rawether`, written in C, that performs the low-level I/O. Raw ethernet access usually also requires root privileges, and running a small program `rawether` with such privileges is somewhat less of a security risk than the whole eCos application, the I/O auxiliary, and various dynamically loaded Tcl scripts.

Because all scripts run in a single interpreter, some care has to be taken to avoid accidental sharing of global variables. The best way to avoid problems is to have each script create its own Tcl namespace, so for example the `ethernet.tcl` script creates a namespace `ethernet::` and all variables and procedures reside in this namespace. Similarly the I/O auxiliary itself makes use of a `synth::` namespace.

Building and Installation

When an eCos device driver or application code instantiates a device, the I/O auxiliary will attempt to load a matching Tcl script. The third argument to `synth_auxiliary_instantiate` specifies the type of device, for example `ethernet`, and the I/O auxiliary will append a `.tcl` suffix and look for a script `ethernet.tcl`.

If the device being instantiated is application-specific rather than part of an eCos package, the I/O auxiliary will look first in the current directory, then in `~/.ecos/synth`. If it is part of an eCos package then the auxiliary will expect to find the Tcl script and any support files below `libexec/ecos` in the install tree - note that the same install tree must be used for the I/O auxiliary itself and for any device driver support. The directory hierarchy below `libexec/ecos` matches the structure of the eCos repository, allowing multiple versions of a package to be installed to allow for incompatible protocol changes.

The preferred way to build host-side software is to use **autoconf** and **automake**. Usually this involves little more than copying the `acinclude.m4`, `configure.in` and `Makefile.am` files from an existing package, for example the synthetic target ethernet driver, and then making minor edits. In `acinclude.m4` it may be necessary to adjust the path to the root of the repository. `configure.in` may require a similar change, and the `AC_INIT` macro invocation will have to be changed to match one of the files in the new package. A critical macro in this file is `ECOS_PACKAGE_DIRS` which will set up the correct install directory. `Makefile.am` may require some more changes, for example to specify the data files that should be installed (including the Tcl script). These files should then be processed using **aclocal**, **autoconf** and **automake** in that order. Actually building the software then just involves **configure**, **make** and **make install**, as per the instructions in the toplevel `README.host` file.

To assist developers, if the environment variable `ECOSYNTH_DEVEL` is set then a slightly different algorithm is used for locating device Tcl scripts. Instead of looking only in the install tree the I/O auxiliary will also look in the source tree, and if the script there is more recent than the installed version it will be used in preference. This allows developers to modify the master copy without having to run **make install** all the time.

If a script needs to know where it has been installed it can examine the Tcl variable `synth::device_install_dir` . This variable gets updated whenever a script is loaded, so if the value may be needed later it should be saved away in a device-specific variable.

Instantiation

The I/O auxiliary will **source** the device-specific Tcl script when the eCos application first attempts to instantiate a device of that type. The script should return a procedure that will be invoked to instantiate a device.

```
namespace eval ethernet {
    …
    proc instantiate { id instance data } {
        …
        return ethernet::handle_request
    }
}
return ethernet::instantiate
```

The `id` argument is a unique identifier for this device instance. It will also be supplied on subsequent calls to the request handler, and will match the return value of `synth_auxiliary_instantiate` on the target side. A common use for this value is as an array index to support multiple instances of this types of device. The `instance` and `data` arguments match the corresponding arguments to `synth_auxiliary_instantiate` on the target side, so a typical value for `instance` would be `eth0`, and `data` is used to pass arbitrary initialization parameters from target to host.

The actual work done by the instantiation procedure is obviously device-specific. It may involve allocating an interrupt vector, adding a device-specific subwindow to the display, opening a real Linux device, establishing a socket connection to some server, spawning a separate process to handle the actual I/O, or a combination of some or all of the above.

If the device is successfully instantiated then the return value should be a handler for subsequent I/O requests. Otherwise the return value should be an empty string, and on the target-side the `synth_auxiliary_instantiate` call will return `-1`. The script is responsible for providing diagnostics explaining why the device could not be instantiated.

Handling Requests

When the target-side calls `synth_auxiliary_xchgmsg`, the I/O auxiliary will end up calling the request handler for the appropriate device instance returned during instantiation:

```
namespace eval ethernet {
    …
    proc handle_request { id request arg1 arg2 txdata txlen max_rxlen } {
        …
        if { <some condition> } {
            synth::send_reply <error code> 0 ""
            return
        }
        …
        synth::send_reply <reply code> $packet_len $packet
    }
    …
}
```

The `id` argument is the same device id that was passed to the instantiate function, and is typically used as an array index to access per-device data. The `request`, `arg1`, `arg2`, and `max_rxlen` are the same values that were passed to `synth_auxiliary_xchgmsg` on the target-side, although since this is a Tcl script obviously the numbers have been converted to strings. The `txdata` buffer is raw data as transmitted by the target, or an empty string if the I/O operation does not involve any additional data. The Tcl procedures **binary scan**, **string index** and **string range** may be found especially useful when manipulating this buffer. `txlen` is provided for convenience, although **string length $txdata** would give the same information.

The code for actually processing the request is of course device specific. If the target does not expect a reply then the request handler should just return when finished. If a reply is expected then there should be a call to **synth::send_reply**. The first argument is the reply code, and will be turned into a 32-bit integer on the target side. The second argument specifies the length of the reply data, and the third argument is the reply data itself. For some devices the Tcl procedure **binary format** may prove useful. If the reply involves just a code and no additional data, the second and third arguments should be `0` and an empty string respectively.

Attempts to send a reply when none is expected, fail to send a reply when one is expected, or send a reply that is larger than the target-side expects, will all be detected by the I/O auxiliary and result in run-time error messages.

It is not possible for the host-side code to send unsolicited messages to the target. If host-side code needs attention from the target, for example because some I/O operation has completed, then an interrupt should be raised.

Interrupts

The I/O auxiliary provides a number of procedures for interrupt handling.

```
synth::interrupt_allocate <name>
synth::interrupt_get_max
synth::interrupt_get_devicename <vector>
synth::interrupt_raise <vector>
```

synth::interrupt_allocate is normally called during device instantiation, and returns the next free interrupt vector. This can be passed on to the target-side device driver in response to a suitable request, and it can then install an interrupt handler on that vector. Interrupt vector `0` is used within the target-side code for the real-time clock, so the allocated vectors will start at `1`. The argument identifies the device, for example `eth0`. This is not actually used internally, but can be accessed by user-initialization scripts that provide some sort of interrupt monitoring facility (typically via the `interrupt` hook). It is possible for a single device to allocate multiple interrupt vectors, but the synthetic target supports a maximum of 32 such vectors.

synth::interrupt_get_max returns the highest interrupt vector that has been allocated, or `0` if there have been no calls to **synth::interrupt_allocate**. **synth::interrupt_get_devicename** returns the string that was passed to **synth::interrupt_allocate** when the vector was allocated.

synth::interrupt_raise can be called any time after initialization. The argument should be the vector returned by **synth::interrupt_allocate** for this device. It will activate the normal eCos interrupt handling mechanism so, subject to interrupts being enabled and this particular interrupt not being masked out, the appropriate ISR will run.

Note

At this time it is not possible for a device to allocate a specific interrupt vector. The order in which interrupt vectors are assigned to devices effectively depends on the order in which the eCos devices get initialized, and that may change if the eCos application is rebuilt. A future extension may allow devices to allocate specific vectors, thus making things more deterministic. However that will introduce new problems, in particular the code will have to start worrying about requests for vectors that have already been allocated.

Flags and Command Line Arguments

The generic I/O auxiliary code will process the standard command line arguments, and will set various flag variables accordingly. Some of these should be checked by device-specific scripts.

`synth::flag_gui`

This is set when the I/O auxiliary is operating in graphical mode rather than text mode. Some functionality such as filters and the GUI layout are only available in graphical mode.

```
    if { $synth::flag_gui } {
        …
    }
```

`synth::flag_verbose`

The user has requested additional information during startup. Each device driver can decide how much additional information, if any, should be produced.

`synth::flag_keep_going`

The user has specified `-k` or `--keep-going`, so even if an error occurs the I/O auxiliary and the various device driver scripts should continue running if at all possible. Diagnostics should still be generated.

Some scripts may want to support additional command line arguments. This facility should be used with care since there is no way to prevent two different scripts from trying to use the same argument. The following Tcl procedures are available:

```
synth::argv_defined <name>
synth::argv_get_value <name>
```

synth::argv_defined returns a boolean to indicate whether or not a particular argument is present. If the argument is the name part of a name/value pair, an = character should be appended. Typical uses might be:

```
    if { [synth::argv_defined "-o13"] } {
        …
    }

    if { [synth::argv_defined "-mark="] } {
        …
    }
```

The first call checks for a flag `-o13` or `--o13` - the code treats options with single and double hyphens interchangeably. The second call checks for an argument of the form `-mark=<value>` or a pair of arguments `-mark <value>`. The value part of a name/value pair can be obtained using **synth::argv_get_value**;

```
    variable speed 1
    if { [synth::argv_defined "-mark="] } {
        set mark [synth::argv_get_value "-mark="]
        if { ![string is integer $mark] || ($mark < 1) || ($mark > 9) } {
            <issue diagnostic>
        } else {
            set speed $mark
        }
    }
```

synth::argv_get_value should only be used after a successful call to **synth::argv_defined**. At present there is no support for some advanced forms of command line argument processing. For example it is not possible to repeat a certain option such as `-v` or `--verbose`, with each occurrence increasing the level of verbosity.

If a script is going to have its own set of command-line arguments then it should give appropriate details if the user specifies `--help`. This involves a hook function:

```
namespace eval my_device {
    proc help_hook { } {
        puts " -o13          : activate the omega 13 device"
        puts " -mark <speed> : set speed. Valid values are 1 to 9."
    }

    synth::hook_add "help" my_device::help_hook
}
```

The Target Definition File

Most device scripts will want to check entries in the target definition file for run-time configuration information. The Tcl procedures for this are as follows:

```
synth::tdf_has_device <name>
synth::tdf_get_devices
synth::tdf_has_option <devname> <option>
synth::tdf_get_option <devname> <option>
synth::tdf_get_options <devname> <option>
synth::tdf_get_all_options <devname>
```

synth::tdf_has_device can be used to check whether or not the target definition file had an entry `synth_device <name>`. Usually the name will match the type of device, so the `console.tcl` script will look for a target definition file entry `console`. **synth::tdf_get_devices** returns a list of all device entries in the target definition file.

Once it is known that the target definition file has an entry for a certain device, it is possible to check for options within the entry. **synth::tdf_has_option** just checks for the presence, returning a boolean:

```
    if { [synth::tdf_has_option "console" "appearance"] } {
        …
    }
```

synth::tdf_get_option returns a list of all the arguments for a given option. For example, if the target definition file contains an entry:

```
synth_device console {
    appearance -foreground white -background black
    filter trace {^TRACE:.*} -foreground HotPink1 -hide 1
    filter xyzzy {.*xyzzy.*} -foreground PapayaWhip
}
```

A call **synth::tdf_get_option console appearance** will return the list `{-foreground white -background black}`. This list can be manipulated using standard Tcl routines such as **llength** and **lindex**. Some options can occur multiple times in one entry, for example `filter` in the `console` entry. **synth::tdf_get_options** returns a list of lists, with one entry for each option occurrence. **synth::tdf_get_all_options** returns a list of lists of all options. This time each entry will include the option name as well.

The I/O auxiliary will not issue warnings about entries in the target definition file for devices which were not loaded, unless the `-v` or `--verbose` command line argument was used. This makes it easier to use a single target definition file for different applications. However the auxiliary will issue warnings about options within an entry that were ignored, because often these indicate a typing mistake of some sort. Hence a script should always call **synth::tdf_has_option**, **synth::;tdf_get_option** or **synth::tdf_get_options** for all valid options, even if some of the options preclude the use of others.

Hooks

Some scripts may want to take action when particular events occur, for example when the eCos application has exited and there is no need for further I/O. This is supported using hooks:

```
namespace eval my_device {
    …
    proc handle_ecos_exit { arg_list } {
        …
    }
    synth::hook_add "ecos_exit" my_device::handle_ecos_exit
}
```

It is possible for device scripts to add their own hooks and call all functions registered for those hooks. A typical use for this is by user initialization scripts that want to monitor some types of I/O. The available Tcl procedures for manipulating hooks are:

```
synth::hook_define <name>
synth::hook_defined <name>
synth::hook_add <name> <function>
synth::hook_call <name> <args>
```

synth::hook_define creates a new hook with the specified name. This hook must not already exist. **synth::hook_defined** can be used to check for the existence of a hook. **synth::hook_add** allows other scripts to register a callback function for this hook, and **synth::hook_call** allows the owner script to invoke all such callback functions. A hook must already be defined before a callback can be attached. Therefore typically device scripts will only use standard hooks and their own hooks, not hooks created by some other device, because the order of device initialization is not sufficiently defined. User scripts run from `mainrc.tcl` can use any hooks that have been defined.

synth::hook_call takes an arbitrary list of arguments, for example:

```
    synth::hook_call "ethernet_rx" "eth0" $packet
```

The callback function will always be invoked with a single argument, a list of the arguments that were passed to **synth::hook_call**:

```
    proc rx_callback { arg_list } {
        set device [lindex $arg_list 0]
        set packet [lindex $arg_list 1]
    }
```

Although it might seem more appropriate to use Tcl's **eval** procedure and have the callback functions invoked with the right number of arguments rather than a single list, that would cause serious problems if any of the data contained special characters such as `[` or `$`. The current implementation of hooks avoids such problems, at the cost of minor inconvenience when writing callbacks.

A number of hooks are defined as standard. Some devices will add additional hooks, and the device-specific documentation should be consulted for those. User scripts can add their own hooks if desired.

`exit`
: This hook is called just before the I/O auxiliary exits. Hence it provides much the same functionality as `atexit` in C programs. The argument list passed to the callback function will be empty.

`ecos_exit`
: This hook is called when the eCos application has exited. It is used mainly to shut down I/O operations: if the application is no longer running then there is no point in raising interrupts or storing incoming packets. The callback argument list will be empty.

`ecos_initialized`
: The synthetic target HAL will send a request to the I/O auxiliary once the static constructors have been run. All devices should now have been instantiated. A script could now check how many instances there are of a given type of device, for example ethernet devices, and create a little monitor window showing traffic on all the devices. The `ecos_initialized` callbacks will be run just before the user's `mainrc.tcl` script. The callback argument list will be empty.

`help`
: This hook is also invoked once static constructors have been run, but only if the user specified `-h` or `--help`. Any scripts that add their own command line arguments should add a callback to this hook which outputs details of the additional arguments. The callback argument list will be empty.

`interrupt`
: Whenever a device calls **synth::interrupt_raise** the `interrupt` hook will be called with a single argument, the interrupt vector. The main use for this is to allow user scripts to monitor interrupt traffic.

Output and Filters

Scripts can use conventional facilities for sending text output to the user, for example calling **puts** or directly manipulating the central text widget `.main.centre.text`. However in nearly all cases it is better to use output facilities provided by the I/O auxiliary itself:

```
synth::report <msg>
synth::report_warning <msg>
synth::report_error <msg>
synth::internal_error <msg>
synth::output <msg> <filter>
```

synth::report is intended for messages related to the operation of the I/O auxiliary itself, especially additional output resulting from `-v` or `--verbose`. If running in text mode the output will go to standard output. If running in graphical mode the output will go to the central text window. In both modes, use of `-l` or `--logfile` will modify the behaviour.

synth::report_warning, **synth::report_error** and **synth::internal_error** have the obvious meaning, including prepending strings such as `Warning:` and `Error:`. When the eCos application informs the I/O auxiliary that all static constructors have run, if at that point there have been any calls to **synth::error** then the I/O auxiliary will exit. This can be suppressed with command line arguments `-k` or `--keep-going`. **synth::internal_error** will output some information about the current state of the I/O auxiliary and then exit immediately. Of course it should never be necessary to call this function.

synth::output is the main routine for outputting text. The second argument identifies a filter. If running in text mode the filter is ignored, but if running in graphical mode the filter can be used to control the appearance of this output. A typical use would be:

```
    synth::output $line "console"
```

This outputs a single line of text using the `console` filter. If running in graphical mode the default appearance of this text can be modified with the `appearance` option in the **synth_device console** entry of the target definition file. The System filters menu option can be used to change the appearance at run-time.

Filters should be created before they are used. The procedures available for this are:

```
synth::filter_exists <name>
synth::filter_get_list
synth::filter_add <name> [options]
synth::filter_parse_options <options> <parsed_options> <message>
synth::filter_add_parsed <name> <parsed_options>
```

synth::filter_exists can be used to check whether or not a particular filter already exists: creating two filters with the same name is not allowed. **synth::filter_get_list** returns a list of the current known filters. **synth::filter_add** can be used to create a new filter. The first argument names the new filter, and the remaining arguments control the initial appearance. A typical use might be:

```
    synth::filter_add "my_device_tx" -foreground yellow -hide 1
```

It is assumed that the supplied arguments are valid, which typically means that they are hard-wired in the script. If instead the data comes out of a configuration file and hence may be invalid, the I/O auxiliary provides a parsing utility. Typical usage would be:

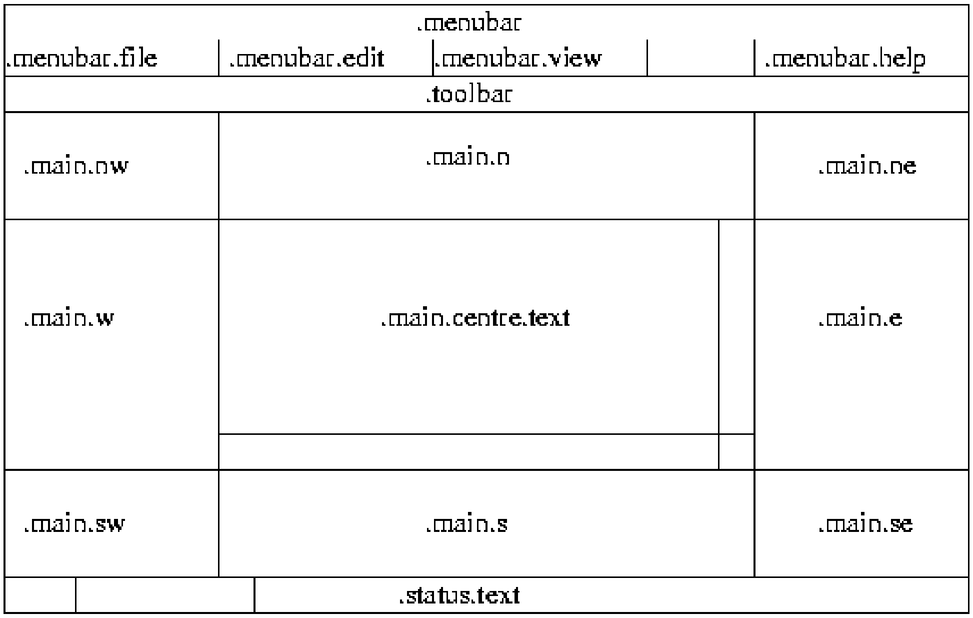
```
    array set parsed_options [list]
    set message ""
    if { ![synth::filter_parse_options $console_appearance parsed_options message] } {
        synth::report_error \
            "Invalid entry in target definition file $synth::target_definition\
             \n  synth_device \"console\", entry \"appearance\"\n$message"
    } else {
        synth::filter_add_parsed "console" parsed_options
    }
```

On success `parsed_options` will be updated with an internal representation of the desired appearance, which can then be used in a call to **synth::filter_add_parsed**. On failure `message` will be updated with details of the parsing error that occurred.

The Graphical Interface

When the I/O auxiliary is running in graphical mode, many scripts will want to update the user interface in some way. This may be as simple as adding another entry to the help menu for the device, or adding a new button to the toolbar. It may also involve adding new subwindows, or even creating entire new toplevel windows. These may be simple monitor windows, displaying additional information about what is going on in the system in a graphical format. Alternatively they may emulate actual I/O operations, for example button widgets could be used to emulate real physical buttons.

The I/O auxiliary does not provide many procedures related to the graphical interface. Instead it is expected that scripts will just update the widget hierarchy directly.



So adding a new item to the Help menu involves a **.menubar.help add** operation with suitable arguments. Adding a new button to the toolbar involves creating a child window in `.toolbar` and packing it appropriately. Scripts can create their own subwindows and then pack it into one of `.main.nw`, `.main.n`, `.main.ne`, `.main.w`, `.main.e`, `.main.sw`, `.main.s` or `.main.se`. Normally the user should be allowed to control this via the target definition file. The central window `.main.centre` should normally be left alone by other scripts since it gets used for text output.

The following graphics-related utilities may be found useful:

```
synth::load_image <image name> <filename>
synth::register_ballon_help <widget> <message>
synth::handle_help <URL>
```

synth::load_image can be used to add a new image to the current interpreter. If the specified file has a `.xbm` extension then the image will be a monochrome bitmap, otherwise it will be a colour image of some sort. A boolean will be returned to indicate success or failure, and suitable diagnostics will be generated if necessary.

synth::register_balloon_help provides balloon help for a specific widget, usually a button on the toolbar.

synth::handle_help is a utility routine that can be installed as the command for displaying online help, for example:

```
    .menubar.help add command -label "my device" -command \
        [list synth::handle_help "file://$path"]
```

Name

Porting — Adding support for other hosts

Description

The initial development effort of the eCos synthetic target happened on x86 Linux machines. Porting to other platforms involves addressing a number of different issues. Some ports should be fairly straightforward, for example a port to Linux on a processor other than an x86. Porting to Unix or Unix-like operating systems other than Linux may be possible, but would involve more effort. Porting to a completely different operating system such as Windows would be very difficult. The text below complements the eCos Porting Guide.

Other Linux Platforms

Porting the synthetic target to a Linux platform that uses a processor other than x86 should be straightforward. The simplest approach is to copy the existing `i386linux` directory tree in the `hal/synth` hierarchy, then rename and edit the ten or so files in this package. Most of the changes should be pretty obvious, for example on a 64-bit processor some new data types will be needed in the `basetype.h` header file. It will also be necessary to update the toplevel `ecos.db` database with an entry for the new HAL package, and a new target entry will be needed.

Obviously a different processor will have different register sets and calling conventions, so the code for saving and restoring thread contexts and for implementing `setjmp` and `longjmp` will need to be updated. The exact way of performing Linux system calls will vary: on x86 linux this usually involves pushing some registers on the stack and then executing an `int 0x080` trap instruction, but on a different processor the arguments might be passed in registers instead and certainly a different trap instruction will be used. The startup code is written in assembler, but needs to do little more than extract the process' argument and environment variables and then jump to the main `linux_entry` function provided by the architectural synthetic target HAL package.

The header file `hal_io.h` provided by the architectural HAL package provides various structure definitions, function prototypes, and macros related to system calls. These are correct for x86 linux, but there may be problems on other processors. For example a structure field that is currently defined as a 32-bit number may in fact may be a 64-bit number instead.

The synthetic target's memory map is defined in two files in the `include/pkgconf` subdirectory. For x86 the default memory map involves eight megabytes of read-only memory for the code at location 0x1000000 and another eight megabytes for data at 0x2000000. These address ranges may be reserved for other purposes on the new architecture, so may need changing. There may be some additional areas of memory allocated by the system for other purposes, for example the startup stack and any environment variables, but usually eCos applications can and should ignore those.

Other HAL functionality such as interrupt handling, diagnostics, and the system clock are provided by the architectural HAL package and should work on different processors with few if any changes. There may be some problems in the code that interacts with the I/O auxiliary because of lurking assumptions about endianness or the sizes of various data types.

When porting to other processors, a number of sources of information are likely to prove useful. Obviously the Linux kernel sources and header files constitute the ultimate authority on how things work at the system call level. The GNU C library sources may also prove very useful: for a normal Linux application it is the C library that provides the startup code and the system call interface.

Other Unix Platforms

Porting to a Unix or Unix-like operating system other than Linux would be somewhat more involved. The first requirement is toolchains: the GNU compilers, gcc and g++, must definitely be used; use of other GNU tools such as the linker may be needed as well, because eCos depends on functionality such as prioritizing C++ static constructors, and other linkers may not implement this or may implement it in a different and incompatible way. A closely related requirement is the use of ELF format for binary executables: if the operating system still uses an older format such as COFF then there are likely to be problems because they do not provide the flexibility required by eCos.

In the architectural HAL there should be very little code that is specific to Linux. Instead the code should work on any operating system that provides a reasonable implementation of the POSIX standard. There may be some problems with program startup, but those could be handled at the architectural level. Some changes may also be required to the exception handling code. However one file which will present a problem is `hal_io.h`, which contains various structure definitions and macros used with the system call interface. It is likely that many of these definitions will need changing, and it may well be appropriate to implement variant HAL packages for the different operating systems where this information can be separated out. Another possible problem is that the generic code assumes that system calls such as `cyg_hal_sys_write` are available. On an operating system other than Linux it is possible that some of these are not simple system calls, and instead wrapper functions will need to be implemented at the variant HAL level.

The generic I/O auxiliary code should be fairly portable to other Unix platforms. However some of the device drivers may contain code that is specific to Linux, for example the `PF_PACKET` socket address family and the ethertap virtual tunnelling interface. These may prove quite difficult to port.

The remaining porting task is to implement one or more platform HAL packages, one per processor type that is supported. This should involve much the same work as a port to another processor running Linux.

When using other Unix operating systems the kernel source code may not be available, which would make any porting effort more challenging. However there is still a good chance that the GNU C library will have been ported already, so its source code may contain much useful information.

Windows Platforms

Porting the current synthetic target code to some version of Windows or to another non-Unix platform is likely to prove very difficult. The first hurdle that needs to be crossed is the file format for binary executables: current Windows implementations do not use ELF, instead they use their own format PE which is a variant of the rather old and limited COFF format. It may well prove easier to first write an ELF loader for Windows executables, rather than try to get eCos to work within the constraints of PE. Of course that introduces new problems, for example existing source-level debuggers will still expect executables to be in PE format.

Under Linux a synthetic target application is not linked with the system's C library or any other standard system library. That would cause confusion, for example both eCos and the system's C library might try to define the `printf` function, and introduce complications such as working with shared libraries. For much the same reasons, a synthetic target application under Windows should not be linked with any Windows DLL's. If an ELF loader has been specially written then this may not be much of a problem.

The next big problem is the system call interface. Under Windows system calls are generally made via DLL's, and it is not clear that the underlying trap mechanism is well-documented or consistent between different releases of Windows.

The current code depends on the operating system providing an implementation of POSIX signal handling. This is used for I/O purposes, for example `SIGALRM` is used for the system clock, and for exceptions. It is not known what equivalent functionality is available under Windows.

Given the above problems a port of the synthetic target to Windows may or may not be technically feasible, but it would certainly require a very large amount of effort.

Part LXXVII. ARM7/ARM9/ XScale/Cortex-A Architecture

Table of Contents

Chapter 234. ARM Architectural Support

Name

`CYGPKG_HAL_ARM` — eCos Support for the ARM Architecture

Description

The ARM architecture HAL provides support for members of the ARM7, ARM9, XScale and Cortex-A families. This includes support for the ARM, Thumb and Thumb2 instruction sets.

The architectural HAL provides support for those features which are common to all members of the ARM family, and for certain features which are present on some, but not all, members. This HAL contains support for CPU initialization, exception and interrupt entry and exit, thread context switching, interrupt masking, timer management, cache management and debugging. A typical eCos configuration will also contain a variant HAL package with support code for a family of processors, possibly a processor HAL package with support for one specific processor, and a platform HAL which contains the code needed for a specific hardware platform. For example the variant or processor HAL may define the exact interrupt controller hardware that is available, and the platform HAL will define the external interrupt vector connections.

If appropriate the variant or platform HAL may also enable support for an ARM architectural hardware Floating Point Unit (FPU). Not all ARM architectures provide hardware floating point as an option, and also not all of the various ARM architectural hardware floating point implementations are currently supported by this architectural HAL.

Support is available for the Advanced SIMD Architecture, commonly known as NEON. Since NEON uses the same registers as the FPU, it is necessary to also have FPU support enabled when NEON support is required.

Support for SMP is available for the Cortex-A processor family based on the ARM MPCore cluster technology. This includes the Generic Interrupt Controller, Snoop Control Unit, private and global timers and the PL310 level 2 cache controller.

Name

Options — Configuring the ARM Architectural HAL Package

Description

The ARM architectural HAL is included in all ecos.db entries for ARM targets, so the package will be loaded automatically when creating a configuration. It should never be necessary to load the package explicitly or to unload it.

The ARM architectural HAL contains a number of configuration points. Few of these should be altered by the user, they are mainly present for the variant and platform HALs to select different architectural features. For example, the CPU family being used, whether the Thumb or Thumb2 instruction sets are supported, if the ARM EABI (Embedded Application Binary Interface) is being used, etc.

CYGINT_HAL_ARM_BIGENDIAN

> This interface controls whether the CPU is capable of being run in big endian mode. It should be implemented by either the variant or platform HAL to reflect the setting of the hardware.

CYGHWR_HAL_ARM_BIGENDIAN

> On targets which are capable of big-endian operation, this option is used to select whether big- or little-endian operation is desired. It provides the main test point for HAL, eCos and application code to test for a big-endian target. It is inactive if `CYGINT_HAL_ARM_BIGENDIAN` is not implemented.

CYGINT_HAL_ARM_FPU

> This interface controls whether the CPU is capable of supporting a hardware FPU (Floating Point Unit). It is the “common” FPU marker and is implemented when either the variant or platform HAL in turn implements a supported FPU type.
>
> For example, a Cortex-A5 target may define `CYGINT_HAL_VFPV4_D16` when it provides the ARMv7 VFPv4-D16 architecture floating point unit.

CYGHWR_HAL_ARM_FPU

> On targets which are capable of hardware FPU operation, this option is used to select whether soft or hard floating point operation is desired. It provides the main test point for HAL, eCos and application code to test for a hard-FP target. It is inactive if `CYGINT_HAL_ARM_FPU` is not implemented.
>
> Even though an architecture may provide a hardware FPU, it is not always suitable for all applications. For example, there is the associated scheduler and RAM cost in preserving FPU context for multi-threaded applications. If `CYGHWR_HAL_ARM_FPU` is enabled then some further configuration options are made available:
>
> CYGHWR_HAL_ARM_FPU_SWITCH
>
> > This option selects the FPU context switching scheme.

Table 234.1. Context Switch

SWITCH	Description
ALL	This mode is the most straightforward, and means that on every context switch, all FPU registers are saved and restored between threads. This mode makes the most sense if you need determinism and/or most or all of your threads will use FP. However if few threads use FP, it can result in a lot of overhead due to saves and restores of unchanged registers.

SWITCH	Description
LAZY	In this mode, if a thread has not used the FPU, the FPU context will not be saved or restored for it. The HAL installs a handler in the ARM undefined instruction exception vector in order to detect the first time the FPU is accessed by that thread. Once the FPU is accessed, the fault handler enables the FPU for that thread, and from then on, the FPU context will be saved and restored when switching from or to that thread. In a system where some or many threads do not use the FPU, this can greatly improve context switch time. However if the system spends most of its time swapping between two or more threads which do both use the FPU, then there may be additional overhead compared to the `ALL` mode (due to the need to check if the FPU was enabled for a particular thread on switch). This means the worst case context switch time is longer than with `ALL` mode. It also reduces determinism as there is an unavoidable latency at the point the thread first accesses the FPU, so that the fault handler can execute to enable the FPU; and determinism is further affected as context switch time depends on whether threads use the FPU. The `LAZY` mode does not save on stack usage, as the number of registers which might need to be saved remains the same.
NONE	In this mode, the FPU is enabled, but no floating point context is stored at any point, which naturally means there is no overhead on context switch. However this means that only one thread or context may use the FPU at a time. If using this mode, either all FP operations must be constrained to a single thread. Or there must be locking to ensure that multiple threads do not access the FPU registers simultaneously. But if you rely on locking, great care must be taken as the compiler has the potential to reorder floating point accesses outside of the critical region if it is still in the same function. The use of the `HAL_REORDER_BARRIER()` call from the `<cyg/hal/hal_arch.h>` HAL header can be useful to prevent reordering across a particular point in the code.

CYGIMP_HAL_ARM_FPU_EXC_SAVE

This option allows the FPU context to be saved over exceptions, including interrupts. Normally this option will be disabled by default, since the hardware FPU instructions will only ever be used from application thread level code.

Disabling this option avoids the unnecessary cost associated with saving the hardware FPU context for interrupt or exception handlers. When disabled this does impose the restriction that the developer should ensure that no attached ISR, DSR or exception handler routine can make use of hardware FPU instructions. For the vast majority of eCos configurations this is *not* an issue.

When this option is enabled the code will preserve the FPU context over exception handlers (including interrupt calls). This option may become a “requirement” if the compiler is configured to make use of hardware FPU registers as temporary storage. However, such use is *not* recommended since the extra performance cost in saving and restoring the FPU context across (for example) *every* interrupt probably far outweighs the gain from being able to use the hardware FPU within such handlers. NOTE: If an interrupt or exception handler does make use of the FPU hardware there is *NO* FPU state preserved across successive handler calls, since each call is a unique instance.

CYGINT_HAL_ARM_NEON

This interface controls whether the CPU contains support for the ARM Advanced SIMD architecture, commonly referred to as NEON. It is usually implemented by the variant or platform HAL to enable NEON support. Since NEON shares registers with the FPU, it is also usually necessary to implement the appropriate FPU option.

For example, a Cortex-A53 target may implement `CYGINT_HAL_VFPV4_D32` when it provides the ARMv7 VFPv4-D32 architecture floating point unit and in addition implement `CYGINT_HAL_ARM_NEON` to indicate that the FPU also implements the NEON architecture.

CYGHWR_HAL_ARM_NEON

On targets which are capable of NEON operation, this option is used to select whether NEON support is enabled. It provides the main test point for HAL, eCos and application code to test for a NEON capable target. It is inactive if either `CYGINT_HAL_ARM_FPU` or `CYGINT_HAL_ARM_NEON` are not implemented. If this option is enabled, then `CYGHWR_HAL_ARM_FPU` will also be enabled automatically. When NEON is enabled, the compiler may generate NEON instructions for data manipulation in any function, including those that may be called from an ISR or DSR. Consequently, `CYGIMP_HAL_ARM_FPU_EXC_SAVE` is also enabled automatically.

Even though an architecture may provide NEON support, it is not always suitable for all applications. NEON shares the FPU registers and the same context save and restore issues apply. Consequently, the FPU options described above for managing the FPU context are active and also apply to NEON applications.

CYGINT_HAL_ARM_SYSTEM_DEBUG_DCC

This interface controls whether the CPU supports the ARM Debug Communications Channel (DCC) feature. It is normally implemented by the variant HAL to reflect the availability of the hardware.

CYGHWR_HAL_ARM_DIAGNOSTICS_INTERFACE

By default the architectural HAL does not implement diagnostic support, with the default `Serial` support being left to the variant or platform HAL implementation.

However, if the CPU variant implements the on-chip Debug Communications Channel feature then selecting `DCC` for this option will configure the system to use the generic architectural HAL DCC diagnostic output support. Accessing DCC diagnostic output will require corresponding support from the hardware debugger host tools being used to connect to the target system.

The `discard` option configures the system so that all diagnostic output is discarded. This can be selected and used when no I/O channel is available for diagnostics.

When DCC is being used for HAL diagnostics the option `CYGDBG_HAL_ARM_DIAGNOSTICS_INTERFACE_DCC_TYPE` should match the attached host debugger interface used to capture the diagnostic output. The `Character` interface simply transfers single characters through the DCC interface, and is compatible with the Linux `ICEDCC` support and with either of the Lauterbach `DCC` and `DCC3` formats. The `LIBDCC` format (as used by OpenOCD for example) uses a specific single character encoding that inter-operates with features not currently supported by eCos.

Compiler Flags

It is normally the responsibility of the platform HAL to define the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos.

Linker Scripts

The architecture HAL does not provide the main linker script. Instead this must be supplied by the variant HAL, and the makefile rules to generate the final target.ld must be included in the variant's CDL file.

Name

HAL Port — Implementation Details

Description

This documentation explains how the eCos HAL specification has been mapped onto the ARM hardware and should be read in conjunction with the relevant Architecture Reference Manual and the Technical Reference Manual for the revision of the ARM architecture being used. It should be noted that the architectural HAL is usually complemented by a variant HAL and a platform HAL, and those may affect or redefine some parts of the implementation.

Exports

The architectural HAL provides header files `cyg/hal/hal_arch.h`, `cyg/hal/hal_intr.h`, `cyg/hal/hal_io.h` and `cyg/hal/hal_mmu.h`. These header files export the functionality provided by all the ARM HALs for a given target, automatically including headers from the lower-level HALs as appropriate. For example the platform HAL may provide a header `cyg/hal/plf_io.h` containing additional I/O functionality, but that header will be automatically included by `cyg/hal/hal_io.h` so there is no need to include it directly.

Additionally, the architecture HAL provides the `cyg/hal/basetype.h` header, which defines the basic properties of the architecture, including endianness, data type sizes and alignment constraints.

Startup

The architectural HAL provides a default implementation of the low-level startup code which will be appropriate in nearly all scenarios. For a ROM startup this includes copying initialized data from flash to RAM. For all startup types it will involve zeroing BSS regions and setting up the general C environment. It will also set up the initial exception priorities, switches the CPU into the correct execution mode, enables the debug monitor and enables error exception handling.

In addition to the setup it does itself, the initialization code calls out to the variant and platform HALs to perform their own initialization via the `hal_hardware_init()` function.

The architectural HAL also initializes the VSR and virtual vector tables, sets up HAL diagnostics, and invokes C++ static constructors, prior to calling the first application entry point `cyg_start`. This code resides in `src/vectors.S`.

Interrupts and Exceptions

The eCos interrupt and exception architecture is built around a table of pointers to Vector Service Routines that translate hardware exceptions and interrupts into the function calls expected by eCos. The ARM vector table provides exactly this functionality, so it is used directly as the eCos VSR table. The `HAL_VSR_GET` and `HAL_VSR_SET` macros therefore manipulate the vector table directly. The `hal_intr.h` header provides definitions for all the standard ARM exception vectors.

The vector table is constructed at runtime. For ROM, ROMRAM and SRAM startup all entries are initialized. For RAM startup only the interrupt vectors are (re-)initialized to point to the VSR in the loaded code, the exception vectors are left pointing to the VSRs of the loading software, usually RedBoot or GDB stubs.

When an exception occurs it is delivered via the relevant handler provided in `vectors.S`. The handler will save the CPU state and call `exception_handler` in `hal_misc.c`, which passes the exception on to either the kernel or the GDB stub handler. If it returns then the CPU state is restored and the code continued.

When an interrupt occurs it is delivered to a shared VSR, `hal_default_irq_vsr`, which saves some state and calls `hal_IRQ_handler`.

The architectural HAL provides default implementations of `HAL_DISABLE_INTERRUPTS`, `HAL_RESTORE_INTERRUPTS`, `HAL_ENABLE_INTERRUPTS` and `HAL_QUERY_INTERRUPTS`. These involve manipulation of the status register `I` flag. Similarly there are default implementations of the interrupt controller macros HAL_INTERRUPT_MASK, HAL_INTERRUPT_UNMASK, HAL_INTERRUPT_ACKNOWLEDGE and HAL_INTERRUPT_CONFIGURE macros.

HAL_INTERRUPT_SET_LEVEL manipulates the relevant interrupt priority registers. The valid range of interrupts supported depends on the number of interrupt priority bits supported by the CPU variant.

Stacks and Stack Sizes

cyg/hal/hal_arch.h defines values for minimal and recommended thread stack sizes, CYGNUM_HAL_STACK_SIZE_MINIMUM and CYGNUM_HAL_STACK_SIZE_TYPICAL. These values depend on a number of configuration options.

A number of system stacks are provided, and their properties controlled in this package's configuration. By default, the ARM HAL will use a separate stack for calling interrupt handlers. This separate interrupt stack means that the worst case overhead of interrupt handling does not need to be considered when determining each thread's maximum stack usage, which reduces overall stack overhead. The size of this interrupt stack is controlled by the common HAL's configuration (CYGNUM_HAL_COMMON_INTERRUPTS_STACK_SIZE) or can be disabled entirely by turning off CYGIMP_HAL_COMMON_INTERRUPTS_USE_INTERRUPT_STACK.

System startup code will also run on the interrupt stack, if enabled, as it is usually sufficiently large for this. Optionally, a separate startup stack can be enabled in this HAL by disabling CYGIMP_HAL_ARM_INT_STACK_IS_STARTUP_STACK, in which case when control is passed to the application by the cyg_start() or cyg_user_start() entry points, this startup stack will then be used. Alternatively, if the interrupt stack has been disabled entirely then a startup stack must be present, and will be used for all initialisation. Its size can be set with CYGNUM_HAL_ARM_STARTUP_STACK_SIZE. Note that global C++ object constructors defined by either the system, or in application code, will have their constructors run on the interrupt stack. Using the C library startup package's "Invoke default static constructors" option (CYGSEM_LIBC_INVOKE_DEFAULT_STATIC_CONSTRUCTORS) would instead ensure the user application constructors are called in the context of main(), which can be more appropriate.

If including GDB stubs in the application, then a separate GDB stub stack is required in order to guarantee that application problems with stack use will not prevent the GDB stub being able to debug the application. Again the size is controlled via this package's CDL (CYGNUM_HAL_ARM_GDB_STACK_SIZE)

Separate small stacks are also created to do the initial handling of Abort Prefetch, Abort Data, Undefined Instruction exceptions, as well as IRQ and FIQ interrupts. Assuming the default eCos VSRs are in place for these exceptions/interrupts, these small stacks are only used very temporarily until the context is switched to supervisor (SVC) mode.

At that point, in the case of the first three exceptions, if GDB stubs are included, the stack then used will be the GDB stack mentioned above. Alternatively, in the case of the first three exceptions without GDB stubs, the stack used will be that of the supervisor mode (SVC) context at the time of the exception. This is usually the running thread, but can also be a DSR running on the interrupt stack.

In the case of the IRQ and FIQ interrupts, these small stacks are only used by the default eCos VSRs temporarily until the stack is switched to the interrupt stack (or if that is disabled, the stack of the interrupted thread).

The above describes the situation when using the normal eCos VSRs for handling the Abort Data, Abort Prefetch, Undefined Instruction, IRQ and FIQ exceptions/interrupts. However if the user overrides the eCos VSRs with their own VSRs, then it may be necessary to change the stack sizes for these contexts depending on the stack use by those new VSRs. Therefore each of the stack sizes corresponding to these exception/interrupt contexts can be changed in the ARM HAL package configuration.

Thread Contexts and setjmp/longjmp

cyg/hal/hal_arch.h defines a thread context data structure, the context-related macros, and the setjmp/longjmp support. The implementations can be found in src/context.S.

Bit Indexing

The architectural HAL provides implementations in the source file hal_misc.c that are referenced by the HAL_LSBIT_INDEX and HAL_MSBIT_INDEX macros.

Idle Thread Processing

Normally the variant HAL provides the `HAL_IDLE_THREAD_ACTION` implementation. It usually implements code that can be used to put the CPU into a low power mode ready to respond quickly to the next interrupt.

Clock Support

The architectural HAL provides default implementations of the various system clock macros such as HAL_CLOCK_INITIALIZE. The variant or platform HAL are responsible for providing the necessary implementation routines.

HAL I/O

The ARM architecture does not have a separate I/O bus. Instead all hardware is assumed to be memory-mapped. Further it is assumed that all peripherals on the memory bus will switch endianness with the processor and that there is no need for any byte swapping. Hence the various HAL macros for performing I/O simply involve pointers to volatile memory.

The variant and platform files included by the `cyg/hal/hal_io.h` header will typically also provide details of some or all of the peripherals, for example register offsets and the meaning of various bits in those registers.

Cache Handling

The architecture HAL does not provide direct support for dealing with caches, since there is no common mechanism for doing this. The cache support is the responsibility of the variant HAL to, which will supply the `cyg/hal/hal_cache.h` header.

Linker Scripts

The architectural HAL will generate the linker script for eCos applications. This involves the architectural file `src/arm.ld` and a `.ldi` memory layout file, typically provided by the platform HAL. It is the `.ldi` file which places code and data in the appropriate places for the startup type, but most of the hard work is done via macros in the `arm.ld` file.

Diagnostic Support

The architectural HAL implements diagnostic support for DCC output if available, or for discarding all output. However, by default, the diagnostics output is left to the variant or platform HAL, depending on whether suitable peripherals are available on-chip or off-chip. The `CYGHWR_HAL_ARM_DIAGNOSTICS_INTERFACE` can be configured to direct the diagnostic output support used to the appropriate destination.

SMP Support

The ARM architectural HAL provides SMP support for Cortex-A class processors. If the configuration option `CYGPKG_HAL_SMP_SUPPORT` is enabled then the `hal_smp.h` header defines the standard SMP macros described in the HAL documentation. The architectural HAL only provides the SMP components that are common to all CPUs. It is the responsibility of variant and platform HALs to complete SMP support.

The variant HAL needs to supply a number of services for SMP. Access to the interrupt controller needs to be multi-core safe. The design of the standard ARM GIC provides this by default, but other controllers may need a spinlock. MMU and cache support are linked since any memory containing a spinlock must be cached and marked shareable. The variant HAL should also contain `cyg_hal_cpu_start()`, which is used to start up the secondary CPUs and `cyg_hal_smp_start()`, which is the initial entry point for secondary CPUs. It must also supply `cyg_hal_cpu_message()`, and associated ISR and DSR, which are used to pass scheduling messages between CPUs.

The platform HAL (which may comprise more than one layer of hardware specific HALs) is responsible for the memory map and initialization. Initialization will usually involve starting clocks, setting up pin multiplexing and configuring the CPU state. Most of this is common to single and multi-core configurations, although there will be some SMP specific settings. This HAL will also need to supply the PLATFORM_SETUP_CPU macro to initialize the secondary CPUs.

Debug Support

The architectural HAL provides basic support for gdb stubs using the debug monitor exceptions. Breakpoints are implemented using a fixed-size list of breakpoints, as per the configuration option CYGNUM_HAL_BREAKPOINT_LIST_SIZE. When a JTAG device is connected to a ARM device, it will steal breakpoints and other exceptions from the running code. Therefore debugging from RedBoot or the GDB stubs can only be done after detaching any JTAG debugger and power-cycling the board.

HAL_DELAY_US() Macro

The variant or platform HAL is responsible for providing an implementation of the HAL_DELAY_US macro. The system timer must be initialized before this macro is used. The include/hal_intr.h defined HAL_CLOCK_INITIALIZE() macro is called during initialization after the variant and platform initialization functions are called, but before constructors are invoked.

Profiling Support

When using local memory based profiling the ARM architectural HAL implements the mcount function, allowing profiling tools like gprof to determine the application's call graph. It does not implement the profiling timer. Instead that functionality needs to be provided by the variant or platform HAL.

Chapter 235. Atmel AT91 Processor Variant Support

Name

eCos Support for the Atmel AT91 Processor Variant — Overview

Description

This document covers the configuration and usage of the Hardware Abstraction Layer (HAL) for the Atmel AT91 processor family which includes the AT91R4xxxx series, the M42xxxx and M55xxxx series and the AT91SAM7S, -X and -A series. It is expected to be read in conjunction with platform HAL-specific documentation, as well as the eCos HAL specification. This processor HAL package complements the ARM architectural HAL, AT91SAM7 variant HAL (where appropriate) and the platform HAL. It provides functionality common to all AT91-based board implementations.

This support is found in the eCos package located at `packages/hal/arm/at91/var` within the eCos source repository.

The AT91 processor HAL package is loaded automatically when eCos is configured for an AT91-based platform. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Supported Hardware

Supported features of the Atmel AT91 processor within this processor HAL package include:

- AT91-specific hardware definitions
- Interrupt controller
- Timer counters
- Serial UARTs

Support for the interrupt-driven serial, SPI, watchdog and wallclock (RTC) features of the AT91 are also present and can be found in separate packages, outside of this processor HAL.

The watchdog hardware may also used within this HAL to perform software reset.

Name

AT91 hardware definitions — Details on obtaining hardware definitions for AT91

Register definitions

The file `<cyg/hal/var_io.h>` provides definitions related to AT91 subsystems. This file should not be included explicitly, but is included automatically whenever `<cyg/hal/hal_io.h>` is included. This file includes register definitions for the interrupt controller, power management controller, clock generator, memory controller, external bus interface, GPIO, USART, MCI, CAN, TWI (I²C®), Ethernet, timer counter, RTC, and SPI subsystems, depending on the exact model.

Name

AT91 interrupt controller — Advanced Interrupt Controller definitions and usage

Interrupt Controller Support

The AT91 variant HAL contains generic support for the AIC (GIC on SAM7XA1 and -A2). It queries the interrupt controller to identify the current interrupt and vectors to the matching service routine. If the device supports the SYSTEM interrupt then the devices that raise this interrupt will be queried individually and vectored to their own interrupt handlers. The mapping between interrupts and vector numbers is defined in the `hal_platform_ints.h` file in either the platform HAL or the AT91SAM7 variant HAL.

As indicated above, further decoding is performed on the SYSTEM interrupt to identify the cause more specifically. Note that as a result, placing an interrupt handler on the SYSTEM interrupt will not work as expected. Conversely, masking a decoded derivative of the SYSTEM interrupt will not work as this would mask other SYSTEM interrupts, but masking the SYSTEM interrupt itself will work. On the other hand, unmasking a decoded SYSTEM interrupt *will* unmask the SYSTEM interrupt as a whole, thus unmasking interrupts for the other units on this shared interrupt.

Interrupt Controller Functions

The source file `src/at91_misc.c` within this package provides most of the support functions to manipulate the interrupt controller. The `hal_IRQ_handler` queries the IRQ status register to determine the interrupt cause. Functions `hal_interrupt_mask` and `hal_interrupt_unmask` enable or disable interrupts within the interrupt controller.

Interrupts are configured in the `hal_interrupt_configure` function, where the `level` and `up` arguments are interpreted as follows:

level	up	interrupt on
0	0	Falling Edge
0	1	Rising Edge
1	0	Low Level
1	1	High Level

To fit into the eCos interrupt model, interrupts essentially must be acknowledged immediately once decoded, and as a result, the `hal_interrupt_acknowledge` function is empty.

The `hal_interrupt_set_level` is used to set the priority level of the supplied interrupt within the Advanced Interrupt Controller.

Note that in all the above, it is not recommended to call the described functions directly. Instead either the HAL macros (`HAL_INTERRUPT_MASK` et al) or preferably the kernel or driver APIs should be used to control interrupts.

Interrupt handling within standalone applications

For non-eCos standalone applications running under RedBoot, it is possible to install an interrupt handler into the interrupt vector table manually. Memory layouts are platform-dependent and so the platform documentation should be consulted, but in general the address of the interrupt table can be determined by analyzing RedBoot's symbol table, and searching for the address of the symbol name `hal_interrupt_handlers`. Table slots correspond to the interrupt numbers defined in the platform or AT91SAM7 HAL. Pointers inserted in this table should be pointers to a C/C++ function with the following prototype:

```
extern unsigned int isr( unsigned int vector, unsigned int data );
```

For non-eCos applications run from RedBoot, the return value can be ignored. The `vector` argument will also be the interrupt vector number. The `data` argument is extracted from a corresponding table named `hal_interrupt_data` which immediately follows the interrupt vector table. It is still the responsibility of the application to enable and configure the interrupt source appropriately if needed.

Name

Timers — Use of on-chip Timer

Hardware Timer

The eCos kernel system clock is implemented using an on-chip Timer. Depending on the device this will either be a Timer Counter, a Simple Timer, or the Periodic Interval Timer. The kind of device used is determined by the platform HAL. By default, the system clock interrupts once every 10ms, corresponding to a 100Hz clock. This can be changed by the configuration option CYGNUM_HAL_RTC_DENOMINATOR which corresponds to the clock frequency. Other clock-related settings are recalculated automatically if the denominator is changed. If the desired frequency cannot be expressed accurately solely with changes to CYGNUM_HAL_RTC_DENOMINATOR, then the configuration option CYGNUM_HAL_RTC_NUMERATOR may also be adjusted, and again clock-related settings will automatically be recalculated.

The selected Timer is also used to implement the HAL microsecond delay function, HAL_DELAY_US. This is used by some device drivers, and in non-kernel configurations such as with RedBoot where this timer is needed for loading program images via X/Y-modem protocols and debugging via TCP/IP. Standalone applications which require RedBoot services, such as debugging, should avoid use of this timer.

Timer-based profiling support

Timer-based profiling support is implemented using timer counter 1 (TC1). If the gprof package, CYGPKG_PROFILE_GPROF, is included in the configuration, then TC1 is reserved for use by the profiler.

Name

Serial UARTs — Configuration and implementation details of serial UART support

Overview

Support is included in this processor HAL package for the AT91's on-chip debug unit UART and serial USART serial devices.

There are two forms of support: HAL diagnostic I/O; and a fully interrupt-driven serial driver. Unless otherwise specified in the platform HAL documentation, for all serial ports the default settings are 38400,8,N,1 with no flow control.

HAL diagnostic I/O

This first form is polled mode HAL diagnostic output, intended primarily for use during debug and development. Operations are usually performed with global interrupts disabled, and thus this mode is not usually suitable for deployed systems. This can operate on any port, according to the configuration settings.

There are several configuration options usually found within a platform HAL which affect the use of this support in the AT91 processor HAL. The CDL option `CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL` selects the serial port channel to use as the console at startup time. This will be the channel that receives output from, for example, `diag_printf()`. The CDL option `CYGNUM_HAL_VIRTUAL_VECTOR_DEBUG_CHANNEL` selects the serial port channel to use for GDB communication by default. Note that when using RedBoot, these options are usually inactive as it is RedBoot that decides which channels are used. Applications may override RedBoot's selections by enabling the `CYGSEM_HAL_VIRTUAL_VECTOR_CLAIM_COMMS` CDL option in the HAL. Baud rates for each channel are set with the `CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL_BAUD` and `CYGNUM_HAL_VIRTUAL_VECTOR_DEBUG_CHANNEL_BAUD` options.

Interrupt-driven serial driver

The second form of support is an interrupt-driven serial driver, which is integrated into the eCos standard serial I/O infrastructure (`CYGPKG_IO_SERIAL`). This support can be enabled on any port.

Note that it is not recommended to share this driver when using the HAL diagnostic I/O on the same port. If the driver is shared with the GDB debugging port, it will prevent ctrl-c operation when debugging.

This driver is contained in the `CYGPKG_IO_SERIAL_ARM_AT91` package. That driver package should also be consulted for documentation and configuration options. The driver is not active until the `CYGPKG_IO_SERIAL_DEVICES` configuration option within the generic serial driver support package `CYGPKG_IO_SERIAL` is enabled in the configuration.

The USARTs are named "/dev/ser0", "/dev/ser1" and so on. The DEBUG serial port is given the name "/dev/dbg". These names are all configurable.

Note that unlike the USART devices, the serial debug port does not support modem control signals such as those used for hardware flow control. In addition, USART devices for a particular platform may also not have these control signals brought out to the physical serial port.

Chapter 236. Atmel AT91SAM7 Processor Variant Support

Name

eCos Support for the Atmel AT91SAM7 Processor Variant — Overview

Description

This document covers the configuration and usage of the Hardware Abstraction Layer (HAL) for the Atmel AT91SAM7 processor family which includes the AT91SAM7S, -X an -A series. It is expected to be read in conjunction with platform HAL-specific documentation, as well as the eCos HAL specification. This HAL package complements the ARM architectural HAL, AT91 variant HAL (where appropriate) and the platform HAL. It provides functionality common to all AT91SAM7-based board implementations.

This support is found in the eCos package located at `packages/hal/arm/at91/at91sam7` within the eCos source repository.

The AT91SAM7 HAL package is loaded automatically when eCos is configured for an AT91SAM7-based platform. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Supported Hardware

Supported features of the Atmel AT91SAM7 within this processor HAL package include:

- AT91SAM7-specific hardware definitions
- Interrupt Vector Definitions

Support for the interrupt-driven serial, SPI, watchdog and wallclock (RTC) features of the AT91SAM7 are also present and can be found in separate packages, outside of this processor HAL.

Name

AT91SAM7 hardware definitions — Details on obtaining hardware definitions for AT91

Device Definitions

The file `<cyg/hal/plf_io.h>` provides definitions related to AT91SAM7 subsystems. This file should not be included explicitly, but is included automatically whenever `<cyg/hal/hal_io.h>` is included. This file mainly includes base address definitions for the interrupt controller, power management controller, clock generator, memory controller, external bus interface, GPIO, USART, MCI, CAN, TWI (I²C®), Ethernet, timer counter, RTC, and SPI subsystems, depending on the exact model.

Name

AT91SAM7 interrupt vector definitions — Advanced Interrupt Controller vector definitions

Interrupt Vector Definitions

The file `<cyg/hal/hal_platform_ints.h>` (located at `hal/arm/arm9/at91sam7/`*`VERSION`*`/include/hal_platform_ints.h` in the eCos source repository) contains interrupt vector number definitions for use with the eCos kernel and driver interrupt APIs. The exact set of vectors supported depends on the AT91SAM7 model:

For the AT91SAM7S family:

```
#define CYGNUM_HAL_INTERRUPT_FIQ        0       // Advanced Interrupt Controller (FIQ)
#define CYGNUM_HAL_INTERRUPT_SYS        1       // System Peripheral (debug unit, system timer)
#define CYGNUM_HAL_INTERRUPT_PIOA       2       // Parallel IO Controller A
#define CYGNUM_HAL_INTERRUPT_ADC        4       // Analog-to-Digital Converter
#define CYGNUM_HAL_INTERRUPT_SPI        5       // Serial Peripheral Interface
#define CYGNUM_HAL_INTERRUPT_USART0     6       // USART 0
#define CYGNUM_HAL_INTERRUPT_USART1     7       // USART 1
#define CYGNUM_HAL_INTERRUPT_SSC        8       // Serial Synchronous Controller
#define CYGNUM_HAL_INTERRUPT_TWI        9       // Two-Wire Interface (I2C)
#define CYGNUM_HAL_INTERRUPT_PWMC       10      // PWM Controller
#define CYGNUM_HAL_INTERRUPT_UDP        11      // USB Device Port
#define CYGNUM_HAL_INTERRUPT_TC0        12      // Timer Counter 0
#define CYGNUM_HAL_INTERRUPT_TC1        13      // Timer Counter 1
#define CYGNUM_HAL_INTERRUPT_TC2        14      // Timer Counter 2
#define CYGNUM_HAL_INTERRUPT_IRQ0       30      // External IRQ0
#define CYGNUM_HAL_INTERRUPT_IRQ1       31      // External IRQ0

// Interrupts which are multiplexed on to the System Interrupt
#define CYGNUM_HAL_INTERRUPT_PITC       32      // Period Interval Timer
#define CYGNUM_HAL_INTERRUPT_RTTC       33      // Real-Time Timer
#define CYGNUM_HAL_INTERRUPT_PMC        34      // Power Management Controller
#define CYGNUM_HAL_INTERRUPT_MC         35      // Memory Controller
#define CYGNUM_HAL_INTERRUPT_WDTC       36      // Watchdog
#define CYGNUM_HAL_INTERRUPT_RSTC       37      // Reset Controller
#define CYGNUM_HAL_INTERRUPT_DEBUG      38      // Debug Serial Port
```

For the AT91SAM7X family:

```
#define CYGNUM_HAL_INTERRUPT_FIQ        0       // Advanced Interrupt Controller (FIQ)
#define CYGNUM_HAL_INTERRUPT_SYS        1       // System Peripheral (debug unit, system timer)
#define CYGNUM_HAL_INTERRUPT_PIOA       2       // Parallel IO Controller A
#define CYGNUM_HAL_INTERRUPT_PIOB       3       // Parallel IO Controller B
#define CYGNUM_HAL_INTERRUPT_SPI        4       // Serial Peripheral Interface
#define CYGNUM_HAL_INTERRUPT_SPI1       5       // Serial Peripheral Interface 1
#define CYGNUM_HAL_INTERRUPT_USART0     6       // USART 0
#define CYGNUM_HAL_INTERRUPT_USART1     7       // USART 1
#define CYGNUM_HAL_INTERRUPT_SSC        8       // Serial Synchronous Controller
#define CYGNUM_HAL_INTERRUPT_TWI        9       // Two-Wire Interface (I2C)
#define CYGNUM_HAL_INTERRUPT_PWMC       10      // PWM Controller
#define CYGNUM_HAL_INTERRUPT_UDP        11      // USB Device Port
#define CYGNUM_HAL_INTERRUPT_TC0        12      // Timer Counter 0
#define CYGNUM_HAL_INTERRUPT_TC1        13      // Timer Counter 1
#define CYGNUM_HAL_INTERRUPT_TC2        14      // Timer Counter 2
#define CYGNUM_HAL_INTERRUPT_CAN        15      // CAN Controller
#define CYGNUM_HAL_INTERRUPT_EMAC       16      // Ethernet MAC
#define CYGNUM_HAL_INTERRUPT_ADC        17      // Analog-to-Digital Converter
#define CYGNUM_HAL_INTERRUPT_IRQ0       30      // External IRQ0
#define CYGNUM_HAL_INTERRUPT_IRQ1       31      // External IRQ0

// Interrupts which are multiplexed on to the System Interrupt
#define CYGNUM_HAL_INTERRUPT_PITC       32      // Period Interval Timer
#define CYGNUM_HAL_INTERRUPT_RTTC       33      // Real-Time Timer
#define CYGNUM_HAL_INTERRUPT_PMC        34      // Power Management Controller
```

```
#define CYGNUM_HAL_INTERRUPT_MC         35      // Memory Controller
#define CYGNUM_HAL_INTERRUPT_WDTC       36      // Watchdog
#define CYGNUM_HAL_INTERRUPT_RSTC       37      // Reset Controller
#define CYGNUM_HAL_INTERRUPT_DEBUG      38      // Debug Serial Port
```

For the AT91SAM7A3:

```
#define CYGNUM_HAL_INTERRUPT_FIQ        0       // Advanced Interrupt Controller (FIQ)
#define CYGNUM_HAL_INTERRUPT_SYS        1       // System Peripheral (debug unit, system timer)
#define CYGNUM_HAL_INTERRUPT_PIOA       2       // Parallel IO Controller A
#define CYGNUM_HAL_INTERRUPT_PIOB       3       // Parallel IO Controller B
#define CYGNUM_HAL_INTERRUPT_CAN0       4       // CAN Controller 0
#define CYGNUM_HAL_INTERRUPT_CAN1       5       // CAN Controller 1
#define CYGNUM_HAL_INTERRUPT_USART0     6       // USART 0
#define CYGNUM_HAL_INTERRUPT_USART1     7       // USART 1
#define CYGNUM_HAL_INTERRUPT_USART2     8       // USART 2
#define CYGNUM_HAL_INTERRUPT_MCI        9       // Multimedia Card Interface
#define CYGNUM_HAL_INTERRUPT_TWI        10      // Two-Wire Interface (I2C)
#define CYGNUM_HAL_INTERRUPT_SPI        11      // Serial Parallel Interface 0
#define CYGNUM_HAL_INTERRUPT_SPI1       12      // Serial Parallel Interface 1
#define CYGNUM_HAL_INTERRUPT_SSC0       13      // Serial Synchronous Controller 0
#define CYGNUM_HAL_INTERRUPT_SSC1       14      // Serial Synchronous Controller 1
#define CYGNUM_HAL_INTERRUPT_TC0        15      // Timer Counter 0
#define CYGNUM_HAL_INTERRUPT_TC1        16      // Timer Counter 1
#define CYGNUM_HAL_INTERRUPT_TC2        17      // Timer Counter 2
#define CYGNUM_HAL_INTERRUPT_TC3        18      // Timer Counter 3
#define CYGNUM_HAL_INTERRUPT_TC4        19      // Timer Counter 4
#define CYGNUM_HAL_INTERRUPT_TC5        20      // Timer Counter 5
#define CYGNUM_HAL_INTERRUPT_TC6        21      // Timer Counter 6
#define CYGNUM_HAL_INTERRUPT_TC7        22      // Timer Counter 7
#define CYGNUM_HAL_INTERRUPT_TC8        23      // Timer Counter 8
#define CYGNUM_HAL_INTERRUPT_ADC0       24      // Analog-to-Digital Converter 0
#define CYGNUM_HAL_INTERRUPT_ADC1       25      // Analog-to-Digital Converter 1
#define CYGNUM_HAL_INTERRUPT_PWMC       26      // PWM Controller
#define CYGNUM_HAL_INTERRUPT_UDP        27      // USB Device Port
#define CYGNUM_HAL_INTERRUPT_IRQ0       28      // External Interrupt 0
#define CYGNUM_HAL_INTERRUPT_IRQ1       29      // External Interrupt 1
#define CYGNUM_HAL_INTERRUPT_IRQ2       30      // External Interrupt 2
#define CYGNUM_HAL_INTERRUPT_IRQ3       31      // External Interrupt 3

// Interrupts which are multiplexed on to the System Interrupt
#define CYGNUM_HAL_INTERRUPT_PITC       32      // Period Interval Timer
#define CYGNUM_HAL_INTERRUPT_RTTC       33      // Real-Time Timer
#define CYGNUM_HAL_INTERRUPT_PMC        34      // Power Management Controller
#define CYGNUM_HAL_INTERRUPT_MC         35      // Memory Controller
#define CYGNUM_HAL_INTERRUPT_WDTC       36      // Watchdog
#define CYGNUM_HAL_INTERRUPT_RSTC       37      // Reset Controller
#define CYGNUM_HAL_INTERRUPT_DEBUG      38      // Debug Serial Port
```

For the AT91SAM7A1 and AT91SAM7A2:

```
#define CYGNUM_HAL_INTERRUPT_FIQ         0      // Advanced Interrupt Controller (FIQ)
#define CYGNUM_HAL_INTERRUPT_SWIIRQ0     1      // Software Interrupt 0
#define CYGNUM_HAL_INTERRUPT_WD          2      // Watchdog
#define CYGNUM_HAL_INTERRUPT_WT          3      // Watch Timer
#define CYGNUM_HAL_INTERRUPT_USART0      4      // USART 0
#define CYGNUM_HAL_INTERRUPT_USART1      5      // USART 1
#define CYGNUM_HAL_INTERRUPT_CAN3        6      // CAN Controller 3
#define CYGNUM_HAL_INTERRUPT_SPI         7      // Serial Peripheral Interface
#define CYGNUM_HAL_INTERRUPT_CAN1        8      // CAN Controller 1
#define CYGNUM_HAL_INTERRUPT_CAN2        9      // CAN Controller 2
#define CYGNUM_HAL_INTERRUPT_ADC0        10      // Analog-to-Digital Converter 0
#define CYGNUM_HAL_INTERRUPT_ADC1        11      // Analog-to-Digital Converter 1
#define CYGNUM_HAL_INTERRUPT_GPT0CH0     12      // General Purpose Timer 0 Channel 0
#define CYGNUM_HAL_INTERRUPT_GPT0CH1     13      // General Purpose Timer 0 Channel 1
#define CYGNUM_HAL_INTERRUPT_GPT0CH2     14      // General Purpose Timer 0 Channel 2
#define CYGNUM_HAL_INTERRUPT_SWIIRQ1     15      // Software Interrupt 1
```

```
#define CYGNUM_HAL_INTERRUPT_SWIIRQ2     16      // Software Interrupt 2
#define CYGNUM_HAL_INTERRUPT_SWIIRQ3     17      // Software Interrupt 3
#define CYGNUM_HAL_INTERRUPT_GPT1CH0     18      // General Purpose Timer 1 Channel 0
#define CYGNUM_HAL_INTERRUPT_PWM         19      // PWM Controller
#define CYGNUM_HAL_INTERRUPT_CAN0        20      // CAN Controller 0
#define CYGNUM_HAL_INTERRUPT_UPIO        21      // Unified Parallel IO Controller
#define CYGNUM_HAL_INTERRUPT_CAPT0       22      // Capture 0
#define CYGNUM_HAL_INTERRUPT_CAPT1       23      // Capture 1
#define CYGNUM_HAL_INTERRUPT_ST0         24      // Simple Timer 0
#define CYGNUM_HAL_INTERRUPT_ST1         25      // Simple Timer 1
#define CYGNUM_HAL_INTERRUPT_SWIIRQ4     26      // Software Interrupt 4
#define CYGNUM_HAL_INTERRUPT_SWIIRQ5     27      // Software Interrupt 5
#define CYGNUM_HAL_INTERRUPT_IRQ0        28      // External Interrupt 0
#define CYGNUM_HAL_INTERRUPT_IRQ1        29      // External Interrupt 1
#define CYGNUM_HAL_INTERRUPT_SWIIRQ6     30      // Software Interrupt 6
#define CYGNUM_HAL_INTERRUPT_SWIIRQ7     31      // Software Interrupt 7
```

Chapter 237. Atmel AT91SAM7A2-EK Board Support

Name

eCos Support for the Atmel AT91SAM7A2-EK — Overview

Description

This document covers the configuration and usage of eCos and RedBoot on the Atmel AT91SAM7A2-EK Evaluation Kit. The AT91SAM7A2-EK Evaluation Kit contains the AT91SAM7A2 processor, 512KiB of SRAM, 2MiB of parallel NOR Flash memory, external connections for a singe serial channel, CAN and LIN ports. eCos support for the devices and peripherals on the AT91SAM7A2 is described below.

For typical eCos development, a RedBoot image is programmed into the flash memory, and the board will boot into this image from reset. RedBoot provides gdb stub functionality so it is then possible to download and debug stand-alone and eCos applications via the gdb debugger. Applications may also be loaded into RAM via a JTAG debugger, or may be programmed into flash.

This documentation is expected to be read in conjunction with the AT91 processor HAL and AT91SAM7 variant HAL documentation and further device support and subsystems are described and documented there.

Supported Hardware

The parallel NOR flash memory consists of 31 blocks of 16KiBytes each, followed by 8 blocks of 8KiBytes each. In a typical setup, the first 128 KiBytes are reserved for the use of the RedBoot image. The topmost 8 blocks are used to manage the flash and hold RedBoot **fconfig** values. The remaining blocks can be used by application code. There are 30 blocks available between 0x40020000 and 0x401EFFFF.

There is a serial driver `CYGPKG_IO_SERIAL_ARM_AT91` which supports both the Debug Unit and USART serial devices. The debug serial port can be used for communication. If RedBoot is installed, it uses the Debug Unit serial device. The serial driver package is loaded automatically when configuring for the AT91SAM7A2-EK target.

There is a watchdog driver `CYGPKG_DEVICES_WATCHDOG_ARM_AT91`. This driver is also loaded automatically when configuring for the AT91SAM7A2-EK target.

In general, devices (PIO, UARTs, etc.) are initialized only as far as is necessary for eCos to run. Other devices (RTC, I²C, SPI etc.) are not touched unless the appropriate driver is loaded, although in some cases, the HAL boot sequence will set up the appropriate PIO configuration.

Tools

The AT91SAM7A2-EK support is intended to work with GNU tools configured for an arm-elf target. The original port was undertaken using arm-elf-gcc version 3.4.4, arm-elf-gdb version 6.3, and binutils version 2.15.

Name

Setup — Preparing the AT91SAM7A2-EK board for eCos Development

Overview

In a typical development environment, the AT91SAM7A2-EK boards boot from the parallel NOR Flash and run the RedBoot ROM monitor directly. eCos applications are configured for RAM startup and then downloaded and run on the board via the debugger **arm-elf-gdb**. Preparing the board therefore usually involves programming a suitable RedBoot image into flash memory.

The following RedBoot configurations are supported:

Configuration	Description	Use	File
ROM	RedBoot running from ROM	`redboot_ROM.ecm`	`redboot_ROM.bin`
RAM	RedBoot running from RAM	`redboot_RAM.ecm`	`redboot_RAM.bin`
JTAG	RedBoot running from RAM, loaded via JTAG	`redboot_JTAG.ecm`	`redboot_JTAG.bin`

For serial communications, all versions run with 8 bits, no parity, and 1 stop bit at 38400 baud.

The ROM version is programmed into flash to boot the system; however, it is not able to reprogram the RedBoot image in flash. The RAM version is provided to allow for updating the resident RedBoot image in Flash. The JTAG version is only used if loading RedBoot into RAM via a JTAG debugger or ICE. The ELF format image of this JTAG version of RedBoot can be loaded and executed from GDB using the Abatron BDI2000 bdiGDB support, to allow it to be debugged.

Initial Installation

For typical eCos development, a RedBoot image is programmed into the flash memory, and the board will boot into this image from reset. Two mechanisms are described below to program RedBoot into Flash. Both of them require a JTAG device. In the following documentation it is assumed that the Abatron BDI2000 is being used. For a different JTAG device, equivalent operations will need to be performed.

Preparing the Abatron BDI2000 JTAG debugger

The BDI2000 must first be configured to allow communication with your local network, and configured with the parameters for interfacing with the target board. The following steps should be followed:

1. Prepare a PC to act as a host PC and start a TFTP server on it.
2. Connect the Abatron BDI2000 JTAG debugger via both serial and ethernet to the host PC and power it on. Use the serial cable supplied with the BDI2000.
3. Install the Abatron BDI2000 bdiGDB support software on the host PC.
4. Locate the file `bdi2000.at91sam7a2ek.cfg` within the eCos platform HAL package in the source repository. This will be in the directory `packages/hal/arm/at91/at91sam7a2ek/`*`VERSION`*`/misc` relative to the root of your eCos installation.
5. Locate the file `reg920t.def` within the installation of the BDI2000 bdiGDB support software.
6. Place the bdi2000.at91sam7a2ek.cfg in a location on the PC accessible to the TFTP server. Later you will configure the BDI2000 to load this file via TFTP as its configuration file.
7. Similarly place the file `reg920t.def` in a location accessible to the TFTP server.

8. Open `bdi2000.at91sam7a2ek.cfg` in an editor such as emacs or notepad and if necessary adjust the path of the `reg920t.def` file in the `[REGS]` section to match its location relative to the TFTP server root.

9. Install and configure the Abatron BDI2000 in line with the bdiGDB instruction manual. Configure the BDI2000 to use the `bdi2000.at91sam7a2ek.cfg` configuration file at the appropriate point of this process.

Preparing the AT91SAM7A2-EK board for programming

Follow the steps in this section in order to allow communication between the board and the host PC, and between the board and the JTAG device.

1. First you must connect a null modem DB9 serial cable between the Serial Debug Port on the board and a serial port on the host computer.

2. Start a suitable terminal emulator on the host computer such as **minicom** or HyperTerminal. Set the communication parameters to 38400 baud, 8 data bits, no parity bit and 1 stop bit with no flow control.

3. Connect the board to the BDI2000 using a 20-pin ARM/Xscale cable from the JTAG interface connector to the Target A port on the BDI2000.

4. Power up the AT91SAM7A2-EK board.

5. Connect to the BDI2000's CLI interface via TCP/IP on the standard telnet port 23. The **telnet** application is suitable for this. You should see usage information followed by the prompt:

   ```
   SAM7A2>
   ```

6. Confirm correct connection with the BDI2000 with the **reset halt** command as follows:

   ```
   SAM7A2> reset halt
   - TARGET: processing reset request
   - TARGET: BDI asserts TRST and RESET
   - TARGET: BDI removes TRST
   - TARGET: Bypass check 0x000000001 => 0x00000001
   - TARGET: JTAG exists check passed
   - Core#0: ID code is 0x1F0F0F0F
   - TARGET: All ICEBreaker access checks passed
   - TARGET: BDI removes RESET
   - TARGET: BDI waits for RESET inactive
   - TARGET: resetting target passed
   - TARGET: processing target startup ....
   - TARGET: processing target startup passed
   SAM7A2>
   ```

7. Locate the `redboot_ROM.bin` image within the `loaders` subdirectory of the base of the eCos installation.

8. Copy the `redboot_ROM.bin` file into a location on the host computer accessible to its TFTP server.

Using the BDI2000 to directly program RedBoot into Flash

As previously mentioned, there are two methods of programming a RedBoot image into the parallel NOR Flash. This method uses the built-in capabilities of the BDI2000.

This is a three stage process. The relevant Flash blocks must first be unlocked, then erased, and finally programmed. This can be accomplished with the following steps:

1. Connect to the BDI2000 telnet port as before.

2. Erase the 8 initial 8Kbyte sized Flash blocks, and the following 64Kbyte Flash block with the following command:

   ```
   SAM7A>erase
   ```

```
Erasing flash at 0x40000000
Erasing flash at 0x40002000
Erasing flash at 0x40004000
Erasing flash at 0x40006000
Erasing flash at 0x40008000
Erasing flash at 0x4000a000
Erasing flash at 0x4000c000
Erasing flash at 0x4000e000
Erasing flash at 0x40010000
Erasing flash passed
SAM7A>
```

3. Program the RedBoot image into Flash with the following command, replacing */RBPATH* with the location of the redboot_ROM.bin file relative to the TFTP server root directory:

   ```
   SAM7A>prog 0x40000000 /RBPATH/redboot_ROM.bin bin
   Programming redboot_ROM.bin , please wait ....
   Programming flash passed
   SAM7A>
   ```

   This operation can take some time.

The RedBoot installation is now complete. This can be tested by powering off the board, disconnecting the JTAG, and then powering on the board again. The RedBoot banner should be visible on the serial port. RedBoot's Flash configuration can be initialized using the same procedure as required in Method 2 below.

If it proves necessary to re-install RedBoot, this may be achieved by repeating the above process. Alternatively, a new image may be downloaded and programmed into flash more directly using RedBoot's own commands. See the RedBoot documentation for details.

Method 2 - Program RedBoot into Flash with RAM RedBoot

With this approach, the BDI2000 is used to load a RAM RedBoot image, which can then in turn be used to load and program a ROMRAM RedBoot image into Flash.

There are three stages, firstly loading the RAM RedBoot image, then initializing RedBoot's Flash configuration, and finally loading and programming the ROMRAM RedBoot.

Loading a RAM RedBoot

1. Locate the `redboot_JTAG.bin` image within the `loaders` subdirectory of the base of the eCos installation.

2. Copy the `redboot_JTAG.bin` file into a location on the host computer accessible to its TFTP server.

3. With the BDI2000 telnet interface, execute the following command, replacing */RBPATH* with the location of the redboot_JTAG.bin file relative to the TFTP server root directory:

   ```
   SAM7A>load 0x48000000 /RBPATH/redboot_JTAG.bin bin
   Loading /RBPATH/redboot_JTAG.bin , please wait ....
   Loading program file passed
   SAM7A>
   ```

4. Run the loaded RAM RedBoot:

   ```
   SAM7A>go 0x48000040
   SAM7A>
   ```

The terminal emulator connected to the serial debug port should now have displayed the RedBoot banner and prompt similar to the following:

```
**Warning** FLASH configuration checksum error or invalid key
Use 'fconfig -i' to [re]initialize database
```

```
RedBoot(tm) bootstrap and debug environment [JTAG]
Non-certified release, version UNKNOWN - built 11:03:18, Oct 27 2006

Platform: Atmel AT91SAM7A2-EK (ARM7TDMI)
Copyright (C) 2000, 2001, 2002, 2003, 2004 Free Software Foundation, Inc.
Copyright (C) 2003, 2004, 2005, 2006 eCosCentric Limited

RAM: 0x48000000-0x48040000, [0x48017078-0x4802d000] available
FLASH: 0x40000000-0x401fffff, 31 x 0x10000 blocks, 8 x 0x2000 blocks
RedBoot>
```

Note

It is also possible to use the RAM startup version of RedBoot and the redboot_RAM.bin file instead of redboot_JTAG.bin above. If so, then the address to the **load** command must be 0x48008000, and the start address given to the **go** command should be 0x48008040.

RedBoot Flash configuration

The following steps describe how to initialize RedBoot's Flash configuration. This must be performed when using a RAM RedBoot to program Flash, but is also applicable to initial configuration of a ROM RedBoot loaded using Method 1.

1. Use the following command to initialize RedBoot's Flash Information System (FIS):

   ```
   RedBoot> fis init -f
   About to initialize [format] FLASH image system - continue (y/n)? y
   *** Initialize FLASH Image System
   ... Erase from 0x40020000-0x401effff: ............................
   ... Erase from 0x401f0000-0x401fffff: ........
   ... Program from 0x48030000-0x48040000 to 0x401f0000: ........
   RedBoot>
   ```

2. Now configure RedBoot's Flash configuration with the command:

   ```
   RedBoot> fconfig -i
   ```

   The following gives an example configuration:

   ```
   RedBoot> fconfig -i
   Initialize non-volatile configuration - continue (y/n)? y
   ... Erase from 0x401f0000-0x401fffff: ........
   ... Program from 0x48030000-0x48040000 to 0x401f0000: ........
   RedBoot>
   ```

Loading and programming the ROM RedBoot

This section describes the steps required to load the ROM RedBoot from the TFTP server and program it into Flash.

1. Load the RedBoot ROM binary image using Y-Modem protocol over the serial line. First give this command to RedBoot:

   ```
   RedBoot> load -r -m y -b %{freememlo}
   C
   RedBoot>
   ```

   Use the Y-Modem protocol support of your terminal emulator to send the `redboot_ROM.bin` file at this point. When the transfer is finished RedBoot will report:

   ```
   Raw file loaded 0x48017400-0x4802abd3, assumed entry at 0x48017400
   xyzModem - CRC mode, 627(SOH)/0(STX)/0(CAN) packets, 8 retries
   RedBoot>
   ```

2. Finally install the loaded image into Flash:

```
RedBoot> fis create RedBoot
An image named 'RedBoot' exists - continue (y/n)? y
... Erase from 0x40000000-0x4001ffff: .........
... Program from 0x48017400-0x4802abd4 to 0x40000000: .........
... Erase from 0x401f0000-0x401fffff: .
... Program from 0x48030000-0x48040000 to 0x401f0000: .
RedBoot>
```

The RedBoot installation is now complete. This can be tested by powering off the board, disconnecting the JTAG, and then powering on the board again. Output similar to the following should be seen on the serial port.

```
+
RedBoot(tm) bootstrap and debug environment [ROM]
Non-certified release, version UNKNOWN - built 11:41:25, Oct 27 2006

Platform: Atmel AT91SAM7A2-EK (ARM7TDMI)
Copyright (C) 2000, 2001, 2002, 2003, 2004 Free Software Foundation, Inc.
Copyright (C) 2003, 2004, 2005, 2006 eCosCentric Limited

RAM: 0x48000000-0x48040000, [0x48005150-0x4802d000] available
FLASH: 0x40000000-0x401fffff, 8 x 0x2000 blocks, 31 x 0x10000 blocks
RedBoot>
```

If it proves necessary to re-install RedBoot, this may be achieved by repeating the above process. Alternatively, a new image may be downloaded and programmed into flash more directly using RedBoot's own commands. See the RedBoot documentation for details.

Name

Configuration — Platform-specific Configuration Options

Overview

The AT91SAM7A2-EK platform HAL package is loaded automatically when eCos is configured for the `at91sam7a2ek` target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The platform HAL package supports three separate startup types:

RAM
: This is the startup type which is normally used during application development. The board has RedBoot programmed into flash and boots into that initially. arm-elf-gdb is then used to load a RAM startup application into memory and debug it. It is assumed that the hardware has already been initialized by the stubs. By default the application will use the eCos virtual vectors mechanism to obtain certain services from the stubs, including diagnostic output.

ROM
: This startup type can be used for finished applications which will be programmed into flash at physical address 0x40000000. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization.

JTAG
: This is an alternative development startup type. The application is loaded into RAM via a JTAG device and is run and debugged from there. The application will be self-contained with no dependencies on services provided by other software. It is expected that hardware setup will have been performed via the JTAG device prior to loading.

RedBoot and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building the GDB Stubs.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then the serial port will be claimed for HAL diagnostics.

Flash Driver

The AT91SAM7A2-EK board contains a quantity of on-chip flash memory. The `CYGPKG_DEVS_FLASH_AMD_AM29XXXXX_V2` package contains the code necessary to support this part and the platform HAL package contains definitions necessary to support this part. This driver is not active until the generic Flash support package, `CYGPKG_IO_FLASH`, is included in the configuration.

Watchdog Driver

The AT91SAM7A2-EK board use the AT91SAM7A2's internal watchdog support. The `CYGPKG_DEVICES_WATCHDOG_ARM_AT91` package contains all the code necessary to support this device. Within that package the `CYGNUM_DEVS_WATCHDOG_ARM_AT91_DESIRED_TIMEOUT_MS` configuration option controls the watchdog timeout, and by default will force a reset of the board upon timeout. This driver is not active until the generic watchdog device support package, `CYGPKG_IO_WATCHDOG`, is included in the configuration.

USART Serial Driver

The AT91SAM7A2-EK board use the AT91SAM7A2's internal USART serial support as described in the AT91 processor HAL documentation. One serial ports is available: USART 0 which is mapped to virtual vector channel 0 and "`/dev/ser0`". USART 0 does not support modem control signals such as those used for hardware flow control.

Compiler Flags

The SAM7 variant HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos. There are just three flags specific to this port:

`-mcpu=arm7tdmi`
: The arm-elf-gcc compiler supports many variants of the ARM architecture. A `-m` option should be used to select the specific variant in use, and with current tools `-mcpu=arm7tdmi` is the correct option for the ARM7TDMI processor in the SAM7A2.

`-mthumb`
: The arm-elf-gcc compiler will compile C and C++ files into the Thumb instruction set when this option is used. The best way to build eCos in Thumb mode is to enable the configuration option `CYGHWR_THUMB`.

`-mthumb-interwork`
: This option allows programs to be created that mix ARM and Thumb instruction sets. Without this option, some memory can be saved. This option should be used if -mthumb is used. The best way to build eCos with Thumb interworking is to enable the configuration option `CYGBLD_ARM_ENABLE_THUMB_INTERWORK`.

Name

JTAG support — Usage

Use of JTAG for debugging

JTAG can be used to single-step and debug loaded RAM applications, or even applications resident in ROM.

Debugging of ROM applications is only possible if using hardware breakpoints. The ARM7TDMI core of the AT91SAM7A2 only supports two such hardware breakpoints, and so they should be used sparingly. If using a GDB front-end such as Eclipse, check it has not set unnecessary extra breakpoints. Some JTAG devices give the option of whether to set hardware or software breakpoints by default. Be sure to configure your device appropriately.

Abatron BDI2000 notes

On the Abatron BDI2000, the `bdi2000.at91sam7a2ek.cfg` file should be used to setup and configure the hardware to an appropriate state to load programs. This includes setting up the PLL and flash memory controller.

The `bdi2000.at91sam7a2ek.cfg` file also contains an option to define whether hardware or software breakpoints are used by default, using the `BREAKMODE` directive in the `[TARGET]` section. Edit this file if you wish to use software break points, and remember to use the **boot** command on the BDI2000 command line interface to make the changes take effect.

On the BDI2000, debugging can be performed either via the telnet interface or using **arm-elf-gdb** and the bdiGDB interface. In the case of the latter, **arm-elf-gdb** needs to connect to TCP port 2001 on the BDI2000's IP address. For example:

```
(gdb) target remote 111.222.333.444:2001
```

By default when the BDI2000 is powered up, the target will always run the initialization section of the `bdi2000.at91sam7a2ek.cfg` file (which configures the SDRAM among other things), and halts the target. This behaviour is repeated with the **reset halt** command.

If the board is reset when in '**reset halt**' mode (either with the '**reset halt**' or '**reset**' commands, or by pressing the reset button) and the '**go**' command is then given, then the board will boot from ROM as normal.

It is also possible for the target to always run, without initialization, after the reset button has been pressed. This mode is selected with the **reset run** command. This conveniently allows the target to be connected to the JTAG debugger, and be able to reset it with the reset button, without being required to always type '**go**' every time. Thereafter, invoking the **reset** command will repeat the previous reset style. Also in this mode, exceptions will be handled by board software, rather than causing the JTAG debugger to halt the CPU.

Suitably configured RAM applications can be loaded either via GDB, or directly via the telnet CLI. For example:

```
SAM7A2>load 0x00201000 /test.bin bin
Loading /test.bin , please wait ....
Loading program file passed
SAM7A2>go 0x00201000
```

Consult the BDI2000 documentation for information on other formats.

Configuration of RAM applications

If the JTAG device has initialized the processor, such as by using the `bdi2000.at91sam7a2ek.cfg` configuration on the BDI2000, applications can be loaded directly into RAM without requiring a ROM monitor. This loading can be done directly through the JTAG device, or where supported by the JTAG device, through GDB.

In order to configure the application to support this mode, some configuration settings are required. Firstly `CYGSEM_HAL_USE_ROM_MONITOR` must be disabled. Secondly the `CYGDBG_HAL_DIAG_TO_DEBUG_CHAN` option should be enabled in order to prevent HAL diagnostic output being encoded into GDB ($O) packets. Selecting the JTAG startup type in the configuration tool sets these options automatically.

Running RAM applications

Once loaded and running via JTAG, HAL diagnostic output will appear by default on the serial port.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the AT91SAM7A2-EK hardware, and should be read in conjunction with that specification. The AT91SAM7A2-EK platform HAL package complements the ARM architectural HAL, the AT91 variant HAL and the AT91SAM7 processor HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset, the HAL will initialize or reinitialize most of the on-chip peripherals. There is an exception for RAM startup applications which depend on a ROM monitor or JTAG device for certain services.

For ROM startup, the HAL will perform additional initialization, setting up the PLL and programming the various internal registers. This is all done in the `PLATFORM_SETUP1` macro in the assembler header file `hal_platform_setup.h`.

Linker Scripts and Memory Maps

The AT91SAM7 processor HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

Flash
This is located at address 0x40000000 of the physical memory space.

RAM
This is located at address 0x48000000 of the physical memory space. During booting this memory is only available at this address, but during the boot process it is also remapped to location 0x00000000 in order to allow the hardware exception vectors to be in RAM. The first 32 bytes are used for hardware exception vectors. The next 32 bytes are used for the VSR table and the next 256 bytes are normally used for the eCos virtual vectors, allowing RAM-based applications to use services provided by the ROM monitor. For ROM startup, all remaining RAM is available. For RAM startup, available RAM starts at location 0x48008000, with the bottom 32KiB reserved for use by RedBoot.

On-chip Peripheral Registers
These are located at address 0xFF000000 in the physical memory space.

Real-time characterization

The tm_basic kernel test gives statistics gathered about the real-time characterization and performance of the kernel. The sample output is shown here for information. The test was built in ARM mode, which provided better performance than Thumb mode.

Example 237.1. at91sam7a2ek Real-time characterization

```
            Startup, main stack : stack used   416 size  3920
             Startup :  Interrupt stack used   148 size  4096
             Startup : Idlethread stack used    88 size  2048

eCos Kernel Timings
Notes: all times are in microseconds (.000001) unless otherwise stated

Reading the hardware clock takes 1 'ticks' overhead
... this value will be factored out of all other measurements
Clock interrupt took   61.00 microseconds (57 raw clock ticks)

Testing parameters:
   Clock samples:            32
   Threads:                  10
   Thread switches:         128
   Mutexes:                  32
   Mailboxes:                32
   Semaphores:               32
   Scheduler operations:    128
   Counters:                 32
   Flags:                    32
   Alarms:                   32


                                   Confidence
     Ave     Min     Max     Var  Ave  Min  Function
  ======  ======  ======  ====== ========== ========
   46.93   43.73   50.13    1.71   40%  30% Create thread
    8.96    8.53    9.60    0.51   60%  60% Yield thread [all suspended]
    9.28    8.53    9.60    0.45   70%  30% Suspend [suspended] thread
   10.03    9.60   10.67    0.51   60%  60% Resume thread
   13.87   13.87   13.87    0.00  100% 100% Set priority
    1.49    1.07    2.13    0.51   60%  60% Get priority
   32.00   32.00   32.00    0.00  100% 100% Kill [suspended] thread
    9.07    8.53    9.60    0.53  100%  50% Yield [no other] thread
   16.96   16.00   17.07    0.19   90%  10% Resume [suspended low prio] thread
    9.60    9.60    9.60    0.00  100% 100% Resume [runnable low prio] thread
   12.80   12.80   12.80    0.00  100% 100% Suspend [runnable] thread
    8.96    8.53    9.60    0.51   60%  60% Yield [only low prio] thread
    9.17    8.53    9.60    0.51   60%  40% Suspend [runnable->not runnable]
   32.00   32.00   32.00    0.00  100% 100% Kill [runnable] thread
   23.15   22.40   23.47    0.45   70%  30% Destroy [dead] thread
   46.51   45.87   46.93    0.51   60%  40% Destroy [runnable] thread
   62.29   60.80   69.33    1.41   60%  90% Resume [high priority] thread
   24.07   23.47   32.00    0.59   50%  49% Thread switch

    1.60    1.07    2.13    0.53  100%  50% Scheduler lock
    6.93    6.40    7.47    0.53  100%  50% Scheduler unlock [0 threads]
    6.93    6.40    7.47    0.53  100%  50% Scheduler unlock [1 suspended]
    6.93    6.40    7.47    0.53  100%  50% Scheduler unlock [many suspended]
    6.93    6.40    7.47    0.53  100%  50% Scheduler unlock [many low prio]

    2.67    2.13    3.20    0.53  100%  50% Init mutex
   10.40    9.60   10.67    0.40   75%  25% Lock [unlocked] mutex
   11.73   11.73   11.73    0.00  100% 100% Unlock [locked] mutex
    9.87    9.60   10.67    0.40   75%  75% Trylock [unlocked] mutex
    8.47    7.47    8.53    0.12   93%   6% Trylock [locked] mutex
    1.07    1.07    1.07    0.00  100% 100% Destroy mutex
   62.57   61.87   62.93    0.48   65%  34% Unlock/Lock mutex

    3.53    3.20    4.27    0.46   68%  68% Create mbox
    1.00    0.00    1.07    0.12   93%   6% Peek [empty] mbox
   11.73   11.73   11.73    0.00  100% 100% Put [first] mbox
    1.00    0.00    1.07    0.12   93%   6% Peek [1 msg] mbox
   11.20   10.67   11.73    0.53  100%  50% Put [second] mbox
    1.00    0.00    1.07    0.12   93%   6% Peek [2 msgs] mbox
   11.67   10.67   11.73    0.12   93%   6% Get [first] mbox
   11.60   10.67   11.73    0.23   87%  12% Get [second] mbox
    9.53    8.53    9.60    0.12   93%   6% Tryput [first] mbox
    8.93    8.53    9.60    0.50   62%  62% Peek item [non-empty] mbox
   10.40    9.60   10.67    0.40   75%  25% Tryget [non-empty] mbox
    8.73    8.53    9.60    0.32   81%  81% Peek item [empty] mbox
    9.13    8.53    9.60    0.52   56%  43% Tryget [empty] mbox
    1.13    1.07    2.13    0.12   93%  93% Waiting to get mbox
    1.13    1.07    2.13    0.12   93%  93% Waiting to put mbox
    3.47    3.20    4.27    0.40   75%  75% Delete mbox
   43.40   42.67   43.73    0.46   68%  31% Put/Get mbox

    2.27    2.13    3.20    0.23   87%  87% Init semaphore
    8.60    8.53    9.60    0.12   93%  93% Post [0] semaphore
    9.60    9.60    9.60    0.00  100% 100% Wait [1] semaphore
    8.00    7.47    8.53    0.53  100%  50% Trywait [0] semaphore
    8.33    7.47    8.53    0.33   81%  18% Trywait [1] semaphore
    2.47    2.13    3.20    0.46   68%  68% Peek semaphore
    1.20    1.07    2.13    0.23   87%  87% Destroy semaphore
   39.17   38.40   39.47    0.43   71%  28% Post/Wait semaphore

    3.73    3.20    4.27    0.53  100%  50% Create counter
    1.33    1.07    2.13    0.40   75%  75% Get counter value
    1.33    1.07    2.13    0.40   75%  75% Set counter value
   10.20    9.60   10.67    0.52   56%  43% Tick counter
    1.27    1.07    2.13    0.32   81%  81% Delete counter

    2.27    2.13    3.20    0.23   87%  87% Init flag
    9.13    8.53    9.60    0.52   56%  43% Destroy flag
    7.93    7.47    8.53    0.52   56%  56% Mask bits in flag
    9.53    8.53    9.60    0.12   93%   6% Set bits in flag [no waiters]
   13.87   13.87   13.87    0.00  100% 100% Wait for flag [AND]
   13.60   12.80   13.87    0.40   75%  25% Wait for flag [OR]
   13.93   13.87   14.93    0.12   93%  93% Wait for flag [AND/CLR]
   13.73   12.80   13.87    0.23   87%  12% Wait for flag [OR/CLR]
    0.93    0.00    1.07    0.23   87%  12% Peek on flag

    5.97    5.33    6.40    0.51   59%  40% Create alarm
   17.60   17.07   18.13    0.53  100%  50% Initialize alarm
    8.23    7.47    8.53    0.43   71%  28% Disable alarm
   16.23   16.00   17.07    0.36   78%  78% Enable alarm
    9.87    9.60   10.67    0.40   75%  75% Delete alarm
   11.87   11.73   12.80    0.23   87%  87% Tick counter [1 alarm]
   74.13   73.60   74.67    0.53  100%  50% Tick counter [many alarms]
   22.67   22.40   23.47    0.40   75%  75% Tick & fire counter [1 alarm]
  452.10  450.13  504.53    3.28   96%  96% Tick & fire counters [>1 together]
   85.70   85.33   86.40    0.48   65%  65% Tick & fire counters [>1 separately]
   58.67   58.67   58.67    0.00  100% 100% Alarm latency [0 threads]
   65.02   58.67   72.53    3.94   42%  30% Alarm latency [2 threads]
   64.89   58.67   72.53    3.99   42%  35% Alarm latency [many threads]
  100.33  100.27  108.80    0.13   99%  99% Alarm -> thread resume latency

   10.68   10.67   11.73    0.00            Clock/interrupt latency

   24.98   21.33  146.13    0.00            Clock DSR latency

  274    232     292  (main stack:   804)  Thread stack used (1360 total)
           All done, main stack : stack used   804 size  3920
            All done :  Interrupt stack used   208 size  4096
            All done : Idlethread stack used   252 size  2048

Timing complete - 30940 ms total

PASS:<Basic timing OK>
EXIT:<done>
```

Other Issues

The AT91SAM7A2-EK platform HAL does not affect the implementation of other parts of the eCos HAL specification. The AT91SAM7 processor HAL, AT91 variant HAL, and the ARM architectural HAL documentation should be consulted for further details.

Chapter 238. Atmel AT91SAM7A3-EK Board Support

Name

eCos Support for the Atmel AT91SAM7A3-EK — Overview

Description

This document covers the configuration and usage of eCos and GDB Stubs on the Atmel AT91SAM7A3-EK Evaluation Kit. The AT91SAM7A3-EK Evaluation Kit contains the AT91SAM7A3 processor, external connections for one serial channel, USB host/device, MMC card. eCos support for the devices and peripherals on the AT91SAM7A3 is described below.

Application development on this board can take one of several approaches. Applications may be loaded into RAM via a JTAG device; however, the application size is limited by the amount of on-chip RAM: 32KiB. Applications may also be loaded into the on-chip flash memory where the RAM limit will only apply to the data portion of the application. Finally, it is possible to program a GDB debugging stub into flash which will then allow applications to be loaded into RAM via the serial port. This allows development to proceed without needing to use a JTAG device, although one will be required to program the debugging stub in the first place, and application size is limited to just 28KiB, since the GDB stub uses the least significant 4KiB.

This documentation is expected to be read in conjunction with the AT91 processor HAL and AT91SAM7 variant HAL documentation and further device support and subsystems are described and documented there.

Supported Hardware

The on-chip NOR flash is organized into 1024 pages of 256 bytes each.

There is a serial driver `CYGPKG_IO_SERIAL_ARM_AT91` which supports the Debug Unit and USART serial devices. The debug serial port at J2 can be used for communication. If the GDB stub ROM is installed, it uses the Debug Unit serial device. The serial driver package is loaded automatically when configuring for the AT91SAM7A3-EK target.

There is a watchdog driver `CYGPKG_DEVICES_WATCHDOG_ARM_AT91WDTC`. This driver is also loaded automatically when configuring for the AT91SAM7A3-EK target.

In general, devices (PIO, UARTs, etc.) are initialized only as far as is necessary for eCos to run. Other devices (RTC, I²C, SPI etc.) are not touched unless the appropriate driver is loaded, although in some cases, the HAL boot sequence will set up the appropriate PIO configuration.

Tools

The AT91SAM7A3-EK support is intended to work with GNU tools configured for an arm-elf target. The original port was undertaken using arm-elf-gcc version 3.4.4, arm-elf-gdb version 6.3, and binutils version 2.15.

Name

Setup — Preparing the AT91SAM7A3-EK board for eCos Development

Overview

eCos applications are either programmed into the on-chip flash, or run from RAM using either a JTAG device or the GDB stubs ROM. To install a flash-resident application, or the GDB stubs requires use of a JTAG device to write to the flash. So, in all cases it is necessary to set up a JTAG device for the board. This document describes how to set up an Abatron BDI2000 and then use it to program an application into the flash.

Initial Installation

Preparing the Abatron BDI2000 JTAG debugger

The BDI2000 must first be configured to allow communication with your local network, and configured with the parameters for interfacing with the target board. The following steps should be followed:

1. Prepare a PC to act as a host PC and start a TFTP server on it.

2. Connect the Abatron BDI2000 JTAG debugger via both serial and ethernet to the host PC and power it on. Use the serial cable supplied with the BDI2000.

3. Install the Abatron BDI2000 bdiGDB support software on the host PC.

4. Locate the file `bdi2000.at91sam7a3ek.cfg` within the eCos platform HAL package in the source repository. This will be in the directory `packages/hal/arm/at91/at91sam7a3ek/`*`VERSION`*`/misc` relative to the root of your eCos installation.

5. Locate the file `reg920t.def` within the installation of the BDI2000 bdiGDB support software.

6. Place the bdi2000.at91sam7a3ek.cfg in a location on the PC accessible to the TFTP server. Later you will configure the BDI2000 to load this file via TFTP as its configuration file.

7. Similarly place the file `reg920t.def` in a location accessible to the TFTP server.

8. Open `bdi2000.at91sam7a3ek.cfg` in an editor such as emacs or notepad and if necessary adjust the path of the `reg920t.def` file in the `[REGS]` section to match its location relative to the TFTP server root.

9. Install and configure the Abatron BDI2000 in line with the bdiGDB instruction manual. Configure the BDI2000 to use the `bdi2000.at91sam7a3ek.cfg` configuration file at the appropriate point of this process.

Preparing the AT91SAM7A3-EK board for programming

Follow the steps in this section in order to allow communication between the board and the host PC, and between the board and the JTAG device.

1. First you must connect a null modem DB9 serial cable between the Serial Debug Port on the board and a serial port on the host computer.

2. Start a suitable terminal emulator on the host computer such as **minicom** or HyperTerminal. Set the communication parameters to 38400 baud, 8 data bits, no parity bit and 1 stop bit with no flow control.

3. Connect the board to the BDI2000 using a 20-pin ARM/Xscale cable from the JTAG interface connector to the Target A port on the BDI2000.

4. Power up the AT91SAM7A3-EK board.

5. Connect to the BDI2000's CLI interface via TCP/IP on the standard telnet port 23. The **telnet** application is suitable for this. You should see usage information followed by the prompt:

   ```
   SAM7A3>
   ```

6. Confirm correct connection with the BDI2000 with the **reset halt** command as follows:

   ```
   SAM7A3> reset halt
   - TARGET: processing reset request
   - TARGET: BDI asserts TRST and RESET
   - TARGET: BDI removes TRST
   - TARGET: Bypass check 0x000000001 => 0x00000001
   - TARGET: JTAG exists check passed
   - Core#0: ID code is 0x3F0F0F0F
   - TARGET: All ICEBreaker access checks passed
   - TARGET: BDI removes RESET
   - TARGET: BDI waits for RESET inactive
   - TARGET: resetting target passed
   - TARGET: processing target startup ....
   - TARGET: processing target startup passed
   SAM7A3>
   ```

Installation into Flash

Installation of an application into the on-chip flash, or the installation of the GDB stubs take exactly the same form:

1. Locate the binary image of the executable to be installed. For the GDB stubs do this by locating the file `gdb.module.bin` within the `loaders` subdirectory of the base of the eCos installation. For applications use **arm-elf-objcopy -O binary** to convert the ELF output of the linker into binary.

2. Copy the file into a location on the host computer accessible to its TFTP server.

3. Connect to the BDI2000 telnet port as before.

4. Give the **unlock** command to ensure that the flash area we want to program is writable:

   ```
   SAM7A3>unlock 0x100000 0x100 256
   Unlocking flash at 0x00100000
   Unlocking flash at 0x00100100
   Unlocking flash at 0x00100200
   ...
   Unlocking flash at 0x0010fe00
   Unlocking flash at 0x0010ff00
   Unlocking flash passed
   SAM7A3>
   ```

   This command unlocks 256 pages, i.e. 64KiB. The number of pages unlocked should match at least the size of the executable to be programmed.

5. Give the **erase** command to clear any previous contents:

   ```
   SAM7A3>erase 0x100000 0x100 256
   Erasing flash at 0x00100000
   Erasing flash at 0x00100100
   Erasing flash at 0x00100200
   ...
   Erasing flash at 0x0010fe00
   Erasing flash at 0x0010ff00
   Erasing flash passed
   SAM7A3>
   ```

As with the **unlock** command, the size of the area erased must be at least the size of the executable to be programmed.

6. Now give the **prog** command to fetch the executable from the TFTP server and program it to the flash.

   ```
   SAM7A3>prog 0x100000 sam7.bin bin
   Programming sam7.bin , please wait ....
   Programming flash passed
   SAM7A3>
   ```

The installation is now complete. This can be tested by powering off the board, disconnecting the JTAG, and then powering on the board again. A ROM based application should start immediately, and any output will be seen on the serial connection. If the GDB stub ROM has been installed, then something similar to the following will be seen on the serial port:

```
+$T050f:cc061000;0d:18082000;#4d
```

Name

Configuration — Platform-specific Configuration Options

Overview

The AT91SAM7A3-EK platform HAL package is loaded automatically when eCos is configured for the `at91sam7a3ek` target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The platform HAL package supports three separate startup types:

RAM
: This is the startup type which is normally used during application development. The board has GDB stubs programmed into flash and boots into that initially. arm-elf-gdb is then used to load a RAM startup application into memory and debug it. It is assumed that the hardware has already been initialized by the stubs. By default the application will use the eCos virtual vectors mechanism to obtain certain services from the stubs, including diagnostic output.

ROM
: This startup type can be used for finished applications which will be programmed into flash at physical address 0x00100000. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization.

JTAG
: This is an alternative development startup type. The application is loaded into RAM via a JTAG device and is run and debugged from there. The application will be self-contained with no dependencies on services provided by other software. It is expected that hardware setup will have been performed via the JTAG device prior to loading.

GDB Stubs and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building the GDB Stubs.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then the serial port will be claimed for HAL diagnostics.

Flash Driver

The AT91SAM7A3-EK board contains a quantity of on-chip flash memory. The `CYGPKG_DEVS_FLASH_AT91` package contains all the code and data definitions necessary to support this part. This driver is not active until the generic Flash support package, `CYGPKG_IO_FLASH`, is included in the configuration.

Watchdog Driver

The AT91SAM7A3-EK board use the AT91SAM7A3's internal watchdog support. The `CYGPKG_DEVICES_WATCHDOG_ARM_AT91WDTC` package contains all the code necessary to support this device. Within that package the `CYGNUM_DEVS_WATCHDOG_ARM_AT91WDTC_DESIRED_TIMEOUT_MS` configuration option controls the watchdog timeout, and by default will force a reset of the board upon timeout. This driver is not active until the generic watchdog device support package, `CYGPKG_IO_WATCHDOG`, is included in the configuration.

Note that on the AT91, the on-chip watchdog peripheral always starts running immediately, and so in configurations that do not include the watchdog driver, it is always disabled via its write-once register. In configurations which include the watchdog driver obviously the watchdog is not disabled otherwise it could not be subsequently re-enabled, and so the application must start and periodically reset the watchdog from the very beginning of execution.

USART Serial Driver

The AT91SAM7A3-EK board use the AT91SAM7A3's internal USART serial support as described in the AT91 processor HAL documentation. Two serial ports are available: the serial debug port which is mapped to virtual vector channel 0 in the HAL diagnostic driver or "`/dev/dbg`" in the interrupt-driven driver; and USART 0 which is mapped to virtual vector channel 1 and "`/dev/ser0`". Only USART 0 supports modem control signals such as those used for hardware flow control.

Compiler Flags

The SAM7 variant HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos. There are just three flags specific to this port:

`-mcpu=arm7tdmi`

The arm-elf-gcc compiler supports many variants of the ARM architecture. A `-m` option should be used to select the specific variant in use, and with current tools `-mcpu=arm7tdmi` is the correct option for the ARM7TDMI processor in the SAM7A3.

`-mthumb`

The arm-elf-gcc compiler will compile C and C++ files into the Thumb instruction set when this option is used. The best way to build eCos in Thumb mode is to enable the configuration option `CYGHWR_THUMB`.

`-mthumb-interwork`

This option allows programs to be created that mix ARM and Thumb instruction sets. Without this option, some memory can be saved. This option should be used if -mthumb is used. The best way to build eCos with Thumb interworking is to enable the configuration option `CYGBLD_ARM_ENABLE_THUMB_INTERWORK`.

Name

JTAG support — Usage

Use of JTAG for debugging

JTAG can be used to single-step and debug loaded RAM applications, or even applications resident in ROM.

Debugging of ROM applications is only possible if using hardware breakpoints. The ARM7TDMI core of the AT91SAM7A3 only supports two such hardware breakpoints, and so they should be used sparingly. If using a GDB front-end such as Eclipse, check it has not set unnecessary extra breakpoints. Some JTAG devices give the option of whether to set hardware or software breakpoints by default. Be sure to configure your device appropriately.

Abatron BDI2000 notes

On the Abatron BDI2000, the `bdi2000.at91sam7a3ek.cfg` file should be used to setup and configure the hardware to an appropriate state to load programs. This includes setting up the PLL and flash memory controller.

The `bdi2000.at91sam7a3ek.cfg` file also contains an option to define whether hardware or software breakpoints are used by default, using the `BREAKMODE` directive in the `[TARGET]` section. Edit this file if you wish to use software break points, and remember to use the **boot** command on the BDI2000 command line interface to make the changes take effect.

On the BDI2000, debugging can be performed either via the telnet interface or using **arm-elf-gdb** and the bdiGDB interface. In the case of the latter, **arm-elf-gdb** needs to connect to TCP port 2001 on the BDI2000's IP address. For example:

```
(gdb) target remote 111.222.333.444:2001
```

By default when the BDI2000 is powered up, the target will always run the initialization section of the `bdi2000.at91sam7a3ek.cfg` file (which configures the SDRAM among other things), and halts the target. This behaviour is repeated with the **reset halt** command.

If the board is reset when in '**reset halt**' mode (either with the '**reset halt**' or '**reset**' commands, or by pressing the reset button) and the '**go**' command is then given, then the board will boot from ROM as normal.

It is also possible for the target to always run, without initialization, after the reset button has been pressed. This mode is selected with the **reset run** command. This conveniently allows the target to be connected to the JTAG debugger, and be able to reset it with the reset button, without being required to always type '**go**' every time. Thereafter, invoking the **reset** command will repeat the previous reset style. Also in this mode, exceptions will be handled by board software, rather than causing the JTAG debugger to halt the CPU.

Suitably configured RAM applications can be loaded either via GDB, or directly via the telnet CLI. For example:

```
SAM7A3>load 0x00201000 /test.bin bin
Loading /test.bin , please wait ....
Loading program file passed
SAM7A3>go 0x00201000
```

Consult the BDI2000 documentation for information on other formats.

Configuration of RAM applications

If the JTAG device has initialized the processor, such as by using the `bdi2000.at91sam7a3ek.cfg` configuration on the BDI2000, applications can be loaded directly into RAM without requiring a ROM monitor. This loading can be done directly through the JTAG device, or where supported by the JTAG device, through GDB.

In order to configure the application to support this mode, some configuration settings are required. Firstly `CYGSEM_HAL_USE_ROM_MONITOR` must be disabled. Secondly the `CYGDBG_HAL_DIAG_TO_DEBUG_CHAN` option should be enabled in order to prevent HAL diagnostic output being encoded into GDB ($O) packets. Selecting the JTAG startup type in the configuration tool sets these options automatically.

Running RAM applications

Once loaded and running via JTAG, HAL diagnostic output will appear by default on the serial debug port. USART 0 can be chosen instead by setting the CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL configuration option in the platform HAL to channel 1.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the AT91SAM7A3-EK hardware, and should be read in conjunction with that specification. The AT91SAM7A3-EK platform HAL package complements the ARM architectural HAL, the AT91 variant HAL and the AT91SAM7 processor HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset, the HAL will initialize or reinitialize most of the on-chip peripherals. There is an exception for RAM startup applications which depend on a ROM monitor or JTAG device for certain services.

For ROM startup, the HAL will perform additional initialization, setting up the PLL and programming the various internal registers. This is all done in the `PLATFORM_SETUP1` macro in the assembler header file `hal_platform_setup.h`.

Linker Scripts and Memory Maps

The AT91SAM7 processor HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

On-chip Flash
This is located at address 0x00100000 of the physical memory space.

On-chip RAM
This is located at address 0x00200000 of the physical memory space. During booting this memory is only available at this address, but during the boot process it is also remapped to location 0x00000000 in order to allow the hardware exception vectors to be in RAM. The first 32 bytes are used for hardware exception vectors. The next 32 bytes are used for the VSR table and the next 256 bytes are normally used for the eCos virtual vectors, allowing RAM-based applications to use services provided by the ROM monitor. For ROM startup, all remaining RAM is available. For RAM startup, available RAM starts at location 0x00201000, with the bottom 4KiB reserved for use by the GDB stubs.

On-chip Peripheral Registers
These are located at address 0xFF000000 in the physical memory space.

Real-time characterization

The tm_basic kernel test gives statistics gathered about the real-time characterization and performance of the kernel. The sample output is shown here for information. The test was built in ARM mode, which provided better performance than Thumb mode.

Example 238.1. at91sam7a3ek Real-time characterization

```
            Startup, main stack : stack used   456 size  3920
             Startup :  Interrupt stack used  4064 size  4096
             Startup : Idlethread stack used   120 size  2048

eCos Kernel Timings
Notes: all times are in microseconds (.000001) unless otherwise stated

Reading the hardware clock takes 3 'ticks' overhead
... this value will be factored out of all other measurements
Clock interrupt took   47.13 microseconds (141 raw clock ticks)

Testing parameters:
   Clock samples:            32
   Threads:                   1
   Thread switches:         128
```

```
 Mutexes:                    32
 Mailboxes:                  21
 Semaphores:                 32
 Scheduler operations:      128
 Counters:                   32
 Flags:                      32
 Alarms:                     32


                               Confidence
   Ave     Min     Max     Var  Ave  Min  Function
======  ======  ======  ====== ========== ========
144.33  144.33  144.33    0.00  100% 100% Create thread
 18.33   18.33   18.33    0.00  100% 100% Yield thread [all suspended]
 14.67   14.67   14.67    0.00  100% 100% Suspend [suspended] thread
 14.67   14.67   14.67    0.00  100% 100% Resume thread
 17.33   17.33   17.33    0.00  100% 100% Set priority
  0.67    0.67    0.67    0.00  100% 100% Get priority
 32.33   32.33   32.33    0.00  100% 100% Kill [suspended] thread
 18.67   18.67   18.67    0.00  100% 100% Yield [no other] thread
 22.67   22.67   22.67    0.00  100% 100% Resume [suspended low prio] thread
 14.67   14.67   14.67    0.00  100% 100% Resume [runnable low prio] thread
 18.00   18.00   18.00    0.00  100% 100% Suspend [runnable] thread
 18.67   18.67   18.67    0.00  100% 100% Yield [only low prio] thread
 14.67   14.67   14.67    0.00  100% 100% Suspend [runnable->not runnable]
 32.00   32.00   32.00    0.00  100% 100% Kill [runnable] thread
 31.33   31.33   31.33    0.00  100% 100% Destroy [dead] thread
 55.67   55.67   55.67    0.00  100% 100% Destroy [runnable] thread
106.67  106.67  106.67    0.00  100% 100% Resume [high priority] thread

  0.85    0.67    1.00    0.16   56%  43% Scheduler lock
 13.31   13.00   13.33    0.04   93%   6% Scheduler unlock [0 threads]
 13.31   13.00   13.33    0.04   93%   6% Scheduler unlock [1 suspended]
 13.31   13.00   13.33    0.04   93%   6% Scheduler unlock [many suspended]
 13.31   13.00   13.33    0.04   93%   6% Scheduler unlock [many low prio]

  2.33    2.33    2.33    0.00  100% 100% Init mutex
 18.58   18.33   18.67    0.13   75%  25% Lock [unlocked] mutex
 20.68   20.67   21.00    0.02   96%  96% Unlock [locked] mutex
 16.58   16.33   16.67    0.13   75%  25% Trylock [unlocked] mutex
 15.96   15.67   16.00    0.07   87%  12% Trylock [locked] mutex
  1.00    1.00    1.00    0.00  100% 100% Destroy mutex
 85.56   85.33   85.67    0.14   68%  31% Unlock/Lock mutex

  3.90    3.67    4.00    0.14   71%  28% Create mbox
  0.29    0.00    0.33    0.08   85%  14% Peek [empty] mbox
 19.90   19.67   20.00    0.14   71%  28% Put [first] mbox
  0.33    0.33    0.33    0.00  100% 100% Peek [1 msg] mbox
 19.92   19.67   20.00    0.12   76%  23% Put [second] mbox
  0.33    0.33    0.33    0.00  100% 100% Peek [2 msgs] mbox
 20.00   20.00   20.00    0.00  100% 100% Get [first] mbox
 20.00   20.00   20.00    0.00  100% 100% Get [second] mbox
 18.92   18.67   19.00    0.12   76%  23% Tryput [first] mbox
 16.49   16.33   16.67    0.17   52%  52% Peek item [non-empty] mbox
 17.49   17.33   17.67    0.17   52%  52% Tryget [non-empty] mbox
 16.33   16.33   16.33    0.00  100% 100% Peek item [empty] mbox
 16.56   16.33   16.67    0.15   66%  33% Tryget [empty] mbox
  0.40    0.33    0.67    0.10   80%  80% Waiting to get mbox
  0.40    0.33    0.67    0.10   80%  80% Waiting to put mbox
  2.16    2.00    2.33    0.17   52%  52% Delete mbox
 68.73   68.67   69.00    0.10   80%  80% Put/Get mbox

  2.33    2.33    2.33    0.00  100% 100% Init semaphore
 14.00   14.00   14.00    0.00  100% 100% Post [0] semaphore
 14.50   14.33   14.67    0.17  100%  50% Wait [1] semaphore
 13.96   13.67   14.00    0.07   87%  12% Trywait [0] semaphore
 14.00   14.00   14.00    0.00  100% 100% Trywait [1] semaphore
```

```
    2.38    2.33    2.67    0.07   87%  87% Peek semaphore
    0.83    0.67    1.00    0.17  100%  50% Destroy semaphore
   58.07   58.00   58.33    0.11   78%  78% Post/Wait semaphore

    4.00    4.00    4.00    0.00  100% 100% Create counter
    0.46    0.33    0.67    0.16   62%  62% Get counter value
    0.46    0.33    0.67    0.16   62%  62% Set counter value
   15.46   15.33   15.67    0.16   62%  62% Tick counter
    0.50    0.33    0.67    0.17  100%  50% Delete counter

    2.25    2.00    2.33    0.13   75%  25% Init flag
   14.33   14.33   14.33    0.00  100% 100% Destroy flag
   13.75   13.67   14.00    0.13   75%  75% Mask bits in flag
   15.67   15.67   15.67    0.00  100% 100% Set bits in flag [no waiters]
   19.83   19.67   20.00    0.17  100%  50% Wait for flag [AND]
   19.58   19.33   19.67    0.13   75%  25% Wait for flag [OR]
   19.83   19.67   20.00    0.17  100%  50% Wait for flag [AND/CLR]
   19.58   19.33   19.67    0.13   75%  25% Wait for flag [OR/CLR]
    0.25    0.00    0.33    0.13   75%  25% Peek on flag

    5.27    5.00    5.33    0.10   81%  18% Create alarm
   20.54   20.33   20.67    0.16   62%  37% Initialize alarm
   13.90   13.67   14.00    0.14   68%  31% Disable alarm
   19.85   19.67   20.00    0.16   56%  43% Enable alarm
   14.65   14.33   14.67    0.04   93%   6% Delete alarm
   17.42   17.33   17.67    0.13   75%  75% Tick counter [1 alarm]
   82.65   82.33   82.67    0.04   93%   6% Tick counter [many alarms]
   26.08   26.00   26.33    0.13   75%  75% Tick & fire counter [1 alarm]
  371.00  369.67  402.67    1.98   96%  96% Tick & fire counters [>1 together]
   91.63   91.33   91.67    0.07   87%  12% Tick & fire counters [>1 separately]
   38.33   38.33   38.33    0.00  100% 100% Alarm latency [0 threads]
   40.89   38.33   44.67    2.44   27%  47% Alarm latency [many threads]
   93.75   93.67  104.67    0.17   99%  99% Alarm -> thread resume latency

    6.66    6.33    6.67    0.00            Clock/interrupt latency

   14.37   13.00  261.33    0.00            Clock DSR latency

 1360   1360    1360  (main stack:  3920)  Thread stack used (1360 total)
           All done, main stack : stack used   896 size  3920
            All done :  Interrupt stack used   212 size  4096
            All done : Idlethread stack used   304 size  2048

Timing complete - 23730 ms total

PASS:<Basic timing OK>
EXIT:<done>
```

Other Issues

The AT91SAM7A3-EK platform HAL does not affect the implementation of other parts of the eCos HAL specification. The AT91SAM7 processor HAL, AT91 variant HAL, and the ARM architectural HAL documentation should be consulted for further details.

Chapter 239. Atmel AT91SAM7S-EK Board Support

Name

eCos Support for the Atmel AT91SAM7S-EK — Overview

Description

This document covers the configuration and usage of eCos and GDB Stubs on the Atmel AT91SAM7S-EK Evaluation Kit. The AT91SAM7S-EK Evaluation Kit contains the AT91SAM7S processor, external connections for two serial channels (one debug, one full), USB host/device. eCos support for the devices and peripherals on the AT91SAM7S is described below.

Application development on this board can take one of several approaches. Applications may be loaded into RAM via a JTAG device; however, the application size is limited by the amount of on-chip RAM, which is 64KiB on the AT91SAM7S256 and AT91SAM7S512, and 32KiB on the AT91SAM7S128. Applications may also be loaded into the on-chip flash memory where the RAM limit will only apply to the data portion of the application. Finally, it is possible to program a GDB debugging stub into flash which will then allow applications to be loaded into RAM via the serial port. This allows development to proceed without needing to use a JTAG device and application size is limited to the RAM size less 4KiB used by the stub. It is therefore recommended that JTAG debugging be used to debug applications since memory is limited on this platform.

This documentation is expected to be read in conjunction with the AT91 processor HAL and AT91SAM7 variant HAL documentation and further device support and subsystems are described and documented there.

Supported Hardware

The on-chip NOR flash is organized into pages of 128 or 256 bytes each. The number of pages is determined by the device variant, from 2048 for the AT91SAM7S512 to 128 for the AT91SAM7S32.

There is a serial driver `CYGPKG_IO_SERIAL_ARM_AT91` which supports both the Debug Unit and USART serial devices. The debug serial port at J3 and DTE port at J2 (connected to USART channel 0) can be used for communication. If the GDB stub ROM is installed, it uses the Debug Unit serial device only. The serial driver package is loaded automatically when configuring for the AT91SAM7S-EK target.

There is a watchdog driver `CYGPKG_DEVICES_WATCHDOG_ARM_AT91WDTC`. This driver is also loaded automatically when configuring for the AT91SAM7S-EK target.

In general, devices (PIO, UARTs, etc.) are initialized only as far as is necessary for eCos to run. Other devices (RTC, I²C, SPI etc.) are not touched unless the appropriate driver is loaded, although in some cases, the HAL boot sequence will set up the appropriate PIO configuration.

Tools

The AT91SAM7S-EK support is intended to work with GNU tools configured for an arm-eabi target. The original port was undertaken using arm-elf-gcc version 3.4.4, arm-elf-gdb version 6.3, and binutils version 2.15.

Name

Setup — Preparing the AT91SAM7S-EK board for eCos Development

Overview

eCos applications are either programmed into the on-chip flash, or run from RAM using either a JTAG device or the GDB stubs ROM. The installation of the GDB stubs or any flash-resident application requires use of a JTAG device to write to the flash, or the Atmel-supplied SAM-BA program that interacts with the on-chip boot program. This document describes how to set up an Abatron BDI2000 Ronetix PEEDI for programming the gdb stubs and applications into the flash, and use the Atmel SAM-BA application to program the gdb stubs into the flash.

Initial Installation with Abatron BDI2000

Preparing the Abatron BDI2000 JTAG debugger

The BDI2000 must first be configured to allow communication with your local network, and configured with the parameters for interfacing with the target board. The following steps should be followed:

1. Prepare a PC to act as a host PC and start a TFTP server on it.
2. Connect the Abatron BDI2000 JTAG debugger via both serial and ethernet to the host PC and power it on. Use the serial cable supplied with the BDI2000.
3. Install the Abatron BDI2000 bdiGDB support software on the host PC.
4. Locate the file `bdi2000.at91sam7sek.cfg` within the eCos platform HAL package in the source repository. This will be in the directory `packages/hal/arm/at91/at91sam7sek/`*`VERSION`*`/misc` relative to the root of your eCos installation.
5. Locate the file `regSAM7S.def` within the installation of the BDI2000 bdiGDB support software.
6. Place the `bdi2000.at91sam7sek.cfg` file in a location on the PC accessible to the TFTP server. Later you will configure the BDI2000 to load this file via TFTP as its configuration file.
7. Similarly place the file `regSAM7S.def` in a location accessible to the TFTP server.
8. Open `bdi2000.at91sam7sek.cfg` in an editor such as emacs or notepad and if necessary adjust the path of the `regSAM7S.def` file in the `[REGS]` section to match its location relative to the TFTP server root.
9. Install and configure the Abatron BDI2000 in line with the bdiGDB instruction manual. Configure the BDI2000 to use the `bdi2000.at91sam7sek.cfg` configuration file at the appropriate point of this process.

Preparing the AT91SAM7S-EK board for programming with BDI2000

Follow the steps in this section in order to allow communication between the board and the host PC, and between the board and the JTAG device.

1. First you must connect a null modem DB9 serial cable between the Serial Debug Port on the board and a serial port on the host computer.
2. Start a suitable terminal emulator on the host computer such as **minicom** or HyperTerminal. Set the communication parameters to 38400 baud, 8 data bits, no parity bit and 1 stop bit with no flow control.
3. Connect the board to the BDI2000 using a 20-pin ARM/Xscale cable from the JTAG interface connector to the Target A port on the BDI2000.

4. Power up the AT91SAM7S-EK board.

5. Connect to the BDI2000's CLI interface via TCP/IP on the standard telnet port 23. The **telnet** application is suitable for this. You should see usage information followed by the prompt:

   ```
   SAM7S>
   ```

6. Confirm correct connection with the BDI2000 with the **reset halt** command as follows:

   ```
   SAM7S> reset halt
   - TARGET: processing reset request
   - TARGET: BDI asserts TRST and RESET
   - TARGET: BDI removes TRST
   - TARGET: Bypass check 0x000000001 => 0x00000001
   - TARGET: JTAG exists check passed
   - Core#0: ID code is 0x3F0F0F0F
   - TARGET: All ICEBreaker access checks passed
   - TARGET: BDI removes RESET
   - TARGET: BDI waits for RESET inactive
   - TARGET: resetting target passed
   - TARGET: processing target startup ....
   - TARGET: processing target startup passed
   SAM7S>
   ```

Initial Installation with Ronetix PEEDI

Preparing the Ronetix PEEDI JTAG debugger

The PEEDI must first be configured to allow communication with your local network, and configured with the parameters for interfacing with the target board. The following steps give a typical outline of setting up the PEEDI using TFTP. Consult the PEEDI documentation for alternative mechanisms.

1. Prepare a PC to act as a host PC and start a TFTP server on it.

2. Connect the PEEDI JTAG debugger via both serial and ethernet to the host PC and power it on. Use the serial cable supplied with the PEEDI (straight through, not null modem).

3. Locate the file `peedi.at91sam7sek.cfg` within the eCos platform HAL package in the source repository. This will be in the directory `packages/hal/arm/at91/at91sam7sek/`*`VERSION`*`/misc` relative to the root of your eCos installation.

4. Place the `peedi.at91sam7sek.cfg` file in a location on the PC accessible to the TFTP server. Later you will configure the PEEDI to load this file via TFTP as its configuration file.

5. Open `at91sam7sek.cfg` in an editor such as emacs or notepad and insert your own license information in the `[LICENSE]` section.

6. Install and configure the PEEDI in line with the PEEDI Quick Start Guide or User's Manual, especially configuring PEEDI's RedBoot with the network information. Configure it to use the `peedi.at91sam7sek.cfg` target configuration file on the TFTP server at the appropriate point of the **fconfig** process, for example with a path such as: `tftp://192.168.7.9/peedi.at91sam7sek.cfg`

7. Reset the PEEDI.

8. Connect to the PEEDI's CLI interface via TCP/IP on the standard telnet port 23. The **telnet** application is suitable for this. You should see output similar to the following:

   ```
   $ telnet 192.168.7.225
   Trying 192.168.7.225...
   Connected to 192.168.7.225.
   Escape character is '^]'.
   ```

```
PEEDI - Powerful Embedded Ethernet Debug Interface
Copyright (c) 2005-2007 www.ronetix.at - All rights reserved
Hw:1.2, Fw:2.0.13, SN: PD-0000-XXXX-XXXX
------------------------------------------------------------

sam7sek>
```

Preparing the AT91SAM7S-EK board for programming with PEEDI

Follow the steps in this section in order to allow communication between the board and the host PC, and between the board and the JTAG device.

1. First you must connect a null modem DB9 serial cable between the Serial Debug Port on the board and a serial port on the host computer.

2. Start a suitable terminal emulator on the host computer such as **minicom** or HyperTerminal. Set the communication parameters to 38400 baud, 8 data bits, no parity bit and 1 stop bit with no flow control.

3. Connect the board to the PEEDI using a 20-pin ARM/Xscale cable from the JTAG interface connector on the board to the Target port on the PEEDI.

4. Power up the AT91SAM7S-EK board.

5. Connect to the PEEDI's CLI on port 23 as before.

6. Confirm correct connection with the PEEDI with the **reset** command as follows:

   ```
   sam7sek> reset
   ++ info: user reset
   sam7sek>
   ++ info: RESET and TRST asserted
   ++ info: TRST released
   ++ info: 1 TAP controller(s) detected
   ++ info: TAP : IDCODE = 0x3F0F0F0F, ARM7TDMI compliant
   ++ info: RESET released
   ++ info: core 0: initialized

   sam7sek>
   ```

Installation into Flash

Installation of an application into the on-chip flash, or the installation of the GDB stubs, using a JTAG programmer takes exactly the same form:

1. Locate the binary image of the executable to be installed. For the GDB stubs do this by locating the file `gdb_module.bin` within the `loaders` subdirectory of the base of the eCos installation. For applications use **arm-eabi-objcopy -O binary** to convert the ELF output of the linker into binary.

2. Copy the file into a location on the host computer accessible to its TFTP server.

3. Connect to the JTAG device telnet port as before.

4. The flash must be unlocked to ensure that the flash area we want to program is writable.

   For the BDI2000, use the **unlock** command:

   ```
   SAM7S>unlock 0x100000 0x100 256
   Unlocking flash at 0x00100000
   Unlocking flash at 0x00100100
   Unlocking flash at 0x00100200
   ```

```
...
Unlocking flash at 0x0010fe00
Unlocking flash at 0x0010ff00
Unlocking flash passed
SAM7S>
```

For the PEEDI, use the **flash unlock** command:

```
sam7sek> flash unlock 0x100000 65536
unlocking region #0 at 0x00100000
unlocking region #1 at 0x00104000
unlocking region #2 at 0x00108000
unlocking region #3 at 0x0010C000

sam7sek>
```

This command unlocks 256 pages, i.e. 64KiB on the AT91SAM7S512. The number of pages unlocked should match at least the size of the executable to be programmed. With the PEEDI you can use an unadorned **flash unlock** to unlock the entire flash.

5. Give the **erase** (BDI2000) or **flash erase** (PEEDI) command to clear any previous contents.

   For the BDI2000:

```
SAM7S>erase 0x100000 0x100 256
Erasing flash at 0x00100000
Erasing flash at 0x00100100
Erasing flash at 0x00100200
...
Erasing flash at 0x0010fe00
Erasing flash at 0x0010ff00
Erasing flash passed
SAM7S>
```

   As with the **unlock** command, the size of the area erased must be at least the size of the executable to be programmed.

   For the PEEDI, only full flash erase is supported:

```
sam7sek> flash erase

done.

sam7sek>
```

6. Now give the **prog** (BDI2000) or **flash program** (PEEDI) command to fetch the executable from the TFTP server and program it to the flash.

   For the BDI2000:

```
SAM7S>prog 0x100000 sam7.bin bin
Programming sam7.bin , please wait ....
Programming flash passed
SAM7S>
```

   For the PEEDI:

```
sam7sek> flash program tftp://192.168.7.9/gdb_module.bin bin 0x100000
++ info: Programming directly
++ info: Programming image file: tftp://192.168.7.9/gdb_module.bin
++ info: At absolute addres:     0x00100000
unlocking   at 0x00100000 (region #0)
programming at 0x00100000
programming at 0x00101000
programming at 0x00102000
programming at 0x00103000
unlocking   at 0x00104000 (region #1)
```

```
programming at 0x00104000
programming at 0x00105000
programming at 0x00106000
programming at 0x00107000

++ info: successfully programmed 32.00 KB in 0.93 sec

sam7sek>
```

The installation is now complete. This can be tested by powering off the board, disconnecting the JTAG, and then powering on the board again. A ROM based application should start immediately, and any output will be seen on the serial connection. If the GDB stub ROM has been installed, then something similar to the following will be seen on the serial port:

```
+$T050f:cc061000;0d:18082000;#4d
```

Programming GDB Stubs into Flash using SAM-BA

The following gives the steps needed to program the gdb stubs into Flash using SAM-BA. The user should refer to the SAM-BA documentation for full details of how to run the program.

1. Download the AT91 In-system Programmer software package from the Atmel website. Install it on a suitable PC running Windows or Linux. The remainder of this section documents the behaviour seen under Windows, although the behaviour on Linux should not be too different.

2. Copy `gdb_module.bin` from either the at91sam7sek_256 or at91sam7sek_512 subdirectories, depending on which of the two boards you are using, to a suitable location on the PC.

3. Connect a null-modem serial cable between the DEBUG serial port of the board and a serial port on a convenient host (which need not be the PC running SAM-BA). Run a terminal emulator (Hyperterm or minicom) at 38400 baud. Connect a USB cable between the PC and the AT91SAM7S-EK board.

4. JP5 (TST) jumper needs to be temporarily closed and USB connected into the PC for 10 seconds. This writes the SAM-BA bootstrap into the boot sectors so the board can then be programmed. USB should then be disconnected and JP5 moved to the open position. If you do not do this then the option of connecting SAM-BA to the \usb\ARM0 will not be available when the USB cable is reconnected to the PC.

5. Power up the board by plugging the USB cable from the AT91SAM7S-EK board into the PC and Windows should now recognize the USB device.

6. Start SAM-BA. Select "\usb\ARM0" for the communication interface, and "at91sam7s256-ek" or "at91sam7s512-ek" for the board to match the board you are about to program. If the USB option does not appear, check the cable and look in the Windows Device Manager for the active device. If all is well, click on "Connect".

7. In the SAM-BA main window, select the "FLASH" tab and in the "Send File Name" field, select the `gdb_module.bin`. Ensure that the Address field contains "0x100000" and click "Send File". The following output should be seen:

   ```
   (AT91-ISP v1.12) 1 % send_file {Flash} "gdb_module.bin" 0x100000 0
   -I- Send File gdb_module.bin at address 0x100000
    first_sector 0 last_sector 1
   -I-     Writing: 0x740C bytes at 0x0 (buffer addr : 0x202B68)
   -I-     0x740C bytes written by applet
   (AT91-ISP v1.12) 1 %
   ```

   You may get a popup asking if you want to unlock sectors 0, 1 of flash. Select "Y" if prompted.

   You may also get a pop-up asking "Do you want to lock involved lock region(s) (0 to 1)?". Select "No" if prompted.

8. Shut down SAM-BA, disconnect and reconnect the USB cable. Press the reset button on the board and something similar to the following should be output for a AT91SAM7S256-EK board on the DEBUG serial line:

```
$T050f:cc051000;0d:e8072000;#7f
```

For a AT91SAM7S512-EK board you should see something similar. For example:

```
$T050f:c0051000;0d:e0072000;#44
```

Name

Configuration — Platform-specific Configuration Options

Overview

The AT91SAM7S-EK platform HAL package is loaded automatically when eCos is configured for the `at91sam7sek` target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The platform HAL package supports three separate startup types:

RAM
: This is the startup type which is normally used during application development. The board has GDB stubs programmed into flash and boots into that initially. arm-eabi-gdb is then used to load a RAM startup application into memory and debug it. It is assumed that the hardware has already been initialized by the stubs. By default the application will use the eCos virtual vectors mechanism to obtain certain services from the stubs, including diagnostic output.

ROM
: This startup type can be used for finished applications which will be programmed into flash at physical address 0x00100000. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization.

JTAG
: This is an alternative development startup type. The application is loaded into RAM via a JTAG device and is run and debugged from there. The application will be self-contained with no dependencies on services provided by other software. It is expected that hardware setup will have been performed via the JTAG device prior to loading.

GDB Stubs and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building the GDB Stubs.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then the serial port will be claimed for HAL diagnostics.

Flash Driver

The AT91SAM7S-EK board contains a quantity of on-chip flash memory. The `CYGPKG_DEVS_FLASH_AT91` package contains all the code and data definitions necessary to support this part. This driver is not active until the generic Flash support package, `CYGPKG_IO_FLASH`, is included in the configuration.

Watchdog Driver

The AT91SAM7S-EK board use the AT91SAM7S's internal watchdog support. The `CYGPKG_DEVICES_WATCHDOG_ARM_AT91WDTC` package contains all the code necessary to support this device. Within that package the `CYGNUM_DEVS_WATCHDOG_ARM_AT91WDTC_DESIRED_TIMEOUT_MS` configuration option controls the watchdog timeout, and by default will force a reset of the board upon timeout. This driver is not active until the generic watchdog device support package, `CYGPKG_IO_WATCHDOG`, is included in the configuration.

Note that on the AT91, the on-chip watchdog peripheral always starts running immediately, and so in configurations that do not include the watchdog driver, it is always disabled via its write-once register. In configurations which include the watchdog driver obviously the watchdog is not disabled otherwise it could not be subsequently re-enabled, and so the application must start and periodically reset the watchdog from the very beginning of execution.

USART Serial Driver

The AT91SAM7S-EK board use the AT91SAM7S's internal USART serial support as described in the AT91 processor HAL documentation. Two serial ports are available: the serial debug port which is mapped to virtual vector channel 0 in the HAL diagnostic driver or "`/dev/dbg`" in the interrupt-driven driver; and USART 0 which is mapped to virtual vector channel 1 and "`/dev/ser0`". Only USART 0 supports modem control signals such as those used for hardware flow control.

Compiler Flags

The SAM7 variant HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos. There are just three flags specific to this port:

`-mcpu=arm7tdmi`
: The arm-eabi-gcc compiler supports many variants of the ARM architecture. A `-m` option should be used to select the specific variant in use, and with current tools `-mcpu=arm7tdmi` is the correct option for the ARM7TDMI processor in the SAM7S.

`-mthumb`
: The arm-eabi-gcc compiler will compile C and C++ files into the Thumb instruction set when this option is used. The best way to build eCos in Thumb mode is to enable the configuration option `CYGHWR_THUMB`.

`-mthumb-interwork`
: This option allows programs to be created that mix ARM and Thumb instruction sets. Without this option, some memory can be saved. This option should be used if -mthumb is used. The best way to build eCos with Thumb interworking is to enable the configuration option `CYGBLD_ARM_ENABLE_THUMB_INTERWORK`.

Name

JTAG support — Usage

Use of JTAG for debugging

JTAG can be used to single-step and debug loaded RAM applications, or even applications resident in ROM.

Debugging of ROM applications is only possible if using hardware breakpoints. The ARM7TDMI core of the AT91SAM7S only supports two such hardware breakpoints, and so they should be used sparingly. If using a GDB front-end such as Eclipse, check it has not set unnecessary extra breakpoints. Some JTAG devices give the option of whether to set hardware or software breakpoints by default. Be sure to configure your device appropriately.

Abatron BDI2000 notes

On the Abatron BDI2000, the `bdi2000.at91sam7sek.cfg` file should be used to setup and configure the hardware to an appropriate state to load programs. This includes setting up the PLL and flash memory controller.

The `bdi2000.at91sam7sek.cfg` file also contains an option to define whether hardware or software breakpoints are used by default, using the `BREAKMODE` directive in the `[TARGET]` section. Edit this file if you wish to use software break points, and remember to use the **boot** command on the BDI2000 command line interface to make the changes take effect.

On the BDI2000, debugging can be performed either via the telnet interface or using **arm-eabi-gdb** and the bdiGDB interface. In the case of the latter, **arm-eabi-gdb** needs to connect to TCP port 2001 on the BDI2000's IP address. For example:

```
(gdb) target remote 111.222.333.444:2001
```

By default when the BDI2000 is powered up, the target will always run the initialization section of the `bdi2000.at91sam7sek.cfg` file (which configures the CPU clock among other things), and halts the target. This behaviour is repeated with the **reset halt** command.

If the board is reset when in '**reset halt**' mode (either with the '**reset halt**' or '**reset**' commands, or by pressing the reset button) and the '**go**' command is then given, then the board will boot from ROM as normal.

It is also possible for the target to always run, without initialization, after the reset button has been pressed. This mode is selected with the **reset run** command. This conveniently allows the target to be connected to the JTAG debugger, and be able to reset it with the reset button, without being required to always type '**go**' every time. Thereafter, invoking the **reset** command will repeat the previous reset style. Also in this mode, exceptions will be handled by board software, rather than causing the JTAG debugger to halt the CPU.

Suitably configured RAM applications can be loaded either via GDB, or directly via the telnet CLI. For example:

```
SAM7S>load 0x00201000 /test.bin bin
Loading /test.bin , please wait ....
Loading program file passed
SAM7S>go 0x00201000
```

Consult the BDI2000 documentation for information on other formats.

Ronetix PEEDI notes

On the Ronetix PEEDI, the `peedi.at91sam7sek.cfg` file should be used to setup and configure the hardware to an appropriate state to load programs. This includes setting up the PLL and flash memory controller.

The `peedi.at91sam7sek.cfg` file also contains an option to define whether hardware or software breakpoints are used by default, using the `CORE0_BREAKMODE` directive in the `[TARGET]` section. The supplied version of the file defaults to software breakpoints. With this default, hardware breakpoints can still be set from GDB using the **hbreak** command. The default can be changed to hardware breakpoints, and remember to use the **reboot** command on the PEEDI command line interface, or press the reset button to make the changes take effect.

On the PEEDI, debugging can be performed either via the telnet interface or using **arm-eabi-gdb**. In the case of the latter, **arm-eabi-gdb** needs to connect to TCP port 2000 on the PEEDI's IP address. For example:

```
(gdb) target remote 111.222.333.444:2000
```

By default when the PEEDI is powered up, the target will always run the initialization section of the `peedi.at91sam7sek.cfg` file (which configures the CPU clock among other things), and halts the target. This behaviour is repeated with the **reset** command.

If the board is reset with the '**reset**' command, or by pressing the reset button and the '**go**' command is then given, then the board will boot from ROM as normal. A similar effect can be achieved in GDB by connecting with **target remote** and immediately typing **continue** or **c**.

It is also possible for the target to always run, without initialization, after the reset button has been pressed. This mode is selected with the `CORE0_STARTUP_MODE` directive in the `[TARGET]` section of the `peedi.at91sam7sek.cfg` file. This conveniently allows the target to be connected to the JTAG debugger, and be able to reset it with the reset button, without being required to always type '**go**' every time.

Suitably configured RAM applications can be loaded either via GDB, or directly via the telnet CLI. For example:

```
sam7sek> memory load tftp://192.168.7.9/test.bin bin 0x201000
++ info: Loading image file:  tftp://192.168.7.9/test.bin
++ info: At absolute address: 0x00201000
loading at 0x201000
loading at 0x205000

Successfully loaded 28KB (29064 bytes) in 0.1s
sam7sek> go 0x201000
```

Consult the PEEDI documentation for information on other formats and loading mechanisms.

Configuration of RAM applications

If the JTAG device has initialized the processor, such as by using the `bdi2000.at91sam7sek.cfg` configuration on the BDI2000 or `peedi.at91sam7sek.cfg` configuration on the PEEDI, applications can be loaded directly into RAM without requiring a ROM monitor. This loading can be done directly through the JTAG device, or where supported by the JTAG device, through GDB.

In order to configure the application to support this mode, some configuration settings are required. Firstly `CYGSEM_HAL_USE_ROM_MONITOR` must be disabled. Secondly the `CYGDBG_HAL_DIAG_TO_DEBUG_CHAN` option should be disabled in order to prevent HAL diagnostic output being encoded into GDB ($O) packets. Selecting the JTAG startup type in the configuration tool sets these options automatically.

Running RAM applications

Once loaded and running via JTAG, HAL diagnostic output will appear by default on the serial debug port. USART 0 can be chosen instead by setting the `CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL` configuration option in the platform HAL to channel 1.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the AT91SAM7S-EK hardware, and should be read in conjunction with that specification. The AT91SAM7S-EK platform HAL package complements the ARM architectural HAL, the AT91 variant HAL and the AT91SAM7 processor HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset, the HAL will initialize or reinitialize most of the on-chip peripherals. There is an exception for RAM startup applications which depend on a ROM monitor or JTAG device for certain services.

For ROM startup, the HAL will perform additional initialization, setting up the PLL and programming the various internal registers. This is all done in the `PLATFORM_SETUP1` macro in the assembler header file `hal_platform_setup.h`.

Linker Scripts and Memory Maps

The AT91SAM7 processor HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

On-chip Flash

This is located at address 0x00100000 of the physical memory space.

On-chip RAM

This is located at address 0x00200000 of the physical memory space. During booting this memory is only available at this address, but during the boot process it is also remapped to location 0x00000000 in order to allow the hardware exception vectors to be in RAM. The first 32 bytes are used for hardware exception vectors. The next 32 bytes are used for the VSR table and the next 256 bytes are normally used for the eCos virtual vectors, allowing RAM-based applications to use services provided by the ROM monitor. For ROM startup, all remaining RAM is available. For RAM startup, available RAM starts at location 0x00201000, with the bottom 4KiB reserved for use by the GDB stubs.

On-chip Peripheral Registers

These are located at address 0xFF000000 in the physical memory space.

Real-time characterization

The tm_basic kernel test gives statistics gathered about the real-time characterization and performance of the kernel. The sample output is shown here for information. The test was built in ARM mode, which provided better performance than Thumb mode.

Example 239.1. at91sam7sek Real-time characterization

```
            Startup, main stack : stack used   416 size  3920
             Startup :  Interrupt stack used   148 size  4096
             Startup : Idlethread stack used    84 size  2048

eCos Kernel Timings
Notes: all times are in microseconds (.000001) unless otherwise stated

Reading the hardware clock takes 3 'ticks' overhead
... this value will be factored out of all other measurements
Clock interrupt took   33.54 microseconds (100 raw clock ticks)

Testing parameters:
   Clock samples:            32
   Threads:                   2
   Thread switches:         128
```

```
 Mutexes:                  32
 Mailboxes:                32
 Semaphores:               32
 Scheduler operations:    128
 Counters:                 32
 Flags:                    32
 Alarms:                   32


                               Confidence
   Ave     Min     Max     Var  Ave  Min  Function
======  ======  ======  ====== ========== ========
 22.00   21.67   22.33    0.33  100%  50% Create thread
  4.50    4.33    4.67    0.17  100%  50% Yield thread [all suspended]
  4.67    4.67    4.67    0.00  100% 100% Suspend [suspended] thread
  5.00    5.00    5.00    0.00  100% 100% Resume thread
  7.33    7.33    7.33    0.00  100% 100% Set priority
  0.67    0.67    0.67    0.00  100% 100% Get priority
 16.67   16.67   16.67    0.00  100% 100% Kill [suspended] thread
  4.67    4.67    4.67    0.00  100% 100% Yield [no other] thread
  8.67    8.33    9.00    0.33  100%  50% Resume [suspended low prio] thread
  5.00    5.00    5.00    0.00  100% 100% Resume [runnable low prio] thread
  6.50    6.33    6.67    0.17  100%  50% Suspend [runnable] thread
  4.50    4.33    4.67    0.17  100%  50% Yield [only low prio] thread
  4.33    4.33    4.33    0.00  100% 100% Suspend [runnable->not runnable]
 16.33   16.33   16.33    0.00  100% 100% Kill [runnable] thread
 11.67   11.67   11.67    0.00  100% 100% Destroy [dead] thread
 23.67   23.67   23.67    0.00  100% 100% Destroy [runnable] thread
 33.50   31.33   35.67    2.17  100%  50% Resume [high priority] thread
 12.62   12.33   17.33    0.14   74%  25% Thread switch

  0.44    0.33    0.67    0.14   68%  68% Scheduler lock
  3.35    3.33    3.67    0.04   93%  93% Scheduler unlock [0 threads]
  3.35    3.33    3.67    0.04   93%  93% Scheduler unlock [1 suspended]
  3.35    3.33    3.67    0.04   93%  93% Scheduler unlock [many suspended]
  3.35    3.33    3.67    0.04   93%  93% Scheduler unlock [many low prio]

  1.00    1.00    1.00    0.00  100% 100% Init mutex
  4.96    4.67    5.00    0.07   87%  12% Lock [unlocked] mutex
  5.84    5.67    6.00    0.17   53%  46% Unlock [locked] mutex
  4.83    4.67    5.00    0.17  100%  50% Trylock [unlocked] mutex
  4.13    4.00    4.33    0.16   62%  62% Trylock [locked] mutex
  0.50    0.33    0.67    0.17  100%  50% Destroy mutex
 32.27   32.00   32.33    0.10   81%  18% Unlock/Lock mutex

  1.46    1.33    1.67    0.16   62%  62% Create mbox
  0.33    0.33    0.33    0.00  100% 100% Peek [empty] mbox
  5.46    5.33    5.67    0.16   62%  62% Put [first] mbox
  0.29    0.00    0.33    0.07   87%  12% Peek [1 msg] mbox
  5.46    5.33    5.67    0.16   62%  62% Put [second] mbox
  0.29    0.00    0.33    0.07   87%  12% Peek [2 msgs] mbox
  5.58    5.33    5.67    0.13   75%  25% Get [first] mbox
  5.58    5.33    5.67    0.13   75%  25% Get [second] mbox
  4.63    4.33    4.67    0.07   87%  12% Tryput [first] mbox
  4.33    4.33    4.33    0.00  100% 100% Peek item [non-empty] mbox
  5.08    5.00    5.33    0.13   75%  75% Tryget [non-empty] mbox
  4.25    4.00    4.33    0.13   75%  25% Peek item [empty] mbox
  4.42    4.33    4.67    0.13   75%  75% Tryget [empty] mbox
  0.42    0.33    0.67    0.13   75%  75% Waiting to get mbox
  0.42    0.33    0.67    0.13   75%  75% Waiting to put mbox
  1.50    1.33    1.67    0.17  100%  50% Delete mbox
 22.02   22.00   22.33    0.04   93%  93% Put/Get mbox

  0.92    0.67    1.00    0.13   75%  25% Init semaphore
  4.00    4.00    4.00    0.00  100% 100% Post [0] semaphore
  4.54    4.33    4.67    0.16   62%  37% Wait [1] semaphore
  4.00    4.00    4.00    0.00  100% 100% Trywait [0] semaphore
```

```
    4.00    4.00    4.00    0.00  100% 100% Trywait [1] semaphore
    1.00    1.00    1.00    0.00  100% 100% Peek semaphore
    0.50    0.33    0.67    0.17  100%  50% Destroy semaphore
   19.92   19.67   20.00    0.13   75%  25% Post/Wait semaphore

    1.54    1.33    1.67    0.16   62%  37% Create counter
    0.46    0.33    0.67    0.16   62%  62% Get counter value
    0.46    0.33    0.67    0.16   62%  62% Set counter value
    4.92    4.67    5.00    0.13   75%  25% Tick counter
    0.50    0.33    0.67    0.17  100%  50% Delete counter

    0.88    0.67    1.00    0.16   62%  37% Init flag
    4.38    4.33    4.67    0.07   87%  87% Destroy flag
    4.00    4.00    4.00    0.00  100% 100% Mask bits in flag
    4.33    4.33    4.33    0.00  100% 100% Set bits in flag [no waiters]
    6.92    6.67    7.00    0.13   75%  25% Wait for flag [AND]
    6.79    6.67    7.00    0.16   62%  62% Wait for flag [OR]
    6.92    6.67    7.00    0.13   75%  25% Wait for flag [AND/CLR]
    6.83    6.67    7.00    0.17  100%  50% Wait for flag [OR/CLR]
    0.33    0.33    0.33    0.00  100% 100% Peek on flag

    2.67    2.67    2.67    0.00  100% 100% Create alarm
    8.63    8.33    8.67    0.07   87%  12% Initialize alarm
    3.92    3.67    4.00    0.13   75%  25% Disable alarm
    7.92    7.67    8.00    0.13   75%  25% Enable alarm
    4.67    4.67    4.67    0.00  100% 100% Delete alarm
    5.88    5.67    6.00    0.16   62%  37% Tick counter [1 alarm]
   40.75   40.67   41.00    0.13   75%  75% Tick counter [many alarms]
   11.54   11.33   11.67    0.16   62%  37% Tick & fire counter [1 alarm]
  232.75  232.67  233.00    0.13   75%  75% Tick & fire counters [>1 together]
   46.75   46.67   47.00    0.13   75%  75% Tick & fire counters [>1 separately]
   32.33   32.33   32.33    0.00  100% 100% Alarm latency [0 threads]
   35.22   33.33   39.67    1.90   78%  78% Alarm latency [2 threads]
   35.22   32.33   39.67    1.90   77%  52% Alarm latency [many threads]
   54.37   54.33   59.00    0.07   99%  99% Alarm -> thread resume latency

    6.65    6.33    7.00    0.00            Clock/interrupt latency

   12.95   11.33   19.67    0.00            Clock DSR latency

  292    292     292  (main stack:   772)  Thread stack used (1360 total)
           All done, main stack : stack used   772 size  3920
            All done :  Interrupt stack used   208 size  4096
            All done : Idlethread stack used   248 size  2048

Timing complete - 30250 ms total

PASS:<Basic timing OK>
EXIT:<done>
```

Other Issues

The AT91SAM7S-EK platform HAL does not affect the implementation of other parts of the eCos HAL specification. The AT91SAM7 processor HAL, AT91 variant HAL, and the ARM architectural HAL documentation should be consulted for further details.

Chapter 240. Atmel AT91SAM7X-EK Board Support

Name

eCos Support for the Atmel AT91SAM7X-EK — Overview

Description

This document covers the configuration and usage of eCos and GDB Stubs on the Atmel AT91SAM7X-EK Evaluation Kit. The AT91SAM7X-EK Evaluation Kit contains the AT91SAM7X256 processor, external connections for two serial channels (one debug, one full), ethernet, USB host/device. eCos support for the devices and peripherals on the AT91SAM7X256 is described below.

Application development on this board can take one of several approaches. Applications may be loaded into RAM via a JTAG device; however, the application size is limited by the amount of on-chip RAM, which is 64KiB on the AT91SAM7X256, 32KiB on the AT91SAM7X128 and 128KiB on the AT91SAM7X512. Applications may also be loaded into the on-chip flash memory where the RAM limit will only apply to the data portion of the application. Finally, it is possible to program a GDB debugging stub into flash which will then allow applications to be loaded into RAM via the serial port. This allows development to proceed without needing to use a JTAG device and application size is limited to the RAM size less 4KiB used by the stub. It is therefore recommended that JTAG debugging be used to debug applications since memory is limited on this platform.

This documentation is expected to be read in conjunction with the AT91 processor HAL and AT91SAM7 variant HAL documentation and further device support and subsystems are described and documented there.

Supported Hardware

The on-chip NOR flash is organized into pages of 256 bytes each. The number of pages is determined by the device variant, from 512 for the AT91SAM7X128 to 2048 for the AT91SAM7X512.

There is a serial driver `CYGPKG_IO_SERIAL_ARM_AT91` which supports both the Debug Unit and USART serial devices. The debug serial port at J3 and DTE port at J2 (connected to USART channel 0) can be used for communication. If the GDB stub ROM is installed, it uses the Debug Unit serial device only. The serial driver package is loaded automatically when configuring for the AT91SAM7X-EK target.

There is a watchdog driver `CYGPKG_DEVICES_WATCHDOG_ARM_AT91WDTC`. This driver is also loaded automatically when configuring for the AT91SAM7X-EK target.

The AT91SAM7 processor HAL contains a driver for the Two-Wire Interface (TWI) controller on the AT91SAM7X. This type of bus is also known as I²C®.

There is a network driver for the on-chip Ethernet and DM9161A PHY. This is only recommended for use with the lwIP TCP/IP stack due to the low RAM requirements.

The AT91SAM7X-EK on-board dataflash device is accessible at virtual address 0x10000000 and a dataflash card connected at J30 is accessible at virtual address 0x20000000.

In general, devices (PIO, UARTs, etc.) are initialized only as far as is necessary for eCos to run. Other devices (RTC, I²C, SPI etc.) are not touched unless the appropriate driver is loaded, although in some cases, the HAL boot sequence will set up the appropriate PIO configuration.

Tools

The AT91SAM7X-EK support is intended to work with GNU tools configured for an arm-eabi target. The original port was undertaken using arm-elf-gcc version 3.4.4, arm-elf-gdb version 6.3, and binutils version 2.15.

Name

Setup — Preparing the AT91SAM7X-EK board for eCos Development

Overview

eCos applications are either programmed into the on-chip flash, or run from RAM using either a JTAG device or the GDB stubs ROM. The installation of the GDB stubs or any flash-resident application requires use of a JTAG device to write to the flash, or the Atmel-supplied SAM-BA program that interacts with the on-chip boot program. So, in all cases it is necessary to set up a JTAG device for the board. This document describes how to set up either an Abatron BDI3000 or Ronetix PEEDI and then use them to program an application into the flash.

Initial Installation with Abatron BDI3000

Preparing the Abatron BDI3000 JTAG debugger

The BDI3000 must first be configured to allow communication with your local network, and configured with the parameters for interfacing with the target board. The following steps should be followed:

1. Prepare a PC to act as a host PC and start a TFTP server on it.
2. Connect the Abatron BDI3000 JTAG debugger via both serial and ethernet to the host PC and power it on. Use the serial cable supplied with the BDI3000.
3. Install the Abatron BDI3000 bdiGDB support software on the host PC.
4. Locate the file `bdi3000.at91sam7xek.cfg` within the eCos platform HAL package in the source repository. This will be in the directory `packages/hal/arm/at91/at91sam7xek/`*`VERSION`*`/misc` relative to the root of your eCos installation.
5. Locate the file `regSAM7S.def` within the installation of the BDI3000 bdiGDB support software.
6. Place the `bdi3000.at91sam7xek.cfg` file in a location on the PC accessible to the TFTP server. Later you will configure the BDI3000 to load this file via TFTP as its configuration file.
7. Similarly place the file `regSAM7S.def` in a location accessible to the TFTP server.
8. Open `bdi3000.at91sam7xek.cfg` in an editor such as emacs or notepad and if necessary adjust the path of the `regSAM7S.def` file in the `[REGS]` section to match its location relative to the TFTP server root.
9. Install and configure the Abatron BDI3000 in line with the bdiGDB instruction manual. Configure the BDI3000 to use the `bdi3000.at91sam7xek.cfg` configuration file at the appropriate point of this process.

Preparing the AT91SAM7X-EK board for programming with BDI3000

Follow the steps in this section in order to allow communication between the board and the host PC, and between the board and the JTAG device.

1. First you must connect a null modem DB9 serial cable between the Serial Debug Port on the board and a serial port on the host computer.
2. Start a suitable terminal emulator on the host computer such as **minicom** or HyperTerminal. Set the communication parameters to 38400 baud, 8 data bits, no parity bit and 1 stop bit with no flow control.
3. Connect the board to the BDI3000 using a 20-pin ARM/Xscale cable from the JTAG interface connector to the Target A port on the BDI3000.

4. Power up the AT91SAM7X-EK board.

5. Connect to the BDI3000's CLI interface via TCP/IP on the standard telnet port 23. The **telnet** application is suitable for this. You should see usage information followed by the prompt:

   ```
   SAM7X>
   ```

6. Confirm correct connection with the BDI3000 with the **reset halt** command as follows:

   ```
   SAM7X> reset halt
   - TARGET: processing reset request
   - TARGET: BDI asserts TRST and RESET
   - TARGET: BDI removes TRST
   - TARGET: Bypass check 0x000000001 => 0x00000001
   - TARGET: JTAG exists check passed
   - Core#0: ID code is 0x3F0F0F0F
   - TARGET: All ICEBreaker access checks passed
   - TARGET: BDI removes RESET
   - TARGET: BDI waits for RESET inactive
   - TARGET: resetting target passed
   - TARGET: processing target startup ....
   - TARGET: processing target startup passed
   SAM7X>
   ```

Initial Installation with Ronetix PEEDI

Preparing the Ronetix PEEDI JTAG debugger

The PEEDI must first be configured to allow communication with your local network, and configured with the parameters for interfacing with the target board. The following steps give a typical outline of setting up the PEEDI using TFTP. Consult the PEEDI documentation for alternative mechanisms.

1. Prepare a PC to act as a host PC and start a TFTP server on it.

2. Connect the PEEDI JTAG debugger via both serial and ethernet to the host PC and power it on. Use the serial cable supplied with the PEEDI (straight through, not null modem).

3. Locate the file `peedi.at91sam7xek.cfg` within the eCos platform HAL package in the source repository. This will be in the directory `packages/hal/arm/at91/at91sam7xek/`*`VERSION`*`/misc` relative to the root of your eCos installation.

4. Place the `peedi.at91sam7xek.cfg` file in a location on the PC accessible to the TFTP server. Later you will configure the PEEDI to load this file via TFTP as its configuration file.

5. Open `at91sam7xek.cfg` in an editor such as emacs or notepad and insert your own license information in the `[LICENSE]` section.

6. Install and configure the PEEDI in line with the PEEDI Quick Start Guide or User's Manual, especially configuring PEEDI's RedBoot with the network information. Configure it to use the `peedi.at91sam7xek.cfg` target configuration file on the TFTP server at the appropriate point of the **fconfig** process, for example with a path such as: `tftp://192.168.7.9/peedi.at91sam7xek.cfg`

7. Reset the PEEDI.

8. Connect to the PEEDI's CLI interface via TCP/IP on the standard telnet port 23. The **telnet** application is suitable for this. You should see output similar to the following:

   ```
   $ telnet 192.168.7.225
   Trying 192.168.7.225...
   Connected to 192.168.7.225.
   Escape character is '^]'.
   ```

```
PEEDI - Powerful Embedded Ethernet Debug Interface
Copyright (c) 2005-2007 www.ronetix.at - All rights reserved
Hw:1.2, Fw:2.0.13, SN: PD-0000-XXXX-XXXX
------------------------------------------------------------

sam7xek>
```

Preparing the AT91SAM7X-EK board for programming with PEEDI

Follow the steps in this section in order to allow communication between the board and the host PC, and between the board and the JTAG device.

1. First you must connect a null modem DB9 serial cable between the Serial Debug Port on the board and a serial port on the host computer.

2. Start a suitable terminal emulator on the host computer such as **minicom** or HyperTerminal. Set the communication parameters to 38400 baud, 8 data bits, no parity bit and 1 stop bit with no flow control.

3. Connect the board to the PEEDI using a 20-pin ARM/Xscale cable from the JTAG interface connector on the board to the Target port on the PEEDI.

4. Power up the AT91SAM7X-EK board.

5. Connect to the PEEDI's CLI on port 23 as before.

6. Confirm correct connection with the PEEDI with the **reset** command as follows:

```
sam7xek> reset
++ info: user reset
sam7xek>
++ info: RESET and TRST asserted
++ info: TRST released
++ info: 1 TAP controller(s) detected
++ info: TAP : IDCODE = 0x3F0F0F0F, ARM7TDMI compliant
++ info: RESET released
++ info: core 0: initialized

sam7xek>
```

Installation into Flash

Installation of an application into the on-chip flash, or the installation of the GDB stubs, using a JTAG programmer takes exactly the same form:

1. Locate the binary image of the executable to be installed. For the GDB stubs do this by locating the file `gdb.module.bin` within the `loaders` subdirectory of the base of the eCos installation. For applications use **arm-eabi-objcopy -O binary** to convert the ELF output of the linker into binary.

2. Copy the file into a location on the host computer accessible to its TFTP server.

3. Connect to the JTAG device telnet port as before.

4. The flash must be unlocked to ensure that the flash area we want to program is writable.

   For the BDI3000, use the **unlock** command:

```
SAM7X>unlock 0x100000 0x100 256
Unlocking flash at 0x00100000
Unlocking flash at 0x00100100
Unlocking flash at 0x00100200
...
Unlocking flash at 0x0010fe00
Unlocking flash at 0x0010ff00
Unlocking flash passed
SAM7X>
```

For the PEEDI, use the **flash unlock** command:

```
sam7xek> flash unlock 0x100000 65536
unlocking region #0 at 0x00100000
unlocking region #1 at 0x00104000
unlocking region #2 at 0x00108000
unlocking region #3 at 0x0010C000

sam7xek>
```

This command unlocks 256 pages, i.e. 64KiB. The number of pages unlocked should match at least the size of the executable to be programmed. With the PEEDI you can use an unadorned **flash unlock** to unlock the entire flash.

5. Give the **erase** (BDI3000) or **flash erase** (PEEDI) command to clear any previous contents.

   For the BDI3000:

   ```
   SAM7X>erase 0x100000 0x100 256
   Erasing flash at 0x00100000
   Erasing flash at 0x00100100
   Erasing flash at 0x00100200
   ...
   Erasing flash at 0x0010fe00
   Erasing flash at 0x0010ff00
   Erasing flash passed
   SAM7X>
   ```

   As with the **unlock** command, the size of the area erased must be at least the size of the executable to be programmed.

   For the PEEDI, only full flash erase is supported:

   ```
   sam7xek> flash erase

   done.

   sam7xek>
   ```

6. Now give the **prog** (BDI3000) or **flash program** (PEEDI) command to fetch the executable from the TFTP server and program it to the flash.

   For the BDI3000:

   ```
   SAM7X>prog 0x100000 sam7.bin bin
   Programming sam7.bin , please wait ....
   Programming flash passed
   SAM7X>
   ```

   For the PEEDI:

   ```
   sam7xek> flash program tftp://192.168.7.9/gdb_module.bin bin 0x100000
   ++ info: Programming directly
   ++ info: Programming image file: tftp://192.168.7.9/gdb_module.bin
   ++ info: At absolute addres:     0x00100000
   unlocking   at 0x00100000 (region #0)
   programming at 0x00100000
   programming at 0x00101000
   programming at 0x00102000
   programming at 0x00103000
   ```

```
unlocking   at 0x00104000 (region #1)
programming at 0x00104000
programming at 0x00105000
programming at 0x00106000
programming at 0x00107000

++ info: successfully programmed 32.00 KB in 0.93 sec

sam7xek>
```

7. Finally, the processor must be switched to boot from the flash rather than the internal ROM. This is done by programming a General Purpose NVM bit in the flash memory. This can be done using memory write commands in the JTAG device telnet interface. On the BDI3000 this is done using the following commands:

   ```
   SAM7X>md 0xffffff60 3
   ffffff60 : 0x00480100      4718848  ..0.
   ffffff64 : 0x00000000            0  ....
   ffffff68 : 0x00000001            1  ....
   SAM7X>mm 0xffffff64 0x5a00020b
   SAM7X>md 0xffffff60 3
   ffffff60 : 0x00480100      4718848  ..0.
   ffffff64 : 0x00000000            0  ....
   ffffff68 : 0x00000401         1025  ....
   SAM7X>
   ```

   And on the PEEDI:

   ```
   sam7xek> mem read 0xffffff60 3

   0xFFFFFF60: 0x00480100 0x00000000 0x00000001
   sam7xek> mem write 0xffffff64 0x5a00020b
   sam7xek> mem read 0xffffff60 3

   0xFFFFFF60: 0x00480100 0x00000000 0x00000401
   sam7xek>
   ```

The installation is now complete. This can be tested by powering off the board, disconnecting the JTAG, and then powering on the board again. A ROM based application should start immediately, and any output will be seen on the serial connection. If the GDB stub ROM has been installed, then something similar to the following will be seen on the serial port:

```
+$T050f:cc061000;0d:18082000;#4d
```

Programming GDB Stubs into Flash using SAM-BA

The following gives the steps needed to program the gdb stubs into Flash using SAM-BA. The user should refer to the SAM-BA documentation for full details of how to run the program.

1. Download the AT91 In-system Programmer software package from the Atmel website. Install it on a suitable PC running Windows or Linux. The remainder of this section documents the behaviour seen under Windows, although the behaviour on Linux should not be too different.

2. Copy `gdb_module.bin` from either the at91sam7xek_256 or at91sam7xek_512 subdirectories, depending on which of the two boards you are using, to a suitable location on the PC.

3. Connect a null-modem serial cable between the DEBUG serial port of the board and a serial port on a convenient host (which need not be the PC running SAM-BA). Run a terminal emulator (Hyperterm or minicom) at 38400 baud. Connect a USB cable between the PC and the AT91SAM7X-EK board.

4. Power up the board by plugging the USB cable from the AT91SAM7X-EK board into the PC and Windows should recognize the USB device. If it does not, then you will need to erase the existing program that has already been programmed into flash. To do this, disconnect the USB cable from the PC, effectively powering down the device, connect jumper J8 (ERASE) on the

AT91SAM7X-EK board and reconnect the USB cable. This step should have erased the flash. Finally disconnect the USB cable followed by J8, reconnect the USB cable and the board should be recognized now. Windows may ask you to install a new driver, in which case follow the instructions.

5. Start SAM-BA. Select "\usb\ARM0" for the communication interface, and "at91sam7x256-ek" or "at91sam7x512-ek" for the board to match the board you are about to program. If the USB option does not appear, check the cable and look in the Windows Device Manager for the active device. If all is well, click on "Connect".

6. In the SAM-BA main window, select the "FLASH" tab and in the "Send File Name" field, select the `gdb_module.bin`. Ensure that the Address field contains "0x100000" and click "Send File". The following output should be seen:

   ```
   (AT91-ISP v1.13) 1 % send_file {Flash} "gdb_module.bin" 0x100000 0
   -I- Send File gdb_module.bin at address 0x100000
    first_sector 0 last_sector 1
   -I-     Writing: 0x7400 bytes at 0x0 (buffer addr : 0x202BC8)
   -I-     0x7400 bytes written by applet
   (AT91-ISP v1.13) 1 %
   ```

   You may get a pop-up asking "Do you want to lock involved lock region(s) (0 to 1)?". Select "No" if prompted.

7. In the Scripts section, select the script "Boot from Flash (GPNVM2)" and press "Execute". The following output should be seen:

   ```
   (AT91-ISP v1.13) 1 % FLASH::ScriptGPNMV 4
   -I- GPNVM2 set
   (AT91-ISP v1.13) 1 %
   ```

8. Shut down SAM-BA, disconnect and reconnect the USB cable. Press the reset button on the board and something similar to the following should be output for a AT91SAM7X256-EK board on the DEBUG serial line:

   ```
   $T050f:cc051000;0d:e8072000;#7f
   ```

   For a AT91SAM7X512-EK board you should see something similar. For example:

   ```
   $T050f:d0051000;0d:e8072000;#4d
   ```

Name

Configuration — Platform-specific Configuration Options

Overview

The AT91SAM7X-EK platform HAL package is loaded automatically when eCos is configured for the `at91sam7xek` target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The platform HAL package supports three separate startup types:

RAM This is the startup type which is normally used during application development. The board has GDB stubs programmed into flash and boots into that initially. arm-eabi-gdb is then used to load a RAM startup application into memory and debug it. It is assumed that the hardware has already been initialized by the stubs. By default the application will use the eCos virtual vectors mechanism to obtain certain services from the stubs, including diagnostic output.

ROM This startup type can be used for finished applications which will be programmed into flash at physical address 0x00100000. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization.

JTAG This is an alternative development startup type. The application is loaded into RAM via a JTAG device and is run and debugged from there. The application will be self-contained with no dependencies on services provided by other software. It is expected that hardware setup will have been performed via the JTAG device prior to loading.

GDB Stubs and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building the GDB Stubs.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then the serial port will be claimed for HAL diagnostics.

Flash Driver

The AT91SAM7X-EK board contains a quantity of on-chip flash memory. The `CYGPKG_DEVS_FLASH_AT91` package contains all the code and data definitions necessary to support this part. This driver is not active until the generic Flash support package, `CYGPKG_IO_FLASH`, is included in the configuration.

Watchdog Driver

The AT91SAM7X-EK board uses the AT91SAM7X's internal watchdog support. The `CYGPKG_DEVICES_WATCHDOG_ARM_AT91WDTC` package contains all the code necessary to support this device. Within that package the `CYGNUM_DEVS_WATCHDOG_ARM_AT91WDTC_DESIRED_TIMEOUT_MS` configuration option controls the watchdog timeout, and by default will force a reset of the board upon timeout. This driver is not active until the generic watchdog device support package, `CYGPKG_IO_WATCHDOG`, is included in the configuration.

Note that on the AT91, the on-chip watchdog peripheral always starts running immediately, and so in configurations that do not include the watchdog driver, it is always disabled via its write-once register. In configurations which include the watchdog driver obviously the watchdog is not disabled otherwise it could not be subsequently re-enabled, and so the application must start and periodically reset the watchdog from the very beginning of execution.

USART Serial Driver

The AT91SAM7X-EK board use the AT91SAM7X's internal USART serial support as described in the AT91 processor HAL documentation. Two serial ports are available: the serial debug port which is mapped to virtual vector channel 0 in the HAL diagnostic driver or "`/dev/dbg`" in the interrupt-driven driver; and USART 0 which is mapped to virtual vector channel 1 and "`/dev/ser0`". Only USART 0 supports modem control signals such as those used for hardware flow control.

Ethernet Driver

The AT91SAM7X-EK board can use the AT91SAM7X's internal ethernet MAC (EMAC) support. The package `CYGPKG_DEVS_ETH_ARM_AT91` contains the necessary device driver support.

Due to the amount of RAM available, it is not possible to use the BSD-derived TCP/IP stack. Instead either the lwIP TCP/IP stack (in `CYGPKG_NET_LWIP`) should be used, or your application can operate the driver directly in standalone mode. To enable the Ethernet driver in your configuration, either include the “Common Ethernet Support” (`CYGPKG_IO_ETH_DRIVERS`) package in your configuration, or base your configuration on an appropriate template, in particular the “lwip_eth” template.

Support for the Davicom DM9161A PHY (which comes from the `CYGPKG_DEVS_ETH_PHY` package) is automatically configured when ethernet support is enabled.

The AT91 ethernet device driver package in fact includes two separate driver implementations: one standard driver suitable for use in standalone mode, or with various TCP/IP stacks including at least RedBoot, BSD and lwIP; and one specific to lwIP. It is strongly recommended that the lwIP-specific driver is used with lwIP, given the low memory constraints. The lwIP-specific driver is a streamlined efficient version designed for very low RAM overhead. As a result it is implemented intentionally at the expense of features irrelevant to the AT91SAM7X, such as multiple network device support, and network debugging under RedBoot. Instead it has improvements such as zero-copy reception and transmission of data packets, leading to both faster operation, and smaller code and data memory footprint.

The choice between using the standard driver, or the lwIP-specific driver is not made within this package, but is instead made in the generic ethernet I/O package `CYGPKG_IO_ETH_DRIVERS` using the options within the lwIP driver model component (`CYGIMP_IO_ETH_DRIVERS_LWIP_DRIVER_MODEL`). The standard driver is the default, and thus needs to be explicitly changed to select the lwIP-specific driver model.

Careful selection and tuning of the configuration settings within the AT91 Ethernet package, and more especially, the lwIP stack, can result in realistic application use of TCP/IP at high speeds, despite the low RAM availability.

Note that a board design issue has required a workaround to be used in order to correctly initialize the PHY. This has two notable consequences: there is an extra delay of 700ms at startup time; and the programmatic use of the NRST line from the processor can confuse attached JTAG units into believing the processor has reset, which can in turn adversely affect debugging sessions. In the case of the Abatron BDI3000, it has been found that ensuring the last reset command was a "reset run", and then connecting via GDB immediately after target reset results in a stable debug session, albeit at the expense of the processor already having run some of the early HAL startup sequence.

Compiler Flags

The SAM7 variant HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos. There are just three flags specific to this port:

`-mcpu=arm7tdmi`

The arm-eabi-gcc compiler supports many variants of the ARM architecture. A `-m` option should be used to select the specific variant in use, and with current tools `-mcpu=arm7tdmi` is the correct option for the ARM7TDMI processor in the SAM7X.

`-mthumb`

The arm-eabi-gcc compiler will compile C and C++ files into the Thumb instruction set when this option is used. The best way to build eCos in Thumb mode is to enable the configuration option `CYGHWR_THUMB`.

`-mthumb-interwork`

This option allows programs to be created that mix ARM and Thumb instruction sets. Without this option, some memory can be saved. This option should be used if -mthumb is used. The best way to build eCos with Thumb interworking is to enable the configuration option `CYGBLD_ARM_ENABLE_THUMB_INTERWORK`.

Name

JTAG support — Usage

Use of JTAG for debugging

JTAG can be used to single-step and debug loaded RAM applications, or even applications resident in ROM.

Debugging of ROM applications is only possible if using hardware breakpoints. The ARM7TDMI core of the AT91SAM7X only supports two such hardware breakpoints, and so they should be used sparingly. If using a GDB front-end such as Eclipse, check it has not set unnecessary extra breakpoints. Some JTAG devices give the option of whether to set hardware or software breakpoints by default. Be sure to configure your device appropriately.

Abatron BDI3000 notes

On the Abatron BDI3000, the `bdi3000.at91sam7xek.cfg` file should be used to setup and configure the hardware to an appropriate state to load programs. This includes setting up the PLL and flash memory controller.

The `bdi3000.at91sam7xek.cfg` file also contains an option to define whether hardware or software breakpoints are used by default, using the `BREAKMODE` directive in the `[TARGET]` section. Edit this file if you wish to use software break points, and remember to use the **boot** command on the BDI3000 command line interface to make the changes take effect.

On the BDI3000, debugging can be performed either via the telnet interface or using **arm-eabi-gdb** and the bdiGDB interface. In the case of the latter, **arm-eabi-gdb** needs to connect to TCP port 2001 on the BDI3000's IP address. For example:

```
(gdb) target remote 111.222.333.444:2001
```

By default when the BDI3000 is powered up, the target will always run the initialization section of the `bdi3000.at91sam7x-ek.cfg` file (which configures the CPU clock among other things), and halts the target. This behaviour is repeated with the **reset halt** command.

If the board is reset when in '**reset halt**' mode (either with the '**reset halt**' or '**reset**' commands, or by pressing the reset button) and the '**go**' command is then given, then the board will boot from ROM as normal.

It is also possible for the target to always run, without initialization, after the reset button has been pressed. This mode is selected with the **reset run** command. This conveniently allows the target to be connected to the JTAG debugger, and be able to reset it with the reset button, without being required to always type '**go**' every time. Thereafter, invoking the **reset** command will repeat the previous reset style. Also in this mode, exceptions will be handled by board software, rather than causing the JTAG debugger to halt the CPU.

Suitably configured RAM applications can be loaded either via GDB, or directly via the telnet CLI. For example:

```
SAM7X>load 0x00201000 /test.bin bin
Loading /test.bin , please wait ....
Loading program file passed
SAM7X>go 0x00201000
```

Consult the BDI3000 documentation for information on other formats.

Ronetix PEEDI notes

On the Ronetix PEEDI, the `peedi.at91sam7xek.cfg` file should be used to setup and configure the hardware to an appropriate state to load programs. This includes setting up the PLL and flash memory controller.

The `peedi.at91sam7xek.cfg` file also contains an option to define whether hardware or software breakpoints are used by default, using the `CORE0_BREAKMODE` directive in the `[TARGET]` section. The supplied version of the file defaults to software breakpoints. With this default, hardware breakpoints can still be set from GDB using the **hbreak** command. The default can be changed to hardware breakpoints, and remember to use the **reboot** command on the PEEDI command line interface, or press the reset button to make the changes take effect.

On the PEEDI, debugging can be performed either via the telnet interface or using **arm-eabi-gdb**. In the case of the latter, **arm-eabi-gdb** needs to connect to TCP port 2000 on the PEEDI's IP address. For example:

```
(gdb) target remote 111.222.333.444:2000
```

By default when the PEEDI is powered up, the target will always run the initialization section of the `peedi.at91sam7xek.cfg` file (which configures the CPU clock among other things), and halts the target. This behaviour is repeated with the **reset** command.

If the board is reset with the '**reset**' command, or by pressing the reset button and the '**go**' command is then given, then the board will boot from ROM as normal. A similar effect can be achieved in GDB by connecting with **target remote** and immediately typing **continue** or **c**.

It is also possible for the target to always run, without initialization, after the reset button has been pressed. This mode is selected with the `CORE0_STARTUP_MODE` directive in the `[TARGET]` section of the `peedi.at91sam7xek.cfg` file. This conveniently allows the target to be connected to the JTAG debugger, and be able to reset it with the reset button, without being required to always type '**go**' every time.

Suitably configured RAM applications can be loaded either via GDB, or directly via the telnet CLI. For example:

```
sam7xek> memory load tftp://192.168.7.9/test.bin bin 0x201000
++ info: Loading image file:  tftp://192.168.7.9/test.bin
++ info: At absolute address: 0x00201000
loading at 0x201000
loading at 0x205000

Successfully loaded 28KB (29064 bytes) in 0.1s
sam7xek> go 0x201000
```

Consult the PEEDI documentation for information on other formats and loading mechanisms.

Configuration of RAM applications

If the JTAG device has initialized the processor, such as by using the `bdi3000.at91sam7xek.cfg` configuration on the BDI3000 or `peedi.at91sam7xek.cfg` configuration on the PEEDI, applications can be loaded directly into RAM without requiring a ROM monitor. This loading can be done directly through the JTAG device, or where supported by the JTAG device, through GDB.

In order to configure the application to support this mode, some configuration settings are required. Firstly `CYGSEM_HAL_USE_ROM_MONITOR` must be disabled. Secondly the `CYGDBG_HAL_DIAG_TO_DEBUG_CHAN` option should be disabled in order to prevent HAL diagnostic output being encoded into GDB ($O) packets. Selecting the JTAG startup type in the configuration tool sets these options automatically.

Running RAM applications

Once loaded and running via JTAG, HAL diagnostic output will appear by default on the serial debug port. USART 0 can be chosen instead by setting the `CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL` configuration option in the platform HAL to channel 1.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the AT91SAM7X-EK hardware, and should be read in conjunction with that specification. The AT91SAM7X-EK platform HAL package complements the ARM architectural HAL, the AT91 variant HAL and the AT91SAM7 processor HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset, the HAL will initialize or reinitialize most of the on-chip peripherals. There is an exception for RAM startup applications which depend on a ROM monitor or JTAG device for certain services.

For ROM startup, the HAL will perform additional initialization, setting up the PLL and programming the various internal registers. This is all done in the `PLATFORM_SETUP1` macro in the assembler header file `hal_platform_setup.h`.

Linker Scripts and Memory Maps

The AT91SAM7 processor HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

On-chip Flash
This is located at address 0x00100000 of the physical memory space.

On-chip RAM
This is located at address 0x00200000 of the physical memory space. During booting this memory is only available at this address, but during the boot process it is also remapped to location 0x00000000 in order to allow the hardware exception vectors to be in RAM. The first 32 bytes are used for hardware exception vectors. The next 32 bytes are used for the VSR table and the next 256 bytes are normally used for the eCos virtual vectors, allowing RAM-based applications to use services provided by the ROM monitor. For ROM startup, all remaining RAM is available. For RAM startup, available RAM starts at location 0x00201000, with the bottom 4KiB reserved for use by the GDB stubs.

On-chip Peripheral Registers
These are located at address 0xFF000000 in the physical memory space.

Real-time characterization

The tm_basic kernel test gives statistics gathered about the real-time characterization and performance of the kernel. The sample output is shown here for information. The test was built in ARM mode, which provided better performance than Thumb mode.

Example 240.1. at91sam7xek Real-time characterization

```
            Startup, main stack : stack used   416 size  3920
             Startup :  Interrupt stack used   148 size  4096
             Startup : Idlethread stack used    84 size  2048

eCos Kernel Timings
Notes: all times are in microseconds (.000001) unless otherwise stated

Reading the hardware clock takes 3 'ticks' overhead
... this value will be factored out of all other measurements
Clock interrupt took   33.54 microseconds (100 raw clock ticks)

Testing parameters:
   Clock samples:            32
   Threads:                   2
   Thread switches:         128
```

```
  Mutexes:                  32
  Mailboxes:                32
  Semaphores:               32
  Scheduler operations:    128
  Counters:                 32
  Flags:                    32
  Alarms:                   32


                               Confidence
   Ave     Min     Max     Var  Ave  Min  Function
 ======  ======  ======  ====== ========== ========
  22.00   21.67   22.33    0.33  100%  50% Create thread
   4.50    4.33    4.67    0.17  100%  50% Yield thread [all suspended]
   4.67    4.67    4.67    0.00  100% 100% Suspend [suspended] thread
   5.00    5.00    5.00    0.00  100% 100% Resume thread
   7.33    7.33    7.33    0.00  100% 100% Set priority
   0.67    0.67    0.67    0.00  100% 100% Get priority
  16.67   16.67   16.67    0.00  100% 100% Kill [suspended] thread
   4.67    4.67    4.67    0.00  100% 100% Yield [no other] thread
   8.67    8.33    9.00    0.33  100%  50% Resume [suspended low prio] thread
   5.00    5.00    5.00    0.00  100% 100% Resume [runnable low prio] thread
   6.50    6.33    6.67    0.17  100%  50% Suspend [runnable] thread
   4.50    4.33    4.67    0.17  100%  50% Yield [only low prio] thread
   4.33    4.33    4.33    0.00  100% 100% Suspend [runnable->not runnable]
  16.33   16.33   16.33    0.00  100% 100% Kill [runnable] thread
  11.67   11.67   11.67    0.00  100% 100% Destroy [dead] thread
  23.67   23.67   23.67    0.00  100% 100% Destroy [runnable] thread
  33.50   31.33   35.67    2.17  100%  50% Resume [high priority] thread
  12.62   12.33   17.33    0.14   74%  25% Thread switch

   0.44    0.33    0.67    0.14   68%  68% Scheduler lock
   3.35    3.33    3.67    0.04   93%  93% Scheduler unlock [0 threads]
   3.35    3.33    3.67    0.04   93%  93% Scheduler unlock [1 suspended]
   3.35    3.33    3.67    0.04   93%  93% Scheduler unlock [many suspended]
   3.35    3.33    3.67    0.04   93%  93% Scheduler unlock [many low prio]

   1.00    1.00    1.00    0.00  100% 100% Init mutex
   4.96    4.67    5.00    0.07   87%  12% Lock [unlocked] mutex
   5.84    5.67    6.00    0.17   53%  46% Unlock [locked] mutex
   4.83    4.67    5.00    0.17  100%  50% Trylock [unlocked] mutex
   4.13    4.00    4.33    0.16   62%  62% Trylock [locked] mutex
   0.50    0.33    0.67    0.17  100%  50% Destroy mutex
  32.27   32.00   32.33    0.10   81%  18% Unlock/Lock mutex

   1.46    1.33    1.67    0.16   62%  62% Create mbox
   0.33    0.33    0.33    0.00  100% 100% Peek [empty] mbox
   5.46    5.33    5.67    0.16   62%  62% Put [first] mbox
   0.29    0.00    0.33    0.07   87%  12% Peek [1 msg] mbox
   5.46    5.33    5.67    0.16   62%  62% Put [second] mbox
   0.29    0.00    0.33    0.07   87%  12% Peek [2 msgs] mbox
   5.58    5.33    5.67    0.13   75%  25% Get [first] mbox
   5.58    5.33    5.67    0.13   75%  25% Get [second] mbox
   4.63    4.33    4.67    0.07   87%  12% Tryput [first] mbox
   4.33    4.33    4.33    0.00  100% 100% Peek item [non-empty] mbox
   5.08    5.00    5.33    0.13   75%  75% Tryget [non-empty] mbox
   4.25    4.00    4.33    0.13   75%  25% Peek item [empty] mbox
   4.42    4.33    4.67    0.13   75%  75% Tryget [empty] mbox
   0.42    0.33    0.67    0.13   75%  75% Waiting to get mbox
   0.42    0.33    0.67    0.13   75%  75% Waiting to put mbox
   1.50    1.33    1.67    0.17  100%  50% Delete mbox
  22.02   22.00   22.33    0.04   93%  93% Put/Get mbox

   0.92    0.67    1.00    0.13   75%  25% Init semaphore
   4.00    4.00    4.00    0.00  100% 100% Post [0] semaphore
   4.54    4.33    4.67    0.16   62%  37% Wait [1] semaphore
   4.00    4.00    4.00    0.00  100% 100% Trywait [0] semaphore
```

```
    4.00    4.00    4.00    0.00  100% 100% Trywait [1] semaphore
    1.00    1.00    1.00    0.00  100% 100% Peek semaphore
    0.50    0.33    0.67    0.17  100%  50% Destroy semaphore
   19.92   19.67   20.00    0.13   75%  25% Post/Wait semaphore

    1.54    1.33    1.67    0.16   62%  37% Create counter
    0.46    0.33    0.67    0.16   62%  62% Get counter value
    0.46    0.33    0.67    0.16   62%  62% Set counter value
    4.92    4.67    5.00    0.13   75%  25% Tick counter
    0.50    0.33    0.67    0.17  100%  50% Delete counter

    0.88    0.67    1.00    0.16   62%  37% Init flag
    4.38    4.33    4.67    0.07   87%  87% Destroy flag
    4.00    4.00    4.00    0.00  100% 100% Mask bits in flag
    4.33    4.33    4.33    0.00  100% 100% Set bits in flag [no waiters]
    6.92    6.67    7.00    0.13   75%  25% Wait for flag [AND]
    6.79    6.67    7.00    0.16   62%  62% Wait for flag [OR]
    6.92    6.67    7.00    0.13   75%  25% Wait for flag [AND/CLR]
    6.83    6.67    7.00    0.17  100%  50% Wait for flag [OR/CLR]
    0.33    0.33    0.33    0.00  100% 100% Peek on flag

    2.67    2.67    2.67    0.00  100% 100% Create alarm
    8.63    8.33    8.67    0.07   87%  12% Initialize alarm
    3.92    3.67    4.00    0.13   75%  25% Disable alarm
    7.92    7.67    8.00    0.13   75%  25% Enable alarm
    4.67    4.67    4.67    0.00  100% 100% Delete alarm
    5.88    5.67    6.00    0.16   62%  37% Tick counter [1 alarm]
   40.75   40.67   41.00    0.13   75%  75% Tick counter [many alarms]
   11.54   11.33   11.67    0.16   62%  37% Tick & fire counter [1 alarm]
  232.75  232.67  233.00    0.13   75%  75% Tick & fire counters [>1 together]
   46.75   46.67   47.00    0.13   75%  75% Tick & fire counters [>1 separately]
   32.33   32.33   32.33    0.00  100% 100% Alarm latency [0 threads]
   35.22   33.33   39.67    1.90   78%  78% Alarm latency [2 threads]
   35.22   32.33   39.67    1.90   77%  52% Alarm latency [many threads]
   54.37   54.33   59.00    0.07   99%  99% Alarm -> thread resume latency

    6.65    6.33    7.00    0.00            Clock/interrupt latency

   12.95   11.33   19.67    0.00            Clock DSR latency

  292    292     292  (main stack:   772)  Thread stack used (1360 total)
           All done, main stack : stack used   772 size  3920
            All done :  Interrupt stack used   208 size  4096
            All done : Idlethread stack used   248 size  2048

Timing complete - 30250 ms total

PASS:<Basic timing OK>
EXIT:<done>
```

Other Issues

The AT91SAM7X-EK platform HAL does not affect the implementation of other parts of the eCos HAL specification. The AT91SAM7 processor HAL, AT91 variant HAL, and the ARM architectural HAL documentation should be consulted for further details.

Chapter 241. NXP LPC2xxx variant HAL

Name

eCos Support for the NXP LPC2xxx ARM microcontrollers — Overview

Description

The NXP LPC2xxx series of ARM microcontrollers is supported by eCos with an eCos processor variant HAL and a number of device drivers supporting some of the on-chip peripherals. These include device drivers for the on-chip serial, watchdog, RTC (wallclock) and Flash devices. In addition it provides common functionality and definitions that LPC2xxx based platform ports may require, as well as definitions useful to application developers.

This documentation covers the LPC2xxx functionality provided but should be read in conjunction with the specific HAL documentation for the platform port. That documentation will cover issues that are platform-specific and are not covered here, and may also describe differences that override or supersede what the LPC2xxx variant HAL provides. The areas that are specific to platform HALs and not the LPC2xxx variant HAL include:

- memory map and related configuration and setup
- memory remapping
- External Memory Controller (EMC) and Memory Accelerator Module (MAM) setup (if applicable)
- Definitions of clock (OSC) inputs to PLL
- PLL setup
- PINSEL and GPIO setup
- Any special handling for external interrupts, or additional interrupts
- Diagnostic I/O baud rates
- Additional diagnostic I/O devices, if any
- LED control

This variant HAL provides helper macros and defines for some of these, but their use or otherwise is governed by the platform HAL.

Name

On-chip subsystems and peripherals — Hardware support

Hardware support

On-chip memory

The NXP LPC2xxx parts include a small amount of on-chip SRAM, and a limited amount of on-chip Flash. Specific capacities vary between parts. The SRAM is generally too small to be useful to RedBoot and some form of external RAM is employed if remote debugging is required, although some limited applications can be run if a GDB stub ROM image is programmed to internal Flash instead. Otherwise for processor models with no external RAM, applications must be programmed directly to internal Flash. The platform HAL may opt to use the SRAM to store the interrupt vectors mapped to address 0, or as a buffer for reprogramming internal flash when using the LPC2xxx Flash driver. The on-chip Flash is generally sufficient to include RedBoot or a GDB stub ROM image, although the on-chip SRAM may not be - again consult the platform HAL documentation. At this time, there is no support for initial programming of the on-chip Flash and so the NXP LPC2000 Flash Utility, Flash Magic or a JTAG/ICE is generally used for this.

Typically, an eCos platform HAL port will expect a RedBoot image to be programmed into the LPC2xxx on-chip Flash memory for development, and the board would boot this image from reset. RedBoot provides gdb stub functionality so it is then possible to download and debug stand-alone and eCos applications via the gdb debugger using serial interfaces or other debug channels.

Serial I/O

The LPC2xxx variant HAL supports basic polled HAL diagnostic I/O over either of the two on-chip serial devices. There is also a fully interrupt-driven serial device driver suitable for eCos applications for both on-chip serial devices. The serial driver consists of two eCos packages: `CYGPKG_IO_SERIAL_GENERIC_16X5X` which is a “generic” package for 16x5x compatible serial devices; and `CYGPKG_IO_SERIAL_ARM_LPC2XXX` which provides more specific definitions for the LPC2xxx on-chip serial devices. Using the HAL diagnostic I/O support, either of these devices can be used by RedBoot for communication with the host. If you are only using UART 0, a small amount of memory can be saved by reducing the number of communication channels with the CDL option `CYGNUM_HAL_VIRTUAL_VECTOR_COMM_CHANNELS` from 2 to 1 if the platform HAL permits this. It is not possible to enable UART 1 but not UART 0 at this time. If a serial device is needed by the application, either directly or via the serial driver, then it cannot also be used for RedBoot communication with the HAL I/O support. The alternative serial port should be used instead, if available on the platform. The serial driver supports the line status and modem control (including hardware handshaking) lines on UART 1 only.

Watchdog

A device driver is included for the on-chip watchdog device. This driver allows the use of the standard eCos watchdog API accessible with the `CYGPKG_IO_WATCHDOG` eCos package. If the watchdog is not reset within a time period defined in the watchdog device driver CDL, an interrupt is generated and a user-supplied function called. Alternatively it may be configured to automatically reset the system.

The watchdog device is also used to implement reset functionality, such as required by the RedBoot **reset** command. It may also be called directly by applications using the following function:

```
#include <cyg/hal/hal_diag.h>
extern void hal_lpc2xxx_reset_cpu(void);
```

RTC/Wallclock

Support is provided for the on-chip RTC (wallclock) device. This allows the use of the standard eCos wallclock API accessible with the `CYGPKG_IO_WALLCLOCK` eCos package. The wallclock is also used by other eCos subsystems such as the C library and POSIX compatibility layer to provide calendar time functionality.

Interrupt controller

eCos manages the on-chip Vectored Interrupt Controller (VIC). The VIC is only configured to use interrupts in non-vectored mode.

Timers

Timer 0 is used to implement the eCos system clock. If the gprof package, CYGPKG_PROFILE_GPROF, is included in the configuration, then timer 1 is reserved for use by the profiler. Any remaining timers are available for application use.

I²C Bus

The on-chip I²C bus devices are supported by the NXPI2C driver package, CYGPKG_DEVS_I2C_NXPI2C. Platform HALs need to enable the attached buses and define the clock speed and lines to be used for SDA and SCL.

SPI Bus

The on-chip SSP SPI devices (not the Legacy SPI device) are supported by the PL022 driver package, CYGPKG_DEVS_SPI_ARM_PL022. This needs some configuration in the platform HAL to enable the attached buses and define the GPIO lines used for chip select.

Other

Other on-chip devices (SPI, PWM, A/D converter, etc.) are not touched by the LPC2xxx variant HAL and unless used by the platform HAL are free for use for applications.

Name

HAL Port — Implementation Details

Overview

This section covers any remaining items of note related to the LPC2xxx variant support, not covered in previous sections.

LEDs

If a platform port has support for display values on LEDs, that support is standardised to be accessible from C with the following function:

```
#include <cyg/infra/hal_diag.h>
extern void hal_diag_led(int leds);
```

PLL configuration

CDL variables related to the PLL configuration are standardised across LPC2xxx implementations. The platform HAL must provide the input oscillator frequency (`CYGNUM_HAL_ARM_LPC2XXX_OSC_FREQ`), as well as the desired PLL multipliers and dividers selected by the user and relevant for the chosen part (`CYGNUM_HAL_ARM_LPC2XXX_PLL_MULTIPLIER` and `CYGNUM_HAL_ARM_LPC2XXX_PLL_DIVIDER`). It must also supply the `VPB` divider (`CYGNUM_HAL_ARM_LPC2XXX_VPB_DIVIDER`) which divides down the core clock (`CCLK`) to generate the peripheral clock (`PCLK`). Where applicable a platform may also define a `CCLK` divider (`CYGNUM_HAL_ARM_LPC2XXX_CCLK_DIVIDER`).

As a result, the variant HAL calculates CDL options for the absolute values of `CCLK` (`CYGNUM_HAL_ARM_LPC2XXX_CCLK_SPEED`) and `PCLK` (`CYGNUM_HAL_ARM_LPC2XXX_PCLK_SPEED`) which are exported to the rest of the system, and accessible to applications from `<pkgconf/hal.h>`.

It is still the responsibility of the platform HAL to initialize the PLL, although assembler helper macros are provided in `<cyg/hal/var_io.h` to ease implementation.

LPC2xxx definitions

The LPC2xxx variant HAL port includes the header file `var_io.h` which provides useful register definitions used by eCos, but can also be freely used by applications. It includes register definitions for subsystems unused by eCos.

It may be found in the `include/cyg/hal` directory relative to your configuration's install tree, or alternatively in the source repository at `hal/arm/lpc2xxx/var/`*`VERSION`*`/include/var_io.h`. However it should be properly included by applications by using:

```
#include <cyg/hal/hal_io.h>
```

This will allow for platform HALs to augment or override any relevant definitions.

Power control

The kernel idle thread is scheduled to run when the system has no other tasks able to run. The idle thread can call a HAL supplied macro to place the chip into an appropriate power saving mode instead of just going around a busy loop. The LPC2xxx variant HAL defines the `HAL_IDLE_THREAD_ACTION` macro to use the LPC2xxx power control support to place the chip into `IDLE` mode which will stop the processor clock, without disabling the on-chip peripherals. This state continues until an interrupt is received. This mode has no deleterious effect on program execution, however it has been known to interfere with JTAG/ICE hardware debuggers. Therefore the CDL option `CYGIMP_HAL_ARM_LPC2XXX_IDLE_THREAD_USES_IDLE` exists in the variant HAL to ensure the processor does not enter the idle mode from the idle thread. It is recommended this option be disabled if hardware debugging solutions are used, especially if reliability is erratic.

Unless specified otherwise by the platform HAL port, no other power saving features are used and no peripherals are disabled, even those unsupported by the eCos LPC2xxx variant port. Therefore if the application wishes to conserve power it is its responsibility to place the relevant peripherals into power down modes.

Memory Acceleration Module support

The variant HAL supplies helper macros to the platform HALs to centralise initialisation code for common subsystems, including the Memory Acceleration Module (MAM).

However it is known that there are errata which affect the MAM, and specifically restrict what mode the MAM should operate in. In some cases, such as the LPC2148, there are conflicting errata, that recommend that the MAM be used only in `Full` mode in some cases to avoid one erratum, or only `Partial` mode in others to avoid a different erratum.

Therefore the choice of MAM mode is left to the user and can be set with the CDL option `CYGHWR_HAL_ARM_LPC2XXX_MAM_MODE` in the variant HAL.

Virtual Vector support

As described in the common HAL documentation, virtual vectors are used to abstract certain services which could be shared between a resident ROM monitor, and an application. In the case where an application is entirely stand-alone, and does not use any resident ROM monitor - for example, if it is itself a ROM application - then virtual vector support is unnecessary.

The CDL configuration option `CYGFUN_HAL_LPC2XXX_VIRTUAL_VECTOR_SUPPORT` can be used to control the presence of virtual vectors, although it is expected that the default value will be selected appropriately in any case. Disabling virtual vector support can save both Flash and RAM use, which can be important for targets with restricted memory.

Chapter 242. Ashling EVBA7 Eval Board Support

Name

eCos Support for the Ashling EVBA7 Eval Board — Overview

Description

The Ashling EVBA7 Eval Board is fitted with a Philips LPC2000 processor rated to 60MHz, which contains up to 64KB of SRAM and up to 256KB of FLASH. The board has two 9-pin RS-232 serial interfaces connected to the LPC2000 on-chip UARTs, an LED bank and JTAG/USB debug interfaces. Refer to the board documentation for full details.

The standard EVBA7 is fitted with an LPC2106 microcontroller. Some versions of the EVBA7 board are fitted with an adaptor that either contains a specific LPC2000 part, or a socket into which one of several LPC2000 parts may be fitted.

For typical eCos development, a RedBoot or GDB Stubrom image is programmed into the LPC2000 on-chip flash memory, and the board will boot this image from reset. Both RedBoot and the GDB stub ROM provide GDB stub functionality so it is then possible to download and debug stand-alone and eCos applications via the gdb debugger using UART 0.

This documentation describes platform-specific elements of the EVBA7 Eval Board support within eCos. Documentation on the Philips LPC2xxx variants is available separately, and should be read in conjunction with this documentation. The LPC2xxx documentation covers various topics including HAL support common to LPC2xxx variants, and on-chip device support. This document complements the LPC2xxx documentation.

Supported Hardware

The EVBA7 Eval Board has up to 128Kbyte of on-chip Flash memory. In a typical setup, RedBoot or the GDB Stubrom will load and run from this internal flash. No support for managing internal Flash is included in this port - the Ashling FlashLPC Utility is required to program the internal Flash. 24Kbytes of internal flash memory should be reserved for the GDB Stubrom, the remainder being free for the application's use. RedBoot will occupy the entire internal ROM.

EVBA7 boards fitted with the PA-EVBA7-144 adaptor also have 1MByte of external RAM on the adaptor. In this case, eCos is configured to use this memory rather than the internal SRAM. These are also the only boards capable of running RedBoot.

The first 64 bytes of on-chip SRAM are mapped by the HAL startup code using the LPC2000 memory mapping control to location 0x00000000 for speed of interrupt vector processing. SRAM from location 0x40000040 to 0x40001000 is used by the GDB Stubrom. The rest of SRAM is available for use by the application.

The Philips LPC2xxx variant HAL includes support for the two LPC2000 on-chip serial devices and is documented in the variant HAL. The interrupt-driven serial driver supports the line status and modem control (including hardware handshaking) lines on UART1 only.

The EVBA7 Eval Board port includes support for the on-chip watchdog, RTC (wallclock), and interrupt controller (VIC). This support is documented in the LPC2xxx variant HAL.

Tools

The EVBA7 Eval Board port is intended to work with GNU tools configured for an arm-eabi target. Thumb mode is supported. The original port was done using arm-elf-gcc version 3.3.3, arm-elf-gdb version 6.1, and binutils version 2.14.

Name

Setup — Preparing the EVBA7 Eval Board for eCos Development

Overview

In a typical development environment, the EVBA7 Eval Board boots from internal flash into the GDB stubrom monitor or RedBoot. eCos applications are configured for RAM startup and then downloaded and run on the board via the debugger **arm-eabi-gdb**. Preparing the board therefore usually involves programming a suitable ROM or RedBoot image into flash memory.

For serial communications, all versions run with 8 bits, no parity, and 1 stop bit at 38400 baud. This baud rate can be changed via the configuration option `CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL_BAUD` and rebuilding the stubrom or RedBoot.

Initial Installation

Flash Installation

This process assumes that a Microsoft Windows machine with the Ashling FlashLPC Utility installed is available. The first step is to connect the RS232 cable supplied with the EVBA7 between port 0 of the EVBA7 and the host PC. Now install the jumper labelled “ENABLE SERIAL ISP”. On the base EVBA7 this is jumper JP6; on boards fitted with one of the FA-EVBA7 adaptors, this is JP5 on the adaptor; on boards fitted with a PA-EVBA7 adaptor, this is JP3 on the adaptor. Refer to the board documentation for full details. Apply the power, or press the reset button.

The board is now running a special Philips boot loader. Start the FlashLPC Utility, and ensure that the selected device matches the device installed on the EVBA7. Choose the appropriate COM port that is being used on your PC and select 115200 baud for both the initial and final baud rates. Set the Crystal KHz value to 14745, ensure that the stop bits value is set to 1 and the packet size is set to 100%. Establish communication with the board by pressing the “Connect” button.

Now in the “Flash Programming” section, select the `stubrom.srec` or `redboot.srec` file. Set the file format to “S-Record Format”,select “Erase individual sectors before programming” under the Programming Options and ensure that “Automatically add checksum to vector table” is ticked. Finally press “Program”, or “Program and Verify” to program the ROM image.

When the process completes, remove the “ENABLE SERIAL ISP” jumper. Verify the programming has been successful by starting a terminal emulation application such as HyperTerminal on the host PC and set the serial communication parameters to 38400 baud, 8 data bits, no parity, 1 stop bit (8N1) and no flow control (handshaking). Reset the board and the stubrom should start. For boards programmed with GDB stubs the output should be similar to the following:

```
+$T050f:ec070000;0d:28080040;#52
```

This is the stubrom attempting to communicate with GDB and indicates that it is functioning correctly.

For boards fitted with RedBoot, you should see the RedBoot startup messages ending with a RedBoot prompt.

Rebuilding GDB Stubrom

Should it prove necessary to rebuild the Stubrom binary, this is done most conveniently at the command line. Assuming your `PATH` and `ECOS_REPOSITORY` environment variables have been set correctly, the steps needed to rebuild the stubrom are:

```
$ mkdir stub_rom
$ cd stub_rom
$ ecosconfig new evba7 stubs
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

At the end of the build the `install/bin` subdirectory should contain the file `stubrom.srec`.

Rebuilding RedBoot

Should it prove necessary to rebuild the RedBoot binary, this is done most conveniently at the command line. Assuming your PATH and ECOS_REPOSITORY environment variables have been set correctly, the steps needed to rebuild RedBoot are:

```
$ mkdir evba7_redboot
$ cd evba7_redboot
$ ecosconfig new evba7_2294 redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/arm/lpc2000/evba7/current/misc/redboot_ROM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

At the end of the build the install/bin subdirectory should contain the file redboot.srec.

Name

Configuration — Platform-specific Configuration Options

Overview

The evba7 platform HAL package is loaded automatically when eCos is configured for a `evba7` target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The evba7 platform HAL package supports two separate startup types:

RAM This is the startup type which is normally used during application development. The board has the GDB stubrom or RedBoot programmed into flash at location 0x0 in internal on-chip Flash and boots from that location. arm-eabi-gdb is then used to load a RAM startup application into memory and debug it. It is assumed that the hardware has already been initialized by the ROM monitor. By default the application will use the eCos virtual vectors mechanism to obtain certain services from the ROM monitor, including diagnostic output.

ROM This startup type can be used for finished applications which will be programmed into internal flash at location 0x0. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization.

The ROM Monitor and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building the GDB Stubrom or RedBoot.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, or as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then serial port UART0 will be claimed for HAL diagnostics.

System Clock

By default, the system clock interrupts once every 10ms, corresponding to a 100Hz clock. This can be changed by the configuration option `CYGNUM_HAL_RTC_DENOMINATOR` which corresponds to the clock frequency. Other clock-related settings are recalculated automatically if the denominator is changed. The PLL multipliers and dividers may be configured to allow a core clock (CCLK) speed of up to 60MHz. The description of the clock-related CDL options may be found in the LPC2xxx variant HAL documentation.

Compiler Flags

The platform HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos.

However there are two flags that are used if Thumb mode is to be supported:

`-mthumb` The arm-eabi-gcc compiler will compile C and C++ files into the Thumb instruction set when this option is used.

`-mthumb-interwork` This option allows programs to be created that mix ARM and Thumb instruction sets. Without this option, some memory can be saved. This option should be used if -mthumb is used.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the EVBA7 Eval Board hardware, and should be read in conjunction with that specification. The EVBA7 Eval Board platform HAL package complements the ARM architectural HAL and the LPC2xxx variant HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset the HAL will initialize or reinitialize many of the on-chip peripherals. This includes the PINSEL functions and LED bank. There is an exception for RAM startup applications which depend on a ROM monitor for certain services.

For ROM startup, the HAL will perform additional initialization, programming the various internal registers including PLL (for the clocks), Memory Mapping control registers to map SRAM to 0x0, and Memory Acceleration Module (MAM). The details of the early hardware startup may be found in the header `cyg/hal/hal_platform_setup.h`.

Linker Scripts and Memory Maps

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

on-chip Flash

This is located at address 0x0 of the memory space, although after hardware initialization, the start of internal SRAM is mapped over locations 0x0 to 0x40. This region ends at 0x20000. No flash driver is provided for the on-chip Flash. The MAM is enabled to accelerate memory reads from this area.

internal SRAM

This is located at address 0x40000000 of the memory space, and is 16, 32 or 64k in size, depending on the chip fitted. The first 64 bytes are mapped to location 0x0000000. When this is the only RAM available, the virtual vector table starts at 0x40000050 and extends to 0x40000150. The remainder of SRAM is available for use by ROM based applications. For RAM startup applications, SRAM below 0x40001000 is reserved for the GDB stubrom and the remainder is available for the application.

On boards fitted with the PA-EVBA7-144 adaptor and where the external SRAM is being used, only the first 64 bytes are used as described above. The remainder of internal SRAM is not used by eCos.

external SRAM

This SRAM is only present if the EVBA7 is fitted with a PA-EVBA7-144 adaptor. It is located at address 0x81000000 of the memory space, and is 1MByte in size. When this memory is being used for applications the virtual vector table starts at 0x81000050 and extends to 0x81000150. The remainder is available for use by ROM based applications. For RAM startup applications, memory below 0x81010000 is reserved for RedBoot and the remainder is available for the application.

on-chip peripherals

These are accessible at location 0xE0000000 onwards. Descriptions of the contents can be found in the LPC2000 User Manual.

Other Issues

The LEDs may be accessed from C with the following function:

```
#include <cyg/infra/hal_diag.h>
extern void hal_diag_led(int leds);
```

Values from 0 to 16 will be displayed on the LED bank representing the binary value with 1 being on and 0 being off, and with P0.7 being the MSB, and P0.4 the LSB.

The LEDs are also used during platform initialization and only P0.4 should be illuminated if booting has been successful. Other LED indications represent the stage in the initialization process that failed.

Chapter 243. Embedded Artists LPC2468 OEM Board Support

Name

eCos Support for the Embedded Artists LPC2468 OEM Board — Overview

Description

The Embedded Artists LPC2468 OEM Board is fitted with an NXP LPC2468 processor rated up to 72MHz, which contains 64KB of SRAM and 512KB of FLASH. When used in conjunction with the OEM base board, it provides access to two on-chip UARTs (one via USB, one via a 9-pin connector), a single GPIO LED, an MMC/SD card socket, and a PHY connected to the on-chip Ethernet MAC. Refer to the board documentation for full details.

Two variants of the LPC2468 OEM board are supported. The LPC2468-16 board provides a 16 bit data bus to external RAM, works with all versions of the base board, and supports UART 1. The LPC2468-32 board provides a 32 bit data bus to external RAM, works only with v1.4 and later versions of the base board, and does not allow UART 1 to be used.

For typical eCos development, a RedBoot image is programmed into the LPC2468 on-chip flash memory, and the board will boot this image from reset. RedBoot provides gdb stub functionality so it is then possible to download and debug stand-alone and eCos applications via the gdb debugger using either UART or via the ethernet.

This documentation describes platform-specific elements of the LPC2468 OEM Board support within eCos. Documentation on the NXP LPC2xxx variants is available separately, and should be read in conjunction with this documentation. The LPC2xxx documentation covers various topics including HAL support common to LPC2xxx variants, and on-chip device support. This document complements the LPC2xxx documentation.

Supported Hardware

The LPC2468 OEM Board has 512Kbyte of on-chip Flash memory. In a typical setup, RedBoot will load and run from this internal flash. An initial image must be programmed into this flash using either the FlashMagic utility, or via a JTAG debugger. Following this, it may be reprogrammed using flash drivers in RedBoot.

The first 64 bytes of on-chip SRAM are mapped by the HAL startup code using the LPC2468 memory mapping control to location 0x00000000 for speed of interrupt vector processing. The rest of SRAM is available for use by the application. 4MB of external NOR flash is available at 0x80000000; the topmost 64K of this is used by RedBoot for configuration data, the rest is available for application use and can be managed by RedBoot's flash file system. 32 MB of SDRAM is available at 0x81000000; the first 1MByte of this is reserved for use by RedBoot, the rest is available for the code and data of loaded applications.

The NXP LPC2xxx variant HAL includes support for the on-chip serial devices which is documented in the variant HAL. The interrupt-driven serial driver supports the line status and modem control (including hardware handshaking) lines on UART1 only.

The LPC2468 OEM Board port includes support for the on-chip watchdog, RTC (wallclock), and interrupt controller (VIC). This support is documented in the LPC2xxx variant HAL.

The on-chip Ethernet MAC is supported. The LPC2468-16 and LPC2468-32 use different PHYs, and both of these are supported.

The on-chip Multimedia Card Interface (MCI) is supported to allow access to Multimedia Cards (MMC) or Secure Digital (SD) cards using the socket on the OEM board.

Tools

The LPC2468 OEM Board port is intended to work with GNU tools configured for an arm-eabi target. Thumb mode is supported. The original port was done using arm-elf-gcc version 3.4.4, arm-elf-gdb version 6.3, and binutils version 2.16.

Name

Setup — Preparing the LPC2468 OEM Board for eCos Development

Overview

In a typical development environment, the LPC2468 OEM Board boots from internal flash into RedBoot. eCos applications are configured for RAM startup and then downloaded and run on the board via the debugger **arm-eabi-gdb**. Preparing the board therefore usually involves programming a suitable RedBoot image into flash memory.

The following RedBoot configurations are supported:

Configuration	Description	Use	File
ROM	RedBoot running from ROM	`redboot_ROM.ecm`	`redboot_ROM.hex`
RAM	RedBoot running from RAM	`redboot_RAM.ecm`	`redboot_RAM.srec`

For serial communications, all versions run with 8 bits, no parity, and 1 stop bit at 38400 baud. This baud rate can be changed via the RedBoot **baud** command.

Initial Installation

Flash Installation

This process assumes that a Microsoft Windows machine with the Embedded Systems Academy Flash Magic utility is available.

The first step is to set up the board as described in the Embedded Artists documentation. The FlashMagic tool from http://www.flashmagictool.com must be installed to program applications or RedBoot into flash. Older versions of Windows may also require the FTDI USB driver from http://www.ftdichip.com/Drivers/VCP.htm be installed and configured as described in the Embedded Artists documentation.

Install the ISP jumpers (P2.10 and RESET) and press the reset button. The board is now running a special NXP boot loader. Start FlashMagic and set the Serial Port to the FTDI USB COM*x* device activated when the target hardware is connected to the host by a suitable USB cable. Select 38400 baud and change the device to LPC2468. Older versions of FlashMagic also require the Interface “None (ISP)” and 12MHz Oscillator Frequency - these settings are available under preferences in newer versions and are set correctly by default.

Test communication with the board by using the “ISP->Read Device Signature” menu entry. If communication is not successful, check that the correct USB cable is used and connected, the ISP jumpers are installed and the correct COM port is being used.

Check “Erase blocks used by Hex File” under “Erase” OR in recent versions of FlashMagic select “Sectors used by File” next to “Erase”. In the “File” section, select the `redboot_ROM.hex` file. Under “Options”, all boxes should be clear except “Verify after programming”. Now press the “Start” button. The utility should show the progress of the flash erase and write operations.

When the process completes, the utility should be closed. Verify the programming has been successful by starting a terminal emulation application such as HyperTerminal or minicom on the host PC and set the serial communication parameters to 38400 baud, 8 data bits, no parity, 1 stop bit (8N1) and no flow control (handshaking). Remove the ISP jumpers. Reset the board and RedBoot should start. The output should be similar to the following:

```
+**Warning** FLASH configuration checksum error or invalid key
Use 'fconfig -i' to [re]initialize database
Ethernet eth0: MAC address 0e:00:00:ea:18:f0
IP: 10.0.2.8/255.0.0.0, Gateway: 10.0.0.3
Default server: 0.0.0.0, DNS server IP: 10.0.0.1

RedBoot(tm) bootstrap and debug environment [ROM]
```

```
eCosCentric certified release, version 4.5.9 - built 10:17:51, Apr 26 2021

Copyright (C) 2000-2009 Free Software Foundation, Inc.
Copyright (C) 2003-2021 eCosCentric Limited
The RedBoot bootloader is a component of the eCos real-time operating system.
Want to know more? Visit www.ecoscentric.com for everything eCos & RedBoot related.
This is free software, covered by the eCosPro Non-Commercial Public License
and eCos Public License. You are welcome to change it and/or distribute copies
of it under certain conditions. Under the license terms, RedBoot's source code
and full license terms must have been made available to you.
Redboot comes with ABSOLUTELY NO WARRANTY.

Platform: Embedded Artists LPC2468 OEM Board (ARM7TDMI)
RAM: 0xa0000000-0xa2000000 [0xa000aa18-0xa1fed000 available]
FLASH: 0x00000000-0x0007dfff, 8 x 0x1000 blocks, 14 x 0x8000 blocks, 6 x 0x1000 blocks
FLASH: 0x80000000-0x803fffff, 64 x 0x10000 blocks
RedBoot>
```

It is now necessary to initialize the flash file system and the flash configuration. This can be done with the following commands:

```
RedBoot> fis init
About to initialize [format] FLASH image system - continue (y/n)? y
*** Initialize FLASH Image System
... Erase from 0x803f0000-0x803fffff: .
... Program from 0xa1ff0000-0xa2000000 to 0x803f0000: .
RedBoot> fconfig -i
Initialize non-volatile configuration - continue (y/n)? y
Run script at boot: false
Use BOOTP for network configuration: true
Default server IP address: 10.0.1.2
Console baud rate: 38400
DNS server IP address: 10.0.0.1
Network hardware address [MAC] for eth0: 0x0E:0x00:0x00:0xEA:0x18:0xF0
GDB connection port: 9000
Force console for special debug messages: false
Network debug at boot time: false
Default network device: lpc2xxx
Update RedBoot non-volatile configuration - continue (y/n)? y
... Erase from 0x803f0000-0x803fffff: .
... Program from 0xa1ff0000-0xa2000000 to 0x803f0000: .
RedBoot>
```

Replace the IP addresses in the above with those for your own network. The above also accepts the default for the MAC address, if more than one LPC2468 is to be used on the same network then different MAC addresses should be used; Embedded Artists boards are supplied with a sticker showing an assigned MAC address, and this should be used by preference.

It is it ever necessary to reinstall RedBoot, the above directions can be repeated. Alternatively, a new RedBoot may be installed from RedBoot itself. It is not possible to do this directly, since RedBoot is executing from the flash that needs to be erased and reprogrammed. Instead it is necessary to run a RAM version of RedBoot, use that to download the new ROM RedBoot to RAM, and then program that to flash.

The following shows an example session to do this. It assumes that `redboot_RAM.srec` and `redboot_ROM.bin` are available via TFTP on the server set up in **fconfig**.

```
RedBoot> load redboot_RAM.srec
Using default protocol (TFTP)
Entry point: 0xa0100040, address range: 0xa0100000-0xa011bab4
RedBoot> go
+Ethernet eth0: MAC address 0e:00:00:ea:18:f0
IP: 10.0.2.8/255.0.0.0, Gateway: 10.0.0.3
Default server: 10.0.1.2, DNS server IP: 10.0.0.1

RedBoot(tm) bootstrap and debug environment [RAM]
eCosCentric certified release, version 4.5.9 - built 10:17:32, Apr 26 2021

```

```
Copyright (C) 2000-2009 Free Software Foundation, Inc.
Copyright (C) 2003-2021 eCosCentric Limited
The RedBoot bootloader is a component of the eCos real-time operating system.
Want to know more? Visit www.ecoscentric.com for everything eCos & RedBoot related.
This is free software, covered by the eCosPro Non-Commercial Public License
and eCos Public License. You are welcome to change it and/or distribute copies
of it under certain conditions. Under the license terms, RedBoot's source code
and full license terms must have been made available to you.
Redboot comes with ABSOLUTELY NO WARRANTY.

Platform: Embedded Artists LPC2468 OEM Board (ARM7TDMI)
RAM: 0xa0000000-0xa2000000 [0xa0126970-0xa1fed000 available]
FLASH: 0x00000000-0x0007dfff, 8 x 0x1000 blocks, 14 x 0x8000 blocks, 6 x 0x1000 blocks
FLASH: 0x80000000-0x803fffff, 64 x 0x10000 blocks
RedBoot> load -r -b %{freememlo} redboot_ROM.bin
Using default protocol (TFTP)
Raw file loaded 0xa0125c00-0xa0142ba3, assumed entry at 0xa0125c00
RedBoot> fis write -f 0x00000000 -b %{freememlo} -l 0x20000
* CAUTION * about to program FLASH
            at 0x00000000..0x0001ffff from 0xa0125c00 - continue (y/n)? y
... Erase from 0x00000000-0x0001ffff: ...........
... Program from 0xa0125c00-0xa0145c00 to 0x00000000: ...........
RedBoot> reset
+Ethernet eth0: MAC address 0e:00:00:ea:18:f0
IP: 10.0.2.8/255.0.0.0, Gateway: 10.0.0.3
Default server: 10.0.1.2, DNS server IP: 10.0.0.1

RedBoot(tm) bootstrap and debug environment [ROM]
eCosCentric certified release, version 4.5.9 - built 10:17:51, Apr 26 2021

Copyright (C) 2000-2009 Free Software Foundation, Inc.
Copyright (C) 2003-2021 eCosCentric Limited
The RedBoot bootloader is a component of the eCos real-time operating system.
Want to know more? Visit www.ecoscentric.com for everything eCos & RedBoot related.
This is free software, covered by the eCosPro Non-Commercial Public License
and eCos Public License. You are welcome to change it and/or distribute copies
of it under certain conditions. Under the license terms, RedBoot's source code
and full license terms must have been made available to you.
Redboot comes with ABSOLUTELY NO WARRANTY.

Platform: Embedded Artists LPC2468 OEM Board (ARM7TDMI)
RAM: 0xa0000000-0xa2000000 [0xa000aa18-0xa1fed000 available]
FLASH: 0x00000000-0x0007dfff, 8 x 0x1000 blocks, 14 x 0x8000 blocks, 6 x 0x1000 blocks
FLASH: 0x80000000-0x803fffff, 64 x 0x10000 blocks
RedBoot>
```

Rebuilding RedBoot

Should it prove necessary to rebuild the RedBoot binary, this is done most conveniently at the command line. Assuming your `PATH` and `ECOS_REPOSITORY` environment variables have been set correctly, the steps needed to rebuild RedBoot for the LPC2468-32 are:

```
$ mkdir redboot_ealpc2468_rom
$ cd redboot_ealpc2468_rom
$ ecosconfig new ea_lpc2468_32 redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/arm/lpc2xxx/ea_lpc2468/current/misc/redboot_ROM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

At the end of the build the `install/bin` subdirectory should contain the file `redboot.hex`.

Substitute 16 for 32 in the above to build RedBoot for the LPC2468-16 module.

Name

Configuration — Platform-specific Configuration Options

Overview

The ea_lpc2468 platform HAL package is loaded automatically when eCos is configured for an `ea_lpc2468_32` or `ea_lpc2468_16` target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The ea_lpc2468 platform HAL package supports two separate startup types:

RAM This is the startup type which is normally used during application development. The board has RedBoot programmed into flash at location 0x0 in internal on-chip Flash and boots from that location. arm-eabi-gdb is then used to load a RAM startup application into memory and debug it. It is assumed that the hardware has already been initialized by RedBoot. By default the application will use the eCos virtual vectors mechanism to obtain certain services from RedBoot, including diagnostic output.

ROM This startup type can be used for finished applications which will be programmed into internal flash at location 0x0. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization.

RedBoot and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building RedBoot.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, or as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then serial port UART0 will be claimed for HAL diagnostics.

Flash Driver

The LPC2468 OEM board contains an SST 39VF3201 NOR flash device. The `CYGPKG_DEVS_FLASH_SST_39VFXXX_V2` package contains all the code necessary to support this part and the platform HAL package contains definitions that customize the driver to the Embedded Artists LPC2468 OEM board.

Ethernet Driver

The LPC2468 contains an ethernet MAC device. The `CYGPKG_DEVS_ETH_ARM_LPC2XXX` package contains all the code necessary to support this device and the platform HAL package contains definitions that customize the driver to the LPC2468 OEM board.

System Clock

By default, the system clock interrupts once every 10ms, corresponding to a 100Hz clock. This can be changed by the configuration option `CYGNUM_HAL_RTC_DENOMINATOR` which corresponds to the clock frequency. Other clock-related settings are recalculated automatically if the denominator is changed. The PLL multipliers and dividers may be configured to allow a core clock

(CCLK) speed of up to 72MHz. However, the platform HAL currently sets the clock to 48MHz, duplicating the configuration in the supplied example code as a consequence of CPU errata affecting various revisions of the LPC2468. Setting the CPU revision with the `CYGHWR_HAL_ARM_LPC2XXX_EA_LPC2468_CPU_REVISION` configuration option can be used to provide default clock settings appropriate to the CPU revision in use. If the CPU revision cannot be guaranteed it should be left as "Initial". The description of the clock-related CDL options may be found in the LPC2xxx variant HAL documentation.

I²C Bus Configuration

The on-chip I²C devices are supported by a driver in the variant HAL package. Each bus for this driver needs to be configured in the platform HAL with the following options:

`CYGPKG_HAL_ARM_LPC2XXX_I2CX`

> This is the master component, enabling this activates all the other configuration options and causes the driver to create the data structures to access this bus.

`CYGPKG_HAL_ARM_LPC2XXX_I2CX_CLOCK`

> Bus clock speed in Hz. Usually frequencies of either 100kHz or 400kHz are chosen, the latter sometimes known as fast mode.

`CYGPKG_HAL_ARM_LPC2XXX_I2CX_SDA`

> This option describes the pin used for SDA on this bus. This takes the form of an invocation of the macro __LPC2XXX_PINSEL_FUNC. Parameters are the port number, pin within that port, and the alternate select function for the pin. See the LPC2468 user manual for details of which pins may be used by each bus.

`CYGPKG_HAL_ARM_LPC2XXX_I2CX_SCL`

> This option describes the pin used for SCL on this bus. Like SDA this takes the form of a call to __LPC2XXX_PINSEL_FUNC.

Note that "I2CX" is a placeholder for a given bus instance: "I2C0", "I2C1" or "I2C2". By default the platform HAL only enables I²C bus 0 in order to access the PCA9532 LED controller on the base board.

SPI Bus Configuration

The on-chip SSP SPI devices (not the Legacy SPI device) are supported by the NXPSSP driver package, `CYGPKG_DEVS_SPI_ARM_NXPSSP`. This needs some configuration in the platform HAL:

`CYGPKG_HAL_ARM_LPC2XXX_SPI`

> This is the master component, enabling this activates all the other configuration options. It also causes `ea_lpc2468_spi.c` to be compiled, which contains descriptions of the devices on the SPI buses.

`CYGPKG_HAL_ARM_LPC2XXX_SPIX`

> This is the master component for each bus. Enabling this activates the other configuration options for this bus, and causes the driver to support this bus.

`CYGPKG_HAL_ARM_LPC2XXX_SPIX_SCLK`

> This option describes the pin used for SCLK on SPIX. It takes the form of an invocation of __LPC2XXX_PINSEL_FUNC. The parameters are the port number, pin within that port, and the alternate select function for the pin. See the LPC2468 user manual for details."

`CYGPKG_HAL_ARM_LPC2XXX_SPIX_MISO`

> This option describes the pin used for MISO on SPIX. Like SCLK it takes the form of a call to __LPC2XXX_PINSEL_FUNC.

CYGPKG_HAL_ARM_LPC2XXX_SPIX_MOSI

This option describes the pin used for MOSI on SPIX. Like SCLK it takes the form of a call to __LPC2XXX_PINSEL_FUNC.

CYGPKG_HAL_ARM_LPC2XXX_SPIX_CS_PINS

This defines the pins to be uses as chip selects for this bus. It is a comma separated list of GPIO pin names, the first for device 0, the second for device 1, and so on. Pin names are defined in the var_io.h header in the LPC2xxx variant HAL.

Note that "SPIX" is a placeholder for a given bus instance: "SPI0" or "SPI1". By default the platform HAL only enables SPI0, for testing only.

MCI peripheral configuration

The on-chip Multimedia Card Interface (MCI) is supported to allow access to Multimedia Cards (MMC) or Secure Digital (SD) cards using the socket on the OEM board. This support is provided in conjunction with the generic MMC/SD driver package (CYGPKG_DEVS_DISK_MMC), the Primecell MCI driver package (CYGPKG_DEVS_MMCSD_ARM_PRIMECELL_MCI) and the LPC2xxx variant HAL in order to provide some elements of the DMA support. Documentation and configuration options within those packages should also be consulted. Note that the miniSD socket on the CPU board is not supported.

In order to configure the hardware for access to the socket, Jumper J47 on the base board must be set with pins 2-3 connected (P0.22 selected for MCIDAT0), and Jumper J27 must be set with MCIPWR active low.

The following CDL configuration options are used to control the behaviour of the MMC/SD card support:

MMC/SD card support (CYGPKG_HAL_ARM_LPC2XXX_EA_LPC2468_MCI)

This option allows the MMC/SD card support as a whole to be enabled or disabled, although the generic disk device driver package (CYGPKG_IO_DISK) must be loaded in order to enable the MMC/SD support.

Use on-chip USB memory for DMA (CYGSEM_HAL_ARM_LPC2XXX_EA_LPC2468_MCI_USE_USB_MEM_FOR_DMA

The LPC2468 cannot always keep up with the data transfer requirements, especially at slower CPU clock speeds. This is because the DMA controller runs at the speed of the CPU clock (CCLK) along with the fact that some LPC2468 have errata which decreases their achievable CPU clock frequency.

Using on-chip memory dedicated to USB helps reduce or remove these problems, depending on CPU frequency. Clearly this option must be disabled if the on-chip USB peripheral is to be used. It is also desirable to disable this option if the CPU frequency is high enough, in order to remove an extra copy on every data transfer, thus improving performance. The USB memory used is 512 bytes at the start of the USB memory space (0x7FD00000).

If this option is disabled and the DMA is not able to proceed quickly enough, this will be visible in the form of I/O errors. In that case, if it is not possible to enable this option it is recommended to adjust the CYGDAT_HAL_ARM_LPC2XXX_EA_LPC2468_MCI_BUS_SPEED_LIMIT configuration option.

Lock AHB bus during DMA transfer (CYGSEM_HAL_ARM_LPC2XXX_EA_LPC2468_MCI_DMA_LOCKS_AHB)

The AMBA Hardware Bus (AHB) is used to connect AMBA peripherals within the LPC2468, including the ARM core, DMA controller and memory controllers. When this option is enabled, the AHB is locked for the duration of MCI DMA transfer bursts. If another AMBA host needs to make a transfer it may be delayed as a result, which may not be desirable.

Disabling this option allows the AHB arbiter to permit other AHB hosts to perform transfers. Of course this may mean the MCI DMA transfers can in turn themselves get delayed, risking data overruns or underruns in MCI transfers, resulting in I/O errors during block reads or writes. This is particularly likely on processors running at slower clock speeds where there may already be difficulties with the DMA servicing data transfers quickly enough.

MMC/SD bus frequency limit (CYGNUM_HAL_ARM_LPC2XXX_EA_LPC2468_MCI_BUS_SPEED_LIMIT)

> The LPC2468 cannot always keep up with the data transfer requirements, especially at slower CPU clock speeds. This is because the DMA controller runs at the speed of the CPU clock (CCLK) along with the fact that some LPC2468 have errata which decreases their achievable CPU clock frequency. The adjacent options to use on-chip USB memory and to lock the AHB bus can help prevent this, but sometimes they are insufficient to prevent data overruns or underruns resulting in I/O errors during block reads or writes. In which case the only remaining recourse is to reduce the required data transfer rate between the MCI and the card.
>
> This option can be used to impose an upper limit on the MMC/SD bus frequency. The value used in this option is measured in Hertz, and the use of 4-bit mode with SD cards is not a factor - this option provides the bus frequency, so a 4-bit bus will transfer four times the amount of data as a 1-bit bus in the same time period.
>
> Note that this option provides a limit, and does not mean the card bus will operate at that frequency. The frequency is also governed by what the card will support, and the resolution of the clock used to derive the MMC/SD clock signal, and how it can be divided down.

Compiler Flags

The platform HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos.

However there are two flags that are used if Thumb mode is to be supported:

-mthumb	The arm-eabi-gcc compiler will compile C and C++ files into the Thumb instruction set when this option is used.
-mthumb-interwork	This option allows programs to be created that mix ARM and Thumb instruction sets. Without this option, some memory can be saved. This option should be used if -mthumb is used.

Onboard NAND

The HAL port includes a low-level driver to access the on-board Samsung K9F1G08U08 NAND flash memory chip. To enable the driver, activate the CDL option CYGHWR_HAL_ARM_LPC2XXX_EA_LPC2468_NAND and ensure that the CYGPKG_DEVS_NAND_SAMSUNG_K9 package is present in your eCos configuration. The driver is capable of operating with or without the NAND_RDY line connected.

CYGHWR_HAL_ARM_LPC2XXX_EA_LPC2468_USE_NAND_RDY

> The EA OEM Base Board provides a jumper which connects the ready line of the NAND chip (NAND_RDY) to pin P2.12 on the CPU. Setting this option indicates to the driver that that jumper, or similar layout with the same effect, is in place. This provides an improvement in efficiency, but must not be set if the jumper is not so connected.

CYGHWR_HAL_ARM_LPC2XXX_EA_LPC2468_NAND_RDY_USE_INTERRUPT

> *(Only active if CYGHWR_HAL_ARM_LPC2XXX_EA_LPC2468_USE_NAND_RDY is set.)* If set, pin P2.12 (see above) is set up as an interrupt (EINT2). Setting this causes the thread invoking the driver to sleep when waiting for a program or erase operation to complete, as opposed to entering a polling loop. This potentially represents an efficiency gain if you have at least one other thread which can carry on performing useful work while the NAND chip works.
>
> If this option is not set, the driver polls the ready line.
>
> When this option is set, the driver automatically detects whether the eCos kernel scheduler is running; if it is not, interrupt mode cannot operate, and the driver falls back to polling the ready line.
>
> Interrupt mode imposes its own overheads on the driver thread. Benchmarking chip program and erase operations alone will necessarily appear to show a slow-down in interrupt mode when the scheduler is running. This option can only improve

efficiency on a holistic basis, and only then in the case where there are other threads which can continue to work while the driver is waiting for the NAND operation to complete.

Partitioning the NAND chip

The NAND chip must be partitioned before it can become available to applications.

A CDL script which allows the chip to be manually partitioned is provided (see `CYGSEM_DEVS_NAND_EA_LPC2468_PARTITION_MANUAL_CONFIG`); if you choose to use this, the relevant data structures will automatically be set up for you when the device is initialised. By default, the manual config CDL script sets up a single partition (number 0) encompassing the entire device.

It is possible to configure the partitions in some other way, should it be appropriate for your setup. To do so you will have to add appropriate code to `ea_lpc2468_nand.c`.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the LPC2468 OEM Board hardware, and should be read in conjunction with that specification. The LPC2468 platform HAL package complements the ARM architectural HAL and the LPC2xxx variant HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset the HAL will initialize or reinitialize many of the on-chip peripherals. There is an exception for RAM startup applications which depend on a ROM monitor to do most of this.

For ROM startup, the HAL will perform additional initialization, programming the various internal registers including PLL (for the clocks); Memory Mapping control registers to map SRAM to 0x0; the memory controller for access to external FLASH and SDRAM; and the Memory Acceleration Module (MAM). The details of the early hardware startup may be found in the header `cyg/hal/hal_platform_setup.h`.

Linker Scripts and Memory Maps

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

on-chip Flash
: This is located at address 0x0 of the memory space, although after hardware initialization, the start of internal SRAM is mapped over locations 0x0 to 0x40. This region ends at 0x80000. The MAM is enabled to accelerate memory reads from this area. A driver is available for using this flash via the eCos flash API.

external Flash
: This is located at address 0x80000000 of the memory space. It is not used by default by eCos, although if RedBoot is asked to manage the Flash, it reserves flash addresses 0x803F0000 thru 0x803FEFFF. If RedBoot stores its configuration data in Flash, then addresses 0x803FF000 thru 0x803FFFFF are reserved by RedBoot.

internal SRAM
: This is located at address 0x40000000 of the memory space, ending at location 0x4000FFFF. The first 64 bytes are mapped to location 0x0000000.

external SDRAM
: This is located at address 0xa0000000 of the memory space, ending at location 0xa2000000. For RAM startup, available SRAM starts at location 0xa1100000, with the bottom 1Mbyte reserved for use by RedBoot.

on-chip peripherals
: These are accessible via location 0xE0000000 onwards. Descriptions of the contents can be found in the LPC2468 User Manual.

Chapter 244. Embedded Artists QuickStart Board Support

Name

eCos Support for the Embedded Artists QuickStart Boards — Overview

Introduction

This platform HAL is designed to support the QuickStart board series from Embedded Artists, fitted with an NXP LPC2xxx microcontroller, and optionally connected to an Embedded Artists QuickStart Prototype Board.

The support for the QuickStart board series provided by this HAL has been initially developed for the Embedded Artists (henceforth 'EA') LPC2148 USB QuickStart Board. This HAL documentation therefore presently corresponds to that particular board instance, and future supported variants will cause this documentation to be updated accordingly.

Description

The Embedded Artists QuickStart Board is fitted with an NXP LPC2xxx processor rated at up to 60MHz, which contains up to 64KB of SRAM and up to 512KB of FLASH, depending on choice of LPC2xxx variant. The board itself has a single 9-pin RS-232 serial interface connected to the LPC2xxx on-chip UART 0, an I²C EEPROM, and a USB device interface. Refer to EA's QuickStart board documentation for full details.

Further peripheral support is available if the board is mounted on to the EA QuickStart Prototype Board, including push buttons, LEDs, JTAG, MMC/SD card socket, buzzer, 7-segment LED, and a further 9-pin RS-232 serial interface connected to the LPC2xxx on-chip UART 1.

The typical mode of operation for eCos development usually depends on the amount of memory available. On LPC2xxx variants with 64Kbytes or more of SRAM, a GDB stub ROM image is programmed into the LPC2xxx on-chip flash memory, and the board will boot this image from reset. While RedBoot may also be used, its larger RAM footprint requirements usually make it unsuitable. Both RedBoot and the GDB stub ROM provide GDB stub functionality so it is then possible to download and debug stand-alone and eCos applications via the gdb debugger using UART 0.

On LPC2xxx variants with less than 64KBytes SRAM, such as the 32KBytes on the LPC2148, it is typically expected that stand-alone applications will be programmed directly to on-chip Flash, either using a hardware JTAG/ICE unit via the QuickStart Prototype Board, or by serial using the on-chip In-System Programming (ISP) mechanism included with NXP LPC2xxx microcontrollers and a suitable host application running on a PC.

This documentation describes platform-specific elements of the EA QuickStart Board support within eCos. Documentation on the NXP LPC2xxx variants is available separately, and should be read in conjunction with this documentation. The LPC2xxx documentation covers various topics including HAL support common to LPC2xxx variants, and on-chip device support. This document complements the LPC2xxx documentation.

Supported Hardware

The NXP LPC2xxx microcontrollers on the EA QuickStart Boards have up to 512Kbytes of on-chip Flash memory. In a typical setup, the GDB stub ROM or the user application will load and run from this internal flash. For initial programming of the internal Flash, external support is required, such as the NXP LPC2000 Flash Utility, the Flash Magic tool, or a hardware JTAG/ICE unit. The latter may be used with its own in-built LPC2xxx flash programming support if it exists, or the eCosPro® **ecoflash** utility. 28Kbytes of internal flash memory should be reserved for the GDB Stub ROM, the remainder being free for the application's use. Note that the LPC2xxx primary boot loader and IAP code reside in boot blocks located at the end of on-chip Flash. To determine the number and size of blocks reserved for their use, consult the specific LPC2xxx variant's datasheet.

The first 64 bytes of on-chip SRAM are mapped by the HAL startup code using the LPC2xxx memory mapping control to location 0x00000000. When loading applications using the GDB stub ROM, SRAM from location 0x40000040 to 0x40001000 is reserved for its use. The rest of SRAM is generally available for use by the application. Programs booted from ROM, or loaded directly into SRAM via JTAG may use all SRAM. In all cases, if using the eCos LPC2xxx Flash driver, the last 32 bytes (or more if a separate program buffer is used) become reserved due to the requirements of the IAP code.

The NXP LPC2xxx variant HAL includes support for the two LPC2xxx on-chip serial devices and is documented in the variant HAL. Although the interrupt-driven serial driver supports the line status and modem control lines on UART 1 (UART 0 not having such support), the QuickStart boards do not connect these pins, and so that functionality is unavailable.

The EA QuickStart port includes support for the on-chip watchdog, RTC (wallclock), interrupt controller (VIC) and on-chip Flash. This support is documented in the LPC2xxx variant HAL.

Tools

The QuickStart Board port is intended to work with GNU tools configured for an arm-eabi target. Thumb mode is supported. The original port was created using arm-elf-gcc version 3.4.4, arm-elf-gdb version 6.3, and binutils version 2.16.

Name

Setup — Preparing the EA QuickStart Board for eCos Development

Overview

In a typical development environment, the EA QuickStart Board boots from internal flash into either the GDB stub ROM monitor or directly into the user application. In the case of microcontrollers with less than 64Kbytes of SRAM, the latter is recommended. eCos applications to be loaded and run from the GDB stub ROM monitor may be configured for RAM startup and then downloaded and run on the board via the debugger **arm-eabi-gdb**. Preparing the board therefore usually involves programming a suitable ROM image into Flash memory, either the GDB stub ROM or application images.

For serial communications, all versions run with 8 bits, no parity, and 1 stop bit at 38400 baud. This baud rate can be changed via the configuration option `CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL_BAUD` and rebuilding the application, or if applicable, GDB stub ROM. A "straight through" 9-pin RS232 serial cable, with Male<->Female connectors is required. Using a "null modem" serial cable will not work.

Initial Installation

Board setup

Jumper settings must be checked and potentially changed on the board to ensure correct operation. This section describes jumper settings that are known to require attention. In general, any board-specific documentation from Embedded Artists takes precedence over the documentation here, as this may reflect hardware which has been modified since the time of writing of this documentation. Most QuickStart boards are very similar to each other, the only change of note being of course the choice of LPC2xxx microcontroller fitted. But if your board does not fit the description here (which has initially been based on the LPC2148 USB QuickStart) then you should consult the board documentation.

Firstly, there are two jumpers located close to the serial connector. In general, these jumpers should only be closed (i.e. jumper fitted and connecting the two pins) when wishing to reprogram the on-chip Flash via ISP. Otherwise they should remain open (jumper not connecting the two pins) so that any unplugging of the serial connector, movement of the serial connector, or use of flow control signals from the host PC, do not cause a spurious reset or interrupt (on the EINT1 line) of the board.

If the board is being powered directly by USB, then a jumper next to the USB connector should be closed. Otherwise it *must* be open. This also means the jumper must be open if the QuickStart board is mounted on the Prototype Board, and power is being provided by either the DC power connector or USB connector on the Prototype Board.

Note that if the Prototype Board is fitted, but the QuickStart board is being powered by its USB connector as opposed to the Prototype board's USB connector (thus meaning that the above jumper would be closed), then pin P0.23 is used as a USB power indication on LPC214x boards. This prevents its use as an SPI chip select line for the 7-segment LED on the Prototype Board.

If mounting the QuickStart LPC2xxx board on the QuickStart Prototype Board, then consult the EA documentation for the correct jumper settings and socket location appropriate to the fitted LPC2xxx model. This includes settings for the JTAG connector. In the case of the LPC213x/LPC214x, the jumper labelled "JTAG" must be closed, and the jumper labelled "DBGSEL" must be open.

It may also be useful to be aware that although eCos configures the PWM pin for the buzzer, it does not directly support it, and so it is probably useful to open the jumper labelled "P0.7" to disable the buzzer.

The jumpers adjacent to the LEDs may remain in their default state of closed, in order to get insight into system operation, and to allow use of the user-configurable LEDs, as described later.

Flash Installation

This process assumes that a Microsoft Windows machine with the Flash Magic utility installed is available. Flash Magic is a tool for programming flash based microcontrollers from NXP using a protocol via the RS232 serial port to communicate with the In-System Programming (ISP) firmware on the LPC2xxx. The Flash Magic utility is sponsored by NXP and available from this website.

The first step is to connect the RS232 cable between the serial port of the QuickStart board and the host PC. Do not use the serial port on the Prototype board. Now close the two jumpers adjacent to the serial port on the QuickStart board. These allow the software on the PC to reset the LPC2xxx and enter the ISP firmware. Finally apply the power.

Start the Flash Magic utility on the host PC, and a window will be displayed allowing various parameters to be configured in a series of steps. For step 1, firstly choose the appropriate COM port that is being used on your PC and set the Baud Rate to 38400 baud. Next select the appropriate LPC2xxx device in use such as LPC2148. The "Interface" should be set to "None (ISP)". And finally for step 1 choose the appropriate Oscillator Frequency for the QuickStart board in use. This may be found in the board documentation, and is usually visibly readable on the surface of the oscillator on the board (in a metal package). For example for the LPC2148 USB QuickStart, the oscillator reads 12.000 indicating 12MHz.

For step 2, it is usually adequate to leave the option "Erase blocks used by hex file" checked, and ignore the other settings. For step 3, you must select the program image to be downloaded, in Intel HEX format. To program the pre-built GDB stub ROM image, locate the file `gdb_module.hex` in the `loaders` subdirectory of your release. To generate an Intel HEX format version of an application you have built yourself run the following command at a shell prompt:

```
 $ arm-eabi-objcopy -O ihex app.elf app.hex
```

This converts the application image in ELF format (as output by the linker), to Intel HEX format in the file `app.hex`. Note that the arm-eabi tools must be on your path at this point. If they are not, run the command below before you run the above **arm-eabi-objcopy** command:

```
 $ . /opt/ecos/ecosenv.sh
```

In step 4, it is recommended to set the option "Verify after programming". Finally it is possible to click on "Start" to program the image into the on-chip Flash.

Tip

If there is a problem communicating with the board, such as a report of a failure to autobaud, then this may imply that the Flash Magic tool was not able to control the serial lines properly. This can happen with some USB-Serial converters. For Prototype Board users, to workaround this issue, power the board off and remove (i.e. open) the two jumpers next to the serial port. Then simultaneously press the buttons marked 'Reset' and 'P0.14' on the prototype board (the latter corresponds to interrupt EINT1), then release the Reset button, and finally release the 'P0.14' button. This is an alternative mechanism of forcing the ISP firmware to be entered. Once this has been successfully performed, the programming operation may be retried from the Flash Magic utility.

When the process completes, remove (i.e. open) the two jumpers next to the serial port. If a GDB stub ROM image has been programmed, verify the programming has been successful by starting a terminal emulation application such as HyperTerminal on the host PC and set the serial communication parameters to 38400 baud, 8 data bits, no parity, 1 stop bit (8N1) and no flow control (handshaking). Reset the board and the stubrom should start. For boards programmed with GDB stubs the output should be similar to the following:

```
+$T050f:ec070000;0d:28080040;#52
```

This is the stubrom attempting to communicate with GDB and indicates that it is functioning correctly.

Rebuilding the GDB Stub ROM

Should it prove necessary to rebuild the GDB Stub ROM binary, this is done most conveniently at the command line. Your `PATH` and `ECOS_REPOSITORY` environment variables must first be set correctly, The following steps given an example of how to rebuild the stubs for a QuickStart board with LPC2148:

```
$ mkdir stub_rom
$ cd stub_rom
$ ecosconfig new ea_quickstart_lpc2148 stubs
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

At the end of the build the `install/bin` subdirectory should contain the files `gdb_module.img` (ELF format), `gdb_module.srec` (Motorola S-Record format), `gdb_module.bin` (raw binary format), and `gdb_module.hex` (Intel HEX format).

Name

Configuration — Platform-specific Configuration Options

Overview

The EA QuickStart platform HAL package is loaded automatically when eCos is configured for a target which uses it. The target names include the LPC2xxx model in use. At this time the only target supported is the `ea_quickstart_lpc2148` target. It should never be necessary to load this platform HAL package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The EA QuickStart platform HAL package supports three separate startup types:

ROM This startup type can be used for finished applications which will be programmed into internal flash at location 0x0. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization. On targets with less than 64Kbytes of SRAM, this is the startup type normally used.

RAM This is the startup type which is normally used during application development on targets with 64Kbytes of SRAM or greater. The board has the GDB stubrom or RedBoot programmed into flash at location 0x0 in internal on-chip Flash and boots from that location. arm-eabi-gdb is then used to load a RAM startup application into memory and debug it. It is assumed that the hardware has already been initialized by the ROM monitor. By default the application will use the eCos virtual vectors mechanism to obtain certain services from the ROM monitor, including diagnostic output. Larger applications may not fit into the available SRAM, in which case ROM startup may be required.

JTAG This is an alternative development startup type. The application is loaded into RAM via a JTAG device and is run and debugged from there. The application will be self-contained with no dependencies on services provided by other software. It is expected that hardware setup will have been performed via the JTAG device prior to loading. Some sample configuration and initialisation scripts for a number of JTAG debugging solutions may be found in the `misc` subdirectory of the platform HAL package within the component repository.

The ROM Monitor and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building the GDB Stubrom or RedBoot.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, or as a testing step before switching to ROM startup. Virtual vector support is usually only required for RAM startup applications or ROM monitors. The CDL option `CYGFUN_HAL_LPC2XXX_VIRTUAL_VECTOR_SUPPORT` within the LPC2xxx variant HAL allows manual control of this facility.

If the application does not rely on a ROM monitor for diagnostic services then by default serial port UART0 will be claimed for HAL diagnostics. If using the Prototype Board, it becomes possible to use UART1 as well, and in order to allow its selection as a potential debug or diagnostic channel, the number of communications channels can be increased from 1 to 2 with the option `CYGNUM_HAL_VIRTUAL_VECTOR_COMM_CHANNELS`. To specify which serial port is used for diagnostics, the configuration option `CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL` may be set accordingly, and its baud rate configured with `CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL_BAUD`.

System Clock

By default, the system clock interrupts once every 10ms, corresponding to a 100Hz clock. This can be changed by the configuration option `CYGNUM_HAL_RTC_DENOMINATOR` which corresponds to the clock frequency. Other clock-related settings are recalculated automatically if the denominator is changed. The PLL multiplier may be configured to allow a core clock (CCLK) speed of up to 60MHz. The description of the clock-related CDL options may be found in the LPC2xxx variant HAL documentation. Note there are frequency constraints on the Current Controlled Oscillator (CCO) within the LPC2xxx, and the datasheet should be consulted to ensure the required specifications are not exceeded.

Compiler Flags

The platform HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos.

However there are two flags that are used if Thumb mode is to be supported:

`-mthumb`
: The arm-eabi-gcc compiler will compile C and C++ files into the Thumb instruction set when this option is used. To build eCos in Thumb mode, enable the `CYGHWR_THUMB` configuration option in the ARM architecture HAL.

`-mthumb-interwork`
: This option allows programs to be created that mix ARM and Thumb instruction sets. For example, allowing a Thumb application to be used with eCos built in normal ARM mode. Without this option, some memory can be saved. This option should be used if -mthumb is used. eCos may be built with Thumb interworking support by enabling the `CYGBLD_ARM_ENABLE_THUMB_INTERWORK` CDL option in the ARM architecture HAL. Use of the LPC2xxx Flash driver requires Thumb interworking support to be enabled as the calls into the IAP firmware are must be made allowing a switch to Thumb mode.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the EA QuickStart board hardware, and should be read in conjunction with that specification. The QuickStart Board platform HAL package complements the ARM architectural HAL and the LPC2xxx variant HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset the HAL will initialize or reinitialize many of the on-chip peripherals. This includes the PINSEL functions and LED bank. There is an exception for RAM startup applications which depend on a ROM monitor for certain services.

For ROM startup, the HAL will perform additional initialization, programming the various internal registers including PLL (for the clocks), Memory Mapping control registers to map SRAM to 0x0, and Memory Acceleration Module (MAM). The details of the early hardware startup may be found in the header `cyg/hal/hal_platform_setup.h`.

Linker Scripts and Memory Maps

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

on-chip Flash
: This is located at address 0x0 of the memory space, although after hardware initialization, the start of internal SRAM is mapped over locations 0x0 to 0x40. The size of this region depends on the LPC2xxx microcontroller variant in use. In the case of the LPC2148, the region is of size 512Kbytes, ending at 0x80000. However the last few blocks of Flash are reserved for use as bootblocks for the ISP/IAP firmware, resulting in a usable Flash size of 500Kbytes, ending at 0x7d000. The MAM is enabled to accelerate memory reads from this area.

internal SRAM
: This is located at address 0x40000000 of the memory space, and is 16, 32 or 64k in size, depending on the chip fitted. The first 64 bytes are mapped to location 0x0000000. If using GDB stubs ROM, or another ROM monitor, the virtual vector table starts at 0x40000050 and extends to 0x40000150. The remainder of SRAM is available for use by applications. For RAM startup applications, SRAM below 0x40001000 is reserved for the GDB stubrom and the remainder is available for the application. An exception is if the on-chip Flash driver is to be used. In that case, the top 32 bytes of SRAM are used by it. This is automatically handled in the port's memory layout files if the flash driver is present in the configuration.

on-chip peripherals
: These are accessible at location 0xE0000000 onwards. Descriptions of the contents can be found in the LPC2xxx User Manual for the appropriate microcontroller variant.

Real-time characterization

The tm_basic kernel test gives statistics gathered about the real-time characterization and performance of the kernel. The sample output is shown here for information. The test was built in ARM mode, which provided better performance than Thumb mode.

Example 244.1. ea_quickstart Real-time characterization

```
            Startup, main stack : stack used   420 size  3920
             Startup :  Interrupt stack used   148 size  4096
             Startup : Idlethread stack used    88 size  2048

eCos Kernel Timings
Notes: all times are in microseconds (.000001) unless otherwise stated
```

```
Reading the hardware clock takes 0 'ticks' overhead
... this value will be factored out of all other measurements
Clock interrupt took   15.31 microseconds (9 raw clock ticks)

Testing parameters:
   Clock samples:            32
   Threads:                   1
   Thread switches:         128
   Mutexes:                  32
   Mailboxes:                21
   Semaphores:               32
   Scheduler operations:    128
   Counters:                 32
   Flags:                    32
   Alarms:                   32


                                 Confidence
     Ave     Min     Max     Var  Ave  Min  Function
  ======  ======  ======  ====== ========== ========
   15.00   15.00   15.00    0.00  100% 100% Create thread
    5.00    5.00    5.00    0.00  100% 100% Yield thread [all suspended]
    3.33    3.33    3.33    0.00  100% 100% Suspend [suspended] thread
    5.00    5.00    5.00    0.00  100% 100% Resume thread
    6.67    6.67    6.67    0.00  100% 100% Set priority
    1.67    1.67    1.67    0.00  100% 100% Get priority
   11.67   11.67   11.67    0.00  100% 100% Kill [suspended] thread
    3.33    3.33    3.33    0.00  100% 100% Yield [no other] thread
    6.67    6.67    6.67    0.00  100% 100% Resume [suspended low prio] thread
    5.00    5.00    5.00    0.00  100% 100% Resume [runnable low prio] thread
    6.67    6.67    6.67    0.00  100% 100% Suspend [runnable] thread
    5.00    5.00    5.00    0.00  100% 100% Yield [only low prio] thread
    5.00    5.00    5.00    0.00  100% 100% Suspend [runnable->not runnable]
   11.67   11.67   11.67    0.00  100% 100% Kill [runnable] thread
    8.33    8.33    8.33    0.00  100% 100% Destroy [dead] thread
   16.67   16.67   16.67    0.00  100% 100% Destroy [runnable] thread
   25.00   25.00   25.00    0.00  100% 100% Resume [high priority] thread

    1.41    0.00    1.67    0.44   84%  15% Scheduler lock
    3.49    3.33    5.00    0.28   90%  90% Scheduler unlock [0 threads]
    3.49    3.33    5.00    0.28   90%  90% Scheduler unlock [1 suspended]
    3.41    3.33    5.00    0.15   95%  95% Scheduler unlock [many suspended]
    3.52    3.33    5.00    0.32   89%  89% Scheduler unlock [many low prio]

    1.82    1.67    3.33    0.28   90%  90% Init mutex
    4.38    3.33    5.00    0.78   62%  37% Lock [unlocked] mutex
    5.00    5.00    5.00    0.00  100% 100% Unlock [locked] mutex
    4.11    3.33    5.00    0.83   53%  53% Trylock [unlocked] mutex
    3.80    3.33    5.00    0.67   71%  71% Trylock [locked] mutex
    1.35    0.00    1.67    0.51   81%  18% Destroy mutex
   21.67   21.67   21.67    0.00  100% 100% Unlock/Lock mutex

    1.98    1.67    3.33    0.51   80%  80% Create mbox
    1.19    0.00    1.67    0.68   71%  28% Peek [empty] mbox
    4.60    3.33    5.00    0.61   76%  23% Put [first] mbox
    1.43    0.00    1.67    0.41   85%  14% Peek [1 msg] mbox
    4.52    3.33    5.00    0.68   71%  28% Put [second] mbox
    0.87    0.00    1.67    0.83   52%  47% Peek [2 msgs] mbox
    4.76    3.33    5.00    0.41   85%  14% Get [first] mbox
    4.76    3.33    5.00    0.41   85%  14% Get [second] mbox
    4.05    3.33    5.00    0.82   57%  57% Tryput [first] mbox
    3.89    3.33    5.00    0.74   66%  66% Peek item [non-empty] mbox
    4.29    3.33    5.00    0.82   57%  42% Tryget [non-empty] mbox
    3.81    3.33    5.00    0.68   71%  71% Peek item [empty] mbox
    4.05    3.33    5.00    0.82   57%  57% Tryget [empty] mbox
    1.35    0.00    1.67    0.51   80%  19% Waiting to get mbox
    1.43    0.00    1.67    0.41   85%  14% Waiting to put mbox
    2.30    1.67    3.33    0.79   61%  61% Delete mbox
   15.00   15.00   15.00    0.00  100% 100% Put/Get mbox

    1.72    1.67    3.33    0.10   96%  96% Init semaphore
    3.91    3.33    5.00    0.75   65%  65% Post [0] semaphore
    3.96    3.33    5.00    0.78   62%  62% Wait [1] semaphore
    3.91    3.33    5.00    0.75   65%  65% Trywait [0] semaphore
    3.75    3.33    5.00    0.63   75%  75% Trywait [1] semaphore
    1.72    1.67    3.33    0.10   96%  96% Peek semaphore
    1.46    0.00    1.67    0.36   87%  12% Destroy semaphore
   13.91   13.33   15.00    0.75   65%  65% Post/Wait semaphore

    1.93    1.67    3.33    0.44   84%  84% Create counter
    1.46    0.00    1.67    0.36   87%  12% Get counter value
    1.35    0.00    1.67    0.51   81%  18% Set counter value
    4.32    3.33    5.00    0.80   59%  40% Tick counter
    1.46    0.00    1.67    0.36   87%  12% Delete counter

    1.72    1.67    3.33    0.10   96%  96% Init flag
    4.06    3.33    5.00    0.82   56%  56% Destroy flag
    3.59    3.33    5.00    0.44   84%  84% Mask bits in flag
    4.06    3.33    5.00    0.82   56%  56% Set bits in flag [no waiters]
    5.63    5.00    6.67    0.78   62%  62% Wait for flag [AND]
    5.63    5.00    6.67    0.78   62%  62% Wait for flag [OR]
    5.57    5.00    6.67    0.75   65%  65% Wait for flag [AND/CLR]
    5.52    5.00    6.67    0.72   68%  68% Wait for flag [OR/CLR]
    1.35    0.00    1.67    0.51   81%  18% Peek on flag

    2.86    1.67    3.33    0.67   71%  28% Create alarm
    6.61    5.00    6.67    0.10   96%   3% Initialize alarm
    3.91    3.33    5.00    0.75   65%  65% Disable alarm
    6.15    5.00    6.67    0.72   68%  31% Enable alarm
    4.38    3.33    5.00    0.78   62%  37% Delete alarm
    4.95    3.33    5.00    0.10   96%   3% Tick counter [1 alarm]
   23.33   23.33   23.33    0.00  100% 100% Tick counter [many alarms]
    8.28    6.67    8.33    0.10   96%   3% Tick & fire counter [1 alarm]
  138.96  138.33  140.00    0.78   62%  62% Tick & fire counters [>1 together]
   26.88   26.67   28.33    0.36   87%  87% Tick & fire counters [>1 separately]
   15.00   15.00   15.00    0.00  100% 100% Alarm latency [0 threads]
   15.00   15.00   15.00    0.00  100% 100% Alarm latency [many threads]
   26.69   26.67   30.00    0.05   99%  99% Alarm -> thread resume latency

    3.38    3.33    5.00    0.00            Clock/interrupt latency

    8.33    8.33    8.33    0.00            Clock DSR latency

  312    312     312  (main stack:   716)  Thread stack used (1360 total)
           All done, main stack : stack used   716 size  3920
            All done :  Interrupt stack used   204 size  4096
            All done : Idlethread stack used   256 size  2048

Timing complete - 23650 ms total

PASS:<Basic timing OK>
EXIT:<done>
```

LED use

LEDs are available on the Prototype board. Most of these are attached to lines associated with peripherals present on the Prototype board, or the QuickStart board itself. However 9 LEDS are available for application use from C. The following C function may be used:

```
#include <cyg/infra/hal_diag.h>
extern void hal_diag_led(int leds);
```

Values from 0 to 511 will be displayed on the LED bank representing the binary value with 1 being on and 0 being off. The LEDs used are connected to P0.10-P0.13 and P0.17-P0.21, and with P0.21 being the MSB, and P0.10 the LSB.

The LEDs are also used during platform initialization and only P0.10 should be illuminated if booting has been successful. Other LED indications represent the stage in the initialization process that failed.

Other Issues

The following pin assignments are configured by default for LPC2148 at board initialisation time:

PINSEL0:
P0.0/P0.1 for UART0
P0.2/P0.3 for I²C
P0.4/P0.5/P0.6 for SPI
P0.7 as PWM2 for buzzer on Prototype board
P0.8/P0.9 for UART1 (which is available on prototype board)
P0.10-P0.13 as GPIO-controlled LEDs
P0.14 EINT1 (available as button on prototype board)
P0.15 EINT2 (available as button on prototype board)

PINSEL1:
P0.16 EINT0 (available as button on prototype board)
P0.17-P0.21 as GPIO-controlled LEDs
P0.22 as GPIO output for SPI_SEL_MMC on prototype board
P0.23 as GPIO output for SPI_SEL_LED on prototype board
P0.29 as GPIO input for MMC/SD card detect on prototype board
P0.30 as EINT3 (available as button on prototype board)
(rather than GPIO input for SD write protect on prototype board)
All other pins set as GPIO inputs

PINSEL2:
P1.26-P1.31 for JTAG
All other pins set as inputs

Chapter 245. IAR KickStart Card Support

Name

eCos Support for the IAR KickStart Cards — Overview

Introduction

This platform HAL is designed to support the KickStart Card series from IAR, fitted with an NXP LPC2xxx microcontroller.

The support for the KickStart board series provided by this HAL has been initially developed for the IAR LPC2106 KickStart Card. This HAL documentation therefore presently corresponds to that particular board instance, and future supported variants will cause this documentation to be updated accordingly.

Description

The IAR KickStart Board is fitted with an NXP LPC2xxx processor rated at up to 60MHz, which contains up to 64KB of SRAM and up to 512KB of FLASH, depending on choice of LPC2xxx variant. The board itself has two 9-pin RS-232 serial interfaces connected to the LPC2xxx on-chip UART 0 and UART 1, push buttons connected to interrupt lines, LEDs, a JTAG debug port, anda prototyping area. Refer to IAR's KickStart board documentation for full details.

The typical mode of operation for eCos development usually depends on the amount of memory available. On LPC2xxx variants with 64Kbytes or more of SRAM, a GDB stub ROM image is programmed into the LPC2xxx on-chip flash memory, and the board will boot this image from reset. While RedBoot may also be used, its larger RAM footprint requirements usually make it unsuitable. Both RedBoot and the GDB stub ROM provide GDB stub functionality so it is then possible to download and debug stand-alone and eCos applications via the gdb debugger using UART 0.

On LPC2xxx variants with less than 64KBytes SRAM, it is typically expected that standalone applications will be programmed directly to on-chip Flash, either using a hardware JTAG/ICE unit, or by serial using the on-chip In-System Programming (ISP) mechanism included with NXP LPC2xxx microcontrollers and a suitable host application running on a PC.

This documentation describes platform-specific elements of the IAR KickStart Board support within eCos. Documentation on the NXP LPC2xxx variants is available separately, and should be read in conjunction with this documentation. The LPC2xxx documentation covers various topics including HAL support common to LPC2xxx variants, and on-chip device support. This document complements the LPC2xxx documentation.

Supported Hardware

The NXP LPC2xxx microcontrollers on the IAR KickStart Boards have up to 512Kbytes of on-chip Flash memory. In a typical setup, the GDB stub ROM or the user application will load and run from this internal flash. For initial programming of the internal Flash, external support is required, such as the NXP LPC2000 Flash Utility, the Flash Magic tool, or a hardware JTAG/ICE unit. The latter may be used with its own in-built LPC2xxx flash programming support if it exists, or the eCosPro® **ecoflash** utility. 28Kbytes of internal flash memory should be reserved for the GDB Stub ROM, the remainder being free for the application's use. Note that the LPC2xxx primary boot loader and IAP code reside in boot blocks located at the end of on-chip Flash. To determine the number and size of blocks reserved for their use, consult the specific LPC2xxx variant's datasheet.

The first 64 bytes of on-chip SRAM are mapped by the HAL startup code using the LPC2xxx memory mapping control to location 0x00000000. When loading applications using the GDB stub ROM, SRAM from location 0x40000040 to 0x40001000 is reserved for its use. The rest of SRAM is generally available for use by the application. Programs booted from ROM, or loaded directly into SRAM via JTAG may use all SRAM. In all cases, if using the eCos LPC2xxx Flash driver, the last 32 bytes (or more if a separate program buffer is used) become reserved due to the requirements of the IAP code.

The NXP LPC2xxx variant HAL includes support for the two LPC2xxx on-chip serial devices and is documented in the variant HAL. Although the interrupt-driven serial driver supports the line status and modem control lines on UART 1 (UART 0 not having such support), the KickStart boards do not connect these pins, and so that functionality is unavailable.

The IAR KickStart port includes support for the on-chip watchdog, RTC (wallclock), interrupt controller (VIC) and on-chip Flash. This support is documented in the LPC2xxx variant HAL.

Tools

The KickStart Board port is intended to work with GNU tools configured for an arm-eabi target. Thumb mode is supported. The original port was created using arm-elf-gcc version 3.4.4, arm-elf-gdb version 6.3, and binutils version 2.16.

Name

Setup — Preparing the IAR KickStart Board for eCos Development

Overview

In a typical development environment, the IAR KickStart Board boots from internal flash into either the GDB stub ROM monitor or directly into the user application. In the case of microcontrollers with less than 64Kbytes of SRAM, the latter is recommended. eCos applications to be loaded and run from the GDB stub ROM monitor may be configured for RAM startup and then downloaded and run on the board via the debugger **arm-eabi-gdb**. Preparing the board therefore usually involves programming a suitable ROM image into Flash memory, either the GDB stub ROM or application images.

For serial communications, all versions run with 8 bits, no parity, and 1 stop bit at 38400 baud. This baud rate can be changed via the configuration option `CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL_BAUD` and rebuilding the application, or if applicable, GDB stub ROM. A "straight through" 9-pin RS232 serial cable, with Male<->Female connectors is required. Using a "null modem" serial cable will not work.

Initial Installation

Board setup

Jumper settings must be checked and potentially changed on the board to ensure correct operation. This section describes jumper settings that are known to require attention. In general, any board-specific documentation from IAR takes precedence over the documentation here, as this may reflect hardware which has been modified since the time of writing of this documentation. Many KickStart boards are similar to each other, the only change of note being of course the choice of LPC2xxx microcontroller fitted. But if your board does not fit the description here (which has initially been based on the LPC2106 KickStart) then you should consult the board documentation.

Firstly, there are two jumpers located close to the serial connectors, labelled JP3 and JP4 on the LPC2106 KickStart. These can both be closed (i.e. jumper fitted and connecting the two pins) in order to permit use of both serial ports.

There are two jumpers labelled EN_SW_ISP and EN_SW_RST close to the push buttons, also labelled JP7 and JP8 respectively on the LPC2106 KickStart. In general, these jumpers should only be closed when wishing to reprogram the on-chip Flash via ISP. Otherwise they should remain open (jumper not connecting the two pins) so that any unplugging of the serial connector, movement of the serial connector, or use of flow control signals from the host PC, do not cause a spurious reset or interrupt (on the EINT1 line) of the board.

The jumpers controlling the LEDs (labelled LED Jumper Block on the LPC2106 KickStart) may remain in their default state of being connected to P0.0-P0.15. This is assumed by code which allows use of the user-configurable LEDs, as described later.

All other jumpers can remain in their factory-supplied default state.

Flash Installation

This process assumes that a Microsoft Windows machine with the Flash Magic utility installed is available. Flash Magic is a tool for programming flash based microcontrollers from NXP using a protocol via the RS232 serial port to communicate with the In-System Programming (ISP) firmware on the LPC2xxx. The Flash Magic utility is sponsored by NXP and available from this website.

The first step is to connect the RS232 cable between UART0 of the KickStart board and the host PC. Do not use UART1. Now close the two jumpers mentioned earlier labelled EN_SW_ISP and EN_SW_RST. These allow the software on the PC to reset the LPC2xxx and enter the ISP firmware. Finally apply the power.

Start the Flash Magic utility on the host PC, and a window will be displayed allowing various parameters to be configured in a series of steps. For step 1, firstly choose the appropriate COM port that is being used on your PC and set the Baud Rate to 38400 baud.

Next select the appropriate LPC2xxx device in use such as LPC2106. The "Interface" should be set to "None (ISP)". And finally for step 1 choose the appropriate Oscillator Frequency for the KickStart board in use. This may be found in the board documentation, and is usually visibly readable on the surface of the oscillator on the board (in a metal package). For example for the LPC2106 KickStart, the oscillator reads T14.7456 indicating 14.7456MHz.

For step 2, it is usually adequate to leave the option "Erase blocks used by hex file" checked, and ignore the other settings. For step 3, you must select the program image to be downloaded, in Intel HEX format. To program the pre-built GDB stub ROM image, locate the file `gdb_module.hex` in the `loaders` subdirectory of your release. To generate an Intel HEX format version of an application you have built yourself run the following command at a shell prompt:

```
$ arm-eabi-objcopy -O ihex app.elf app.hex
```

This converts the application image in ELF format (as output by the linker), to Intel HEX format in the file `app.hex`. Note that the arm-eabi tools must be on your path at this point. If they are not, run the command below before you run the above **arm-eabi-objcopy** command:

```
$ . /opt/ecos/ecosenv.sh
```

In step 4, it is recommended to set the option "Verify after programming". Finally it is possible to click on "Start" to program the image into the on-chip Flash.

Tip

If there is a problem communicating with the board, such as a report of a failure to autobaud, then this may imply that the Flash Magic tool was not able to control the serial lines properly. This can happen with some USB-Serial converters. To workaround this issue, power the board off and remove (i.e. open) the EN_SW_ISP and EN_SW_RST jumpers. Then simultaneously press the buttons marked 'Reset' and 'ISP/INT1' then release the Reset button, and finally release the 'ISP/INT1' button. This is an alternative mechanism of forcing the ISP firmware to be entered. Once this has been successfully performed, the programming operation may be retried from the Flash Magic utility.

When the process completes, remove (i.e. open) the two jumpers next to the serial port. If a GDB stub ROM image has been programmed, verify the programming has been successful by starting a terminal emulation application such as HyperTerminal on the host PC and set the serial communication parameters to 38400 baud, 8 data bits, no parity, 1 stop bit (8N1) and no flow control (handshaking). Reset the board and the stubrom should start. For boards programmed with GDB stubs the output should be similar to the following:

```
+$T050f:ec070000;0d:28080040;#52
```

This is the stubrom attempting to communicate with GDB and indicates that it is functioning correctly.

Rebuilding the GDB Stub ROM

Should it prove necessary to rebuild the GDB Stub ROM binary, this is done most conveniently at the command line. Your `PATH` and `ECOS_REPOSITORY` environment variables must first be set correctly, The following steps given an example of how to rebuild the stubs for a KickStart board with LPC2106:

```
$ mkdir stub_rom
$ cd stub_rom
$ ecosconfig new iar_kickstart_lpc2106 stubs
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

At the end of the build the `install/bin` subdirectory should contain the files `gdb_module.img` (ELF format), `gdb_module.srec` (Motorola S-Record format), `gdb_module.bin` (raw binary format), and `gdb_module.hex` (Intel HEX format).

Name

Configuration — Platform-specific Configuration Options

Overview

The IAR KickStart platform HAL package is loaded automatically when eCos is configured for a target which uses it. The target names include the LPC2xxx model in use. At this time the only target supported is the `iar_kickstart_lpc2106` target. It should never be necessary to load this platform HAL package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The IAR KickStart platform HAL package supports three separate startup types:

ROM
: This startup type can be used for finished applications which will be programmed into internal flash at location 0x0. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization. On targets with less than 64Kbytes of SRAM, this is the startup type normally used.

RAM
: This is the startup type which is normally used during application development on targets with 64Kbytes of SRAM or greater. The board has the GDB stubrom or RedBoot programmed into flash at location 0x0 in internal on-chip Flash and boots from that location. arm-eabi-gdb is then used to load a RAM startup application into memory and debug it. It is assumed that the hardware has already been initialized by the ROM monitor. By default the application will use the eCos virtual vectors mechanism to obtain certain services from the ROM monitor, including diagnostic output. Larger applications may not fit into the available SRAM, in which case ROM startup may be required.

JTAG
: This is an alternative development startup type. The application is loaded into RAM via a JTAG device and is run and debugged from there. The application will be self-contained with no dependencies on services provided by other software. It is expected that hardware setup will have been performed via the JTAG device prior to loading. Some sample configuration and initialisation scripts for a number of JTAG debugging solutions may be found in the `misc` subdirectory of the platform HAL package within the component repository.

The ROM Monitor and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building the GDB Stubrom or RedBoot.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, or as a testing step before switching to ROM startup. Virtual vector support is usually only required for RAM startup applications or ROM monitors. The CDL option `CYGFUN_HAL_LPC2XXX_VIRTUAL_VECTOR_SUPPORT` within the LPC2xxx variant HAL allows manual control of this facility.

If the application does not rely on a ROM monitor for diagnostic services then by default serial port UART0 will be claimed for HAL diagnostics. If using the Prototype Board, it becomes possible to use UART1 as well, and in order to allow its selection as a potential debug or diagnostic channel, the number of communications channels can be increased from 1 to 2 with the option `CYGNUM_HAL_VIRTUAL_VECTOR_COMM_CHANNELS`. To specify which serial port is used for diagnostics, the configuration option `CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL` may be set accordingly, and its baud rate configured with `CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL_BAUD`.

System Clock

By default, the system clock interrupts once every 10ms, corresponding to a 100Hz clock. This can be changed by the configuration option `CYGNUM_HAL_RTC_DENOMINATOR` which corresponds to the clock frequency. Other clock-related settings are recalculated automatically if the denominator is changed. The PLL multiplier may be configured to allow a core clock (CCLK) speed of up to 60MHz. The description of the clock-related CDL options may be found in the LPC2xxx variant HAL documentation. Note there are frequency constraints on the Current Controlled Oscillator (CCO) within the LPC2xxx, and the datasheet should be consulted to ensure the required specifications are not exceeded.

Compiler Flags

The platform HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos.

However there are two flags that are used if Thumb mode is to be supported:

`-mthumb`
: The arm-eabi-gcc compiler will compile C and C++ files into the Thumb instruction set when this option is used. To build eCos in Thumb mode, enable the `CYGHWR_THUMB` configuration option in the ARM architecture HAL.

`-mthumb-interwork`
: This option allows programs to be created that mix ARM and Thumb instruction sets. For example, allowing a Thumb application to be used with eCos built in normal ARM mode. Without this option, some memory can be saved. This option should be used if -mthumb is used. eCos may be built with Thumb interworking support by enabling the `CYGBLD_ARM_ENABLE_THUMB_INTERWORK` CDL option in the ARM architecture HAL. Use of the LPC2xxx Flash driver requires Thumb interworking support to be enabled as the calls into the IAP firmware are must be made allowing a switch to Thumb mode.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the IAR KickStart board hardware, and should be read in conjunction with that specification. The KickStart Board platform HAL package complements the ARM architectural HAL and the LPC2xxx variant HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset the HAL will initialize or reinitialize many of the on-chip peripherals. This includes the PINSEL functions and LED bank. There is an exception for RAM startup applications which depend on a ROM monitor for certain services.

For ROM startup, the HAL will perform additional initialization, programming the various internal registers including PLL (for the clocks), Memory Mapping control registers to map SRAM to 0x0, and Memory Acceleration Module (MAM). The details of the early hardware startup may be found in the header `cyg/hal/hal_platform_setup.h`.

Linker Scripts and Memory Maps

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

on-chip Flash

This is located at address 0x0 of the memory space, although after hardware initialization, the start of internal SRAM is mapped over locations 0x0 to 0x40. The size of this region depends on the LPC2xxx microcontroller variant in use. In the case of the LPC2106, the region is of size 128Kbytes, ending at 0x20000. However the last few blocks of Flash are reserved for use as bootblocks for the ISP/IAP firmware, resulting in a usable Flash size of 120Kbytes, ending at 0x1e000. The MAM is enabled to accelerate memory reads from this area.

internal SRAM

This is located at address 0x40000000 of the memory space, and is 16, 32 or 64k in size, depending on the chip fitted. The first 64 bytes are mapped to location 0x0000000. If using GDB stubs ROM, or another ROM monitor, the virtual vector table starts at 0x40000050 and extends to 0x40000150. The remainder of SRAM is available for use by applications. For RAM startup applications, SRAM below 0x40001000 is reserved for the GDB stubrom and the remainder is available for the application. An exception is if the on-chip Flash driver is to be used. In that case, the top 32 bytes of SRAM are used by it. This is automatically handled in the port's memory layout files if the flash driver is present in the configuration.

on-chip peripherals

These are accessible at location 0xE0000000 onwards. Descriptions of the contents can be found in the LPC2xxx User Manual for the appropriate microcontroller variant.

Real-time characterization

The tm_basic kernel test gives statistics gathered about the real-time characterization and performance of the kernel. The sample output is shown here for information. The test was built in ARM mode, which provided better performance than Thumb mode.

Example 245.1. iar_kickstart Real-time characterization

```
            Startup, main stack : stack used   420 size  3920
             Startup :  Interrupt stack used   148 size  4096
             Startup : Idlethread stack used    80 size  2048

eCos Kernel Timings
Notes: all times are in microseconds (.000001) unless otherwise stated
```

```
Reading the hardware clock takes 0 'ticks' overhead
... this value will be factored out of all other measurements
Clock interrupt took   14.94 microseconds (8 raw clock ticks)

Testing parameters:
   Clock samples:            32
   Threads:                   2
   Thread switches:         128
   Mutexes:                  32
   Mailboxes:                32
   Semaphores:               32
   Scheduler operations:    128
   Counters:                 32
   Flags:                    32
   Alarms:                   32


                                 Confidence
     Ave     Min     Max     Var  Ave  Min  Function
  ======  ======  ======  ====== ========== ========
   13.56   13.56   13.56    0.00  100% 100% Create thread
    3.39    3.39    3.39    0.00  100% 100% Yield thread [all suspended]
    4.24    3.39    5.09    0.85  100%  50% Suspend [suspended] thread
    3.39    3.39    3.39    0.00  100% 100% Resume thread
    5.09    5.09    5.09    0.00  100% 100% Set priority
    1.70    1.70    1.70    0.00  100% 100% Get priority
   11.87   11.87   11.87    0.00  100% 100% Kill [suspended] thread
    4.24    3.39    5.09    0.85  100%  50% Yield [no other] thread
    5.93    5.09    6.78    0.85  100%  50% Resume [suspended low prio] thread
    4.24    3.39    5.09    0.85  100%  50% Resume [runnable low prio] thread
    5.93    5.09    6.78    0.85  100%  50% Suspend [runnable] thread
    3.39    3.39    3.39    0.00  100% 100% Yield [only low prio] thread
    3.39    3.39    3.39    0.00  100% 100% Suspend [runnable->not runnable]
   11.87   11.87   11.87    0.00  100% 100% Kill [runnable] thread
    8.48    8.48    8.48    0.00  100% 100% Destroy [dead] thread
   15.26   15.26   15.26    0.00  100% 100% Destroy [runnable] thread
   22.04   20.35   23.74    1.70  100%  50% Resume [high priority] thread
    8.38    6.78   11.87    0.23   92%   7% Thread switch

    1.38    0.00    1.70    0.52   81%  18% Scheduler lock
    3.31    1.70    3.39    0.15   95%   4% Scheduler unlock [0 threads]
    3.31    1.70    3.39    0.15   95%   4% Scheduler unlock [1 suspended]
    3.31    1.70    3.39    0.15   95%   4% Scheduler unlock [many suspended]
    3.31    1.70    3.39    0.15   95%   4% Scheduler unlock [many low prio]

    1.75    1.70    3.39    0.10   96%  96% Init mutex
    4.24    3.39    5.09    0.85  100%  50% Lock [unlocked] mutex
    4.82    3.39    5.09    0.45   84%  15% Unlock [locked] mutex
    4.08    3.39    5.09    0.82   59%  59% Trylock [unlocked] mutex
    3.71    3.39    5.09    0.52   81%  81% Trylock [locked] mutex
    1.38    0.00    1.70    0.52   81%  18% Destroy mutex
   20.35   20.35   20.35    0.00  100% 100% Unlock/Lock mutex

    1.96    1.70    3.39    0.45   84%  84% Create mbox
    1.38    0.00    1.70    0.52   81%  18% Peek [empty] mbox
    4.50    3.39    5.09    0.76   65%  34% Put [first] mbox
    1.27    0.00    1.70    0.64   75%  25% Peek [1 msg] mbox
    4.56    3.39    5.09    0.73   68%  31% Put [second] mbox
    1.22    0.00    1.70    0.68   71%  28% Peek [2 msgs] mbox
    4.56    3.39    5.09    0.73   68%  31% Get [first] mbox
    4.56    3.39    5.09    0.73   68%  31% Get [second] mbox
    3.97    3.39    5.09    0.76   65%  65% Tryput [first] mbox
    3.39    3.39    3.39    0.00  100% 100% Peek item [non-empty] mbox
    4.40    3.39    5.09    0.82   59%  40% Tryget [non-empty] mbox
    3.66    3.39    5.09    0.45   84%  84% Peek item [empty] mbox
    3.92    3.39    5.09    0.73   68%  68% Tryget [empty] mbox
    1.38    0.00    1.70    0.52   81%  18% Waiting to get mbox
    1.43    0.00    1.70    0.45   84%  15% Waiting to put mbox
    2.44    1.70    3.39    0.83   56%  56% Delete mbox
   13.88   13.56   15.26    0.52   81%  81% Put/Get mbox

    1.70    1.70    1.70    0.00  100% 100% Init semaphore
    3.71    3.39    5.09    0.52   81%  81% Post [0] semaphore
    3.97    3.39    5.09    0.76   65%  65% Wait [1] semaphore
    3.66    3.39    5.09    0.45   84%  84% Trywait [0] semaphore
    3.60    3.39    5.09    0.37   87%  87% Trywait [1] semaphore
    1.75    1.70    3.39    0.10   96%  96% Peek semaphore
    1.70    1.70    1.70    0.00  100% 100% Destroy semaphore
   13.56   13.56   13.56    0.00  100% 100% Post/Wait semaphore

    2.01    1.70    3.39    0.52   81%  81% Create counter
    1.70    1.70    1.70    0.00  100% 100% Get counter value
    1.22    0.00    1.70    0.68   71%  28% Set counter value
    4.13    3.39    5.09    0.83   56%  56% Tick counter
    1.43    0.00    1.70    0.45   84%  15% Delete counter

    1.70    1.70    1.70    0.00  100% 100% Init flag
    4.13    3.39    5.09    0.83   56%  56% Destroy flag
    3.50    3.39    5.09    0.20   93%  93% Mask bits in flag
    4.03    3.39    5.09    0.79   62%  62% Set bits in flag [no waiters]
    5.40    5.09    6.78    0.52   81%  81% Wait for flag [AND]
    5.30    5.09    6.78    0.37   87%  87% Wait for flag [OR]
    5.30    5.09    6.78    0.37   87%  87% Wait for flag [AND/CLR]
    5.35    5.09    6.78    0.45   84%  84% Wait for flag [OR/CLR]
    1.22    0.00    1.70    0.68   71%  28% Peek on flag

    2.70    1.70    3.39    0.82   59%  40% Create alarm
    6.46    5.09    6.78    0.52   81%  18% Initialize alarm
    3.81    3.39    5.09    0.64   75%  75% Disable alarm
    6.04    5.09    6.78    0.83   56%  43% Enable alarm
    4.29    3.39    5.09    0.84   53%  46% Delete alarm
    4.82    3.39    5.09    0.45   84%  15% Tick counter [1 alarm]
   22.15   22.04   23.74    0.20   93%  93% Tick counter [many alarms]
    8.05    6.78    8.48    0.64   75%  25% Tick & fire counter [1 alarm]
  138.82  137.33  139.03    0.37   87%  12% Tick & fire counters [>1 together]
   25.70   25.43   27.13    0.45   84%  84% Tick & fire counters [>1 separately]
   13.56   13.56   13.56    0.00  100% 100% Alarm latency [0 threads]
   15.54   13.56   18.65    1.49   50%  32% Alarm latency [2 threads]
   15.54   13.56   18.65    1.49   50%  32% Alarm latency [many threads]
   27.14   27.13   28.82    0.02   99%  99% Alarm -> thread resume latency

    3.39    3.39    3.39    0.00            Clock/interrupt latency

    7.93    6.78   11.87    0.00            Clock DSR latency

  272    272     272  (main stack:   764)  Thread stack used (1360 total)
           All done, main stack : stack used   764 size  3920
            All done :  Interrupt stack used   204 size  4096
            All done : Idlethread stack used   248 size  2048

Timing complete - 30220 ms total

PASS:<Basic timing OK>
EXIT:<done>
```

LED use

LEDs are available on the KickStart boards although most of these are attached to lines associated with peripherals. However 4 LEDS are available for application use from C. The following C function may be used:

```
#include <cyg/infra/hal_diag.h>
extern void hal_diag_led(int leds);
```

Values from 0 to 15 will be displayed on the LED bank representing the binary value with 1 being on and 0 being off. The LEDs used are connected to P0.10-P0.13 P0.13 being the MSB, and P0.10 the LSB.

The LEDs are also used during platform initialization and only P0.10 should be illuminated if booting has been successful. Other LED indications represent the stage in the initialization process that failed.

Other Issues

The following pin assignments are configured by default for LPC2106 at board initialisation time:

PINSEL0:
    P0.0/P0.1 for UART0
    P0.2/P0.3 for I2C
    P0.4/P0.5/P0.6/P0.7 for SPI
    P0.8/P0.9 for UART1
    P0.10-P0.13 as GPIO-controlled LEDs
    P0.14 EINT1
    P0.15 EINT2

    PINSEL1:
    P0.16 EINT0
    P0.17-P0.21 as GPIO, although in practice these are used for JTAG
    if a JTAG unit is connected.
    P0.22-P0.31 as GPIO inputs

Chapter 246. Keil MCB2387 Board Support

Name

eCos Support for the Keil MCB2387 Board — Overview

Description

The Keil MCB2387 Board is fitted with an NXP LPC2387 processor rated up to 72MHz, which contains 64KB of SRAM and 512KB of FLASH. It provides access to two on-chip UARTs, an MMC/SD card socket, and a PHY connected to the on-chip Ethernet MAC. Refer to the board documentation for full details.

For typical eCos development, a GDB StubROM image is programmed into the LPC2387 on-chip flash memory, and the board will boot this image from reset. This provides gdb stub functionality so it is then possible to download and debug stand-alone and eCos applications via the gdb debugger using UART 0.

This documentation describes platform-specific elements of the MCB2387 Board support within eCos. Documentation on the NXP LPC2xxx variants is available separately, and should be read in conjunction with this documentation. The LPC2xxx documentation covers various topics including HAL support common to LPC2xxx variants, and on-chip device support. This document complements the LPC2xxx documentation.

Supported Hardware

The MCB2387 board has 512Kbyte of on-chip Flash memory. In a typical setup, the stubrom will run from this internal flash. An image must be programmed into this flash using either the FlashMagic utility, or via a JTAG debugger.

The first 64 bytes of on-chip SRAM are mapped by the HAL startup code using the LPC2387 memory mapping control to location 0x00000000 for speed of interrupt vector processing. The rest of SRAM is available for use by the application.

The NXP LPC2xxx variant HAL includes support for the on-chip serial devices which is documented in the variant HAL. While the interrupt-driven serial driver supports the line status and modem control features of the UART devices, none of these lines are made available on the COM0 or COM1 connectors.

The MCB2387 board port includes support for the on-chip watchdog, RTC (wallclock), and interrupt controller (VIC). This support is documented in the LPC2xxx variant HAL.

The on-chip Ethernet MAC is supported.

The on-chip Multimedia Card Interface (MCI) is supported to allow access to Multimedia Cards (MMC) or Secure Digital (SD) cards using the socket on the OEM board.

Drivers for I²C and SPI are present. However, since there are no on-board devices connected to these busses, they have only been tested using external devices attached to the board for the purpose.

Tools

The MCB2387 board port is intended to work with GNU tools configured for an arm-eabi target. Thumb mode is supported. The original port was done using arm-elf-gcc version 3.4.4, arm-elf-gdb version 6.3, and binutils version 2.16.

Name

Setup — Preparing the MCB2387 Board for eCos Development

Overview

In a typical development environment, the MCB2387 board boots from internal flash into the GDB stubrom. eCos applications are configured for RAM startup and then downloaded and run on the board via the debugger **arm-eabi-gdb**. Preparing the board therefore usually involves programming a suitable image into flash memory.

Initial Installation

Flash Installation

This process assumes that a Microsoft Windows machine with the Embedded Systems Academy Flash Magic utility is available. Install Flash Magic from http://www.flashmagictool.com. Connect a 9-pin serial cable from a COM port on the PC to the COM0 connector on the MCB2387 Board. Power the board via a USB cable.

Set the ISP jumpers (J9(RST) and J10(ISP) on, J13(ETM) off) and press the reset button. The board is now running a special NXP boot loader. Start Flash Magic and set the Communications section to select the COM port used above, 38400 baud, device LPC2387, Interface “None (ISP)” and 12MHz Oscillator Frequency. Test communication with the board by using the “ISP->Read Device Signature” menu entry. If communication is not successful, check that the serial cable is connected, the ISP jumpers are installed and the correct COM port is being used.

Check “Erase blocks used by Hex File” under “Erase”. In the “Hex File” section, select the `stubrom.hex` file. Under “Options”, all boxes should be clear except “Verify after programming”. Now press the “Start” button. The utility should show the progress of the upload.

When the process completes, the utility should be closed. Reset the ISP jumpers (J9(RST) and J10(ISP) off, J13(ETM) on). Verify the programming has been successful by starting a terminal emulation application such as HyperTerminal or minicom on the host PC and set the serial communication parameters to 38400 baud, 8 data bits, no parity, 1 stop bit (8N1) and no flow control (handshaking). Reset the board and the stubrom should start. The output should be similar to the following:

```
+$T050f:98070000;0d:48070040;#fc
```

This is the stubrom reporting it's state using the GDB remote protocol.

Rebuilding the Stubrom

Should it prove necessary to rebuild the Stubrom binary, this is done most conveniently at the command line. Assuming your `PATH` and `ECOS_REPOSITORY` environment variables have been set correctly, the steps needed to rebuild RedBoot for the MCB2387 are:

```
$ mkdir stub_mcb2387_rom
$ cd stub_mcb2387_rom
$ ecosconfig new mcb2387 stubs
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

At the end of the build the `install/bin` subdirectory should contain the file `stubrom.hex`.

Name

Configuration — Platform-specific Configuration Options

Overview

The mcb2387 platform HAL package is loaded automatically when eCos is configured for an `mcb2387` target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The mcb2387 platform HAL package supports three separate startup types:

JTAG
: This is the startup type which is normally used during JTAG based application development. arm-eabi-gdb is then used to connect to the JTAG device and load a JTAG startup application into memory and debug it. It is assumed that the basic hardware has already been initialized via the JTAG device's initialization script. Otherwise the application is entirly self contained and should contain drivers for all hardware used.

RAM
: This is the startup type which is normally used during stubrom based application development. The board has the stubrom programmed into flash at location 0x0 in internal on-chip Flash and boots from that location. arm-eabi-gdb is then used to load a RAM startup application into memory and debug it. It is assumed that the basic hardware has already been initialized by the stubs. By default the application will use the eCos virtual vectors mechanism to obtain certain services from the stubrom, including diagnostic output.

ROM
: This startup type can be used for finished applications which will be programmed into internal flash at location 0x0. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization.

Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building the stubrom.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, or as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then serial port UART0 will be claimed for HAL diagnostics.

Flash Driver

The `CYGPKG_DEVS_FLASH_LPC2XXX` package contains all the code necessary to support the on-chip flash.

System Clock

By default, the system clock interrupts once every 10ms, corresponding to a 100Hz clock. This can be changed by the configuration option `CYGNUM_HAL_RTC_DENOMINATOR` which corresponds to the clock frequency. Other clock-related settings are recalculated automatically if the denominator is changed. The PLL multipliers and dividers may be configured to allow a core clock (CCLK) speed of up to 72MHz. However, the platform HAL currently sets the clock to 48MHz, duplicating the configuration in the supplied example code as a consequence of CPU errata affecting various revisions of the LPC2387. Setting the CPU revision with the `CYGHWR_HAL_ARM_LPC2XXX_MCB2387_CPU_REVISION` configuration option can be used to provide default clock settings appropriate to the CPU revision in use. If the CPU revision cannot be guaranteed it should be left as "Initial". The description of the clock-related CDL options may be found in the LPC2xxx variant HAL documentation.

I²C Bus Configuration

The on-chip I²C devices are supported by a driver in the variant HAL package. Each bus for this driver needs to be configured in the platform HAL with the following options:

`CYGPKG_HAL_ARM_LPC2XXX_I2CX`

> This is the master component, enabling this activates all the other configuration options and causes the driver to create the data structures to access this bus.

`CYGPKG_HAL_ARM_LPC2XXX_I2CX_CLOCK`

> Bus clock speed in Hz. Usually frequencies of either 100kHz or 400kHz are chosen, the latter sometimes known as fast mode.

`CYGPKG_HAL_ARM_LPC2XXX_I2CX_SDA`

> This option describes the pin used for SDA on this bus. This takes the form of an invocation of the macro __LPC2XXX_PINSEL_FUNC. Parameters are the port number, pin within that port, and the alternate select function for the pin. See the LPC2387 user manual for details of which pins may be used by each bus.

`CYGPKG_HAL_ARM_LPC2XXX_I2CX_SCL`

> This option describes the pin used for SCL on this bus. Like SDA this takes the form of a call to __LPC2XXX_PINSEL_FUNC.

Note that "I2CX" is a placeholder for a given bus instance: "I2C0", "I2C1" or "I2C2". By default the platform HAL does not enable any I²C buses since there are no on-board devices.

SPI Bus Configuration

The on-chip SSP SPI devices (not the Legacy SPI device) are supported by the NXPSSP driver package, `CYGPKG_DEVS_SPI_ARM_NXPSSP`. This needs some configuration in the platform HAL:

`CYGPKG_HAL_ARM_LPC2XXX_SPI`

> This is the master component, enabling this activates all the other configuration options. It also causes `mcb2387_spi.c` to be compiled, which contains descriptions of the devices on the SPI buses.

`CYGPKG_HAL_ARM_LPC2XXX_SPIX`

> This is the master component for each bus. Enabling this activates the other configuration options for this bus, and causes the driver to support this bus.

`CYGPKG_HAL_ARM_LPC2XXX_SPIX_SCLK`

> This option describes the pin used for SCLK on SPIX. It takes the form of an invocation of __LPC2XXX_PINSEL_FUNC. The parameters are the port number, pin within that port, and the alternate select function for the pin. See the LPC2387 user manual for details."

`CYGPKG_HAL_ARM_LPC2XXX_SPIX_MISO`

> This option describes the pin used for MISO on SPIX. Like SCLK it takes the form of a call to __LPC2XXX_PINSEL_FUNC.

`CYGPKG_HAL_ARM_LPC2XXX_SPIX_MOSI`

> This option describes the pin used for MOSI on SPIX. Like SCLK it takes the form of a call to __LPC2XXX_PINSEL_FUNC.

CYGPKG_HAL_ARM_LPC2XXX_SPIX_CS_PINS

> This defines the pins to be uses as chip selects for this bus. It is a comma separated list of GPIO pin names, the first for device 0, the second for device 1, and so on. Pin names are defined in the `var_io.h` header in the LPC2xxx variant HAL.

Note that "SPIX" is a placeholder for a given bus instance: "SPI0" or "SPI1". By default the platform HAL does not enable any SPI busses since there are on on-chip devices.

MCI peripheral configuration

The on-chip Multimedia Card Interface (MCI) is supported to allow access to Multimedia Cards (MMC) or Secure Digital (SD) cards using the socket on the board. This support is provided in conjunction with the generic MMC/SD driver package (`CYGPKG_DEVS_DISK_MMC`), the Primecell MCI driver package (`CYGPKG_DEVS_MMCSD_ARM_PRIMECELL_MCI`) and the LPC2xxx variant HAL in order to provide some elements of the DMA support. Documentation and configuration options within those packages should also be consulted.

The following CDL configuration options are used to control the behaviour of the MMC/SD card support:

MMC/SD card support (`CYGPKG_HAL_ARM_LPC2XXX_MCB2387_MCI`)

> This option allows the MMC/SD card support as a whole to be enabled or disabled, although the generic disk device driver package (`CYGPKG_IO_DISK`) must be loaded in order to enable the MMC/SD support.

Use on-chip USB memory for DMA (`CYGSEM_HAL_ARM_LPC2XXX_MCB2387_MCI_USE_USB_MEM_FOR_DMA`

> The LPC2387 cannot always keep up with the data transfer requirements, especially at slower CPU clock speeds. This is because the DMA controller runs at the speed of the CPU clock (CCLK) along with the fact that some LPC2387 have errata which decreases their achievable CPU clock frequency.
>
> Using on-chip memory dedicated to USB helps reduce or remove these problems, depending on CPU frequency. Clearly this option must be disabled if the on-chip USB peripheral is to be used. It is also desirable to disable this option if the CPU frequency is high enough, in order to remove an extra copy on every data transfer, thus improving performance. The USB memory used is 512 bytes at the start of the USB memory space (0x7FD00000).
>
> If this option is disabled and the DMA is not able to proceed quickly enough, this will be visible in the form of I/O errors. In that case, if it is not possible to enable this option it is recommended to adjust the `CYGDAT_HAL_ARM_LPC2XXX_MCB2387_MCI_BUS_SPEED_LIMIT` configuration option.

Lock AHB bus during DMA transfer (`CYGSEM_HAL_ARM_LPC2XXX_MCB2387_MCI_DMA_LOCKS_AHB`)

> The AMBA Hardware Bus (AHB) is used to connect AMBA peripherals within the LPC2387, including the ARM core, DMA controller and memory controllers. When this option is enabled, the AHB is locked for the duration of MCI DMA transfer bursts. If another AMBA host needs to make a transfer it may be delayed as a result, which may not be desirable.
>
> Disabling this option allows the AHB arbiter to permit other AHB hosts to perform transfers. Of course this may mean the MCI DMA transfers can in turn themselves get delayed, risking data overruns or underruns in MCI transfers, resulting in I/O errors during block reads or writes. This is particularly likely on processors running at slower clock speeds where there may already be difficulties with the DMA servicing data transfers quickly enough.

MMC/SD bus frequency limit (`CYGNUM_HAL_ARM_LPC2XXX_MCB2387_MCI_BUS_SPEED_LIMIT`)

> The LPC2387 cannot always keep up with the data transfer requirements, especially at slower CPU clock speeds. This is because the DMA controller runs at the speed of the CPU clock (CCLK) along with the fact that some LPC2387 have errata which decreases their achievable CPU clock frequency. The adjacent options to use on-chip USB memory and to lock the AHB bus can help prevent this, but sometimes they are insufficient to prevent data overruns or underruns resulting in I/O errors

during block reads or writes. In which case the only remaining recourse is to reduce the required data transfer rate between the MCI and the card.

This option can be used to impose an upper limit on the MMC/SD bus frequency. The value used in this option is measured in Hertz, and the use of 4-bit mode with SD cards is not a factor - this option provides the bus frequency, so a 4-bit bus will transfer four times the amount of data as a 1-bit bus in the same time period.

Note that this option provides a limit, and does not mean the card bus will operate at that frequency. The frequency is also governed by what the card will support, and the resolution of the clock used to derive the MMC/SD clock signal, and how it can be divided down.

Compiler Flags

The platform HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos.

However there are two flags that are used if Thumb mode is to be supported:

`-mthumb`
: The arm-eabi-gcc compiler will compile C and C++ files into the Thumb instruction set when this option is used.

`-mthumb-interwork`
: This option allows programs to be created that mix ARM and Thumb instruction sets. Without this option, some memory can be saved. This option should be used if -mthumb is used.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the MCB2387 board hardware, and should be read in conjunction with that specification. The LPC2387 platform HAL package complements the ARM architectural HAL and the LPC2xxx variant HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset the HAL will initialize or reinitialize many of the on-chip peripherals. There is an exception for RAM startup applications which depend on a ROM monitor to do most of this and for JTAG startup, where some initialization will be done by the JTAG device.

For ROM startup, the HAL will perform additional initialization, programming the various internal registers including PLL (for the clocks); Memory Mapping control registers to map SRAM to 0x0 and the Memory Acceleration Module (MAM). The details of the early hardware startup may be found in the header `cyg/hal/hal_platform_setup.h`.

Linker Scripts and Memory Maps

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

on-chip Flash

This is located at address 0x0 of the memory space, although after hardware initialization, the start of internal SRAM is mapped over locations 0x0 to 0x40. This region ends at 0x80000. The MAM is enabled to accelerate memory reads from this area. A driver is available for using this flash via the eCos flash API.

internal SRAM

This is located at address 0x40000000 of the memory space, ending at location 0x4000FFFF. The first 64 bytes are mapped to location 0x0000000.

on-chip peripherals

These are accessible via location 0xE0000000 onwards. Descriptions of the contents can be found in the LPC2387 User Manual.

Chapter 247. Phytec phyCORE LPC2294 Board Support

Name

eCos Support for the Phytec phyCORE LPC2294 Board — Overview

Description

The Phytec phyCORE LPC2294 Board is fitted with a Philips LPC2294 processor rated to 60MHz, which contains up to 64KB of SRAM and up to 256KB of FLASH. When used in conjunction with the phyCORE HD200 development board, it provides two 9-pin RS-232 serial interfaces connected to the LPC2294 on-chip UARTs, a single LED, two CAN interfaces and an SMSC LAN91C111 ethernet interface. Refer to the board documentation for full details.

For typical eCos development, a RedBoot image is programmed into the LPC2294 on-chip flash memory, and the board will boot this image from reset. RedBoot provides gdb stub functionality so it is then possible to download and debug stand-alone and eCos applications via the gdb debugger using UART 0 or via the ethernet.

This documentation describes platform-specific elements of the phyCORE LPC2294 Board support within eCos. Documentation on the Philips LPC2xxx variants is available separately, and should be read in conjunction with this documentation. The LPC2xxx documentation covers various topics including HAL support common to LPC2xxx variants, and on-chip device support. This document complements the LPC2xxx documentation.

Supported Hardware

The phyCORE LPC2294 Board has 128Kbyte of on-chip Flash memory. In a typical setup, RedBoot will load and run from this internal flash. An initial image must be programmed into this flash using either the FlashMagic utility, or via a JTAG debugger. Following this, it may be reprogrammed using flash drivers in RedBoot.

The first 64 bytes of on-chip SRAM are mapped by the HAL startup code using the LPC2294 memory mapping control to location 0x00000000 for speed of interrupt vector processing. The rest of SRAM is available for use by the application. One MByte of SRAM is available at 0x81000000; the first 64KBytes of this is reserved for use by RedBoot, the rest is available for the code and data of loaded applications.

The Philips LPC2xxx variant HAL includes support for the two on-chip serial devices and is documented in the variant HAL. The interrupt-driven serial driver supports the line status and modem control (including hardware handshaking) lines on UART1 only. These handshaking lines are not accessible at the DB9 connector (P1B).

The phyCORE LPC2294 Board port includes support for the on-chip watchdog, RTC (wallclock), and interrupt controller (VIC). This support is documented in the LPC2xxx variant HAL.

The SMSC LAN91C111 ethernet MAC is supported. However, due to PCB tracking problems, it is only capable of running at 10MBit/s and the driver forces the device to only operate at that speed.

Tools

The phyCORE LPC2294 Board port is intended to work with GNU tools configured for an arm-eabi target. Thumb mode is supported. The original port was done using arm-elf-gcc version 3.3.3, arm-elf-gdb version 6.1, and binutils version 2.14.

Name

Setup — Preparing the phyCORE LPC2294 Board for eCos Development

Overview

In a typical development environment, the phyCORE LPC2294 Board boots from internal flash into RedBoot. eCos applications are configured for RAM startup and then downloaded and run on the board via the debugger **arm-eabi-gdb**. Preparing the board therefore usually involves programming a suitable stubrom image into flash memory.

The following RedBoot configurations are supported:

Configuration	Description	Use	File
ROM	RedBoot running from ROM	`redboot_ROM.ecm`	`redboot_ROM.hex`
RAM	RedBoot running from RAM	`redboot_RAM.ecm`	`redboot_RAM.srec`

For serial communications, all versions run with 8 bits, no parity, and 1 stop bit at 38400 baud. This baud rate can be changed via the configuration option `CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL_BAUD` and rebuilding the stubrom.

Initial Installation

Flash Installation

This process assumes that a Microsoft Windows machine with the Embedded Systems Academy Flash Magic utility is available.

Install Flash Magic from http://www.flashmagictool.com.

Connect the RS232 cable supplied with the phyCORE LPC2294 between port 0 of the phyCORE LPC2294 and the host PC. Apply power to the board and with the Boot button (S_1) held down, press and release the reset button (S_2). The board is now running a special NXP boot loader. Start Flash Magic and set the Communications section to select the appropraite serial port, 38400 baud, device LPC2294, Interface “None (ISP)” and 12MHz Oscillator Frequency. Test communication with the board by using the “ISP->Read Device Signature” menu entry. If communication is not successful, check that the serial cable is connected correctly, that the board was booted with the Boot button (S_1) held down and that the correct communication parameters have been selected in Flash Magic.

Check “Erase blocks used by Hex File” under “Erase”. In the “Hex File” section, select the `redboot_ROM.hex` file. Under “Options”, all boxes should be clear except “Verify after programming”. Now press the “Start” button. The utility should show the progress of the upload.

When the process completes, the utility should be closed. Verify that programming has been successful by starting a terminal emulation application such as HyperTerminal or minicom on the host PC and set the serial communication parameters to 38400 baud, 8 data bits, no parity, 1 stop bit (8N1) and no flow control (handshaking). Reset the board and RedBoot should start. The output should be similar to the following:

```
+... Read from 0x801e0000-0x801fffff to 0x810e0000:
... Read from 0x801ff000-0x801fffff to 0x810df000:
... waiting for BOOTP information
Ethernet eth0: MAC address 00:50:c2:32:ad:40
IP: 10.0.0.200/255.255.255.0, Gateway: 10.0.0.3
Default server: 10.0.0.102, DNS server IP: 10.0.0.1

RedBoot(tm) bootstrap and debug environment [ROM]
Non-certified release, version UNKNOWN - built 17:10:02, Dec  1 2004

Platform: Phytec phyCORE LPC229x Board (ARM7TDMI)
Copyright (C) 2000, 2001, 2002, 2003, 2004 Free Software Foundation, Inc.
Copyright (C) 2003, 2004, 2005 eCosCentric Limited

RAM: 0x81000000-0x81100000, [0x8100b1a8-0x810dd000] available
FLASH: 0x80000000 - 0x801fffff 16 x 0x20000 blocks
RedBoot>
```

If is it ever necessary to reinstall RedBoot, the above directions can be repeated. Alternatively, a new RedBoot may be installed from RedBoot itself. It is not possible to do this directly, since RedBoot is executing from the flash that needs to be erased and reprogrammed. Instead it is necessary to run a RAM version of RedBoot, use that to download the new ROM RedBoot to RAM, and then program that to flash.

The following shows an example session to do this. It assumes that `redboot_RAM.srec` and `redboot_ROM.bin` are available via TFTP on the server set up in **fconfig**.

```
RedBoot> load redboot_RAM.srec
Using default protocol (TFTP)
Entry point: 0x81010040, address range: 0x81010000-0x8102c04c
RedBoot> go
+Ethernet eth0: MAC address 00:50:c2:3b:aa:9d
IP: 192.168.7.251/255.255.255.0, Gateway: 192.168.7.11
Default server: 192.168.7.5
DNS server IP: 192.168.7.11, DNS domain name: <null>

RedBoot(tm) bootstrap and debug environment [RAM]
eCosCentric certified release, version v2_0_105 - built 15:42:30, Dec  5 2008

Platform: Phytec phyCORE LPC229x Board (ARM7TDMI)
Copyright (C) 2000, 2001, 2002, 2003, 2004 Free Software Foundation, Inc.
Copyright (C) 2003, 2004, 2005, 2006, 2007, 2008 eCosCentric Limited

RAM: 0x81000000-0x81800000, [0x81035f90-0x817dd000] available
FLASH: 0x00000000-0x0003dfff, 8 x 0x2000 blocks, 2 x 0x10000 blocks, 7 x 0x2000s
FLASH: 0x80000000-0x80ffffff, 63 x 0x20000 blocks, 8 x 0x4000 blocks, 63 x 0x20s
RedBoot> load -r -b %{freememlo} redboot_ROM.bin
Raw file loaded 0x81036000-0x81053463, assumed entry at 0x81036000
RedBoot> fis write -f 0x00000000 -b %{freememlo} -l 0x20000
* CAUTION * about to program FLASH
            at 0x00000000..0x0001ffff from 0x81036000 - continue (y/n)? y
... Erase from 0x00000000-0x0001ffff: .........
... Program from 0x81036000-0x81056000 to 0x00000000: .........
RedBoot> reset
+Ethernet eth0: MAC address 00:50:c2:3b:aa:9d
IP: 192.168.7.251/255.255.255.0, Gateway: 192.168.7.11
Default server: 192.168.7.5
DNS server IP: 192.168.7.11, DNS domain name: <null>

RedBoot(tm) bootstrap and debug environment [ROM]
eCosCentric certified release, version v2_0_105 - built 15:42:52, Dec  5 2008

Platform: Phytec phyCORE LPC229x Board (ARM7TDMI)
Copyright (C) 2000, 2001, 2002, 2003, 2004 Free Software Foundation, Inc.
Copyright (C) 2003, 2004, 2005, 2006, 2007, 2008 eCosCentric Limited

RAM: 0x81000000-0x81800000, [0x8100acc8-0x817dd000] available
FLASH: 0x00000000-0x0003dfff, 8 x 0x2000 blocks, 2 x 0x10000 blocks, 7 x 0x2000s
FLASH: 0x80000000-0x80ffffff, 63 x 0x20000 blocks, 8 x 0x4000 blocks, 63 x 0x20s
RedBoot>
```

Rebuilding RedBoot

Should it prove necessary to rebuild the RedBoot binary, this is done most conveniently at the command line. Assuming your `PATH` and `ECOS_REPOSITORY` environment variables have been set correctly, the steps needed to rebuild RedBoot are:

```
$ mkdir redboot_phycore_rom
$ cd redboot_phycore_rom
$ ecosconfig new phycore_lpc2294 redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/arm/lpc2xxx/phycore_lpc229x/current/misc/redboot_ROM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

At the end of the build the `install/bin` subdirectory should contain the file `redboot.hex`.

Note

The PhyCORE LPC2294 board can be fitted with a wide range of flash and SRAM parts. So it may be necessary to adjust the configuration after importing the `redboot_ROM.ecm` file to match the hardware being used. The Memory Configuration section contains full details of the options available for this.

Name

Configuration — Platform-specific Configuration Options

Overview

The phycore platform HAL package is loaded automatically when eCos is configured for an `phycore_lpc2294` target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The phyCORE LPC229x platform HAL package supports two separate startup types:

RAM This is the startup type which is normally used during application development. The board has RedBoot programmed into flash at location 0x0 in internal on-chip Flash and boots from that location. arm-eabi-gdb is then used to load a RAM startup application into memory and debug it. It is assumed that the hardware has already been initialized by the stubrom. By default the application will use the eCos virtual vectors mechanism to obtain certain services from the stubrom, including diagnostic output.

ROM This startup type can be used for finished applications which will be programmed into internal flash at location 0x0. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization.

RedBoot and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building RedBoot.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, or as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then serial port UART0 will be claimed for HAL diagnostics.

Flash Driver

The phyCORE LPC2294 board contains a number of AMD flash devices. The `CYGPKG_DEVS_FLASH_AMD_AM29XXXXX_V2` package contains all the code necessary to support these parts and the `CYGPKG_DEVS_FLASH_ARM_PHYCORE` package contains definitions that customize the driver to the phyCORE LPC2294 board.

Ethernet Driver

The phyCORE-LPC229x board contains an SMSC LAN91C111 ethernet MAC. The `CYGPKG_DEVS_ETH_SMSC_LAN91CXX` package contains all the code necessary to support this device and the `CYGPKG_DEVS_ETH_ARM_PHYCORE` package contains definitions that customize the driver to the phyCORE LPC2294 board.

System Clock

By default, the system clock interrupts once every 10ms, corresponding to a 100Hz clock. This can be changed by the configuration option `CYGNUM_HAL_RTC_DENOMINATOR` which corresponds to the clock frequency. Other clock-related settings are recalculated automatically if the denominator is changed. The PLL multipliers and dividers may be configured to allow a core clock (CCLK) speed of up to 60MHz. The description of the clock-related CDL options may be found in the LPC2xxx variant HAL documentation.

Memory Configuration

The PhyCORE LPC2294 board can be fitted with a wide range of flash and SRAM parts. The following options adjust the configuration of eCos and RedBoot to accommodate these variations:

`CYGHWR_HAL_ARM_LPC2XXX_PHYCORE_MEMORY_CONFIGURATION_FLASH`

> This option describes the flash devices fitted to the board. Possible values are: AM29DL800BT, AM29LV800BT, AM29LV160BT and AM29LV320BT. Of these only the AM29DL800BT and AM29LV320BT variants have been tested.

`CYGHWR_HAL_ARM_LPC2XXX_PHYCORE_MEMORY_CONFIGURATION_FLASH_COUNT`

> This option defines the number of flash devices fitted. Flash devices can only be fitted in pairs, and there is only space for up to 4, so this value can only be 2 or 4.

`CYGHWR_HAL_ARM_LPC2XXX_PHYCORE_MEMORY_CONFIGURATION_SRAM_SIZE`

> This option defines the total SRAM size. The board can be fitted with two or four SRAM devices, of 512KB, 1MB or 2MB each, giving Possible possible SRAM sizes of: 0x100000, 0x200000, 0x400000 or 0x800000.

Compiler Flags

The platform HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos.

However there are two flags that are used if Thumb mode is to be supported:

`-mthumb`
: The arm-eabi-gcc compiler will compile C and C++ files into the Thumb instruction set when this option is used.

`-mthumb-interwork`
: This option allows programs to be created that mix ARM and Thumb instruction sets. Without this option, some memory can be saved. This option should be used if -mthumb is used.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the phyCORE LPC2294 Board hardware, and should be read in conjunction with that specification. The phyCORE LPC229x platform HAL package complements the ARM architectural HAL and the LPC2xxx variant HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset the HAL will initialize or reinitialize many of the on-chip peripherals. This includes the PINSEL functions and LED bank. There is an exception for RAM startup applications which depend on a ROM monitor for certain services.

For ROM startup, the HAL will perform additional initialization, programming the various internal registers including PLL (for the clocks); Memory Mapping control registers to map SRAM to 0x0; the memory controller for access to external FLASH, SRAM and ethernet; and the Memory Acceleration Module (MAM). The details of the early hardware startup may be found in the header `cyg/hal/hal_platform_setup.h`.

Linker Scripts and Memory Maps

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

on-chip Flash
: This is located at address 0x0 of the memory space, although after hardware initialization, the start of internal SRAM is mapped over locations 0x0 to 0x40. This region ends at 0x40000. The MAM is enabled to accelerate memory reads from this area. A driver is available for using this flash via the eCos flash API.

external Flash
: This is located at address 0x80000000 of the memory space. It is not used by default by eCos, although if RedBoot is asked to manage the Flash, it reserves flash addresses 0x801E0000 thru 0x801FF000. If RedBoot stores its configuration data in Flash, then addresses 0x801FF000 thru 0x801FFFFF are reserved by RedBoot. RedBoot also reserves the first block of Flash at 0x80000000 thru 0x80001FFFF to ensure that it remains erased and does not therefore inhibit the execution of RedBoot from the internal Flash. External flash is 32bits wide and accessed with 7 wait states.

internal SRAM
: This is located at address 0x40000000 of the memory space, ending at location 0x40004000. The first 64 bytes are mapped to location 0x0000000.

external SRAM
: This is located at address 0x81000000 of the memory space, ending at location 0x81100000. For RAM startup, available SRAM starts at location 0x81010000, with the bottom 64Kbytes reserved for use by RedBoot.

on-chip peripherals
: These are accessible via location 0xE0000000 onwards. Descriptions of the contents can be found in the LPC2294 User Manual.

Other Issues

The LEDs may be accessed from C with the following function:

```
#include <cyg/infra/hal_diag.h>
extern void hal_diag_led(int leds);
```

Values from 0 to 16 will be displayed on the LED bank representing the binary value with 1 being on and 0 being off, and with P0.7 being the MSB, and P0.4 the LSB.

The LEDs are also used during platform initialization and only P0.4 should be illuminated if booting has been successful. Other LED indications represent the stage in the initialization process that failed.

Chapter 248. ST STR7XX variant HAL

Name

eCos Support for the ST Microelectronics STR7XX ARM microcontrollers — Overview

Description

The ST STR7XX series of ARM microcontrollers is supported by eCos with an eCos processor variant HAL and a number of device drivers supporting some of the on-chip peripherals. These include device drivers for the on-chip flash, serial and watchdog devices. In addition it provides common functionality and definitions that STR7XX based platform ports may require, as well as definitions useful to application developers.

This documentation covers the STR7XX functionality provided but should be read in conjunction with the specific HAL documentation for the platform port. That documentation will cover issues that are platform-specific and are not covered here, and may also describe differences that override or supersede what the STR7XX variant HAL provides. The areas that are specific to platform HALs and not the STR7XX variant HAL include:

- memory map and related configuration and setup
- memory remapping
- Clock parameters
- GPIO setup
- Any special handling for external interrupts, or additional interrupts
- Diagnostic I/O baud rates
- Additional diagnostic I/O devices, if any
- LED/LCD control

Name

On-chip Subsystems and Peripherals — Hardware Support

Hardware support

On-chip memory

The ST STR7XX parts include on-chip SRAM, and on-chip FLASH. The RAM consists of a single 64KiB block. The FLASH comprises a block of program memory which is either 64KiB, 128KiB or 256KiB in size depending on model, plus a 16KiB area of higher durability data memory. There is also support in some models for external SRAM and flash, which eCos may use where available.

Typically, an eCos platform HAL port will expect a GDB stub ROM monitor or RedBoot image to be programmed into either the external FLASH or the STR7XX on-chip ROM memory for development, and the board would boot this image from reset. The stub ROM/RedBoot provides GDB stub functionality so it is then possible to download and debug stand-alone and eCos applications via the gdb debugger using serial interfaces or other debug channels. The JTAG interface may also be used for development if a suitable JTAG device is available. If RedBoot is present it may also be used to manage the on-chip and external flash memory. For production purposes, applications are programmed into the external or on-chip ROM and will be self-booting.

Serial I/O

The STR7XX variant HAL supports basic polled HAL diagnostic I/O over any of the on-chip serial devices. There is also a fully interrupt-driven serial device driver suitable for eCos applications for all on-chip serial devices. The serial driver consists of an eCos package: `CYGPKG_IO_SERIAL_ARM_STR7XX` which provides all support for the STR7XX on-chip serial devices. Using the HAL diagnostic I/O support, any of these devices can be used by the ROM monitor or RedBoot for communication with GDB. If a device is needed by the application, either directly or via the serial driver, then it cannot also be used for GDB communication using the HAL I/O support. An alternative serial port should be used instead.

The STR7XX UARTs only provide the minimal TX and RX data lines; hardware flow control using RTS/CTS is not supported. The eCos device drivers have been extended to permit the use of a pair of GPIO lines as flow control lines. It is the responsibility of the platform HAL to enable this functionality and define the GPIO lines to be used in this way.

I^{2}C Support

The I²C® driver uses the STR7XX's internal support. This is controlled within the STR7XX variant HAL. The `CYGPKG_HAL_STR7XX_I2C` CDL component controls whether the I^{2}C driver is enabled. Within that component, there are two sub-options:

- `CYGNUM_HAL_STR7XX_I2C_BUS0_CLOCK` sets the speed of the I^{2}C bus 0 clock in Hz. This is usually 100kHz, but can be set up to 400kHz (fast mode) if the devices on the bus support this speed. Other values below 400kHz can also be chosen, subject to the accuracy of the clock waveform generation parameters.
- `CYGNUM_HAL_STR7XX_I2C_BUS1_CLOCK` sets the speed of the I^{2}C bus 1 clock in Hz. This is usually 100kHz, but can be set up to 400kHz (fast mode) if the devices on the bus support this speed. Other values below 400kHz can also be chosen, subject to the accuracy of the clock waveform generation parameters.

The I^{2}C driver is accessed via the generic I^{2}C driver package `CYGPKG_IO_I2C`. Documentation for its API may be found elsewhere.

This driver only operates in interrupt mode. It does not operate in polled mode, and thus does not operate when interrupts are disabled. It cannot therefore be used in an initialization context, before the eCos kernel thread scheduler starts, and it cannot be used with RedBoot.

Watchdog

A device driver is included for the on-chip watchdog device. This driver allows the use of the standard eCos watchdog API accessible with the `CYGPKG_IO_WATCHDOG` eCos package. If the watchdog is not reset within a time period defined in the watchdog device driver CDL, then the system is automatically reset.

The watchdog device is also used to implement reset functionality, it may also be called directly by applications using the following function:

```
#include <cyg/hal/hal_diag.h>
extern void hal_str7xx_reset_cpu(void);
```

Interrupt controller

eCos manages the on-chip Enhanced Interrupt Controller (EIC). The EIC is configured to use interrupts in non-vectored mode, although the vector mechanism is used to aid interrupt source decoding. External interrupts controlled by the XTI unit are also decoded into individual vectors.

Timer 0 is used to implement the eCos system clock. Timer-based profiling support is implemented using timer 1. If the gprof package, `CYGPKG_PROFILE_GPROF`, is included in the configuration, then timer 1 is reserved for use by the profiler. Timers 2 and 3 are free for use by applications.

Other

Other on-chip devices (SPI, USB, CAN, HDLC etc.) are not touched by the STR7XX variant HAL and unless used by the platform HAL are free for use for applications.

Name

HAL Port — Implementation Details

Overview

This section covers any remaining items of note related to the STR7xx variant support, not covered in previous sections.

LEDs

If a platform port has support for the display of values on LEDs, that support is standardised to be accessible from C with the following function:

```
#include <cyg/infra/hal_diag.h>
extern void hal_diag_led(int leds);
```

Clock Control

The platform HAL must provide the input clock frequency (`CYGARC_HAL_STR7XX_INPUT_CLOCK`) in its CDL file.

STR7XX definitions

The STR7XX variant HAL port includes the header file `var_io.h` which provides useful register definitions used by eCos, that can also be freely used by applications. It includes only limited register definitions for subsystems unused by eCos.

It may be found in the `include/cyg/hal` directory relative to your configuration's install tree, or alternatively in the source repository at `hal/arm/str7xx/var/`*`VERSION`*`/include/var_io.h`. However it should be properly included by applications by using the following to allow for platform HALs to augment or override any relevant definitions:

```
#include <cyg/hal/hal_io.h>
```

Power Control

The kernel idle thread is scheduled to run when the system has no other tasks able to run. The idle thread can call a HAL supplied macro to place the chip into an appropriate power saving mode instead of just going around a busy loop. The STR7XX variant HAL defines the `HAL_IDLE_THREAD_ACTION` macro to use the STR7XX power control support to place the chip into `WFI` mode which will stop the processor clock, without disabling the on-chip peripherals. This state continues until an interrupt is received.

Further power saving can be achieved by reducing the system clock frequencies using `hal_str7xx_clocks_setup()` described above. This function only changes the clock frequencies; it may also be necessary to change the values of dividers in various peripherals to compensate. There are several routines supplied in the HAL to do this.

Calling `hal_str7xx_uart_reinit()` will cause all the UART baud rate dividers to be reset to match the current value of PCLK1. Note that there are limitations to the range of baud rates that can be set, PCLK1 must be at least 16 times the required rate. Also, the resolution of the baud rate divider may make certain baud rates less accurate at different PCLK1 frequencies.

Calling `hal_str7xx_watchdog_init(CYGNUM_DEVS_WATCHDOG_ARM_STR7XX_DESIRED_TIMEOUT_US)` will cause the watchdog to be reinitialized with a timeout based on the current value of PCLK2. The resolution of the prescaler and the size of the 16 bit counter may render certain watchdog timeouts unachievable at some clock rates.

Calling `HAL_CLOCK_INITIALIZE(CYGNUM_HAL_RTC_PERIOD)` will cause the main system timer to be reinitialized based on the current value of PCLK2.

Name

Power Management — Details

Synopsis

#include <cyg/hal/hal_io.h>

void **hal_str7xx_clocks_setup**(*index*);

void **hal_str7xx_set_clock_speed**(*index*);

cyg_uint32 **hal_str7xx_get_clock_speed**();

cyg_bool **hal_str7xx_mode_stop**();

void **hal_str7xx_mode_standby**();

cyg_uint32 **hal_str7xx_startup_mode**();

void **hal_str7xx_uart_setbaud**(*uart*, *baud*);

void **hal_str7xx_uart_reinit**();

void **hal_str7xx_i2c_init**(*bus*, *clock*);

void **hal_str7xx_i2c_reinit**();

void **hal_str7xx_watchdog_init**(*timeout*);

void **hal_str7xx_can_init**(*devno*, *clock*);

void **hal_str7xx_can_reinit**();

void **hal_str7xx_adc_init**(*rate*);

void **hal_str7xx_adc_reinit**();

void **hal_str7xx_rtc_init**();

void **hal_str7xx_rtc_alarm_set**(*secs*);

void **hal_str7xx_rtc_alarm_cancel**();

cyg_uint32 **hal_str7xx_rtc_counter**();

cyg_uint32 **hal_str7xx_rtc_counter_set**(*secs*);

Description

The STR7XX variant HAL provides support for managing the power consumption of the device. This consists of a collection of functions that may be used to adjust clock frequencies, system modes and other aspects of the device. These routines mainly comprise a "kit of parts" from which applications may construct their own power management policy. The reader is referred to the STR7XX hardware documentation for full details of clock and power management.

The main function is `hal_str7xx_clocks_setup(index)` which controls the frequencies of the main clocks: MCLK, which supplies the CPU and memories; PCLK1, which supplies APB1 including the I^{2}C, SPI and UARTs; and PCLK2, which supplies APB2 including IO ports, Timers, RTC etc. The single argument to this function is an index into a table of hal_str7xx_clock_params structures, which is defined by the platform HAL. Each entry in the table has the following structure:

```
typedef struct
{
    char        *name;                  // Name string
    cyg_uint8   clk2_divider;           // 1, 2
    cyg_uint8   pll1_multiplier;        // 12, 16, 20, 24
    cyg_uint8   pll1_divider;           // 1..7
    cyg_uint8   rclk_select;            // RCLK_SELECT_* below
    cyg_uint8   mclk_divider;           // 1, 2, 4, 8
    cyg_uint8   pclk1_divider;          // 1, 2, 4, 8
    cyg_uint8   pclk2_divider;          // 1, 2, 4, 8
} hal_str7xx_clock_params;

#define RCLK_SELECT_CLK2        0
#define RCLK_SELECT_CLK2_16     1
#define RCLK_SELECT_PLL1        2
#define RCLK_SELECT_AF          3

__externC cyg_uint32 hal_str7xx_clock_param_index;
```

Each entry in the table corresponds to a single configuration of the clock hardware. Some care must be taken in specifying these entries, the resulting clocks will depend on the system input clock and the various multipliers and dividers; many configurations will result in out of range or otherwise illegal clock frequencies. See the manuals for the STR7XX variant and the platform for details.

It is recommended that the *clk2_divider* field is always set to 2. This causes the ocillator input to be divided by 2 and provides a more stable input to the rest of the clock circuity. Also, some registers in the RCCU are only accessible if MCLK is equal to RCLK, so the *mclk_divider* field should always be 1. These restrictions may be relaxed in special circumstances.

The last entry in this table should be all zeros, to mark the end of the table. A typical table would appear as follows:

```
const hal_str7xx_clock_params hal_str7xx_clock_param_table[] =
{
    // name               clk/  pll1* pll1/ rclk source          mclk/ pclk1/ pclk2/
    {  "32KHz"          ,   2,     0,    0, RCLK_SELECT_AF     ,    1,     1,     1 },
    {  "500KHz"         ,   2,     0,    0, RCLK_SELECT_CLK2_16,    1,     1,     1 },
    {  "8MHz"           ,   2,     0,    0, RCLK_SELECT_CLK2   ,    1,     1,     1 },
    {  "24/6/3MHz"      ,   2,    12,    4, RCLK_SELECT_PLL1   ,    1,     4,     8 },
    {  "32/32MHz"       ,   2,    16,    4, RCLK_SELECT_PLL1   ,    1,     1,     1 },
    {  "40/10/5MHz"     ,   2,    20,    4, RCLK_SELECT_PLL1   ,    1,     4,     8 },
    {  "40/40MHz"       ,   2,    20,    4, RCLK_SELECT_PLL1   ,    1,     1,     1 },
    {  "48/12/6MHz"     ,   2,    12,    2, RCLK_SELECT_PLL1   ,    1,     4,     8 },
    {  "48/24/12MHz"    ,   2,    12,    2, RCLK_SELECT_PLL1   ,    1,     2,     4 },

    {  0                ,   0,     0,    0,                   0,    0,     0,     0 }
};

cyg_uint32 hal_str7xx_clock_param_index = 5;
```

The naming convention used above is that a single frequency implies that all 3 clocks (MCLK, PCLK1 and PCLK2) are set to the same value. Two frequencies mean that MCLK is set to the first and both PCLK1 and PCLK2 are set to the second. Three frequencies show the values for MCLK/PCLK1/PCLK2 in order.

The variable `hal_str7xx_clock_param_index` indicates the table entry of the parameter set that is currently set. This should be initialized by the platform HAL to the index of the default parameter set, which will be used during initialization. `hal_str7xx_clocks_setup()` also sets a number of other global variables with the clock rates resulting from the parameter set in use:

```
__externC cyg_uint32 hal_str7xx_pclk1;          // PCLK1 frequency in Hz
__externC cyg_uint32 hal_str7xx_pclk2;          // PCLK2 frequency in Hz
__externC cyg_uint32 hal_str7xx_mclk;           // MCLK  frequency in Hz
```

Functions to initialize baud rate generators or prescaler dividers for various devices are also present:

`hal_str7xx_uart_setbaud(uart, baud)` sets the baud rate generator of the given UART to the given baud rate, based on the current value of PCLK1. UARTs are numbered 0 to 3, corresponding to the UARTs available on the device and baud rate is given in Hz. `hal_str7xx_uart_reinit()` causes all UARTs baud rate generators to be reinitialized using the last baud rate setting and the current PCLK1 value. It is usually called after changing the system clocks.

`hal_str7xx_i2c_init(bus, clock)` initializes the clock divider of the given I^2C bus to the given value based on the current value of PCLK1. The bus numbers are either 0 or 1, and the clock rate is given in Hz. `hal_str7xx_i2c_reinit()` causes all I^2C busses clock dividers to be reinitialized using the last clock rate setting and the current PCLK1 value.

`hal_str7xx_watchdog_init(timeout)` initializes the watchdog timeout based on the current PCLK2 setting. The timeout is given in microseconds. Some care is needed in setting this value since the resolution of the prescaler and the width of the 16 bit counter mean that certain timeouts may not be achievable at different PCLK2 frequencies.

`hal_str7xx_can_init(devno, clock)` initializes the clock divider of the given CAN device to the given baud rate based on the current PCLK1 setting. This function sets the entire Bit Timing Register, including the bit segment lengths and the synchronization jump width as well as the clock divider. While this interface is designed to support multiple CAN devices, the current implementation only supports a single CAN bus. The return value from this function indicates whether the requested clock frequency can be supported: zero if it is, -1 if not. `hal_str7xx_can_reinit()` causes the bit timings for all CAN busses to be reinitialized based on the current value of PCLK1.

`hal_str7xx_adc_init( rate )` initializes the prescaler for the ADC device based on the current PCLK2 setting. The rate argument gives the sample rate for each channel in samples per second. All channels share the same sample rate and are sampled on a round-robin basis. Therefore the combined sample rate, and hence maximum interrupt rate, will be four times this frequency. `hal_str7xx_adc_reinit()` causes the rate to be reinitialized based on the current value of PCLK2.

`hal_str7xx_mode_stop()` puts the STR7XX into STOP mode. Aside from entering STOP mode, all this routine does is set the WKUP-INT bit in the XTI CTRL register so that any of the external interrupt lines may be used to restart the system from STOP. However, it does not configure or unmask these lines. Instead, they may be unmasked and configured using the standard interrupt control API (`cyg_interrupt_unmask()`, `cyg_interrupt_configure()` etc.) It is also possible to configure the RTC to wake the STR7XX from STOP mode. The value returned from this function indicates whether STOP mode was entered: `true` if it was, `false` if not. It is usually adequate to just retry in the case of failed entry.

`hal_str7xx_mode_standby()` puts the STR7XX into STANDBY mode. As with entering STOP mode, it is the responsibility of the caller to configure the external interrupt lines and RTC to bring the system out of STANDBY mode. Exit from STANDBY mode causes the STR7XX to reboot, so if this function returns, then an error has occurred during entry to STANDBY. It is usually adequate to just retry in this case.

Note

At the time of writing, it has not been possible, at least on the STR710-EVAL board, to test RTC wakeup from STANDBY mode. It is believed that this is due to a silicon bug in the version of the STR710 present on the board.

`hal_str7xx_startup_mode()` returns the reason for the last restart. It indicates whether the restart was as a result of one of the following events:

- `STARTUP_MODE_RESET`: Standard power-on reset.
- `STARTUP_MODE_WAKEUP`: External WAKEUP event.
- `STARTUP_MODE_LOW_VOLTAGE`: Low voltage detected.
- `STARTUP_MODE_RTC_ALARM`: RTC alarm.
- `STARTUP_MODE_WATCHDOG`: Watchdog expiry.
- `STARTUP_MODE_SOFTWARE`: Software reset.

`hal_str7xx_set_clock_speed(index)` is a wrapper function that reprograms all the clocks and baud rate generators. The argument is the same index into the platform HAL supplied parameter table as given to `hal_str7xx_clocks_setup()`. After calling that function it also calls `hal_str7xx_uart_reinit()`, `hal_str7xx_i2c_reinit()`, `hal_str7xx_can_reinit()`, `HAL_CLOCK_INITIALIZE()` and `hal_str7xx_watchdog_init()`. Unlike `hal_str7xx_clocks_setup()`, this function checks that the supplied index is valid, and returns `false` if it is not.

`hal_str7xx_get_clock_speed()` returns the parameter table index given to the last call to `hal_str7xx_clocks_setup()` or `hal_str7xx_set_clock_speed()`.

The HAL also contains functions to control the Real Time Clock, RTC. These are mainly oriented towards using the RTC to resume the system from STOP or STANDBY mode. Before making any other calls to the RTC routines, the application must call `hal_str7xx_rtc_init()` to initialize the device. Calling `hal_str7xx_rtc_alarm_set(secs)` sets the alarm to fire after the given number of seconds. After the alarm had fired, or to prevent it firing, call `hal_str7xx_rtc_alarm_cancel()`. Calling `hal_str7xx_rtc_counter()` returns the current value of the RTC counter, which counts seconds. Function `hal_str7xx_rtc_counter_set( secs )` sets the RTC counter to the given value. These last two function may be used by a wallclock driver to provide time and date functionality.

RedBoot Support

The STR7XX HAL installs a number of RedBoot commands to allow testing of the power management support.

speed [-l] [index]

This command reports and sets the clock speed of the STR7XX. Giving the command on its own, with no arguments, lists the available speed settings:

```
RedBoot> speed
   0        32KHz
   1       500KHz
   2         8MHz
   3    24/6/3MHz
   4   40/10/5MHz
*  5   48/12/6MHz
   6  48/24/12MHz
RedBoot>
```

The index numbers on the left are used as arguments to the **speed** command. The names on the right correspond to the clock parameter set names described before. The parameter set currently in force is indicated by an asterisk in the first column. If the `-l` option is given, then more details of the parameter sets are given, together with the current settings of MCLK, PCLK1 and PCLK2:

```
RedBoot> speed -l
MCLK: 48000000, PCLK1 12000000, PCLK2 6000000
C Ix         Name CLK/ PLL1* PLL1/    RCLK MCLK/ PCLK1/ PCLK2/
   0        32KHz   2     0     0       AF   1     1      1
   1       500KHz   2     0     0  CLK2/16   1     1      1
   2         8MHz   2     0     0     CLK2   1     1      1
   3    24/6/3MHz   2    12     4     PLL1   1     4      8
   4   40/10/5MHz   2    20     4     PLL1   1     4      8
*  5   48/12/6MHz   2    12     2     PLL1   1     4      8
   6  48/24/12MHz   2    24     2     PLL1   2     4      8
RedBoot>
```

Supplying the **speed** command with the index number of a parameter set will change the STR7XX to use that set of clock parameters:

```
RedBoot> speed 3
Set clock speed 3, please wait...
Now running at new speed
RedBoot> speed -l
MCLK: 32000000, PCLK1 32000000, PCLK2 32000000
C Ix         Name CLK/ PLL1* PLL1/    RCLK MCLK/ PCLK1/ PCLK2/
```

```
   0        32KHz   2     0     0        AF   1     1      1
   1       500KHz   2     0     0  CLK2/16   1     1      1
   2         8MHz   2     0     0     CLK2   1     1      1
   3    24/6/3MHz   2    12     4     PLL1   1     4      8
*  4     32/32MHz   2    16     4     PLL1   1     1      1
   5   40/10/5MHz   2    20     4     PLL1   1     4      8
   6     40/40MHz   2    20     4     PLL1   1     1      1
   7   48/12/6MHz   2    12     2     PLL1   1     4      8
   8  48/24/12MHz   2    12     2     PLL1   1     2      4
RedBoot>
```

Note that setting too low a speed may result in RedBoot not being able to program the serial baud rate generator to maintain the current speed.

stop

The **stop** command puts the STR7XX into STOP mode. The RTC will be programmed to wake the system up after 5 seconds. The device may also be woken up before that timeout using the WAKEUP line, if it is connected to a switch (as it is on the STR710-EVAL board).

standby

The **standby** command puts the STR7XX into STANDBY mode. The RTC will be programmed to wake the system up after 5 seconds. The device may also be woken up before that timeout using the WAKEUP line, if it is connected to a switch (as it is on the STR710-EVAL board).

Chapter 249. ST STR710-EVAL Board HAL

Name

eCos Support for the ST STR710-EVAL Board — Overview

Description

The ST STR710-EVAL Board is fitted with an STR710FZ2T6 microcontroller to provide a development environment for all STR71X microcontrollers. The board is fitted with 4MiB of external RAM and 4MiB of external FLASH memory. The board has two 9-pin RS-232 serial interfaces connected to two of the STR710 on-chip UARTs, LEDs, and LCD display, and a JTAG debug interface. Refer to the board documentation and the STR7XX documentation for full details.

For typical eCos development, a RedBoot image is programmed into the external FLASH and the switches set so that the board will boot this image from reset. RedBoot provides gdb stub functionality so it is then possible to download and debug stand-alone and eCos applications via the gdb debugger using either UART0 or UART1.

This documentation describes platform-specific elements of the STR710-EVAL Board support within eCos. The STR7XX documentation covers various topics including HAL support common to STR7XX variants, and on-chip device support. This document complements the STR7XX documentation.

Supported Hardware

The STR7XX has two on-chip memory regions. A RAM region of 64KiB is present at 0x20000000, which is mapped to 0x00000000 after booting. A FLASH region is present at 0x40000000 and is comprised of 64KiB, 128KiB or 256KiB of program memory plus 16KiB of higher durability data flash. The STR710FZ2T6 on the STR710-EVAL board is equipped with 256KiB.

On-board memory consists of 4MiB of SRAM mapped to 0x62000000 and 4MiB of FLASH mapped to 0x60000000. During booting the external flash is mapped to 0x00000000 but will be replaced with the internal flash for normal execution.

The STR7XX variant HAL includes support for the four on-chip serial devices which are documented in the variant HAL. Only two of these serial devices are connected to external connectors on the board, so only these are normally usable.

The STR710-EVAL board port includes support for the on-chip watchdog and interrupt controller. This support is documented in the STR7XX variant HAL.

Tools

The STR710-EVAL Board port is intended to work with GNU tools configured for an arm-eabi target. Thumb mode is supported. The original port was done using arm-elf-gcc version 3.4.4, arm-elf-gdb version 6.3, and binutils version 2.16.

Name

Setup — Preparing the STR710-EVAL Board for eCos Development

Overview

In a typical development environment, the STR710-EVAL board boots from external flash into RedBoot. eCos applications are configured for RAM startup and then downloaded and run on the board via the debugger **arm-eabi-gdb**.

The following RedBoot configurations are supported:

Configuration	Description	Use	File
ROM	RedBoot running from external FLASH	`redboot_ROM.ecm`	`redboot_ROM.bin`
ROM_INT	RedBoot running from internal FLASH	`redboot_ROM_INT.ecm`	`redboot_ROM_INT.bin`
ROM_INT_EXTRAM	RedBoot running from internal FLASH, using external RAM for DATA, BSS and heap	`redboot_ROM_INT_EXTRAM.ecm`	`redboot_ROM_INT_EXTRAM.bin`
RAM	RedBoot running from external RAM	`redboot_RAM.ecm`	`redboot_RAM.bin`
JTAG	RedBoot running from external RAM, loaded via JTAG	`redboot_JTAG.ecm`	`redboot_JTAG.bin`

For serial communications, all versions run with 8 bits, no parity, and 1 stop bit at 38400 baud. This baud rate can be changed via the configuration option `CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL_BAUD` and rebuilding the stubrom.

Under normal circumstances, RedBoot runs in-place from the external Flash. The RAM version is provided to allow for updating the resident RedBoot image in Flash. The JTAG version is only used if loading RedBoot into RAM via a JTAG debugger or ICE. It is similar to the RAM version, but loads at a lower address within RAM, and so can be used to load eCos applications, as if it is the normal resident boot monitor. The ELF format image of this JTAG version of RedBoot can also be loaded and executed from GDB using the Abatron BDI2000 bdiGDB support, to allow it to be debugged. The ROM_INT version does not contain support for the flash filesystem or flash config since it uses only internal RAM, which is not large enough for the necessary data structures. The ROM_INT_EXTRAM version uses external RAM instead of internal RAM and is therefore able to contain a full implementation of the flash filesystem and configuration.

Initial Installation

Two mechanisms are described below to program RedBoot into the external Flash. Both of them require a JTAG device. In the following documentation it is assumed that the Abatron BDI2000 is being used. For a different JTAG device, equivalent operations will need to be performed.

Preparing the Abatron BDI2000 JTAG debugger

The BDI2000 must first be configured to allow communication with your local network, and configured with the parameters for interfacing with the target board. The following steps should be followed:

1. Prepare a PC to act as a host PC and start a TFTP server on it.
2. Connect the Abatron BDI2000 JTAG debugger via both serial and ethernet to the host PC and power it on. Use the serial cable supplied with the BDI2000.

3. Install the Abatron BDI2000 bdiGDB support software on the host PC.

4. Locate the file `bdi2000.str710eval.cfg` within the eCos platform HAL package in the source repository. This will be in the directory packages/hal/arm/str710eval/*VERSION*/misc relative to the root of your eCos installation.

5. Place the bdi2000.str710eval.cfg file in a location on the PC accessible to the TFTP server. Later you will configure the BDI2000 to load this file via TFTP as its configuration file.

6. Install and configure the Abatron BDI2000 in line with the bdiGDB instruction manual. Configure the BDI2000 to use the `bdi2000.str710eval.cfg` configuration file at the appropriate point of this process.

Preparing the STR710-EVAL board for programming

Follow the steps in this section in order to allow communication between the board and the host PC, and between the board and the JTAG device.

1. First you must connect a null modem DB9 serial cable between the Serial Debug Port on the board and a serial port on the host computer.

2. Start a suitable terminal emulator on the host computer such as **minicom** or HyperTerminal. Set the communication parameters to 38400 baud, 8 data bits, no parity bit and 1 stop bit with no flow control.

3. Connect the board to the BDI2000 using a 20-pin ARM/Xscale cable from the JTAG interface connector to the Target A port on the BDI2000.

4. Locate switches SW13, SW14 and SW15 on the board, they are in a bank of 3 next to the processor. Set all these switches to the 2-3 position, this is the position away from the processor. In due course this will ensure that the board boots RedBoot from the external Flash device.

5. Power up the STR710-EVAL board.

6. Connect to the BDI2000's CLI interface via TCP/IP on the standard telnet port 23. The **telnet** application is suitable for this. You should see usage information followed by the prompt:

   ```
   Core#0>
   ```

7. Confirm correct connection with the BDI2000 with the **reset** command as follows:

   ```
   Core#0> reset
   - TARGET: processing reset request
   - TARGET: BDI asserts TRST and RESET
   - TARGET: BDI removes TRST
   - TARGET: Bypass check 0x00000001 => 0xFFFFFFFF
   - TARGET: JTAG exists check failed
   - TARGET: Remove RESET and try again
   - TARGET: BDI waits for RESET inactive
   - TARGET: Bypass check 0x00000001 => 0x00000001
   - TARGET: JTAG exists check passed
   - Core#0: ID code is 0x3F0F0F0F
   - TARGET: All ICEBreaker access checks passed
   - TARGET: BDI removes RESET
   - TARGET: BDI waits for RESET inactive
   - TARGET: resetting target passed
   - TARGET: processing target startup ....
   - TARGET: processing target startup passed
   Core#0>
   ```

8. Locate the `redboot_ROM.bin` image within the `loaders` subdirectory of the base of the eCos installation.

9. Copy the `redboot_ROM.bin` file into a location on the host computer accessible to its TFTP server.

Method 1 - Using the BDI2000 to directly program RedBoot into Flash

As previously mentioned, there are two methods of programming a RedBoot image into the parallel NOR Flash. This method uses the built-in capabilities of the BDI2000.

This is a three stage process. The relevant Flash blocks must first be unlocked, then erased, and finally programmed. This can be accomplished with the following steps:

1. Connect to the BDI2000 telnet port as before.

2. Cut and paste the following commands into the BDI2000 telnet session. They are used to unlock the relevant Flash blocks that will contain RedBoot.

```
Core#0>unlock 0x60000000 0x2000 8
Unlocking flash at 0x60000000
Unlocking flash at 0x60002000
Unlocking flash at 0x60004000
Unlocking flash at 0x60006000
Unlocking flash at 0x60008000
Unlocking flash at 0x6000a000
Unlocking flash at 0x6000c000
Unlocking flash at 0x6000e000
Unlocking flash passed
Core#0>unlock 0x60010000 0x10000 2
Unlocking flash at 0x60010000
Unlocking flash at 0x60020000
Unlocking flash passed
```

3. Erase the 8 initial 8Kbyte sized Flash blocks, and the following 2 64Kbyte Flash blocks with the following command (the blocks to erase are defined in the .cfg file):

```
Core#0>erase
Erasing flash at 0x60000000
Erasing flash at 0x60002000
Erasing flash at 0x60004000
Erasing flash at 0x60006000Information System (FIS):
Erasing flash at 0x60008000
Erasing flash at 0x6000a000
Erasing flash at 0x6000c000
Erasing flash at 0x6000e000
Erasing flash at 0x60010000
Erasing flash at 0x60020000
Erasing flash passed
Core#0>
```

4. Program the RedBoot image into Flash with the following command, replacing *`/RBPATH`* with the location of the redboot_ROM.bin file relative to the TFTP server root directory:

```
Core#0>prog 0x60000000 /RBPATH/redboot_ROM.bin bin
Programming /RBPATH/redboot_ROM.bin , please wait ....
Programming flash passed
Core#0>
```

   This operation can take some time.

The RedBoot installation is now complete. This can be tested by powering off the board, disconnecting the JTAG, and then powering on the board again. The RedBoot banner should be visible on the serial port. RedBoot's Flash configuration can be initialized using the same procedure as required in Method 2 below.

If it proves necessary to re-install RedBoot, this may be achieved by repeating the above process. Alternatively, a new image may be downloaded and programmed into flash more directly using RedBoot's own commands. See the RedBoot documentation for details.

Method 2 - Program RedBoot into External Flash with RAM RedBoot

With this approach, the BDI2000 is used to load a RedBoot image into RAM, which can then in turn be used to load and program a ROM RedBoot image into Flash.

There are three stages, firstly loading the RAM RedBoot image, then initializing RedBoot's Flash configuration, and finally loading and programming the ROM RedBoot.

Loading a RAM RedBoot

1. Locate the `redboot_JTAG.bin` image within the `loaders` subdirectory of the base of the eCos installation.

2. Copy the `redboot_JTAG.bin` file into a location on the host computer accessible to its TFTP server.

3. With the BDI2000 telnet interface, execute the following command, replacing */RBPATH* with the location of the redboot_JTAG.bin file relative to the TFTP server root directory:

```
Core#0>load 0x62000000 /RBPATH/redboot_JTAG.bin bin
Loading /RBPATH/redboot_JTAG.bin , please wait ....
Loading program file passed
Core#0>
```

4. Run the loaded RAM RedBoot:

```
Core#0>go 0x62000000
Core#0>
```

The terminal emulator connected to the serial debug port should now have displayed the RedBoot banner and prompt similar to the following:

```
+RedBoot(tm) bootstrap and debug environment [JTAG]
Non-certified release, version UNKNOWN - built 17:29:34, Apr 10 2006

Platform: ST STR710-EVAL Board (ARM7TDMI)
Copyright (C) 2000, 2001, 2002, 2003, 2004 Free Software Foundation, Inc.
Copyright (C) 2003, 2004, 2005, 2006 eCosCentric Limited

RAM: 0x62000000-0x62400000, [0x62019a20-0x623ed000] available
     0x00000000-0x00010000, [0x00000000-0x00010000] available
FLASH: 0x40000000-0x40c40000, 4 x 0x2000 blocks, 1 x 0x8000 blocks, 3 x 0x10000 blocks, 8 x 0x10000 blocks, 2 x 0x2000 blocks
FLASH: 0x60000000-0x603fffff, 8 x 0x2000 blocks, 63 x 0x10000 blocks
RedBoot>
```

Note

It is also possible to use the RAM startup version of RedBoot and the redboot_RAM.bin file instead of redboot_JTAG.bin above. If so, then the address to the **load** command must be 0x62020000, as must be the address to the **go** command.

RedBoot Flash Configuration

The following steps describe how to initialize RedBoot's Flash configuration. This must be performed when using a JTAG or RAM RedBoot to program Flash, but is also applicable to initial configuration of a ROM RedBoot loaded using Method 1.

1. Use the following command to initialize RedBoot's Flash Information System (FIS):

```
RedBoot> fis init
About to initialize [format] FLASH image system - continue (y/n)? y
*** Initialize FLASH Image System
... Unlocking from 0x603f0000-0x603fffff: .
... Erase from 0x603f0000-0x603fffff: .
... Program from 0x623f0000-0x62400000 to 0x603f0000: .
... Locking from 0x603f0000-0x603fffff: .
RedBoot>
```

2. Now configure RedBoot's Flash configuration with the command:

```
RedBoot> fconfig -i
Initialize non-volatile configuration - continue (y/n)? y
Run script at boot: false
Console baud rate: 38400
Update RedBoot non-volatile configuration - continue (y/n)? y
... Unlocking from 0x603f0000-0x603fffff: .
... Erase from 0x603f0000-0x603fffff: .
... Program from 0x623f0000-0x62400000 to 0x603f0000: .
... Locking from 0x603f0000-0x603fffff: .
RedBoot>
```

Loading and programming the ROM RedBoot

This section describes the steps required to load the ROM RedBoot via the serial line and program it into Flash.

1. Load the RedBoot ROM binary image from the serial line. Use the following command:

```
RedBoot> load -r -m y -b %{freememlo}
CRaw file loaded 0x62038c00-0x6204eacb, assumed entry at 0x62038c00
xyzModem - CRC mode, 704(SOH)/0(STX)/0(CAN) packets, 3 retries
RedBoot>
```

2. Finally install the loaded image into Flash:

```
RedBoot> fis create RedBoot
An image named 'RedBoot' exists - continue (y/n)? y
... Unlocking from 0x60000000-0x6001ffff: .........
... Erase from 0x60000000-0x6001ffff: .........
... Program from 0x62038c00-0x6204eacc to 0x60000000: .........
... Locking from 0x60000000-0x6001ffff: .........
... Unlocking from 0x603f0000-0x603fffff: .
... Erase from 0x603f0000-0x603fffff: .
... Program from 0x623f0000-0x62400000 to 0x603f0000: .
... Locking from 0x603f0000-0x603fffff: .
RedBoot>
```

It is also possible to use the **fis write** command to write the image into Flash, but if so, the relevant Flash blocks must also be explicitly unlocked with the command:

```
RedBoot> fis unlock -f 0x60000000 -l 0x20000
```

The RedBoot installation is now complete. This can be tested by powering off the board, disconnecting the JTAG, and then powering on the board again. Output similar to the following should be seen on the serial port.

```
+RedBoot(tm) bootstrap and debug environment [ROM]
Non-certified release, version UNKNOWN - built 10:22:53, Apr 10 2006

Platform: ST STR710-EVAL Board (ARM7TDMI)
Copyright (C) 2000, 2001, 2002, 2003, 2004 Free Software Foundation, Inc.
Copyright (C) 2003, 2004, 2005, 2006 eCosCentric Limited

RAM: 0x62000000-0x62400000, [0x62004910-0x623ed000] available
     0x00000000-0x00010000, [0x00000000-0x00010000] available
FLASH: 0x40000000-0x40c40000, 4 x 0x2000 blocks, 1 x 0x8000 blocks, 3 x 0x10000 blocks, 8 x 0x10000 blocks, 2 x 0x2000 blocks
FLASH: 0x60000000-0x603fffff, 8 x 0x2000 blocks, 63 x 0x10000 blocks
RedBoot>
```

If it proves necessary to re-install RedBoot, this may be achieved by repeating the above process. Alternatively, a new image may be downloaded and programmed into flash more directly using RedBoot's own commands. See the RedBoot documentation for details.

Method 3 - Program RedBoot into Internal Flash with RAM RedBoot

This is a variant of Method 2, which puts RedBoot into the internal flash of the STR710, rather than the external M28W320CB flash device.

The first part of this is exactly the same as Method 2: load a RAM version of RedBoot as described in the Loading a RAM RedBoot section and configure is as described in the RedBoot Flash Configuration section. Now load the `redboot_ROM_INT.bin` binary image from the serial line as follows:

```
RedBoot> load -r -m y -b %{freememlo}
CRaw file loaded 0x6201a800-0x6202c733, assumed entry at 0x6201a800
xyzModem - CRC mode, 577(SOH)/0(STX)/0(CAN) packets, 4 retries
RedBoot>
```

Install the loaded image into Flash:

```
RedBoot> fis write -f 0x40000000 -b %{freememlo} -l 0x20000
* CAUTION * about to program FLASH
            at 0x40000000..0x4001ffff from 0x6201a800 - continue (y/n)? y
... Erase from 0x40000000-0x4001ffff: ......
... Program from 0x6201a800-0x6203a800 to 0x40000000: ......
RedBoot>
```

The RedBoot installation is now complete. This can be tested by powering off the board, disconnecting the JTAG, and then powering on the board again. Output similar to the following should be seen on the serial port.

```
+
RedBoot(tm) bootstrap and debug environment [ROM]
Non-certified release, version UNKNOWN - built 18:41:12, Jun  1 2006

Platform: ST STR710-EVAL Board (ARM7TDMI)
Copyright (C) 2000, 2001, 2002, 2003, 2004 Free Software Foundation, Inc.
Copyright (C) 2003, 2004, 2005, 2006 eCosCentric Limited

RAM: 0x00000000-0x00010000, [0x000048b0-0x00010000] available
FLASH: 0x40000000-0x40c40000, 4 x 0x2000 blocks, 1 x 0x8000 blocks, 3 x 0x10000 blocks, 8 x 0x10000 blocks, 2 x 0x2000 blocks
FLASH: 0x60000000-0x603fffff, 8 x 0x2000 blocks, 63 x 0x10000 blocks
RedBoot>
```

Rebuilding RedBoot

Should it prove necessary to rebuild a RedBoot binary, this is done most conveniently at the command line. The steps needed to rebuild the the ROM version of RedBoot are:

```
$ mkdir redboot_str710eval_rom
$ cd redboot_str710eval_rom
$ ecosconfig new str710eval redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/arm/str7xx/str710eval/VERSION/misc/redboot_ROM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

At the end of the build the `install/bin` subdirectory should contain the file `redboot.bin`.

The other versions of RedBoot - ROM, RAM or JTAG - may be similarly built by choosing the appropriate alternative .ecm file.

Name

Configuration — Platform-specific Configuration Options

Overview

The STR710-EVAL Board platform HAL package is loaded automatically when eCos is configured for an `str710eval` target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The STR710-EVAL Board platform HAL package supports four separate startup types:

RAM

> This is the startup type which is normally used during application development. The board has RedBoot programmed into external Flash at location 0x60000000 and uses external RAM at location 0x62000000. arm-eabi-gdb is then used to load a RAM startup application into memory from 0x62020000 and debug it. It is assumed that the hardware has already been initialized by RedBoot. By default the application will use the eCos virtual vectors mechanism to obtain certain services from RedBoot, including diagnostic output.

ROM

> This startup type can be used for finished applications which will be programmed into external ROM at location 0x60000000. The application will be self-contained with no dependencies on services provided by other software. The program expects to boot from reset with ROM mapped at location zero. It will then transfer control to the 0x20000000 region and map internal RAM to location zero. eCos startup code will perform all necessary hardware initialization.

ROM_INT

> This startup type can be used for finished applications which will be programmed into internal Flash at location 0x40000000. The application will be self-contained with no dependencies on services provided by other software. The program expects to boot from reset with ROM mapped at location zero. It will then transfer control to the 0x40000000 region and map internal RAM to location zero. eCos startup code will perform all necessary hardware initialization.
>
> This startup is enabled by setting the `CYGHWR_HAL_STR7XX_FLASH_INTERNAL` option, when `CYG_HAL_STARTUP` is set to ROM.

ROM_INT_EXTRAM

> This startup type can be used for finished applications which will be programmed into internal Flash at location 0x40000000. The application will be self-contained with no dependencies on services provided by other software. The program expects to boot from reset with ROM mapped at location zero. It will then transfer control to the 0x40000000 region and map internal RAM to location zero. eCos startup code will perform all necessary hardware initialization. The application will also use external RAM at 0x62000000 for its DATA, BSS and heap, rather than the internal SRAM.
>
> This startup is enabled by setting the `CYGHWR_HAL_STR7XX_FLASH_INTERNAL` option, when `CYG_HAL_STARTUP` is set to ROM and additionally setting `CYGHWR_HAL_ARM_STR710EVAL_EXT_RAM`.

JTAG

> This is the startup type used to build applications that are loaded via a JTAG interface. The application will be self-contained with no dependencies on services provided by other software. The program expects to be loaded from 0x62000000 and entered at that address. It will then map internal RAM to location zero. eCos startup code will perform all necessary hardware initialization and the system will be in a condition suitable for loading and running RAM applications.

The Stubrom and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option CYGSEM_HAL_ROM_MONITOR should be set. Typically this option is set only when building RedBoot.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option CYGSEM_HAL_USE_ROM_MONITOR should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then serial port 0 will be claimed for HAL diagnostics.

Flash Drivers

The STR710-EVAL board contains a 4Mbyte ST M28W320CB parallel Flash device. The CYGPKG_DEVS_FLASH_STRATA_V2 package contains all the code necessary to support this part and the platform HAL package contains definitions that customize the driver to the STR710-EVAL board. This driver is not active until the generic Flash support package, CYGPKG_IO_FLASH, is included in the configuration.

A driver is also present for the internal Flash. The package CYGPKG_DEVS_FLASH_STR7XX contains all the code necessary to support this memory and the platform HAL package contains definitions that customize the driver to the STR710-EVAL board. This driver is not active until the generic Flash support package, CYGPKG_IO_FLASH, is included in the configuration.

Watchdog Driver

The STR710-EVAL board use the STR7XX's internal watchdog support. The CYGPKG_DEVICES_WATCHDOG_ARM_STR7XX package contains all the code necessary to support this device. Within that package the CYGNUM_DEVS_WATCHDOG_ARM_STR7XX_DESIRED_TIMEOUT_US configuration option controls the watchdog timeout, and by default will force a reset of the board upon timeout. This driver is not active until the generic watchdog device support package, CYGPKG_IO_WATCHDOG, is included in the configuration.

UART Serial Driver

The STR710-EVAL boards use the STR7XX's internal UART serial support. As well as the polled HAL diagnostic interface, there is also a CYGPKG_IO_SERIAL_ARM_STR7XX package which contains all the code necessary to support interrupt-driven operation with greater functionality. All four UARTs can be supported by this driver, although only UARTs 0 and 1 are actually routed to external connectors. Note that it is not recommended to enable this driver on the port used for HAL diagnostic I/O. This driver is not active until the CYGPKG_IO_SERIAL_DEVICES configuration option within the generic serial driver support package CYGPKG_IO_SERIAL is enabled in the configuration.

The STR7XX UARTs only provide the minimal TX and RX data lines; hardware flow control using RTS/CTS is not supported. The eCos device drivers have been extended to permit the use of a pair of GPIO lines to be use as flow control lines. These must be defined by the platform for each UART using the following CDL options:

CYGHWR_HAL_ARM_STR7XX_UARTX_RTS

> This value encodes which PIO line will be used as the RTS line for UARTX. The value of this option is an invocation of the macro UART_PIO(), which takes three arguments: the first is the PIO port number and the second is the bit number in that port for the PIO line. The third argument gives the polarity of the line, 0 if it is active low, 1 if it is active high.

CYGHWR_HAL_ARM_STR7XX_UARTX_CTS

> This value encodes which PIO line will be used as the CTS line for UARTX. UART_PIO() takes three arguments: the first is the PIO port number and the second is the bit number in that port for the PIO line. The third argument gives the polarity of the line, 0 if it is active low, 1 if it is active high.

CYGHWR_HAL_ARM_STR7XX_UARTX_CTS_INT

> This must be the name of the interrupt vector, from var_intr.h, that corresponds to the PIO bit selected for CTS. It is essential that the PIO bit selected be capable of generating an interrupt, so only those that have an XTI interrupt vector can be used. The polarity of the CTS line will decide whether this interrupt occurs on a rising or falling edge.

I^{2}C Support

Support for the two I²C® busses is provided by the variant HAL (`CYGPKG_HAL_ARM_STR7XX`). The STR710-EVAL board carries an ST M24C08 I^{2}C serial EEPROM connected to bus 0.

The M24C08 is an 8Kibit device, 1024 bytes. This memory is addressed by using a single byte for the least significant 8 bits of the address plus 2 bits from the device's I^{2}C address. Within eCos, this EEPROM is presented as four separate I^{2}C devices at addresses 0xA8, 0xAA, 0xAC, 0xAE. These are instantiated as four I^{2}C device objects, named `cyg_i2c_str710eval_m24c08_0`, `cyg_i2c_str710eval_m24c08_1`, `cyg_i2c_str710eval_m24c08_2` and `cyg_i2c_str710eval_m24c08_3` respectively.

A test application for use with the EEPROM is provided within the `tests` subdirectory of the `CYGPKG_HAL_ARM_STR7XX_STR710EVAL` package. This test communicates with the I^{2}C EEPROM on the board to perform read and write operations using I^{2}C. Since it overwrites the contents of the EEPROM, this test is not built by default. It may be built by enabling the configuration option `CYGBLD_HAL_ARM_STR7XX_STR710EVAL_TEST_M24C08`.

System Clock

By default, the system clock interrupts once every 10ms, corresponding to a 100Hz clock. This can be changed by the configuration option `CYGNUM_HAL_RTC_DENOMINATOR` which corresponds to the clock frequency. Other clock-related settings are recalculated automatically if the denominator is changed. The description of the clock-related options may be found in the STR7XX variant HAL documentation.

Compiler Flags

The platform HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos.

The option "-mcpu=arm7tdmi" should be set for all compilations for this platform.

There are two flags that are used if Thumb mode is to be supported:

`-mthumb`

The arm-eabi-gcc compiler will compile C and C++ files into the Thumb instruction set when this option is used.

`-mthumb-interwork`

This option allows programs to be created that mix ARM and Thumb instruction sets. Without this option, some memory can be saved. This option should be used if -mthumb is used.

Name

JTAG support — Usage

Use of JTAG for debugging

JTAG can be used to single-step and debug loaded applications, or even applications resident in ROM, including RedBoot.

Debugging of ROM applications is only possible if using hardware breakpoints. The ARM7TDMI core of the STR7XX only supports two such hardware breakpoints, so they should be used sparingly. If using a GDB front-end such as Eclipse, check it has not set unnecessary extra breakpoints. Some JTAG devices give the option of whether to set hardware or software breakpoints by default. Be sure to configure your device appropriately.

Abatron BDI2000 notes

On the Abatron BDI2000, the `bdi2000.str710eval.cfg` file should be used to setup and configure the hardware to an appropriate state to load programs.

The `bdi2000.str710eval.cfg` file also contains an option to define whether hardware or software breakpoints are used by default, using the `BREAKMODE` directive in the `[TARGET]` section. Edit this file if you wish to use software break points, and remember to use the **boot** command on the BDI2000 command line interface to make the changes take effect.

On the BDI2000, debugging can be performed either via the telnet interface or using **arm-eabi-gdb** and the bdiGDB interface. In the case of the latter, **arm-eabi-gdb** needs to connect to TCP port 2001 on the BDI2000's IP address. For example:

```
(gdb) target remote 111.222.333.444:2001
```

By default when the BDI2000 is powered up, the target will always run the initialization section of the `bdi2000.str710eval.cfg` file, and halt the target. This behaviour is repeated with the **reset** command.

If the board is reset when in '**reset**' mode (either with the '**reset halt**' or '**reset**' commands, or by pressing the reset button) and the '**go**' command is then given, then the board will boot as normal. If a ROM RedBoot is resident in Flash, it will be run.

It is also possible for the target to always run, without initialization, after the reset button has been pressed. This mode is selected with the **reset run** command. This conveniently allows the target to be connected to the JTAG debugger, and be able to reset it with the reset button, without being required to always type '**go**' every time. Thereafter, invoking the **reset** command will repeat the previous reset style. Also in this mode, exceptions will be handled by board software, rather than causing the JTAG debugger to halt the CPU.

Suitably configured JTAG applications can be loaded either via GDB, or directly via the telnet CLI. For example:

```
Core#0>load 0x62000000 test.bin bin
Loading /test.bin , please wait ....
Loading program file passed
Core#0>go 0x620000000
```

Consult the BDI2000 documentation for information on other formats.

Configuration of JTAG applications

JTAG applications can be loaded directly into RAM without requiring a ROM monitor. This loading can be done directly through the JTAG device, or where supported by the JTAG device, through GDB.

In order to configure the application to support this mode, some configuration settings are required. Firstly `CYGSEM_HAL_USE_ROM_MONITOR` must be disabled. Secondly the `CYGDBG_HAL_DIAG_TO_DEBUG_CHAN` option should be enabled in order to prevent HAL diagnostic output being encoded into GDB ($O) packets. Both of these settings are made automatically if the JTAG startup type is selected.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the STR710-EVAL Board hardware, and should be read in conjunction with that specification. The STR710-EVAL Board platform HAL package complements the ARM architectural HAL and the STR7XX variant HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset the HAL will initialize or reinitialize many of the on-chip peripherals. There is an exception for RAM startup applications which depend on a ROM monitor for certain services.

For ROM and JTAG startup, the HAL will perform additional initialization, programming the various internal registers including the PLL, peripheral clocks, GPIO pins and memory mapping control to map internal RAM 0x0. The details of the early hardware startup may be found in the header `cyg/hal/hal_platform_setup.h`.

Linker Scripts and Memory Maps

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

External RAM

This is located at address 0x62000000 of the memory space, and is 4MiB long. For ROM applications, all of RAM is available for use. For RAM startup applications, RAM below 0x62020000 is reserved for RedBoot and the remainder is available for the application.

External ROM

This is located at address 0x60000000 of the memory space. If switches SW13, SW14 and SW15 are all set to 2-3 this region will be mapped to 0x00000000 at reset. This region is 4MiB in size. RedBoot is normally programmed into this memory and the rest managed by the FIS flash file system.

Internal RAM

This is located at address 0x20000000 of the memory space, and is 64KiB in size. Normally this RAM area will be mapped to location 0x00000000 after bootstrap. The CPU vector table and the eCos VSR table occupy the bottom 64 bytes. The virtual vector table starts at 0x00000050 and extends to 0x00000150. The remainder of internal RAM is available for use by applications.

Internal ROM

This is located at address 0x40000000 of the memory space. If switches SW13 and SW14 are set to 1-2 and SW15 to 2-3 this region will be mapped to 0x00000000 at reset. This region is 256KiB in size. Applications may be configured to run from this memory by setting the `CYGHWR_HAL_STR7XX_FLASH_INTERNAL` option. This memory is not managed by RedBoot's FIS system, but it can be written using the **fis write** command and erased using the **fis erase** command.

on-chip peripherals

These are accessible at locations 0xC0000000 and 0xE0000000 upwards, depending on which APB bus they are on. Descriptions of the contents can be found in the STR7XX User Manual.

Chapter 250. Atmel AT91RM9200 Processor Support

Name

eCos Support for the Atmel AT91RM9200 Processor — Overview

Description

This document covers the configuration and usage of the Hardware Abstraction Layer (HAL) for the Atmel AT91RM9200 processor. It is expected to be read in conjunction with platform HAL-specific documentation, as well as the eCos HAL specification. This processor HAL package complements the ARM architectural HAL, ARM9 variant HAL and the platform HAL. It provides functionality common to AT91RM9200-based board implementations.

This support is found in the eCos package located at `packages/hal/arm/arm9/at91rm9200` within the eCos source repository.

The AT91RM9200 processor HAL package is loaded automatically when eCos is configured for an AT91RM9200-based platform. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Supported Hardware

Supported features of the Atmel AT91RM9200 processor within this processor HAL package include:

- AT91RM9200-specific hardware definitions
- Interrupt controller
- Timer counters
- Serial UARTs
- MultiMedia Card Interface (MCI)
- Two-Wire Interface (TWI)
- Power saving

Support for the on-chip ethernet, interrupt-driven serial, SPI, watchdog and wallclock (RTC) features of the AT91RM9200 are also present and can be found in separate packages, outside of this processor HAL.

The watchdog hardware is also used within this HAL to perform software reset.

Name

AT91RM9200 hardware definitions — Details on obtaining hardware definitions for AT91RM9200

Register definitions

The file `<cyg/hal/at91rm9200.h>` can be included from application and eCos package sources to provide definitions related to AT91RM9200 subsystems. These include register definitions for the interrupt controller, power management controller, clock generator, memory controller, external bus interface, GPIO, USART, MCI, TWI (I²C®), Ethernet, timer counter, RTC, and SPI subsystems.

Initialization helper macros

The file `<cyg/hal/at91rm9200_init.inc>` contains definitions of helper macros which may be used by AT91RM9200 platform HALs in order to initialise common AT91RM9200 subsystems without excessive duplication between the platform HALs. Typically this file will be included by the `hal_platform_setup.h` header in the platform HAL, in turn included from the architectural HAL file `vectors.S`.

This file is solely intended to be used by platform HALs. At the same time, it is only present to assist initialization, and platform HALs are not obliged to use it if their startup requirements vary.

Name

AT91RM9200 interrupt controller — Advanced Interrupt Controller definitions and usage

Interrupt controller definitions

The file `<cyg/hal/var_ints.h>` (located at `hal/arm/arm9/at91rm9200/`*`VERSION`*`/include/var_ints.h` in the eCos source repository) contains interrupt vector number definitions for use with the eCos kernel and driver interrupt APIs:

```
#define CYGNUM_HAL_INTERRUPT_FIQ     0 // Advanced Interrupt Controller (FIQ)
#define CYGNUM_HAL_INTERRUPT_SYSTEM  1 // System Peripheral (debug unit, system timer)
#define CYGNUM_HAL_INTERRUPT_PIOA    2 // Parallel IO Controller A
#define CYGNUM_HAL_INTERRUPT_PIOB    3 // Parallel IO Controller B
#define CYGNUM_HAL_INTERRUPT_PIOC    4 // Parallel IO Controller C
#define CYGNUM_HAL_INTERRUPT_PIOD    5 // Parallel IO Controller D
#define CYGNUM_HAL_INTERRUPT_US0     6 // USART 0
#define CYGNUM_HAL_INTERRUPT_US1     7 // USART 1
#define CYGNUM_HAL_INTERRUPT_US2     8 // USART 2
#define CYGNUM_HAL_INTERRUPT_US3     9 // USART 3
#define CYGNUM_HAL_INTERRUPT_MCI    10 // Multimedia Card Interface
#define CYGNUM_HAL_INTERRUPT_UDP    11 // USB Device Port
#define CYGNUM_HAL_INTERRUPT_TWI    12 // Two-Wire Interface
#define CYGNUM_HAL_INTERRUPT_SPI    13 // Serial Peripheral Interface
#define CYGNUM_HAL_INTERRUPT_SSC0   14 // Serial Synchronous Controller 0
#define CYGNUM_HAL_INTERRUPT_SSC1   15 // Serial Synchronous Controller 1
#define CYGNUM_HAL_INTERRUPT_SSC2   16 // Serial Synchronous Controller 2
#define CYGNUM_HAL_INTERRUPT_TC0    17 // Timer Counter 0
#define CYGNUM_HAL_INTERRUPT_TC1    18 // Timer Counter 1
#define CYGNUM_HAL_INTERRUPT_TC2    19 // Timer Counter 2
#define CYGNUM_HAL_INTERRUPT_TC3    20 // Timer Counter 3
#define CYGNUM_HAL_INTERRUPT_TC4    21 // Timer Counter 4
#define CYGNUM_HAL_INTERRUPT_TC5    22 // Timer Counter 5
#define CYGNUM_HAL_INTERRUPT_UHP    23 // USB Host port
#define CYGNUM_HAL_INTERRUPT_EMAC   24 // Ethernet MAC
#define CYGNUM_HAL_INTERRUPT_IRQ0   25 // Advanced Interrupt Controller (IRQ0)
#define CYGNUM_HAL_INTERRUPT_IRQ1   26 // Advanced Interrupt Controller (IRQ1)
#define CYGNUM_HAL_INTERRUPT_IRQ2   27 // Advanced Interrupt Controller (IRQ2)
#define CYGNUM_HAL_INTERRUPT_IRQ3   28 // Advanced Interrupt Controller (IRQ3)
#define CYGNUM_HAL_INTERRUPT_IRQ4   29 // Advanced Interrupt Controller (IRQ4)
#define CYGNUM_HAL_INTERRUPT_IRQ5   30 // Advanced Interrupt Controller (IRQ5)
#define CYGNUM_HAL_INTERRUPT_IRQ6   31 // Advanced Interrupt Controller (IRQ6)

// The following interrupts are derived from the SYSTEM interrupt
#define CYGNUM_HAL_INTERRUPT_DEBUG  32 // Debug unit
#define CYGNUM_HAL_INTERRUPT_PMC    33 // Power Management Controller
#define CYGNUM_HAL_INTERRUPT_RTCH   34 // Real Time Clock
#define CYGNUM_HAL_INTERRUPT_PIT    35 // System Timer Period Interval Timer
#define CYGNUM_HAL_INTERRUPT_WDOVF  36 // System Timer Watchdog Overflow
#define CYGNUM_HAL_INTERRUPT_RTTINC 37 // System Timer Real-Time Timer Increment
#define CYGNUM_HAL_INTERRUPT_ALM    38 // System Timer Alarm
```

As indicated above, further decoding is performed on the SYSTEM interrupt to identify the cause more specifically. Note that as a result, placing an interrupt handler on the SYSTEM interrupt will not work as expected. Conversely, masking a decoded derivative of the SYSTEM interrupt will not work as this would mask other SYSTEM interrupts, but masking the SYSTEM interrupt itself will work. On the other hand, unmasking a decoded SYSTEM interrupt *will* unmask the SYSTEM interrupt as a whole, thus unmasking interrupts for the other units on this shared interrupt.

The list of interrupt vectors may be augmented on a per-platform basis. Consult the platform HAL documentation for your platform for whether this is the case.

Interrupt controller functions

The source file src/at91rm9200_misc.c within this package provides most of the support functions to manipulate the interrupt controller. The hal_IRQ_handler queries the IRQ status register to determine the interrupt cause. Functions hal_interrupt_mask and hal_interrupt_unmask enable or disable interrupts within the interrupt controller.

Interrupts are configured in the hal_interrupt_configure function, where the level and up arguments are interpreted as follows:

level	up	interrupt on
0	0	Falling Edge
0	1	Rising Edge
1	0	Low Level
1	1	High Level

To fit into the eCos interrupt model, interrupts essentially must be acknowledged immediately once decoded, and as a result, the hal_interrupt_acknowledge function is empty.

The hal_interrupt_set_level is used to set the priority level of the supplied interrupt within the Advanced Interrupt Controller.

Note that in all the above, it is not recommended to call the described functions directly. Instead either the HAL macros (HAL_INTERRUPT_MASK et al) or preferably the kernel or driver APIs should be used to control interrupts.

Using the Advanced Interrupt Controller for VSRs

The AT91RM9200 HAL has been designed to exploit benefits of the on-chip Advanced Interrupt Controller (AIC) on the AT91RM9200. Support has been included for exploiting its ability to provide hardware vectoring for VSR interrupt handlers.

This support is dependent on definitions that may only be provided by the platform HAL and therefore is only enabled if the platform HAL package implements the CYGINT_HAL_AT91RM9200_AIC_VSR CDL interface. The necessary definitions are available to all platform HALs which use the facilities of the at91rm9200_init.inc header file.

The interrupt decoding path has been optimised by allowing the AIC to be interrogated for the interrupt handler VSR to use. These vectored interrupts are by default still configured to point to the default ARM architecture HAL IRQ and FIQ VSRs. However applications may set their own VSRs to override this default behaviour to allow optimised interrupt handling.

The VSR vector numbers to use when overriding are defined as follows:

```
#define CYGNUM_HAL_VECTOR_FIQ     7 // FIQ
#define CYGNUM_HAL_VECTOR_SYSTEM  8 // System Peripheral (debug unit, system timer)
#define CYGNUM_HAL_VECTOR_PIOA    9 // Parallel IO Controller A
#define CYGNUM_HAL_VECTOR_PIOB   10 // Parallel IO Controller B
#define CYGNUM_HAL_VECTOR_PIOC   11 // Parallel IO Controller C
#define CYGNUM_HAL_VECTOR_PIOD   12 // Parallel IO Controller D
#define CYGNUM_HAL_VECTOR_US0    13 // USART 0
#define CYGNUM_HAL_VECTOR_US1    14 // USART 1
#define CYGNUM_HAL_VECTOR_US2    15 // USART 2
#define CYGNUM_HAL_VECTOR_US3    16 // USART 3
#define CYGNUM_HAL_VECTOR_MCI    17 // Multimedia Card Interface
#define CYGNUM_HAL_VECTOR_UDP    18 // USB Device Port
#define CYGNUM_HAL_VECTOR_TWI    19 // Two-Wire Interface
#define CYGNUM_HAL_VECTOR_SPI    20 // Serial Peripheral Interface
#define CYGNUM_HAL_VECTOR_SSC0   21 // Serial Synchronous Controller 0
#define CYGNUM_HAL_VECTOR_SSC1   22 // Serial Synchronous Controller 1
#define CYGNUM_HAL_VECTOR_SSC2   23 // Serial Synchronous Controller 2
```

```
#define CYGNUM_HAL_VECTOR_TC0    24 // Timer Counter 0
#define CYGNUM_HAL_VECTOR_TC1    25 // Timer Counter 1
#define CYGNUM_HAL_VECTOR_TC2    26 // Timer Counter 2
#define CYGNUM_HAL_VECTOR_TC3    27 // Timer Counter 3
#define CYGNUM_HAL_VECTOR_TC4    28 // Timer Counter 4
#define CYGNUM_HAL_VECTOR_TC5    29 // Timer Counter 5
#define CYGNUM_HAL_VECTOR_UHP    30 // USB Host port
#define CYGNUM_HAL_VECTOR_EMAC   31 // Ethernet MAC
#define CYGNUM_HAL_VECTOR_IRQ0   32 // Advanced Interrupt Controller (IRQ0)
#define CYGNUM_HAL_VECTOR_IRQ1   33 // Advanced Interrupt Controller (IRQ1)
#define CYGNUM_HAL_VECTOR_IRQ2   34 // Advanced Interrupt Controller (IRQ2)
#define CYGNUM_HAL_VECTOR_IRQ3   35 // Advanced Interrupt Controller (IRQ3)
#define CYGNUM_HAL_VECTOR_IRQ4   36 // Advanced Interrupt Controller (IRQ4)
#define CYGNUM_HAL_VECTOR_IRQ5   37 // Advanced Interrupt Controller (IRQ5)
#define CYGNUM_HAL_VECTOR_IRQ6   38 // Advanced Interrupt Controller (IRQ6)
```

Consult the kernel and generic HAL documentation for more information on VSRs and how to set them.

Interrupt handling within standalone applications

For non-eCos standalone applications running under RedBoot, it is possible to install an interrupt handler into the interrupt vector table manually. Memory mappings are platform-dependent and so the platform documentation should be consulted, but in general the address of the interrupt table can be determined by analyzing RedBoot's symbol table, and searching for the address of the symbol name `hal_interrupt_handlers`. Table slots correspond to the interrupt numbers above. Pointers inserted in this table should be pointers to a C/C++ function with the following prototype:

```
extern unsigned int isr( unsigned int vector, unsigned int data );
```

For non-eCos applications run from RedBoot, the return value can be ignored. The `vector` argument will also be the interrupt vector number. The `data` argument is extracted from a corresponding table named `hal_interrupt_data` which immediately follows the interrupt vector table. It is still the responsibility of the application to enable and configure the interrupt source appropriately if needed.

Name

Timer counters — Use of on-chip timer counters

Timer counter 0

The eCos kernel system clock is implemented using Timer Counter 0. By default, the system clock interrupts once every 10ms, corresponding to a 100Hz clock. This can be changed by the configuration option `CYGNUM_HAL_RTC_DENOMINATOR` which corresponds to the clock frequency. Other clock-related settings are recalculated automatically if the denominator is changed. If the desired frequency cannot be expressed accurately solely with changes to `CYGNUM_HAL_RTC_DENOMINATOR`, then the configuration option `CYGNUM_HAL_RTC_NUMERATOR` may also be adjusted, and again clock-related settings will automatically be recalculated.

Timer Counter 0 is also used to implement the HAL microsecond delay function, `HAL_DELAY_US`. This is used by some device drivers, and in non-kernel configurations such as with RedBoot where this timer is needed for loading program images via X/Y-modem protocols and debugging via TCP/IP. Standalone applications which require RedBoot services, such as debugging, should avoid use of this timer.

Timer-based profiling support

Timer-based profiling support is implemented using timer counter 1 (TC1). If the gprof package, `CYGPKG_PROFILE_GPROF`, is included in the configuration, then TC1 is reserved for use by the profiler.

Name

Serial UARTs — Configuration and implementation details of serial UART support

Overview

Support is included in this processor HAL package for the AT91RM9200's on-chip debug unit UART and four serial USART serial devices.

There are two forms of support: HAL diagnostic I/O; and a fully interrupt-driven serial driver. Unless otherwise specified in the platform HAL documentation, for all serial ports the default settings are 115200,8,N,1 with no flow control.

HAL diagnostic I/O

This first form is polled mode HAL diagnostic output, intended primarily for use during debug and development. Operations are usually performed with global interrupts disabled, and thus this mode is not usually suitable for deployed systems. This can operate on any port, according to the configuration settings.

There are several configuration options usually found within a platform HAL which affect the use of this support in the AT91RM9200 processor HAL. The CDL option `CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL` selects the serial port channel to use as the console at startup time. This will be the channel that receives output from, for example, `diag_printf()`. The CDL option `CYGNUM_HAL_VIRTUAL_VECTOR_DEBUG_CHANNEL` selects the serial port channel to use for GDB communication by default. Note that when using RedBoot, these options are usually inactive as it is RedBoot that decides which channels are used. Applications may override RedBoot's selections by enabling the `CYGSEM_HAL_VIRTUAL_VECTOR_CLAIM_COMMS` CDL option in the HAL. Baud rates for each channel are set with the `CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL_BAUD` and `CYGNUM_HAL_VIRTUAL_VECTOR_DEBUG_CHANNEL_BAUD` options.

Interrupt-driven serial driver

The second form of support is an interrupt-driven serial driver, which is integrated into the eCos standard serial I/O infrastructure (`CYGPKG_IO_SERIAL`). This support can be enabled on any port.

Note that it is not recommended to share this driver when using the HAL diagnostic I/O on the same port. If the driver is shared with the GDB debugging port, it will prevent ctrl-c operation when debugging.

This driver is contained in the `CYGPKG_IO_SERIAL_ARM_AT91` package. That driver package should also be consulted for documentation and configuration options. The driver is not active until the `CYGPKG_IO_SERIAL_DEVICES` configuration option within the generic serial driver support package `CYGPKG_IO_SERIAL` is enabled in the configuration.

Note that unlike the USART devices, the serial debug port does not support modem control signals such as those used for hardware signals. In addition, USART devices for a particular platform may also not have these control signals brought out to the physical serial port.

Name

Multimedia Card Interface (MCI) driver — Using MMC/SD cards with block drivers and filesystems

Overview

The MultiMedia Card Interface (MCI) driver in the AT91RM9200 processor HAL allows use of MultiMedia Cards (MMC cards) and Secure Digital (SD) flash storage cards within eCos, exported as block devices. This makes them suitable for use as the underlying devices for filesystems such as FAT.

Configuration

This driver provides the necessary support for the generic MMC/SD bus layer within the `CYGPKG_DEVS_DISK_MMC` package to export a disk block device. The disk block device is only available if the generic disk I/O layer found in the package `CYGPKG_IO_DISK` is included in the configuration.

The block device may then be used as the device layer for a filesystem such as FAT. Example devices are "`/dev/mmc0/1`" to refer to the first partition on the card, or "`/dev/mmc0/0`" to address the whole device including potentially the partition table at the start.

The driver may be forcibly disabled within this processor HAL package with the configuration option `CYGPKG_HAL_ARM_ARM9_AT91RM9200_MCI`.

If the driver is enabled, there are only two AT91RM9200 specific options:

`CYGIMP_HAL_ARM_ARM9_AT91RM9200_MCI_INTMODE`
: This indicates that the driver should operate in interrupt-driven mode if possible. This is enabled by default if the eCos kernel is enabled. Note though that if the driver finds that global interrupts are off when running, then it will fall back to polled mode even if this option is enabled. This allows for use of the MCI driver in an initialisation context.

`CYGNUM_HAL_ARM_ARM9_AT91RM9200_MCI_POWERSAVE_DIVIDER`
: The AT91RM9200 MCI peripheral allows the MCI clock to be divided down if told to enter power saving mode. This option specifies the divider to use. The driver itself does not implement any power saving - it is up to the application to enable power saving in the MCI control register if it is required.

Usage notes

MMC/SD cards may only be used in a MMC/SD card slot, and not a dataflash slot. The driver will detect the appropriate card sizes. Hotswapping of cards is supported by the driver, and in the case of eCosPro, the FAT filesystem. Although any cards removed before explicit unmounting or a `sync()` call to flush filesystem buffers will likely result in a corrupted filesystem on the removed card.

The MMC/SD bus layer will parse partition tables, although it can be configured to allow handling of cards with no partition table.

Name

Two-Wire Interface (TWI) driver — Configuration and implementation details of TWI (I²C®) driver

Overview

The AT91RM9200 processor HAL contains a driver for the Two-Wire Interface (TWI) controller on the AT91RM9200. This type of bus is also known as I²C®. The API for this may be found within the `CYGPKG_IO_I2C` package.

I²C®/TWI driver configuration

The I²C® driver uses the AT91RM9200's internal Two-Wire Interface (TWI) support. This is controlled within the AT91RM9200 processor HAL (`CYGPKG_HAL_AT91RM9200`). The `CYGPKG_HAL_AT91RM9200_TWI` CDL component controls whether the TWI driver is enabled. Within that component, there are two sub-options:

- `CYGNUM_HAL_AT91RM9200_TWI_CLOCK` sets the speed of the TWI bus clock in Hz. This is usually 100kHz, but can be set up to 400kHz if the devices on the bus support this speed, also known as fast mode. However other values below 400kHz can also be chosen, subject to the accuracy of the clock waveform generation parameters.

- The second option within the `CYGPKG_HAL_AT91RM9200_TWI` component is `CYGNUM_HAL_AT91RM9200_TWI_CKDIV`. This is the clock divider used when configuring the TWI_CWGR register. Consult the AT91RM9200 datasheet description of the TWI_CWGR register for the formula used to determine the clock frequency. Increasing the divider will decrease the accuracy in practice of the generated I²C bus clock compared to `CYGNUM_HAL_AT91RM9200_TWI_CLOCK`. But the divider must also be sufficiently low that the relevant factors do not overflow valid values for CHDIV/CLDIV in TWI_CWGR. Note that when the AT91RM9200 is using a 60MHz MCK, then for 100kHz operation, a value for this option of 1 is most appropriate. For 400kHz, a value for this option of 0 is most appropriate. The default value of this CDL is an appropriate value for CKDIV assuming a 60MHz MCK and a TWI clock between 29kHz and 400kHz.

  To be specific, the CLDIV/CHDIV fields of the TWI_CWGR are considered equal. The value of, for example, CLDIV, can be expressed as:

$$CLDIV = \frac{f_{MCK} - 6 f_{TWI}}{2^{CKDIV+1} \cdot f_{TWI}}$$

To use the I²C/TWI driver, the generic I²C driver package `CYGPKG_IO_I2C` must be used. Documentation for its API may be found elsewhere.

Usage notes

This driver only operates in interrupt mode. It does not operate in polled mode, and thus does not operate when interrupts are disabled. It cannot therefore be used in an initialization context, before the eCos kernel thread scheduler starts. And it cannot be used with RedBoot.

Due to the characteristics of the AT91RM9200's operation, it is not possible to provide support for repeated starts with the I²C package API. Similarly indicating a NACK when performing a receive is equivalent to also sending a STOP.

A test application for use with the Aardvark I²C/SPI Activity Board is provided within the `tests` subdirectory of the `CYGPKG_HAL_AT91RM9200` package. This test communicates with the I²C EEPROM on the board to perform read and write operations using I²C. This test is not built by default. It may be built by enabling the configuration option `CYGBLD_HAL_ARM_ARM9_AT91RM9200_TEST_TWI_AT24C02A` within the AT91RM9200 processor HAL.

Name

Power saving support — Extensions for saving power

Overview

There is support in the AT91RM9200 processor HAL for a simple power saving mechanism. This is provided by two functions:

```
#include <cyg/hal/hal_intr.h>

__externC void cyg_hal_at91rm9200_powersave_init( cyg_uint32 ip_addr );

__externC void cyg_hal_at91rm9200_powerdown( void );
```

The powersaving system is initialized by calling `cyg_hal_at91rm9200_powersave_init()`. The argument should be the IP address of this machine in network order. This can usually be fetched from the bootp data for an interface after completion of the call to `init_all_network_interfaces()`. e.g. `eth0_bootp_data.bp_ciaddr.s_addr`.

A call to `cyg_hal_at91rm9200_powerdown()` will put the machine into a low power mode. This will involve switching to a slower system clock speed, disabling all peripherals except those that are defined to cause the system to wake up and return from this function.

Configuration

The exact behaviour of the power saving system is controlled by the following configuration options:

CYGPKG_HAL_ARM_ARM9_AT91RM9200_POWERSAVE

> This option controls the overall inclusion of the power saving system.
>
> Default value: on

CYGSEM_HAL_ARM_ARM9_AT91RM9200_POWERSAVE_POLL_ETHERNET

> This option enables polling of the ethernet interface for relevant ARP packets and unicast IP packets. It is necessary for the CPU to run at a higher CPU speed for this option to work.
>
> Default value: off

CYGSEM_HAL_ARM_ARM9_AT91RM9200_POWERSAVE_IDLE

> If this option is set, the CPU will go into idle mode, which will cause it to halt until an interrupt is delivered.
>
> Default value: off

CYGVAR_HAL_ARM_ARM9_AT91RM9200_POWERSAVE_ACTIVE_DEVICES

> This option defines the devices that are to be kept running during power down mode. An interrupt from one of these devices is usually the only way of bringing the system out of idle mode. The value of this option is a bit mask with bits set for each device that is to be kept active. The bits correspond to the peripheral identifiers described in the AT91RM9200 documentation.
>
> Default value: 0x00000000

CYGSEM_HAL_ARM_ARM9_AT91RM9200_POWERSAVE_POLL_GPIO

> This option control whether the power saving system will poll GPIO pins during power saving. For this to work the CPU cannot be put into idle mode.

Default value: on

CYGVAR_HAL_ARM_ARM9_AT91RM9200_POWERSAVE_PIO_HI

This is an array of bitmasks of the bits in the PIO PDSR registers. Within the array, index 0 corresponds to PIOA, index 1 to PIOB and so on. For each set bit in these masks, if the value is seen to be 1, then the low power mode will be terminated.

Default value: 0, 0, 0, 0

CYGVAR_HAL_ARM_ARM9_AT91RM9200_POWERSAVE_PIO_LO

This is an array of bitmasks of the bits in the PIO PDSR registers. Within the array, index 0 corresponds to PIOA, index 1 to PIOB and so on. For each set bit in these masks, if the value is seen to be 0, then the low power mode will be terminated.

Default value: 0, 0, 0, 0

CYGVAR_HAL_ARM_ARM9_AT91RM9200_POWERSAVE_PIO_CHANGE

This is an array of bitmasks of the bits in the PIO PDSR registers. Within the array, index 0 corresponds to PIOA, index 1 to PIOB and so on. For each set bit in these masks, if the value is seen to change between successive polls, then the low power mode will be terminated.

Default value: 0, 0, 0, 0

CYGBLD_HAL_ARM_ARM9_AT91RM9200_TEST_POWERSAVE

This option controls whether a simple test is built to exercise power saving support. The test is not built by default as an external means is required to wake the processor up by one of the above configured mechanisms.

Default value: 0

Chapter 251. Atmel AT91RM9200 Development Kit/Evaluation Kit Board Support

Name

eCos Support for the Atmel AT91RM9200 Development Kit/Evaluation Kit — Overview

Description

This document covers the configuration and usage of eCos and RedBoot on the Atmel AT91RM9200 Development Kit and Atmel AT91RM9200 Evaluation Kit. The AT91RM9200 Evaluation Kit (EK board) contains the AT91RM9200 processor, 8Mbytes of SDRAM, 8Mbytes of parallel NOR flash memory, a Davicom DM9161A PHY, a SD/MMC/DataFlash socket, a DAC, external connections for two serial channels (one debug, one full), ethernet, USB host/device, graphics, and the various other peripherals supported by the AT91RM9200. The AT91RM9200 Development Kit (DK board) is similar but also comes with a 128Kbytes TWI (I2C) EEPROM, IrDA port, and 8Mbytes of SPI DataFlash, although only 2Mbytes of parallel NOR flash memory. eCos support for the many devices and peripherals on the boards and the AT91RM9200 is described below.

In this document, an EK board will be assumed for the purposes of examples and output.

For typical eCos development, a RedBoot image is programmed into the flash memory, and the board will boot into this image from reset. RedBoot provides gdb stub functionality so it is then possible to download and debug stand-alone and eCos applications via the gdb debugger. This can happen over either a serial line or over ethernet.

This documentation is expected to be read in conjunction with the AT91RM9200 processor HAL documentation and further device support and subsystems are described and documented there.

Supported Hardware

On the EK board, the parallel NOR flash memory consists of 8 blocks of 8Kbytes each, followed by 127 blocks of 64Kbytes each. In a typical setup, the first 192 Kbytes are reserved for the use of the ROMRAM RedBoot image. The topmost block is used to manage the flash and hold RedBoot **fconfig** values. The remaining blocks can be used by application code. There are 125 blocks available between 0x60030000 and 0x607EFFFF.

There is a serial driver `CYGPKG_IO_SERIAL_ARM_AT91` which supports both the Debug Unit and USART serial devices. The debug serial port at J10 and DTE port at J14 (connected to USART channel 1) can be used by RedBoot for communication with the host. If either of these devices is needed by the application, either directly or via the serial driver, then it cannot also be used for RedBoot communication. Another communication channel such as ethernet should be used instead. The serial driver package is loaded automatically when configuring for the AT91RM9200-EK or AT91RM9200-DK targets.

There is an ethernet driver `CYGPKG_DEVS_ETH_ARM_AT91RM9200` for the on-chip ethernet device. The platform HAL package is responsible for configuring this generic driver to the EK/DK hardware. This driver is also loaded automatically when configuring for the EK or DK targets.

There is a watchdog driver `CYGPKG_DEVICES_WATCHDOG_ARM_AT91RM9200`. This driver is also loaded automatically when configuring for the EK or DK targets.

There is a driver for the on-chip real-time clock (RTC) at `CYGPKG_DEVICES_WALLCLOCK_ARM_AT91`. This driver is also loaded automatically when configuring for the EK or DK targets.

The AT91RM9200 processor HAL contains a driver for the Two-Wire Interface (TWI) controller on the AT91RM9200. This type of bus is also known as I²C®. Further documentation may be found in the AT91RM9200 processor HAL documentation.

The AT91RM9200 processor HAL contains a driver for the MultiMedia Card Interface (MCI). This driver is loaded automatically when configuring for the EK or DK targets and allows use of MMC and Secure Digital (SD) flash storage cards within eCos, exported as block devices. Further documentation may be found in the AT91RM9200 processor HAL documentation.

There is a driver `CYGPKG_DEVS_SPI_ARM_ATMEL_AT91RM9200_KITS` to allow access to devices on the SPI bus. This driver provides information to the more general AT91 SPI driver (`CYGPKG_DEVS_SPI_ARM_AT91`) which in turn provides the underlying implementation for the SPI API layer in the `CYGPKG_IO_SPI` package. All these packages are automatically loaded when configuring for the EK or DK targets.

Furthermore, the platform HAL package contains support for SPI dataflash cards. The HAL support integrates with the `CYGPKG_DEVS_FLASH_ATMEL_DATAFLASH` package as well as the above SPI packages. That package is automatically loaded when configuring for the EK or DK targets. Dataflash media is then accessed as a Flash device, using the Flash I/O API within the `CYGPKG_IO_FLASH` package, if that package is loaded in the configuration.

In general, devices (Caches, PIO, UARTs, EMAC) are initialized only as far as is necessary for eCos to run. Other devices (RTC, I²C, SPI, MCI etc.) are not touched unless the appropriate driver is loaded, although in some cases, the HAL boot sequence will set up the appropriate PIO configuration.

Tools

The AT91RM9200-EK and AT91RM9200-DK support is intended to work with GNU tools configured for an arm-eabi target. The original port was undertaken using arm-elf-gcc version 3.4.4, arm-elf-gdb version 6.3, and binutils version 2.15.

Name

Setup — Preparing the AT91RM9200-EK and AT91RM9200-DK boards for eCos Development

Overview

In a typical development environment, the AT91RM9200-EK/DK boards boot from the parallel NOR Flash and run the RedBoot ROM monitor directly. eCos applications are configured for RAM startup and then downloaded and run on the board via the debugger **arm-eabi-gdb**. Preparing the board therefore usually involves programming a suitable RedBoot image into flash memory.

The following RedBoot configurations are supported:

Configuration	Description	Use	File
ROM	RedBoot running from ROM	`redboot_ROM.ecm`	`redboot_ROM.bin`
RAM	RedBoot running from RAM	`redboot_RAM.ecm`	`redboot_RAM.bin`
ROMRAM	RedBoot running from RAM, but contained in the board's flash boot sector	`redboot_ROMRAM.ecm`	`redboot_ROMRAM.bin`
JTAG	RedBoot running from RAM, loaded via JTAG	`redboot_JTAG.ecm`	`redboot_JTAG.bin`

For serial communications, all versions run with 8 bits, no parity, and 1 stop bit at 115200 baud. RedBoot also supports ethernet communication and flash management.

The ample provision of RAM memory on the board allows the ROMRAM version of RedBoot to be preferential to the standard ROM version which executes directly from Flash. Alternatively, if the ROM version is to be chosen, then the RAM version is provided to allow for updating the resident RedBoot image in Flash. The JTAG version is only used if loading RedBoot into RAM via a JTAG debugger or ICE. It is similar to the RAM version, but loads at a lower address within RAM, and so can be used to in turn load eCos applications, as if it is the normal resident boot monitor. The ELF format image of this JTAG version of RedBoot can also be loaded and executed from GDB using the Abatron BDI2000 bdiGDB support, to allow it to be debugged.

Initial Installation

The on-chip boot program on the AT91RM9200 is only capable of loading programs into 12Kbytes of on-chip SRAM and is therefore quite restrictive. Consequently two mechanisms are described below to program RedBoot into Flash. Both of them require a JTAG device. In the following documentation it is assumed that the Abatron BDI2000 is being used. For a different JTAG device, equivalent operations will need to be performed.

Preparing the Abatron BDI2000 JTAG debugger

The BDI2000 must first be configured to allow communication with your local network, and configured with the parameters for interfacing with the target board. The following steps should be followed:

1. Prepare a PC to act as a host PC and start a TFTP server on it.

2. Connect the Abatron BDI2000 JTAG debugger via both serial and ethernet to the host PC and power it on. Use the serial cable supplied with the BDI2000.

3. Install the Abatron BDI2000 bdiGDB support software on the host PC.

4. Locate the file `bdi2000.at91rm9200ek.cfg` within the eCos platform HAL package in the source repository. This will be in the directory `packages/hal/arm/arm9/atmel-at91rm9200-kits/`*`VERSION`*`/misc` relative to the root of your eCos installation.

5. Locate the file `reg920t.def` within the installation of the BDI2000 bdiGDB support software.

6. Place the bdi2000.at91rm9200ek.cfg in a location on the PC accessible to the TFTP server. Later you will configure the BDI2000 to load this file via TFTP as its configuration file.

7. Similarly place the file `reg920t.def` in a location accessible to the TFTP server.

8. Open `bdi2000.at91rm9200ek.cfg` in an editor such as emacs or notepad and if necessary adjust the path of the `reg920t.def` file in the `[REGS]` section to match its location relative to the TFTP server root.

9. Install and configure the Abatron BDI2000 in line with the bdiGDB instruction manual. Configure the BDI2000 to use the `bdi2000.at91rm9200ek.cfg` configuration file at the appropriate point of this process.

Preparing the AT91RM9200-EK/DK board for programming

Follow the steps in this section in order to allow communication between the board and the host PC, and between the board and the JTAG device.

1. First you must connect a null modem DB9 serial cable between the Serial Debug Port on the board and a serial port on the host computer.

2. Start a suitable terminal emulator on the host computer such as **minicom** or HyperTerminal. Set the communication parameters to 115200 baud, 8 data bits, no parity bit and 1 stop bit with no flow control.

3. Connect the board to your host PC's LAN with an Ethernet cable.

4. You should designate the board with a new Ethernet MAC address. The RedBoot binary image contains a default address, but each board requires its own unique address. It is advisable to mark each board with its programmed MAC address for future identification.

5. Connect the board to the BDI2000 using a 20-pin ARM/Xscale cable from the ICE interface connector to the Target A port on the BDI2000.

6. Locate jumper J15 on the board, which is by default set to `INT`. It should be reset to `EXT`. In due course this will ensure that the board boots RedBoot from the external parallel Flash device.

7. Power up the AT91RM9200-EK board. You should see the three ethernet LEDs illuminate.

8. Connect to the BDI2000's CLI interface via TCP/IP on the standard telnet port 23. The **telnet** application is suitable for this. You should see usage information followed by the prompt:

   ```
   Core#0>
   ```

9. Confirm correct connection with the BDI2000 with the **reset halt** command as follows:

   ```
   Core#0> reset halt
   - TARGET: processing reset request
   - TARGET: BDI asserts TRST and RESET
   - TARGET: BDI removes TRST
   - TARGET: Bypass check 0x000000001 => 0x00000001
   - TARGET: JTAG exists check passed
   - Core#0: ID code is 0x05B0203F
   - TARGET: All ICEBreaker access checks passed
   - TARGET: BDI removes RESET
   - TARGET: BDI waits for RESET inactive
   - TARGET: resetting target passed
   - TARGET: processing target startup ....
   - TARGET: processing target startup passed
   Core#0>
   ```

10.Locate the `redboot_ROMRAM.bin` image within the `loaders` subdirectory of the base of the eCos installation.

11.Copy the `redboot_ROMRAM.bin` file into a location on the host computer accessible to its TFTP server.

Method 1 - Using the BDI2000 to directly program RedBoot into Flash

As previously mentioned, there are two methods of programming a RedBoot image into the parallel NOR Flash. This method uses the built-in capabilities of the BDI2000.

This is a three stage process. The relevant Flash blocks must first be unlocked, then erased, and finally programmed. This can be accomplished with the following steps:

1. Connect to the BDI2000 telnet port as before.
2. Cut and paste the following commands into the BDI2000 telnet session. They are used to unlock the relevant Flash blocks that will contain RedBoot. The BDI2000 does have an **unlock** command, however this only works with Intel StrataFLASH and is therefore not suitable.

```
mmh 0x1000aaaa 0x00aa
mmh 0x10000000 0x0070
mmh 0x1000aaaa 0x00aa
mmh 0x10002000 0x0070
mmh 0x1000aaaa 0x00aa
mmh 0x10004000 0x0070
mmh 0x1000aaaa 0x00aa
mmh 0x10006000 0x0070
mmh 0x1000aaaa 0x00aa
mmh 0x10008000 0x0070
mmh 0x1000aaaa 0x00aa
mmh 0x1000a000 0x0070
mmh 0x1000aaaa 0x00aa
mmh 0x1000c000 0x0070
mmh 0x1000aaaa 0x00aa
mmh 0x1000e000 0x0070
mmh 0x1000aaaa 0x00aa
mmh 0x10010000 0x0070
mmh 0x1000aaaa 0x00aa
mmh 0x10020000 0x0070
```

3. Erase the 8 initial 8Kbyte sized Flash blocks, and the following 2 64Kbyte Flash blocks with the following commands:

```
Core#0>erase 0x10000000 0x2000 8
Erasing flash at 0x10000000
Erasing flash at 0x10002000
Erasing flash at 0x10004000
Erasing flash at 0x10006000
Erasing flash at 0x10008000
Erasing flash at 0x1000a000
Erasing flash at 0x1000c000
Erasing flash at 0x1000e000
Erasing flash passed
Core#0>erase 0x10010000 0x10000 2
Erasing flash at 0x10010000
Erasing flash at 0x10020000
Erasing flash passed
Core#0>
```

4. Program the RedBoot image into Flash with the following command, replacing `/RBPATH` with the location of the redboot_ROMRAM.bin file relative to the TFTP server root directory:

```
Core#0>prog 0x10000000 /RBPATH/redboot_ROMRAM.bin bin
Programming /RBPATH/redboot_ROMRAM.bin , please wait ....
Programming flash passed
Core#0>
```

This operation can take some time.

The RedBoot installation is now complete. This can be tested by powering off the board, disconnecting the JTAG, and then powering on the board again. The RedBoot banner should be visible on the serial port. RedBoot's Flash configuration can be initialized using the same procedure as required in Method 2 below.

If it proves necessary to re-install RedBoot, this may be achieved by repeating the above process. Alternatively, a new image may be downloaded and programmed into flash more directly using RedBoot's own commands. See the RedBoot documentation for details.

Method 2 - Program RedBoot into Flash with RAM RedBoot

With this approach, the BDI2000 is used to load a RAM RedBoot image, which can then in turn be used to load and program a ROMRAM RedBoot image into Flash.

There are three stages, firstly loading the RAM RedBoot image, then initializing RedBoot's Flash configuration, and finally loading and programming the ROMRAM RedBoot.

Loading a RAM RedBoot

1. Locate the `redboot_JTAG.bin` image within the `loaders` subdirectory of the base of the eCos installation.

2. Copy the `redboot_JTAG.bin` file into a location on the host computer accessible to its TFTP server.

3. With the BDI2000 telnet interface, execute the following command, replacing */RBPATH* with the location of the redboot_JTAG.bin file relative to the TFTP server root directory:

   ```
   Core#0>load 0x20008000 /RBPATH/redboot_JTAG.bin bin
   Loading /RBPATH/redboot_JTAG.bin , please wait ....
   Loading program file passed
   Core#0>
   ```

4. Run the loaded RAM RedBoot:

   ```
   Core#0>go 0x20008000
   Core#0>
   ```

The terminal emulator connected to the serial debug port should now have displayed the RedBoot banner and prompt similar to the following:

```
+**Warning** FLASH configuration checksum error or invalid key
Use 'fconfig -i' to [re]initialize database
PHY: Davicom DM9161A
AT91RM9200 ETH: Waiting for link to come up.
AT91RM9200 ETH: 100Mb/Full Duplex
... waiting for BOOTP information
Ethernet eth0: MAC address 00:23:31:37:00:1c
IP: 192.168.7.190/255.255.255.0, Gateway: 192.168.7.1
Default server: 192.168.7.11, DNS server IP: 192.168.7.11

RedBoot(tm) bootstrap and debug environment [RAM]
eCosCentric certified release, version v2_XX - built 18:51:18, Aug 25 2005

Platform: Atmel AT91RM9200-EK (ARM9)
Copyright (C) 2000, 2001, 2002, 2003, 2004 Free Software Foundation, Inc.
Copyright (C) 2003, 2004, 2005 eCosCentric Limited

RAM: 0x20000000-0x20800000, [0x2002f4e0-0x207ed000] available
FLASH: 0x60000000 - 0x607fffff 8 x 0x2000 blocks 127 x 0x10000 blocks
RedBoot>
```

In the above output, a local BOOTP/DHCP server was able to serve an address to the device.

Note

It is also possible to use the RAM startup version of RedBoot and the redboot_RAM.bin file instead of redboot_JTAG.bin above. If so, then the address to the **load** command must be 0x20040000, as must be the address to the **go** command.

RedBoot Flash configuration

The following steps describe how to initialize RedBoot's Flash configuration. This must be performed when using a RAM RedBoot to program Flash, but is also applicable to initial configuration of a ROMRAM RedBoot loaded using Method 1.

1. Use the following command to initialize RedBoot's Flash Information System (FIS):

```
RedBoot> fis init -f
About to initialize [format] FLASH image system - continue (y/n)? y
*** Initialize FLASH Image System
... Erase from 0x60030000-0x607effff: .........................................
... Unlocking from 0x607f0000-0x607fffff: .
... Erase from 0x607f0000-0x607fffff: .
... Program from 0x207f0000-0x20800000 to 0x607f0000: .
... Locking from 0x607f0000-0x607fffff: .
RedBoot>
```

2. Now configure RedBoot's Flash configuration with the command:

```
RedBoot> fconfig -i
```

Remember to substitute the appropriate MAC address for this board at the appropriate step. If a BOOTP/DHCP server is not available, then IP configuration may be set manually. The default server IP address can be set to a PC that will act as a TFTP host for future RedBoot load operations, or may be left unset. The following gives an example configuration:

```
RedBoot> fconfig -i
Initialize non-volatile configuration - continue (y/n)? y
Run script at boot: false
Use BOOTP for network configuration: false
Gateway IP address: 192.168.7.11
Local IP address: 192.168.7.222
Local IP address mask: 255.255.255.0
Default server IP address: 192.168.7.9
Console baud rate: 115200
DNS server IP address: 192.168.7.11
Network hardware address [MAC]: 0x00:0x23:0x31:0x37:0x00:0x4e
GDB connection port: 9000
Force console for special debug messages: false
Network debug at boot time: false
Default network device: at91rm9200_eth
Update RedBoot non-volatile configuration - continue (y/n)? y
... Unlocking from 0x607f0000-0x607fffff: .
... Erase from 0x607f0000-0x607fffff: .
... Program from 0x207f0000-0x20800000 to 0x607f0000: .
... Locking from 0x607f0000-0x607fffff: .
RedBoot>
```

Loading and programming the ROMRAM RedBoot

This section describes the steps required to load the ROMRAM RedBoot from the TFTP server and program it into Flash.

1. Load the RedBoot ROMRAM binary image from the TFTP server. Use the following command, replacing *`111.222.333.444`* with the TFTP server IP address (or domain name if a DNS server has been configured), and *`/RBPATH`* with the location of the `redboot_ROMRAM.bin` file relative to the TFTP server root directory:

```
RedBoot> load -r -b %{freememlo} -h 111.222.333.444 /RBPATH/redboot_ROMRAM.bin
Using default protocol (TFTP)
```

```
Raw file loaded 0x2002f800-0x2004e367, assumed entry at 0x2002f800
RedBoot>
```

2. Finally install the loaded image into Flash:

   ```
   RedBoot> fis create RedBoot
   An image named 'RedBoot' exists - continue (y/n)? y
   ... Unlocking from 0x60000000-0x6002ffff: ..........
   ... Erase from 0x60000000-0x6002ffff: ..........
   ... Program from 0x0002f800-0x0004e368 to 0x60000000: .........
   ... Locking from 0x60000000-0x6002ffff: ..........
   ... Unlocking from 0x607f0000-0x607fffff: .
   ... Erase from 0x607f0000-0x607fffff: .
   ... Program from 0x007f0000-0x00800000 to 0x607f0000: .
   ... Locking from 0x607f0000-0x607fffff: .
   RedBoot>
   ```

   It is also possible to use the **fis write** command to write the image into Flash, but if so, the relevant Flash blocks must also be explicitly unlocked with the command:

   ```
   RedBoot> fis unlock -f 0x60000000 -l 0x30000
   ```

The RedBoot installation is now complete. This can be tested by powering off the board, disconnecting the JTAG, and then powering on the board again. Output similar to the following should be seen on the serial port. Verify the IP settings are as expected.

```
+PHY: Davicom DM9161A
AT91RM9200 ETH: Waiting for link to come up.
AT91RM9200 ETH: 100Mb/Full Duplex
Ethernet eth0: MAC address 00:23:31:37:00:3d
IP: 192.168.7.222/255.255.255.0, Gateway: 192.168.7.1
Default server: 192.168.7.9, DNS server IP: 192.168.7.11

RedBoot(tm) bootstrap and debug environment [ROMRAM]
eCosCentric certified release, version v2_XX - built 18:55:20, Aug 25 2005

Platform: Atmel AT91RM9200-EK (ARM9)
Copyright (C) 2000, 2001, 2002, 2003, 2004 Free Software Foundation, Inc.
Copyright (C) 2003, 2004, 2005 eCosCentric Limited

RAM: 0x20000000-0x20800000, [0x20030470-0x207ed000] available
FLASH: 0x60000000 - 0x607fffff 8 x 0x2000 blocks 127 x 0x10000 blocks
RedBoot>
```

If it proves necessary to re-install RedBoot, this may be achieved by repeating the above process. Alternatively, a new image may be downloaded and programmed into flash more directly using RedBoot's own commands. See the RedBoot documentation for details.

Rebuilding RedBoot

Should it prove necessary to rebuild a RedBoot binary, this is done most conveniently at the command line. The steps needed to rebuild the ROMRAM version of RedBoot for the AT91RM9200-EK are:

```
$ mkdir redboot_at91rm9200ek_romram
$ cd redboot_at91rm9200ek_romram
$ ecosconfig new at91rm9200ek redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/arm/arm9/atmel-at91rm9200-kits/VERSION/misc/redboot_ROMRAM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

At the end of the build the `install/bin` subdirectory should contain the file `redboot.bin`.

The other versions of RedBoot - ROM, RAM or JTAG - may be similarly built by choosing the appropriate alternative .ecm file.

Name

Configuration — Platform-specific Configuration Options

Overview

The AT91RM9200-EK/DK platform HAL package is loaded automatically when eCos is configured for `at91rm9200ek` or `at91rm9200dk` targets. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The platform HAL package supports three separate startup types:

RAM
: This is the startup type which is normally used during application development. The board has RedBoot programmed into flash and boots into that initially. arm-eabi-gdb is then used to load a RAM startup application into memory and debug it. It is assumed that the hardware has already been initialized by RedBoot. By default the application will use the eCos virtual vectors mechanism to obtain certain services from RedBoot, including diagnostic output.

ROM
: This startup type can be used for finished applications which will be programmed into flash at physical address 0x10000000. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization.

ROM-RAM
: This startup type can be used for finished applications which will be programmed into flash at physical location 0x10000000. However, when it starts up, the application will first copy itself to RAM at virtual address 0x00000000 and then run from there. RAM is generally faster than flash memory, so the program will run more quickly than a ROM-startup application. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization.

RedBoot and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building RedBoot.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then the serial port will be claimed for HAL diagnostics.

Flash Driver

The AT91RM9200-EK board contains an 8Mbyte Atmel AT49BV6416 parallel Flash device. The `CYGPKG_DEVS_FLASH_AMD_AM29XXXXX_V2` package contains all the code necessary to support this part and the platform HAL package contains definitions that customize the driver to the AT91RM9200-EK board. This driver is not active until the generic Flash support package, `CYGPKG_IO_FLASH`, is included in the configuration.

Ethernet Driver

The AT91RM9200-EK/DK boards use the AT91RM9200's internal EMAC ethernet device attached to an external Davicom DM9161A PHY. The `CYGPKG_DEVS_ETH_ARM_AT91RM9200` package contains all the code necessary to support this device and the platform HAL package contains definitions that customize the driver to the AT91RM9200-EK/DK boards. This driver is not active until the generic Ethernet support package, `CYGPKG_IO_ETH_DRIVERS`, is included in the configuration.

RTC Driver

The AT91RM9200-EK/DK boards use the AT91RM9200's internal RTC support. The `CYGPKG_DEVICES_WALLCLOCK_ARM_AT91` package contains all the code necessary to support this device. This driver is not active until the generic wallclock device support package, `CYGPKG_IO_WALLCLOCK`, is included in the configuration.

Watchdog Driver

The AT91RM9200-EK/DK boards use the AT91RM9200's internal watchdog support. The `CYGPKG_DEVICES_WATCHDOG_ARM_AT91RM9200` package contains all the code necessary to support this device. Within that package the `CYGNUM_DEVS_WATCHDOG_ARM_AT91RM9200_DESIRED_TIMEOUT_MS` configuration option controls the watchdog timeout, and by default will force a reset of the board upon timeout. This driver is not active until the generic watchdog device support package, `CYGPKG_IO_WATCHDOG`, is included in the configuration.

USART Serial Driver

The AT91RM9200-EK/DK boards use the AT91RM9200's internal USART serial support as described in the AT91RM9200 processor HAL documentation. Two serial ports are available: the serial debug port which is mapped to virtual vector channel 0 in the HAL diagnostic driver or "`/dev/dbg`" in the interrupt-driven driver; and USART 1 which is mapped to virtual vector channel 1 and "`/dev/ser1`". Only USART 1 supports modem control signals such as those used for hardware flow control.

MCI Driver

As the AT91RM9200 MCI driver is part of the AT91RM9200 HAL, nothing is required to load it. Similarly the MMC/SD bus driver layer (`CYGPKG_DEVS_DISK_MMC`) is automatically included as part of the hardware-specific configuration for this target. All that is required to enable the support is to include the generic disk I/O infrastructure package (`CYGPKG_IO_DISK`), along with the intended filesystem, typically, the FAT filesystem (`CYGPKG_FS_FAT`) and any of its package dependencies (including `CYGPKG_LIBC_STRING` and `CYGPKG_LINUX_COMPAT` for FAT).

Various options can be used to control specific of the AT91RM9200 MCI driver. Consult the AT91RM9200 HAL documentation for information on its configuration.

On this target, the MMC/SD socket allows detection of when cards are inserted and removed. This may be used with the removeable media support and disk insertion/removal event notification system in the disk I/O package so that the application or other eCos subsystems are informed when cards are inserted and removed. This in turn allows use of the automounter contained within the File I/O package (`CYGPKG_FILEIO`) to mount and unmount cards automatically.

Caution

Remember that the ability to unmount cards after removal does not prevent those cards containing corrupt filesystems - instead cards should be preferably unmounted before removal, or at least have the filesystem's in-memory buffers flushed to the media using the `sync()` function).

The MMC/SD socket also allows detection of the write-protect (or "lock") switch present on SD cards. "Locked" cards will be detected and mounted read-only, and attempts to write to them will fail.

Compiler Flags

The platform HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos. There are just three flags specific to this port:

`-mcpu=arm9`
: The arm-eabi-gcc compiler supports many variants of the ARM architecture. A `-m` option should be used to select the specific variant in use, and with current tools `-mcpu=arm9` is the correct option for the ARM920T CPU in the AT91RM9200.

`-mthumb`
: The arm-eabi-gcc compiler will compile C and C++ files into the Thumb instruction set when this option is used. The best way to build eCos in Thumb mode is to enable the configuration option `CYGHWR_THUMB`.

`-mthumb-interwork`
: This option allows programs to be created that mix ARM and Thumb instruction sets. Without this option, some memory can be saved. This option should be used if -mthumb is used. The best way to build eCos with Thumb interworking is to enable the configuration option `CYGBLD_ARM_ENABLE_THUMB_INTERWORK`.

Name

JTAG support — Usage

Use of JTAG for debugging

JTAG can be used to single-step and debug loaded RAM applications, or even applications resident in ROM, including RedBoot.

Debugging of ROM applications is only possible if using hardware breakpoints. The ARM920T core of the AT91RM9200 only supports two such hardware breakpoints, and so they should be used sparingly. If using a GDB front-end such as Eclipse, check it has not set unnecessary extra breakpoints. Some JTAG devices give the option of whether to set hardware or software breakpoints by default. Be sure to configure your device appropriately.

Abatron BDI2000 notes

On the Abatron BDI2000, the `bdi2000.at91rm9200ek.cfg` file should be used to setup and configure the hardware to an appropriate state to load programs. This includes setting up the SDRAM controller.

The `bdi2000.at91rm9200ek.cfg` file also contains an option to define whether hardware or software breakpoints are used by default, using the `BREAKMODE` directive in the `[TARGET]` section. Edit this file if you wish to use software break points, and remember to use the **boot** command on the BDI2000 command line interface to make the changes take effect.

On the BDI2000, debugging can be performed either via the telnet interface or using **arm-eabi-gdb** and the bdiGDB interface. In the case of the latter, **arm-eabi-gdb** needs to connect to TCP port 2001 on the BDI2000's IP address. For example:

```
(gdb) target remote 111.222.333.444:2001
```

By default when the BDI2000 is powered up, the target will always run the initialization section of the `bdi2000.at91rm9200ek.cfg` file (which configures the SDRAM among other things), and halts the target. This behavior is repeated with the **reset halt** command.

If the board is reset when in '**reset halt**' mode (either with the '**reset halt**' or '**reset**' commands, or by pressing the reset button) and the '**go**' command is then given, then the board will boot as normal. If a ROMRAM RedBoot is resident in Flash, it will be run.

It is also possible for the target to always run, without initialization, after the reset button has been pressed. This mode is selected with the **reset run** command. This conveniently allows the target to be connected to the JTAG debugger, and be able to reset it with the reset button, without being required to always type '**go**' every time. Thereafter, invoking the **reset** command will repeat the previous reset style. Also in this mode, exceptions will be handled by board software, rather than causing the JTAG debugger to halt the CPU.

Suitably configured RAM applications can be loaded either via GDB, or directly via the telnet CLI. For example:

```
Core#0>load 0x20008000 /test.bin bin
Loading /test.bin , please wait ....
Loading program file passed
Core#0>go 0x20008000
```

Consult the BDI2000 documentation for information on other formats.

Configuration of RAM applications

If the JTAG device has initialized the SDRAM, such as by using the `bdi2000.at91rm9200ek.cfg` configuration on the BDI2000, RAM applications can be loaded directly into SDRAM without requiring a ROM monitor. This loading can be done directly through the JTAG device, or where supported by the JTAG device, through GDB.

In order to configure the application to support this mode, some configuration settings are required. Firstly `CYGSEM_HAL_USE_ROM_MONITOR` must be disabled. This will also cause the `CYGBLD_HAL_ARM9_ATMEL_AT91R-`

M9200_KITS_LOAD_LOW_RAM configuration option to be enabled allowing the application to be built with a set of memory layout files that will configure the linker script to set the program load address to be within the physical SDRAM space. Secondly the CYGDBG_HAL_DIAG_TO_DEBUG_CHAN option should be disabled in order to prevent HAL diagnostic output being encoded into GDB ($O) packets.

Running RAM applications

Once loaded and running via JTAG, HAL diagnostic output will appear by default on the serial debug port. USART 1 can be chosen instead by setting the CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL configuration option in the platform HAL to channel 1.

Warning

If resetting the board using the JTAG device, such as by using the BDI2000 **reset** command, the Ethernet PHY fails to interface correctly with the AT91RM9200, and consequently all subsequent ethernet operations are impossible. Only a reset by pressing the reset button or due to a watchdog timeout will cause the PHY to reset correctly.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the AT91RM9200-EK/DK hardware, and should be read in conjunction with that specification. The AT91RM9200-EK/DK platform HAL package complements the ARM architectural HAL, the ARM9 variant HAL and the AT91RM9200 processor HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset, the HAL will initialize or reinitialize most of the on-chip peripherals. There is an exception for RAM startup applications which depend on a ROM monitor for certain services.

For ROM or ROMRAM startup, the HAL will perform additional initialization, setting up the external RAM and programming the various internal registers. This is all done in the `PLATFORM_SETUP1` macro in the assembler header file `hal_platform_setup.h`.

Linker Scripts and Memory Maps

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

Flash

This is located at address 0x10000000 of the physical memory space. The HAL uses the MMU to locate it at virtual address 0x60000000 after initialization. It remains accessible at address 0x1000000 but accesses to this address range are uncached.

SDRAM

This is located at address 0x20000000 of the physical memory space. The HAL configures the MMU to retain the SDRAM at virtual address 0x20000000, but in order to assign hardware exception vectors vectors at address 0x00000000, the HAL also uses the MMU to create a clone of this memory at virtual address 0x00000000. The same memory is also accessible uncached and unbuffered at virtual location 0x30000000 for use by devices. The first 32 bytes are used for hardware exception vectors. The next 32 bytes are used for the VSR table and the next 256 bytes are normally used for the eCos virtual vectors, allowing RAM-based applications to use services provided by the ROM monitor. Memory is required for the MMU tables, and must be aligned on a 16Kbyte boundary. These therefore occupy memory from 0x4000 to 0x8000. For ROM/ROMRAM startup, all remaining SDRAM is available. For RAM startup, available RAM starts at virtual location 0x00040000, with the bottom 256kB reserved for use by RedBoot.

On-chip SRAM

This is located at address 0x00200000 of the physical memory space. However the HAL uses the MMU to relocate this to virtual address 0x70000000. The same memory is also accessible uncached and unbuffered at virtual location 0x70100000 for use by devices. At present this memory is entirely reserved for use by the ethernet interface, since there are problems using external SDRAM for ethernet buffers.

On-chip ROM

This is located at address 0x00100000 of the physical memory space. However the HAL uses the MMU to relocate this to virtual address 0x71000000. The same memory is also accessible uncached and unbuffered at virtual location 0x71800000.

USB host port

The USB host port (UHP) registers are located at address 0x00300000 of the physical memory space. However the HAL uses the MMU to relocate this to virtual address 0x72800000. Memory accessed at this address is uncached and unbuffered. There is no cached variant.

SPI dataflash
: SPI Dataflash media can only be accessed with the Flash API. For the purposes of this API a placeholder address range has been allocated as if the Flash is present at this address. The base of this address range is 0x30000000 and the extent will clearly depend on the Dataflash capacity. This reserved range is not real memory and any attempt to access it directly by the processor other than via the Flash API will result in a memory address exception.

On-chip Peripheral Registers
: These are located at address 0xFF000000 in the physical memory space. When the MMU is enabled, it sets up a direct, uncached, unbuffered mapping so that these registers remain accessible at their physical locations.

Off-chip Peripherals
: eCos uses the SDRAM, parallel NOR flash, ethernet PHY, SPI dataflash and MCI facilities on the AT91RM9200-EK/DK boards. eCos does not currently make any use of any other off-chip peripherals present on these boards.

Advanced Interrupt Controller

This port has been designed to exploit benefits of the Advanced Interrupt Controller of the AT91RM9200, using the facilities of the AT91RM9200 processor HAL. Consult the documentation in that package for details.

SPI Dataflash

eCos supports SPI access to Dataflash on the AT91RM9200. Two physical slots are provided on the board, but only the upper one may be used for SPI dataflash, not the one on the underside. This is due to an AT91RM9200 errata affecting SPI chip selects.

Accesses to Dataflash are performed via the Flash API, using 0x30000000 as the nominal address of the device, although it does not truly exist in the processor address space. On driver initialisation, eCos and RedBoot can detect the presence of a card in the socket. In particular, on reset RedBoot will indicate the presence of Flash at the 0x30000000 address range in its startup banner if it has been successfully detected. Hot swapping is not possible.

Since Dataflash is not directly addressable, access from RedBoot is only possible using **fis** command operations. Flash partitions within the FIS can be created, although users should be aware that the FIS partition data is stored in the NOR flash, and not on a per-Dataflash card basis. Therefore if a second Dataflash card is inserted it will appear to have the same FIS partitions residing on the card. Care must be taken if swapping between cards with differing partition layouts.

The MCI driver cannot be enabled simultaneously with the SPI driver, as the drivers need differing pin configurations for the same pins on this board due to the shared socket.

Real-time characterization

The tm_basic kernel test gives statistics gathered about the real-time characterization and performance of the kernel. The sample output is shown here for information. The test was built in ARM mode, which provided better performance than Thumb mode.

Example 251.1. atmel-at91rm9200-kits Real-time characterization

```
            Startup, main stack : stack used   412 size  3920
             Startup :  Interrupt stack used   524 size  4096
             Startup : Idlethread stack used    80 size  2048

eCos Kernel Timings
Notes: all times are in microseconds (.000001) unless otherwise stated

Reading the hardware clock takes 0 'ticks' overhead
... this value will be factored out of all other measurements
Clock interrupt took   13.02 microseconds (6 raw clock ticks)

Testing parameters:
   Clock samples:            32
   Threads:                  64
```

```
  Thread switches:         128
  Mutexes:                  32
  Mailboxes:                32
  Semaphores:               32
  Scheduler operations:    128
  Counters:                 32
  Flags:                    32
  Alarms:                   32


                                Confidence
    Ave     Min     Max     Var  Ave  Min  Function
 ======  ======  ======  ====== ========== ========
  14.09    8.55   19.23    2.46   45%  25% Create thread
   2.10    0.00    4.27    0.13   95%   3% Yield thread [all suspended]
   2.74    2.14    6.41    0.88   73%  73% Suspend [suspended] thread
   2.54    2.14    4.27    0.65   81%  81% Resume thread
   3.30    2.14    6.41    1.09   51%  46% Set priority
   0.47    0.00    2.14    0.73   78%  78% Get priority
   7.24    6.41   14.96    1.09   65%  65% Kill [suspended] thread
   2.10    0.00    4.27    0.13   95%   3% Yield [no other] thread
   4.21    2.14    6.41    0.19   93%   4% Resume [suspended low prio] thread
   2.40    2.14    4.27    0.47   87%  87% Resume [runnable low prio] thread
   2.94    2.14    4.27    1.00   62%  62% Suspend [runnable] thread
   2.14    0.00    4.27    0.07   96%   1% Yield [only low prio] thread
   2.17    2.14    4.27    0.06   98%  98% Suspend [runnable->not runnable]
   7.08    6.41   12.82    0.96   71%  71% Kill [runnable] thread
   5.41    4.27    8.55    1.10   50%  48% Destroy [dead] thread
  10.32    8.55   14.96    0.72   78%  20% Destroy [runnable] thread
  12.69   10.68   19.23    0.81   70%  20% Resume [high priority] thread
   5.56    4.27    8.55    1.04   58%  40% Thread switch

   0.52    0.00    2.14    0.78   75%  75% Scheduler lock
   1.77    0.00    2.14    0.61   82%  17% Scheduler unlock [0 threads]
   1.77    0.00    2.14    0.61   82%  17% Scheduler unlock [1 suspended]
   1.77    0.00    2.14    0.61   82%  17% Scheduler unlock [many suspended]
   1.78    0.00    4.27    0.61   82%  17% Scheduler unlock [many low prio]

   0.87    0.00    2.14    1.03   59%  59% Init mutex
   2.74    2.14    6.41    0.90   75%  75% Lock [unlocked] mutex
   3.07    2.14    4.27    1.05   56%  56% Unlock [locked] mutex
   2.34    2.14    4.27    0.36   90%  90% Trylock [unlocked] mutex
   2.14    2.14    2.14    0.00  100% 100% Trylock [locked] mutex
   0.53    0.00    2.14    0.80   75%  75% Destroy mutex
  12.89   12.82   14.96    0.13   96%  96% Unlock/Lock mutex

   0.80    0.00    4.27    1.05   65%  65% Create mbox
   0.80    0.00    2.14    1.00   62%  62% Peek [empty] mbox
   2.60    2.14    4.27    0.73   78%  78% Put [first] mbox
   0.33    0.00    2.14    0.56   84%  84% Peek [1 msg] mbox
   2.67    2.14    4.27    0.80   75%  75% Put [second] mbox
   0.53    0.00    2.14    0.80   75%  75% Peek [2 msgs] mbox
   2.60    2.14    4.27    0.73   78%  78% Get [first] mbox
   2.47    2.14    4.27    0.56   84%  84% Get [second] mbox
   2.47    2.14    4.27    0.56   84%  84% Tryput [first] mbox
   2.20    2.14    4.27    0.13   96%  96% Peek item [non-empty] mbox
   2.54    2.14    4.27    0.65   81%  81% Tryget [non-empty] mbox
   2.07    0.00    2.14    0.13   96%   3% Peek item [empty] mbox
   2.20    2.14    4.27    0.13   96%  96% Tryget [empty] mbox
   0.47    0.00    2.14    0.73   78%  78% Waiting to get mbox
   0.47    0.00    2.14    0.73   78%  78% Waiting to put mbox
   2.67    2.14    6.41    0.83   78%  78% Delete mbox
   6.61    6.41   12.82    0.38   96%  96% Put/Get mbox

   0.73    0.00    2.14    0.96   65%  65% Init semaphore
   2.20    0.00    4.27    0.26   90%   3% Post [0] semaphore
   2.20    2.14    4.27    0.13   96%  96% Wait [1] semaphore
```

```
    2.00    0.00    2.14    0.25   93%   6% Trywait [0] semaphore
    2.07    0.00    2.14    0.13   96%   3% Trywait [1] semaphore
    0.73    0.00    2.14    0.96   65%  65% Peek semaphore
    0.33    0.00    2.14    0.56   84%  84% Destroy semaphore
    7.28    6.41   10.68    1.08   62%  62% Post/Wait semaphore

    1.00    0.00    2.14    1.06   53%  53% Create counter
    0.93    0.00    2.14    1.05   56%  56% Get counter value
    0.60    0.00    2.14    0.86   71%  71% Set counter value
    2.74    2.14    4.27    0.86   71%  71% Tick counter
    0.53    0.00    2.14    0.80   75%  75% Delete counter

    0.73    0.00    2.14    0.96   65%  65% Init flag
    2.54    2.14    4.27    0.65   81%  81% Destroy flag
    2.00    0.00    4.27    0.38   87%   9% Mask bits in flag
    2.40    2.14    4.27    0.47   87%  87% Set bits in flag [no waiters]
    3.40    2.14    6.41    1.11   53%  43% Wait for flag [AND]
    3.27    2.14    4.27    1.06   53%  46% Wait for flag [OR]
    3.27    2.14    4.27    1.06   53%  46% Wait for flag [AND/CLR]
    3.34    2.14    4.27    1.05   56%  43% Wait for flag [OR/CLR]
    0.40    0.00    2.14    0.65   81%  81% Peek on flag

    1.53    0.00    2.14    0.86   71%  28% Create alarm
    4.41    2.14    8.55    0.51   84%   6% Initialize alarm
    2.00    0.00    4.27    0.38   87%   9% Disable alarm
    4.34    4.27    6.41    0.13   96%  96% Enable alarm
    2.40    2.14    4.27    0.47   87%  87% Delete alarm
    3.14    2.14    4.27    1.06   53%  53% Tick counter [1 alarm]
   14.69   12.82   14.96    0.47   87%  12% Tick counter [many alarms]
    5.01    4.27    6.41    0.96   65%  65% Tick & fire counter [1 alarm]
   86.34   85.47   87.61    1.03   59%  59% Tick & fire counters [>1 together]
   16.89   14.96   17.09    0.36   90%   9% Tick & fire counters [>1 separately]
   10.75   10.68   19.23    0.13   99%  99% Alarm latency [0 threads]
   13.59   10.68   21.37    1.64   76%  22% Alarm latency [2 threads]
   13.89   10.68   25.64    1.30   92%   4% Alarm latency [many threads]
   19.63   19.23   61.97    0.77   96%  96% Alarm -> thread resume latency

    2.15    2.14    6.41    0.00            Clock/interrupt latency

    5.05    2.14    8.55    0.00            Clock DSR latency

   48       0     296  (main stack:  1408)  Thread stack used (1360 total)
           All done, main stack : stack used  1408 size  3920
            All done :  Interrupt stack used   208 size  4096
            All done : Idlethread stack used   732 size  2048

Timing complete - 29590 ms total

PASS:<Basic timing OK>
EXIT:<done>
```

Other Issues

The AT91RM9200-EK/DK platform HAL does not affect the implementation of other parts of the eCos HAL specification. The AT91RM9200 processor HAL, ARM9 variant HAL, and the ARM architectural HAL documentation should be consulted for further details.

Chapter 252. Cogent CSB337 Board Support

Name

eCos Support for the CSB337 Board — Overview

Description

This document covers the Cogent CSB337 single board computer based on the Atmel AT91RM9200. The CSB337 contains the AT91RM9200 processor, 32Mb of SDRAM, 8MB of flash memory, an Intel LXT971 PHY and external connections for two serial channels, ethernet and the various other peripherals supported by the AT91RM9200. The CSB337 is usually plugged into a breakout board such as a Cogent CSB300 or CSB300CF.

For typical eCos development, a RedBoot image is programmed into the flash memory, and the board will boot this image from reset. RedBoot provides gdb stub functionality so it is then possible to download and debug stand-alone and eCos applications via the gdb debugger. This can happen over either a serial line or over ethernet.

Supported Hardware

The flash memory consists of 64 blocks of 128k bytes each. In a typical setup, the first flash block is used for the ROMRAM RedBoot image. The topmost block is used to manage the flash and hold RedBoot **fconfig** values. The remaining 60 blocks between 0x60020000 and 0x607DFFFF can be used by application code.

There is a serial driver `CYGPKG_IO_SERIAL_ARM_AT91` which supports both the Debug Unit and USART serial devices. These devices can be used by RedBoot for communication with the host. If either of these devices is needed by the application, either directly or via the serial driver, then it cannot also be used for RedBoot communication. Another communication channel such as ethernet should be used instead. The serial driver package is loaded automatically when configuring for the CSB337 target.

There is an ethernet driver `CYGPKG_DEVS_ETH_ARM_AT91RM9200` for the on-chip ethernet device. A second package `CYGPKG_DEVS_ETH_ARM_CSB337` is responsible for configuring this generic driver to the CSB337 hardware. These drivers are also loaded automatically when configuring for the CSB337 target.

The on-chip TWI device is not supported. Instead there is a bit-banged I²C bus using GPIO pins PA25 and PA26, with one attached device: a DS1307 battery-backed wallclock. The bus is supported by the `CYGPKG_IO_I2C` package and some platform-specific support. The platform HAL provides a cyg_i2c_bus structure `hal_csb337_i2c_bus`, and one cyg_i2c_device structure `cyg_i2c_wallclock_ds1307`. The wallclock is used mainly by the DS1307 device driver, but it also provides 56 bytes of non-volatile storage which can be used by the application. Any unused I²C functionality will be eliminated at link-time.

`CYGPKG_DEVICES_WALLCLOCK_DALLAS_DS1307` provides support for the DS1307 clock. This will be inactive unless the generic wallclock support `CYGPKG_IO_WALLCLOCK` is loaded. Some templates load this automatically, otherwise it must be loaded explicitly. The wallclock is not normally accessed directly. Instead it provides support for the standard C library time-related routines such as `time` and `asctime`, and can be updated by an eCos-specific function `cyg_libc_time_settime`.

eCos manages the on-chip interrupt controller. Timer counter 0 is used to implement the eCos system clock and the microsecond delay function. Other on-chip devices (Caches, PIO, UARTs, EMAC) are initialized only as far as is necessary for eCos to run. Other devices (SPI, MCI etc.) are not touched.

Tools

The CSB337 port is intended to work with GNU tools configured for an arm-eabi target. The original port was undertaken using arm-elf-gcc version 3.2.1, arm-elf-gdb version 5.3, and binutils version 2.13.1.

Name

Setup — Preparing the CSB337 board for eCos Development

Overview

In a typical development environment, the CSB337 board boots from flash into the RedBoot ROM monitor. eCos applications are configured for RAM startup and then downloaded and run on the board via the debugger **arm-eabi-gdb**. Preparing the board therefore usually involves programming a suitable RedBoot image into flash memory.

The following RedBoot configurations are supported:

Configuration	Description	Use	File
RAM	RedBoot running from RAM	`redboot_RAM.ecm`	`redboot_RAM.bin`
ROMRAM	RedBoot running from RAM, but contained in the board's flash boot sector.	`redboot_ROMRAM.ecm`	`redboot_ROMRAM.bin`

For serial communications, all versions run with 8 bits, no parity, and 1 stop bit at 38400 baud. RedBoot also supports ethernet communication and flash management.

Initial Installation

Flash Installation

The CSB337 boards are shipped from Cogent with a version of Micromonitor installed.

Installing RedBoot is a matter of downloading a new binary image and overwriting the existing Micromonitor ROM image. This is a two stage process, you must first download a RAM-resident version of RedBoot and then use that to download the ROM image to be programmed into the flash memory.

Connect a serial cable between the CSB337 board serial port 0 and a host computer and start a terminal emulator such as HyperTerminal. Experiments indicate that the version of the Xmodem protocol used by Micromonitor is incompatible with that used by the Linux minicom program. It does work with HyperTerminal, so at present RedBoot must be installed from a Windows host.

When Micromonitor starts up you will see something similar to this:

```
TFS Scanning //FLASH/...
EMAC: Auto-Negotiate Complete, Link = 100MBIT, Full Duplex.
MICRO MONITOR
CPU: AT91RM9200 ARM920T
Platform: Cogent CSB337 - AT91RM9200 SBC
Built: Jan_29,2004 @ 11:42:57
Monitor RAM: 0x20000000-0x2001a044
Application RAM Base: 0x20100000
MAC: 00:23:31:37:00:01
IP: 192.168.254.210
uMON>
```

Start the download by giving the following command to Micromonitor:

```
uMON>xmodem -d -a 0x20040000
```

You may get a sequence of binary characters, which indicate that Micromonitor is waiting for the download to start. Use HyperTerminal's X-Modem file transfer option to send the file `redboot_RAM.bin`.

When the transfer is finished you will see something like:

```
Rcvd 868 pkts (111104 bytes)
EMAC: Auto-Negotiate Complete, Link = 100MBIT, Full Dupex.
uMON>
```

Start RedBoot with the **call** command, which should result in RedBoot starting up.

```
uMON>call 0x20040040
+Ethernet eth0: MAC address 00:23:31:37:00:1c
IP: 10.0.0.210/255.255.255.0, Gateway: 10.0.0.1
Default server: 10.0.0.102, DNS server IP: 10.0.0.1

RedBoot(tm) bootstrap and debug environment [RAM]
Non-certified release, version v2_0_11a1 - built 13:21:01, Feb 12 2004

Platform: Cogent CSB337 (ARM9)
Copyright (C) 2000, 2001, 2002, Free Software Foundation, Inc.

RAM: 0x00000000-0x02000000, [0x00065e40-0x01fdd000] available
FLASH: 0x60000000 - 0x60800000, 64 blocks of 0x00020000 bytes each.
RedBoot>
```

Now the ROM image can be downloaded using the following RedBoot command:

```
RedBoot> load -r -b %{FREEMEMLO} -m xmodem
```

Again, use HyperTerminal's Xmodem support to send the file `redboot_ROMRAM.bin`. This should result in something like the following output:

```
Raw file loaded 0x00030000-0x0004d15f, assumed entry at 0x00030000
xyzModem - CRC mode, 932(SOH)/0(STX)/0(CAN) packets, 3 retries
RedBoot>
```

Once the file has been uploaded, you can check that it has been transferred correctly using the **cksum** command. On the host (Linux or Cygwin) run the **cksum** program on the binary file:

```
$ cksum redboot_ROMRAM.bin
2299507324 119136 redboot_ROMRAM.bin
```

In RedBoot, run the **cksum** command on the data that has just been loaded:

```
RedBoot> cksum -b %{FREEMEMLO} -l 119136
POSIX cksum = 2299507324 119136 (0x890fb27c 0x0001d160)
```

The second number in the output of the host **cksum** program is the file size, which should be used as the argument to the `-l` option in the RedBoot **cksum** command. The first numbers in each instance are the checksums, which should be equal.

If the program has downloaded successfully, then it can be programmed into the flash using the following commands:

```
RedBoot> fis init -f
About to initialize [format] FLASH image system - continue (y/n)? y
*** Initialize FLASH Image System
... Erase from 0x60020000-0x607e0000: ..................................
... Erase from 0x60800000-0x60800000:
... Unlock from 0x607e0000-0x60800000: .
... Erase from 0x607e0000-0x60800000: .
... Program from 0x01fe0000-0x02000000 at 0x607e0000: .
... Lock from 0x607e0000-0x60800000: .
RedBoot> fis create -b %{FREEMEMLO} RedBoot
An image named 'RedBoot' exists - continue (y/n)? y
... Erase from 0x60000000-0x60020000: .
... Program from 0x00100000-0x00120000 at 0x60000000: .
... Unlock from 0x607e0000-0x60800000: .
... Erase from 0x607e0000-0x60800000: .
... Program from 0x01fe0000-0x02000000 at 0x607e0000: .
... Lock from 0x607e0000-0x60800000: .
RedBoot>
```

The CBS337 board may now be reset either by cycling the power, pressing the reset switch, or with the **reset** command. It should then display the startup screen for the ROMRAM version of RedBoot.

Rebuilding RedBoot

Should it prove necessary to rebuild a RedBoot binary, this is done most conveniently at the command line. The steps needed to rebuild the the ROMRAM version of RedBoot for the CSB337 are:

```
$ mkdir redboot_csb337_romram
$ cd redboot_csb337_romram
$ ecosconfig new csb337 redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/arm/arm9/csb337/current/misc/redboot_ROMRAM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

To rebuild the RAM version of RedBoot:

```
$ mkdir redboot_csb337_ram
$ cd redboot_csb337_ram
$ ecosconfig new csb337 redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/arm/arm9/csb337/current/misc/redboot_RAM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

At the end of the build the `install/bin` subdirectory should contain the file `redboot.bin`. This is the case for both the above builds, take care not to mix the two files up, since programming the RAM RedBoot into the ROM will render the board unbootable.

Name

Configuration — Platform-specific Configuration Options

Overview

The CSB337 platform HAL package is loaded automatically when eCos is configured for a `csb337` target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The CSB337 platform HAL package supports three separate startup types:

RAM
: This is the startup type which is normally used during application development. The board has RedBoot programmed into flash and boots into that initially. arm-eabi-gdb is then used to load a RAM startup application into memory and debug it. It is assumed that the hardware has already been initialized by RedBoot. By default the application will use the eCos virtual vectors mechanism to obtain certain services from RedBoot, including diagnostic output.

ROM
: This startup type can be used for finished applications which will be programmed into flash at physical address 0x10000000. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization.

ROM-RAM
: This startup type can be used for finished applications which will be programmed into flash at physical location 0x10000000. However, when it starts up the application will first copy itself to RAM at 0x00000000 and then run from there. RAM is generally faster than flash memory, so the program will run more quickly than a ROM-startup application. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization.

RedBoot and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building RedBoot.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then the serial port will be claimed for HAL diagnostics.

Flash Driver

The CBS337 board contains an 8Mb Intel StrataFlash flash device. The `CYGPKG_DEVS_FLASH_STRATA` package contains all the code necessary to support these parts and the `CYGPKG_DEVS_FLASH_CSB337` package contains definitions that customize the driver to the CSB337 board.

Ethernet Driver

The CSB337 board uses the AT91RM9200's internal EMAC ethernet device attached to an external Intel LXT971 PHY. The `CYGPKG_DEVS_ETH_ARM_AT91RM9200` package contains all the code necessary to support this device and the `CYGPKG_DEVS_ETH_ARM_CSB337` package contains definitions that customize the driver to the CSB337 board.

System Clock

By default, the system clock interrupts once every 10ms, corresponding to a 100Hz clock. This can be changed by the configuration option `CYGNUM_HAL_RTC_DENOMINATOR` which corresponds to the clock frequency. Other clock-related settings are recalculated automatically if the denominator is changed.

Compiler Flags

The platform HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos. There is just one flag specific to this port:

`-mcpu=arm9`
: The arm-eabi-gcc compiler supports many variants of the ARM architecture. A `-m` option should be used to select the specific variant in use, and with current tools `-mcpu=arm9` is the correct option for the ARM920T CPU in the AT91RM9200.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the CSB337 hardware, and should be read in conjunction with that specification. The CSB337 platform HAL package complements the ARM architectural HAL and the ARM9 variant HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset the HAL will initialize or reinitialize most of the on-chip peripherals. There is an exception for RAM startup applications which depend on a ROM monitor for certain services.

For ROM or ROMRAM startup, the HAL will perform additional initialization, setting up the external RAM and programming the various internal registers. This is all done in the `PLATFORM_SETUP1` macro in the assembler header file `hal_platform_setup.h`.

Linker Scripts and Memory Maps

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

Flash

This is located at address 0x10000000 of the physical memory space. However, the HAL uses the MMU to relocate it to virtual address 0x60000000 after initialization.

SDRAM

This is located at address 0x20000000 of the physical memory space. However the HAL uses the MMU to relocate this to virtual address 0x00000000. The same memory is also accessible uncached and unbuffered at virtual location 0x20000000 for use by devices. The first 32 bytes are used for hardware exception vectors. The next 32 bytes are used for the VSR table and the next 256 bytes are normally used for the eCos virtual vectors, allowing RAM-based applications to use services provided by the ROM monitor. For ROM/ROMRAM startup, all remaining SDRAM is available. For RAM startup, available RAM starts at virtual location 0x00040000, with the bottom 256kB reserved for use by RedBoot.

On-chip SRAM

This is located at address 0x00200000 of the physical memory space. However the HAL uses the MMU to relocate this to virtual address 0x70000000. The same memory is also accessible uncached and unbuffered at virtual location 0x70100000 for use by devices. At present this memory is entirely reserved for use by the ethernet interface, since there are problems using external SDRAM for ethernet buffers.

On-chip Peripheral Registers

These are located at address 0xFF000000 in the physical memory space. When the MMU is enabled, it sets up a direct, uncached, unbuffered mapping so that these registers remain accessible at their physical locations.

Off-chip Peripherals

Apart from the SDRAM, flash and ethernet PHY, eCos does not currently make any use of the off-chip peripherals present on the CSB337.

Other Issues

The CSB337 platform HAL does not affect the implementation of other parts of the eCos HAL specification. The ARM9 variant HAL, and the ARM architectural HAL documentation should be consulted for further details.

Chapter 253. SSV DNP/9200 with DNP/EVA9 Board Support

Name

eCos Support for the SSV DNP/9200 with DNP/EVA9 Evaluation Board — Overview

Description

This document covers the configuration and usage of eCos and RedBoot on the DNP/9200 with DNP/EVA9 evaluation board as provided in the SSV SK23 starter kit. This kit provides the AT91RM9200 processor, 32Mbytes of SDRAM, 16Mbytes of parallel NOR flash memory, a Davicom DM9161A PHY, a SD/MMC/DataFlash socket, a DAC, external connections for two serial channels, ethernet, USB host/device and the various other peripherals supported by the AT91RM9200. eCos support for the many devices and peripherals on the boards and the AT91RM9200 is described below.

For typical eCos development, a RedBoot image is programmed into the flash memory, and the board will boot into this image from reset. RedBoot provides gdb stub functionality so it is then possible to download and debug stand-alone and eCos applications via the gdb debugger. This can happen over either a serial line or over ethernet.

This documentation is expected to be read in conjunction with the AT91RM9200 processor HAL documentation and further device support and subsystems are described and documented there.

Supported Hardware

The parallel NOR flash memory consists of 128 blocks of 128Kbytes each. In a typical setup, the first 256 Kbytes are reserved for the use of the ROMRAM RedBoot image. The topmost block is used to manage the flash and hold RedBoot **fconfig** values. The remaining blocks can be used by application code. There are 126 blocks available between 0x60030000 and 0x60FDFFFF.

There is a serial driver `CYGPKG_IO_SERIAL_ARM_AT91` which supports both the Debug Unit and USART serial devices. The serial port at J6 COM1 (connected to USART channel 1) and DTE port at J7 COM2 (connected to USART channel 2) can be used by RedBoot for communication with the host. If either of these devices is needed by the application, either directly or via the serial driver, then it cannot also be used for RedBoot communication. Another communication channel such as ethernet should be used instead. The serial driver package is loaded automatically when configuring for the DNP/9200 target.

There is an ethernet driver `CYGPKG_DEVS_ETH_ARM_AT91RM9200` for the on-chip ethernet device. The platform HAL package is responsible for configuring this generic driver to the hardware. This driver is also loaded automatically when configuring for the DNP/9200 target.

There is a watchdog driver `CYGPKG_DEVICES_WATCHDOG_ARM_AT91RM9200`. This driver is also loaded automatically when configuring for the DNP/9200 target.

There is a driver for the on-chip real-time clock (RTC) at `CYGPKG_DEVICES_WALLCLOCK_ARM_AT91`. This driver is also loaded automatically when configuring for the DNP/9200 target.

The AT91RM9200 processor HAL contains a driver for the Two-Wire Interface (TWI) controller on the AT91RM9200. This type of bus is also known as I²C®. Further documentation may be found in the AT91RM9200 processor HAL documentation.

The AT91RM9200 processor HAL contains a driver for the MultiMedia Card Interface (MCI). This driver is loaded automatically when configuring for the DNP/9200 target and allows use of MMC and Secure Digital (SD) flash storage cards within eCos, exported as block devices. Further documentation may be found in the AT91RM9200 processor HAL documentation.

There is a general AT91 SPI driver (`CYGPKG_DEVS_SPI_ARM_AT91`) which provides the underlying implementation for the SPI API layer in the `CYGPKG_IO_SPI` package. All these packages are automatically loaded when configuring for the DNP/9200 target.

In general, devices (Caches, PIO, UARTs, EMAC) are initialized only as far as is necessary for eCos to run. Other devices (RTC, I²C, SPI, MCI etc.) are not touched unless the appropriate driver is loaded, although in some cases, the HAL boot sequence will set up the appropriate PIO configuration.

Tools

The DNP/9200 support is intended to work with GNU tools configured for an arm-eabi target. The original port was undertaken using arm-elf-gcc version 3.4.4, arm-elf-gdb version 6.3, and binutils version 2.15.

Name

Setup — Preparing the DNP/9200 with DNP/EVA9 evaluation board for eCos Development

Overview

In a typical development environment, the DNP/9200 board boots from the parallel NOR Flash and run the RedBoot ROM monitor directly. eCos applications are configured for RAM startup and then downloaded and run on the board via the debugger **arm-eabi-gdb**. Preparing the board therefore usually involves programming a suitable RedBoot image into flash memory.

The following RedBoot configurations are supported:

Configuration	Description	Use	File
ROM	RedBoot running from ROM	`redboot_ROM.ecm`	`redboot_ROM.bin`
RAM	RedBoot running from RAM	`redboot_RAM.ecm`	`redboot_RAM.bin`
ROMRAM	RedBoot running from RAM, but contained in the board's flash boot sector	`redboot_ROMRAM.ecm`	`redboot_ROMRAM.bin`
JTAG	RedBoot running from RAM, loaded via JTAG	`redboot_JTAG.ecm`	`redboot_JTAG.bin`
UBOOT	RedBoot running from RAM, loaded via U-Boot	`redboot_UBOOT.ecm`	`redboot_UBOOT.bin`

For serial communications, all versions run with 8 bits, no parity, and 1 stop bit at 115200 baud. RedBoot also supports ethernet communication and flash management.

The ample provision of RAM memory on the board allows the ROMRAM version of RedBoot to be preferential to the standard ROM version which executes directly from Flash. Alternatively, if the ROM version is to be chosen, then the RAM version is provided to allow for updating the resident RedBoot image in Flash. The JTAG version is only used if loading RedBoot into RAM via a JTAG debugger or ICE. It is similar to the RAM version, but loads at a lower address within RAM, and so can be used to in turn load eCos applications, as if it is the normal resident boot monitor. The ELF format image of this JTAG version of RedBoot can also be loaded and executed from GDB using the Abatron BDI2000 bdiGDB support, to allow it to be debugged. The UBOOT version may be used to load RedBoot into RAM using the UBOOT bootloader. This is usually only used to then load a ROMRAM executable for programming into flash. Like the JTAG version it loads at a lower RAM address and can therefore be used to load RAM applications.

Initial Installation

The on-chip boot program on the AT91RM9200 is only capable of loading programs into 12Kbytes of on-chip SRAM and is therefore quite restrictive. Consequently two mechanisms are described below to program RedBoot into Flash. The first requires a JTAG device while the other makes use of the U-Boot bootloader that is shipped with the board.

Method 1 - Program RedBoot into Flash with RAM RedBoot loaded by JTAG

With this approach, the BDI2000 is used to load a RAM RedBoot image, which can then in turn be used to load and program a ROMRAM RedBoot image into Flash. In the following documentation it is assumed that the Abatron BDI2000 is being used. For a different JTAG device, equivalent operations will need to be performed.

There are three stages, firstly loading the RAM RedBoot image, then initializing RedBoot's Flash configuration, and finally loading and programming the ROMRAM RedBoot. First, however, we must set up the BDI2000 and the board.

Preparing the Abatron BDI2000 JTAG debugger

The BDI2000 must first be configured to allow communication with your local network, and configured with the parameters for interfacing with the target board. The following steps should be followed:

1. Prepare a PC to act as a host PC and start a TFTP server on it.

2. Connect the Abatron BDI2000 JTAG debugger via both serial and ethernet to the host PC and power it on. Use the serial cable supplied with the BDI2000.

3. Install the Abatron BDI2000 bdiGDB support software on the host PC.

4. Locate the file `bdi2000.dnp_sk23.cfg` within the eCos platform HAL package in the source repository. This will be in the directory `packages/hal/arm/arm9/dnp_sk23/`*`VERSION`*`/misc` relative to the root of your eCos installation.

5. Locate the file `reg920t.def` within the installation of the BDI2000 bdiGDB support software.

6. Place the bdi2000.dnp_sk23.cfg in a location on the PC accessible to the TFTP server. Later you will configure the BDI2000 to load this file via TFTP as its configuration file.

7. Similarly place the file `reg920t.def` in a location accessible to the TFTP server.

8. Open `bdi2000.dnp_sk23.cfg` in an editor such as emacs or notepad and if necessary adjust the path of the `reg920t.def` file in the `[REGS]` section to match its location relative to the TFTP server root.

9. Install and configure the Abatron BDI2000 in line with the bdiGDB instruction manual. Configure the BDI2000 to use the `bdi2000.dnp_sk23.cfg` configuration file at the appropriate point of this process.

Preparing the DNP/9200 with DNP/EVA9 evaluation board for programming

Follow the steps in this section in order to allow communication between the board and the host PC, and between the board and the JTAG device.

1. First you must connect a null modem DB9 serial cable between COM1 on the board and a serial port on the host computer.

2. Start a suitable terminal emulator on the host computer such as **minicom** or HyperTerminal. Set the communication parameters to 115200 baud, 8 data bits, no parity bit and 1 stop bit with no flow control.

3. Connect the board to your host PC's LAN with an Ethernet cable.

4. You should designate the board with a new Ethernet MAC address. The RedBoot binary image contains a default address, but each board requires its own unique address. It is advisable to mark each board with its programmed MAC address for future identification.

5. Connect the board to the BDI2000 using a 20-pin ARM/Xscale cable from the JTAG interface connector on the DNP/9200 to the Target A port on the BDI2000. Since the JTAG connector on the DNP/9200 is non-standard, this will require an adaptor cable.

6. Power up the DNP/EVA9 board. You should see the power LED and some of the ethernet LEDs illuminate.

7. Connect to the BDI2000's CLI interface via TCP/IP on the standard telnet port 23. The **telnet** application is suitable for this. You should see usage information followed by the prompt:

   ```
   Core#0>
   ```

8. Confirm correct connection with the BDI2000 with the **reset halt** command as follows:

```
Core#0> reset halt
- TARGET: processing reset request
- TARGET: BDI asserts TRST and RESET
- TARGET: BDI removes TRST
- TARGET: Bypass check 0x00000001 => 0x00000001
- TARGET: JTAG exists check passed
- Core#0: ID code is 0x05B0203F
- TARGET: All ICEBreaker access checks passed
- TARGET: BDI removes RESET
- TARGET: BDI waits for RESET inactive
- TARGET: resetting target passed
- TARGET: processing target startup ....
- TARGET: processing target startup passed
Core#0>
```

9. Locate the `redboot_ROMRAM.bin` image within the `loaders` subdirectory of the base of the eCos installation.

10. Copy the `redboot_ROMRAM.bin` file into a location on the host computer accessible to its TFTP server.

Loading a RAM RedBoot

1. Locate the `redboot_JTAG.bin` image within the `loaders` subdirectory of the base of the eCos installation.

2. Copy the `redboot_JTAG.bin` file into a location on the host computer accessible to its TFTP server.

3. With the BDI2000 telnet interface, execute the following command, replacing *`/RBPATH`* with the location of the redboot_JTAG.bin file relative to the TFTP server root directory:

```
Core#0>load 0x20010000 /RBPATH/redboot_JTAG.bin bin
Loading /RBPATH/redboot_JTAG.bin , please wait ....
Loading program file passed
Core#0>
```

4. Run the loaded RAM RedBoot:

```
Core#0>go 0x20010040
Core#0>
```

The terminal emulator connected to the serial debug port should now have displayed the RedBoot banner and prompt similar to the following:

```
+**Warning** FLASH configuration checksum error or invalid key
Use 'fconfig -i' to [re]initialize database
PHY: Davicom DM9161A
AT91RM9200 ETH: Waiting for link to come up.
AT91RM9200 ETH: 100Mb
... waiting for BOOTP information
Ethernet eth0: MAC address 00:23:31:37:00:1c
IP: 10.0.2.9/255.0.0.0, Gateway: 10.0.0.3
Default server: 10.0.1.2, DNS server IP: 10.0.0.1

RedBoot(tm) bootstrap and debug environment [RAM]
Non-certified release, version UNKNOWN - built 17:10:26, Jun  8 2006

Platform: DNP/9200 with DNP/EVA9 (ARM9)
Copyright (C) 2000, 2001, 2002, 2003, 2004 Free Software Foundation, Inc.
Copyright (C) 2003, 2004, 2005, 2006 eCosCentric Limited

RAM: 0x20000000-0x20800000, [0x2002f2e0-0x207dd000] available
FLASH: 0x60000000-0x60ffffff, 128 x 0x20000 blocks
RedBoot>
```

In the above output, a local BOOTP/DHCP server was able to serve an address to the device.

Note

It is also possible to use the RAM startup version of RedBoot and the redboot_RAM.bin file instead of redboot_JTAG.bin above. If so, then the address to the **load** command must be 0x20100000, as must be the address to the **go** command.

RedBoot Flash configuration

The following steps describe how to initialize RedBoot's Flash configuration. This must be performed when using a RAM RedBoot to program Flash.

1. Use the following command to initialize RedBoot's Flash Information System (FIS):

```
RedBoot> fis init
About to initialize [format] FLASH image system - continue (y/n)? y
*** Initialize FLASH Image System
... Unlocking from 0x60fe0000-0x60ffffff: .
... Erase from 0x60fe0000-0x60ffffff: .
... Program from 0x207e0000-0x20800000 to 0x60fe0000: .
... Locking from 0x60fe0000-0x60ffffff: .
RedBoot>
```

2. Now configure RedBoot's Flash configuration with the command:

```
RedBoot> fconfig -i
```

Remember to substitute the appropriate MAC address for this board at the appropriate step. If a BOOTP/DHCP server is not available, then IP configuration may be set manually. The default server IP address can be set to a PC that will act as a TFTP host for future RedBoot load operations, or may be left unset. The following gives an example configuration:

```
RedBoot> fconfig -i
Initialize non-volatile configuration - continue (y/n)? y
Run script at boot: false
Use BOOTP for network configuration: false
Gateway IP address: 192.168.7.11
Local IP address: 192.168.7.222
Local IP address mask: 255.255.255.0
Default server IP address: 192.168.7.9
Console baud rate: 115200
DNS server IP address: 192.168.7.11
Network hardware address [MAC]: 0x00:0x23:0x31:0x37:0x00:0x4e
GDB connection port: 9000
Force console for special debug messages: false
Network debug at boot time: false
Default network device: at91rm9200_eth
Update RedBoot non-volatile configuration - continue (y/n)? y
... Unlocking from 0x60fe0000-0x60ffffff: .
... Erase from 0x60fe0000-0x60ffffff: .
... Program from 0x207e0000-0x20800000 to 0x60fe0000: .
... Locking from 0x60fe0000-0x60ffffff: .
RedBoot>
```

Loading and programming the ROMRAM RedBoot

This section describes the steps required to load the ROMRAM RedBoot from the TFTP server and program it into Flash.

1. Load the RedBoot ROMRAM binary image from the TFTP server. Use the following command, replacing *`111.222.333.444`* with the TFTP server IP address (or domain name if a DNS server has been configured), and *`/RBPATH`* with the location of the `redboot_ROMRAM.bin` file relative to the TFTP server root directory:

```
RedBoot> load -r -b %{freememlo} -h 111.222.333.444 /RBPATH/redboot_ROMRAM.bin
Using default protocol (TFTP)
Raw file loaded 0x20030000-0x2004e91b, assumed entry at 0x20030000
```

```
RedBoot>
```

2. Finally install the loaded image into Flash:

   ```
   RedBoot> fis create RedBoot
   An image named 'RedBoot' exists - continue (y/n)? y
   ... Erase from 0x60000000-0x6003ffff: ..
   ... Program from 0x20030000-0x2004e91c to 0x60000000: .
   ... Locking from 0x60000000-0x6003ffff: ..
   ... Unlocking from 0x60fe0000-0x60ffffff: .
   ... Erase from 0x60fe0000-0x60ffffff: .
   ... Program from 0x21fe0000-0x22000000 to 0x60fe0000: .
   ... Locking from 0x60fe0000-0x60ffffff: .
   RedBoot>
   ```

   It is also possible to use the **fis write** command to write the image into Flash, but if so, the relevant Flash blocks must also be explicitly unlocked with the command:

   ```
   RedBoot> fis unlock -f 0x60000000 -l 0x30000
   ```

The RedBoot installation is now complete. This can be tested by powering off the board, disconnecting the JTAG, and then powering on the board again. Output similar to the following should be seen on the serial port. Verify the IP settings are as expected.

```
+PHY: Davicom DM9161A
AT91RM9200 ETH: 100Mb
... waiting for BOOTP information
Ethernet eth0: MAC address 00:23:31:37:00:1c
IP: 10.0.2.9/255.0.0.0, Gateway: 10.0.0.3
Default server: 10.0.1.2, DNS server IP: 10.0.0.1

RedBoot(tm) bootstrap and debug environment [ROMRAM]
Non-certified release, version UNKNOWN - built 18:10:00, Jun  8 2006

Platform: DNP/9200 with DNP/EVA9 (ARM9)
Copyright (C) 2000, 2001, 2002, 2003, 2004 Free Software Foundation, Inc.
Copyright (C) 2003, 2004, 2005, 2006 eCosCentric Limited

RAM: 0x20000000-0x22000000, [0x2002ff78-0x21fdd000] available
FLASH: 0x60000000-0x60ffffff, 128 x 0x20000 blocks
RedBoot>
```

If it proves necessary to re-install RedBoot, this may be achieved by repeating the above process. Alternatively, a new image may be downloaded and programmed into flash more directly using RedBoot's own commands. See the RedBoot documentation for details.

Method 2 - Program RedBoot into Flash with RAM Redboot loaded by U-Boot

With this approach, the existing U-Boot bootloader is used to load a RAM RedBoot which is then used to load and program a ROMRAM RedBoot image to replace U-Boot in Flash. A JTAG debugger is not needed for this method.

There are three stages, firstly loading the RAM RedBoot image, then initializing RedBoot's Flash configuration, and finally loading and programming the ROMRAM RedBoot. The first of these stages is described here, the remaining two stages are identical to the equivalent stages in Method 1, above.

Loading a RAM RedBoot

1. First you must connect a null modem DB9 serial cable between COM1 on the board and a serial port on the host computer.

2. Start a suitable terminal emulator on the host computer such as **minicom** or HyperTerminal. Set the communication parameters to 115200 baud, 8 data bits, no parity bit and 1 stop bit with no flow control.

3. Connect the board to your host PC's LAN with an Ethernet cable.

4. You should designate the board with a new Ethernet MAC address. The RedBoot binary image contains a default address, but each board requires its own unique address. It is advisable to mark each board with its programmed MAC address for future identification.

5. Locate the `redboot_UBOOT.bin` image within the `loaders` subdirectory of the base of the eCos installation.

6. Copy the `redboot_UBOOT.bin` file into a location on the host computer accessible to its TFTP server.

7. Power up the DNP/EVA9 board. You should see the power LED and some of the ethernet LEDs illuminate.

8. After a few seconds the following output should be seen on the serial line. Hit a key to stop the auto boot:

```
U-Boot 1.1.2 (Dec 14 2005 - 13:12:14)

U-Boot code: 21F00000 -> 21F1666C  BSS: -> 21F1AC44
RAM Configuration:
Bank #0: 20000000 32 MB
Flash: 16 MB

In:    serial
Out:   serial
Err:   serial
Hit any key to stop autoboot:  0
U-Boot>
```

9. Set up the environment of U-Boot to download the RAM RedBoot. Users with access to a TFTP server should substitute their own IP address, TFTP server address and netmask in the following commands:

```
U-Boot> setenv ipaddr 10.0.2.22
U-Boot> setenv serverip 10.0.1.2
U-Boot> setenv netmask 255.0.0.0
U-Boot> setenv bootfile redboot_UBOOT.bin
```

10. Download the RAM RedBoot. Users with access to a TFTP server should use the following command:

```
U-Boot> tftpboot 0x20040000
TFTP from server 10.0.1.2; our IP address is 10.0.2.22
Filename 'redboot_UBOOT.bin'.
Load address: 0x20040000
Loading: ########################
done
Bytes transferred = 120760 (1d7b8 hex)
```

11. Users without access to a TFTP server may use Kermit to send the file `redboot_UBOOT.bin`:

```
U-Boot> loadb 0x20040000
```

12. The downloaded RedBoot image may now be executed:

```
U-Boot> go 0x20040000
.+PHY: Davicom DM9161A
AT91RM9200 ETH: 100Mb
... waiting for BOOTP information
Ethernet eth0: MAC address 00:23:31:37:00:1c
IP: 10.0.2.9/255.0.0.0, Gateway: 10.0.0.3
Default server: 10.0.1.2, DNS server IP: 10.0.0.1

RedBoot(tm) bootstrap and debug environment [UBOOT]
Non-certified release, version UNKNOWN - built 18:18:48, Jun  8 2006

Platform: DNP/9200 with DNP/EVA9 (ARM9)
Copyright (C) 2000, 2001, 2002, 2003, 2004 Free Software Foundation, Inc.
Copyright (C) 2003, 2004, 2005, 2006 eCosCentric Limited

RAM: 0x20000000-0x22000000, [0x20067040-0x21fdd000] available
```

```
FLASH: 0x60000000-0x60ffffff, 128 x 0x20000 blocks
RedBoot>
```

13.Now follow the directions in the previous method to initialize the flash and program the ROMRAM RedBoot.

Rebuilding RedBoot

Should it prove necessary to rebuild a RedBoot binary, this is done most conveniently at the command line. The steps needed to rebuild the the ROMRAM version of RedBoot for the DNP/9200 with DNP/EVA9 are:

```
$ mkdir redboot_dnp_sk23_romram
$ cd redboot_dnp_sk23_romram
$ ecosconfig new dnp_sk23 redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/arm/arm9/dnp_sk23/VERSION/misc/redboot_ROMRAM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

At the end of the build the `install/bin` subdirectory should contain the file `redboot.bin`.

The other versions of RedBoot - ROM, RAM, JTAG or UBOOT - may be similarly built by choosing the appropriate alternative .ecm file.

Name

Configuration — Platform-specific Configuration Options

Overview

The DNP/9200 with DNP/EVA9 platform HAL package is loaded automatically when eCos is configured for `dnp_sk23` target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The platform HAL package supports three separate startup types:

RAM
: This is the startup type which is normally used during application development. The board has RedBoot programmed into flash and boots into that initially. arm-eabi-gdb is then used to load a RAM startup application into memory and debug it. It is assumed that the hardware has already been initialized by RedBoot. By default the application will use the eCos virtual vectors mechanism to obtain certain services from RedBoot, including diagnostic output.

ROM
: This startup type can be used for finished applications which will be programmed into flash at physical address 0x10000000. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization.

ROMRAM
: This startup type can be used for finished applications which will be programmed into flash at physical location 0x10000000. However, when it starts up, the application will first copy itself to RAM at virtual address 0x00000000 and then run from there. RAM is generally faster than flash memory, so the program will run more quickly than a ROM-startup application. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization.

RedBoot and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building RedBoot.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then the serial port will be claimed for HAL diagnostics.

Flash Driver

The DNP/9200 board contains a 16Mbyte Intel 28F128J3 parallel Flash device. The `CYGPKG_DEVS_FLASH_STRATA_V2` package contains all the code necessary to support this part and the platform HAL package contains definitions that customize the driver to the board. This driver is not active until the generic Flash support package, `CYGPKG_IO_FLASH`, is included in the configuration.

Ethernet Driver

The DNP/9200 board uses the AT91RM9200's internal EMAC ethernet device attached to an external Davicom DM9161 PHY. The `CYGPKG_DEVS_ETH_ARM_AT91RM9200` package contains all the code necessary to support this device and the platform HAL package contains definitions that customize the driver to the DNP/9200 board. This driver is not active until the generic Ethernet support package, `CYGPKG_IO_ETH_DRIVERS`, is included in the configuration.

RTC Driver

The DNP/9200 board uses the AT91RM9200's internal RTC support. The `CYGPKG_DEVICES_WALLCLOCK_ARM_AT91` package contains all the code necessary to support this device. This driver is not active until the generic wallclock device support package, `CYGPKG_IO_WALLCLOCK`, is included in the configuration.

Watchdog Driver

The DNP/9200 board uses the AT91RM9200's internal watchdog support. The `CYGPKG_DEVICES_WATCHDOG_ARM_AT91RM9200` package contains all the code necessary to support this device. Within that package the `CYGNUM_DEVS_WATCHDOG_ARM_AT91RM9200_DESIRED_TIMEOUT_MS` configuration option controls the watchdog timeout, and by default will force a reset of the board upon timeout. This driver is not active until the generic watchdog device support package, `CYGPKG_IO_WATCHDOG`, is included in the configuration.

USART Serial Driver

The DNP/9200 board uses the AT91RM9200's internal USART serial support as described in the AT91RM9200 processor HAL documentation. Two serial ports are available: USART 1 which is mapped to virtual vector channel 0 in the HAL diagnostic driver or "`/dev/ser1`" in the interrupt-driven driver; and USART 2 which is mapped to virtual vector channel 1 and "`/dev/ser2`". Both UARTs support modem control signals such as those used for hardware flow control.

MCI Driver

As described in the SSV board documentation, in order to use the MMC/SD socket on the EVA9 board, the JP8 jumper block must have all jumpers in place, i.e. closed . The SPI jumper block JP9 must have all jumpers removed, i.e. open.

As the AT91RM9200 MCI driver is part of the AT91RM9200 HAL, nothing is required to load it. Similarly the MMC/SD bus driver layer (`CYGPKG_DEVS_DISK_MMC`) is automatically included as part of the hardware-specific configuration for this target. All that is required to enable the support is to include the generic disk I/O infrastructure package (`CYGPKG_IO_DISK`), along with the intended filesystem, typically, the FAT filesystem (`CYGPKG_FS_FAT`) and any of its package dependencies (including `CYGPKG_LIBC_STRING` and `CYGPKG_LINUX_COMPAT` for FAT).

Various options can be used to control specific of the AT91RM9200 MCI driver. Consult the AT91RM9200 HAL documentation for information on its configuration.

On this target, it is not possible to detect from the MMC/SD socket whether cards have been inserted or removed. Thus the disk I/O layer's removeable media support will not detect when cards have been inserted or removed, and therefore the only way to detect if a card has been inserted is to attempt mounts.

The MMC/SD socket also does not permit detection of the write-protect (or "lock") switch present on SD cards. "Locked" cards will therefore not be detected which means that despite the switch position, it is still possible to write to them since the lock switch does not physically enforce write protection.

Compiler Flags

The platform HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos. There are just three flags specific to this port:

`-mcpu=arm9`
: The arm-eabi-gcc compiler supports many variants of the ARM architecture. A `-m` option should be used to select the specific variant in use, and with current tools `-mcpu=arm9` is the correct option for the ARM920T CPU in the AT91RM9200.

`-mthumb`
The arm-eabi-gcc compiler will compile C and C++ files into the Thumb instruction set when this option is used. The best way to build eCos in Thumb mode is to enable the configuration option `CYGHWR_THUMB`.

`-mthumb-interwork`
This option allows programs to be created that mix ARM and Thumb instruction sets. Without this option, some memory can be saved. This option should be used if -mthumb is used. The best way to build eCos with Thumb interworking is to enable the configuration option `CYGBLD_ARM_ENABLE_THUMB_INTERWORK`.

Name

JTAG support — Usage

Use of JTAG for debugging

JTAG can be used to single-step and debug loaded RAM applications, or even applications resident in ROM, including RedBoot.

Debugging of ROM applications is only possible if using hardware breakpoints. The ARM920T core of the AT91RM9200 only supports two such hardware breakpoints, so they should be used sparingly. If using a GDB front-end such as Eclipse, check it has not set unnecessary extra breakpoints. Some JTAG devices give the option of whether to set hardware or software breakpoints by default. Be sure to configure your device appropriately.

Abatron BDI2000 notes

On the Abatron BDI2000, the `bdi2000.dnp_sk23.cfg` file should be used to setup and configure the hardware to an appropriate state to load programs. This includes setting up the SDRAM controller.

The `bdi2000.dnp_sk23.cfg` file also contains an option to define whether hardware or software breakpoints are used by default, using the `BREAKMODE` directive in the `[TARGET]` section. Edit this file if you wish to use software break points, and remember to use the **boot** command on the BDI2000 command line interface to make the changes take effect.

On the BDI2000, debugging can be performed either via the telnet interface or using **arm-eabi-gdb** and the bdiGDB interface. In the case of the latter, **arm-eabi-gdb** needs to connect to TCP port 2001 on the BDI2000's IP address. For example:

```
(gdb) target remote 111.222.333.444:2001
```

By default when the BDI2000 is powered up, the target will always run the initialization section of the `bdi2000.dnp_sk23.cfg` file (which configures the SDRAM among other things), and halts the target. This behavior is repeated with the **reset halt** command.

If the board is reset when in '**reset halt**' mode (either with the '**reset halt**' or '**reset**' commands, or by pressing the reset button) and the '**go**' command is then given, then the board will boot as normal. If a ROMRAM RedBoot is resident in Flash, it will be run.

It is also possible for the target to always run, without initialization, after the reset button has been pressed. This mode is selected with the **reset run** command. This conveniently allows the target to be connected to the JTAG debugger, and be able to reset it with the reset button, without being required to always type '**go**' every time. Thereafter, invoking the **reset** command will repeat the previous reset style. Also in this mode, exceptions will be handled by board software, rather than causing the JTAG debugger to halt the CPU.

Suitably configured RAM applications can be loaded either via GDB, or directly via the telnet CLI. For example:

```
Core#0>load 0x20008000 /test.bin bin
Loading /test.bin , please wait ....
Loading program file passed
Core#0>go 0x20008000
```

Consult the BDI2000 documentation for information on other formats.

Configuration of RAM applications

If the JTAG device has initialized the SDRAM, such as by using the `bdi2000.dnp_sk23.cfg` configuration on the BDI2000, RAM applications can be loaded directly into SDRAM without requiring a ROM monitor. This loading can be done directly through the JTAG device, or where supported by the JTAG device, through GDB.

In order to configure the application to support this mode, some configuration settings are required. Firstly `CYGSEM_HAL_USE_ROM_MONITOR` must be disabled. This will also cause the `CYGBLD_HAL_ARM9_ATMEL_AT91RM9200_KITS_LOAD_LOW_RAM` configuration option to be enabled allowing the application to be built with a set of memory layout files that will configure the linker script to set the program load address to be within the physical SDRAM space. Secondly the `CYGDBG_HAL_DIAG_TO_DEBUG_CHAN` option should be enabled in order to prevent HAL diagnostic output being encoded into GDB ($O) packets.

Running RAM applications

Once loaded and running via JTAG, HAL diagnostic output will appear by default on the serial debug port. USART 1 can be chosen instead by setting the `CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL` configuration option in the platform HAL to channel 1.

Warning

If resetting the board using the JTAG device, such as by using the BDI2000 **reset** command, the Ethernet PHY fails to interface correctly with the AT91RM9200, and consequently all subsequent ethernet operations are impossible. Only a reset by pressing the reset button, cycling the power or due to a watchdog timeout will cause the PHY to reset correctly.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the DNP/9200 and DNP/EVA9 hardware, and should be read in conjunction with that specification. The DNP/9200 with DNP/EVA9 platform HAL package complements the ARM architectural HAL, the ARM9 variant HAL and the AT91RM9200 processor HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset, the HAL will initialize or reinitialize most of the on-chip peripherals. There is an exception for RAM startup applications which depend on a ROM monitor for certain services.

For ROM or ROMRAM startup, the HAL will perform additional initialization, setting up the external RAM and programming the various internal registers. This is all done in the `PLATFORM_SETUP1` macro in the assembler header file `hal_platform_setup.h`.

Linker Scripts and Memory Maps

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

Flash

This is located at address 0x10000000 of the physical memory space. The HAL uses the MMU to locate it at virtual address 0x60000000 after initialization. It remains accessible at address 0x1000000 but accesses to this address range are uncached.

SDRAM

This is located at address 0x20000000 of the physical memory space. The HAL configures the MMU to retain the SDRAM at virtual address 0x20000000, but in order to assign hardware exception vectors at address 0x00000000, the HAL also uses the MMU to create a clone of this memory at virtual address 0x00000000. The same memory is also accessible uncached and unbuffered at virtual location 0x30000000 for use by devices. The first 32 bytes are used for hardware exception vectors. The next 32 bytes are used for the VSR table and the next 256 bytes are normally used for the eCos virtual vectors, allowing RAM-based applications to use services provided by the ROM monitor. Memory is required for the MMU tables, and must be aligned on a 16Kbyte boundary. These therefore occupy memory from 0x4000 to 0x8000. For ROM/ROMRAM startup, all remaining SDRAM is available. For RAM startup, available RAM starts at virtual location 0x00100000, with the bottom 1MB reserved for use by RedBoot.

On-chip SRAM

This is located at address 0x00200000 of the physical memory space. However the HAL uses the MMU to relocate this to virtual address 0x70000000. The same memory is also accessible uncached and unbuffered at virtual location 0x70100000 for use by devices. At present this memory is entirely reserved for use by the ethernet interface, since there are problems using external SDRAM for ethernet buffers.

On-chip ROM

This is located at address 0x00100000 of the physical memory space. However the HAL uses the MMU to relocate this to virtual address 0x71000000. The same memory is also accessible uncached and unbuffered at virtual location 0x71800000.

USB host port

The USB host port (UHP) registers are located at address 0x00300000 of the physical memory space. However the HAL uses the MMU to relocate this to virtual address 0x72800000. Memory accessed at this address is uncached and unbuffered. There is no cached variant.

On-chip Peripheral Registers	These are located at address 0xFF000000 in the physical memory space. When the MMU is enabled, it sets up a direct, uncached, unbuffered mapping so that these registers remain accessible at their physical locations.
Off-chip Peripherals	eCos uses the SDRAM, parallel NOR flash, ethernet PHY of the DNP/9200 board. eCos does not currently make any use of any other off-chip peripherals present on this board.

Advanced Interrupt Controller

This port has been designed to exploit benefits of the Advanced Interrupt Controller of the AT91RM9200, using the facilities of the AT91RM9200 processor HAL. Consult the documentation in that package for details.

Real-time characterization

The tm_basic kernel test gives statistics gathered about the real-time characterization and performance of the kernel. The sample output is shown here for information. The test was built in ARM mode, which provided better performance than Thumb mode.

Example 253.1. dnp_sk23 Real-time characterization

```
            Startup, main stack : stack used   416 size  3920
             Startup :  Interrupt stack used   524 size  4096
             Startup : Idlethread stack used    92 size  2048

eCos Kernel Timings
Notes: all times are in microseconds (.000001) unless otherwise stated

Reading the hardware clock takes 0 'ticks' overhead
... this value will be factored out of all other measurements
Clock interrupt took   13.69 microseconds (6 raw clock ticks)

Testing parameters:
   Clock samples:            32
   Threads:                  64
   Thread switches:         128
   Mutexes:                  32
   Mailboxes:                32
   Semaphores:               32
   Scheduler operations:    128
   Counters:                 32
   Flags:                    32
   Alarms:                   32


                                 Confidence
     Ave     Min     Max     Var  Ave  Min  Function
  ======  ======  ======  ====== ========== ========
   13.99    8.55   19.23    2.40   43%  26% Create thread
    2.10    0.00    4.27    0.13   95%   3% Yield thread [all suspended]
    2.64    2.14    6.41    0.78   78%  78% Suspend [suspended] thread
    2.90    2.14    4.27    0.98   64%  64% Resume thread
    3.71    2.14   10.68    0.98   67%  31% Set priority
    0.93    0.00    2.14    1.05   56%  56% Get priority
    8.11    6.41   19.23    0.96   70%  28% Kill [suspended] thread
    2.10    0.00    4.27    0.13   95%   3% Yield [no other] thread
    4.17    2.14    6.41    0.25   92%   6% Resume [suspended low prio] thread
    2.94    2.14    6.41    1.03   64%  64% Resume [runnable low prio] thread
    3.60    2.14    6.41    0.96   65%  32% Suspend [runnable] thread
    2.14    0.00    4.27    0.07   96%   1% Yield [only low prio] thread
    2.77    2.14    4.27    0.89   70%  70% Suspend [runnable->not runnable]
    7.68    6.41   19.23    1.23   50%  48% Kill [runnable] thread
    6.51    6.41   12.82    0.19   98%  98% Destroy [dead] thread
   10.75    8.55   21.37    0.46   87%   7% Destroy [runnable] thread
   15.19   12.82   25.64    0.57   90%   3% Resume [high priority] thread
```

```
    5.57    4.27   10.68    1.06   58%  40% Thread switch

    0.48    0.00    2.14    0.75   77%  77% Scheduler lock
    1.74    0.00    2.14    0.65   81%  18% Scheduler unlock [0 threads]
    1.78    0.00    2.14    0.59   83%  16% Scheduler unlock [1 suspended]
    1.78    0.00    2.14    0.59   83%  16% Scheduler unlock [many suspended]
    1.74    0.00    2.14    0.65   81%  18% Scheduler unlock [many low prio]

    0.80    0.00    2.14    1.00   62%  62% Init mutex
    2.60    2.14    6.41    0.76   81%  81% Lock [unlocked] mutex
    3.21    2.14    8.55    1.20   96%  56% Unlock [locked] mutex
    2.54    2.14    4.27    0.65   81%  81% Trylock [unlocked] mutex
    2.34    2.14    4.27    0.36   90%  90% Trylock [locked] mutex
    0.47    0.00    2.14    0.73   78%  78% Destroy mutex
   13.09   12.82   19.23    0.50   93%  93% Unlock/Lock mutex

    1.40    0.00    2.14    0.96   65%  34% Create mbox
    0.47    0.00    2.14    0.73   78%  78% Peek [empty] mbox
    3.00    2.14    4.27    1.03   59%  59% Put [first] mbox
    0.67    0.00    2.14    0.92   68%  68% Peek [1 msg] mbox
    3.00    2.14    4.27    1.03   59%  59% Put [second] mbox
    0.80    0.00    2.14    1.00   62%  62% Peek [2 msgs] mbox
    3.07    2.14    4.27    1.05   56%  56% Get [first] mbox
    3.14    2.14    6.41    1.13   56%  56% Get [second] mbox
    2.60    2.14    4.27    0.73   78%  78% Tryput [first] mbox
    2.80    2.14    4.27    0.92   68%  68% Peek item [non-empty] mbox
    3.00    2.14    6.41    1.08   62%  62% Tryget [non-empty] mbox
    2.54    2.14    4.27    0.65   81%  81% Peek item [empty] mbox
    2.47    2.14    4.27    0.56   84%  84% Tryget [empty] mbox
    0.80    0.00    2.14    1.00   62%  62% Waiting to get mbox
    1.00    0.00    2.14    1.06   53%  53% Waiting to put mbox
    3.34    2.14    6.41    1.13   50%  46% Delete mbox
    6.61    6.41   12.82    0.38   96%  96% Put/Get mbox

    0.73    0.00    2.14    0.96   65%  65% Init semaphore
    2.20    0.00    4.27    0.26   90%   3% Post [0] semaphore
    2.40    2.14    4.27    0.47   87%  87% Wait [1] semaphore
    2.20    2.14    4.27    0.13   96%  96% Trywait [0] semaphore
    1.34    0.00    2.14    1.00   62%  37% Trywait [1] semaphore
    0.87    0.00    2.14    1.03   59%  59% Peek semaphore
    0.67    0.00    2.14    0.92   68%  68% Destroy semaphore
    8.75    8.55   14.96    0.38   96%  96% Post/Wait semaphore

    1.20    0.00    2.14    1.05   56%  43% Create counter
    0.40    0.00    2.14    0.65   81%  81% Get counter value
    0.67    0.00    2.14    0.92   68%  68% Set counter value
    2.60    2.14    4.27    0.73   78%  78% Tick counter
    1.00    0.00    2.14    1.06   53%  53% Delete counter

    0.73    0.00    2.14    0.96   65%  65% Init flag
    2.34    2.14    6.41    0.37   93%  93% Destroy flag
    1.87    0.00    2.14    0.47   87%  12% Mask bits in flag
    2.40    2.14    4.27    0.47   87%  87% Set bits in flag [no waiters]
    3.40    2.14    6.41    1.11   53%  43% Wait for flag [AND]
    3.27    2.14    4.27    1.06   53%  46% Wait for flag [OR]
    3.47    2.14    6.41    1.08   56%  40% Wait for flag [AND/CLR]
    3.40    2.14    6.41    1.11   53%  43% Wait for flag [OR/CLR]
    0.47    0.00    2.14    0.73   78%  78% Peek on flag

    1.47    0.00    4.27    1.01   62%  34% Create alarm
    4.41    2.14    8.55    0.38   90%   3% Initialize alarm
    2.27    2.14    4.27    0.25   93%  93% Disable alarm
    4.21    2.14    8.55    0.39   87%   9% Enable alarm
    2.67    2.14    4.27    0.80   75%  75% Delete alarm
    2.74    2.14    4.27    0.86   71%  71% Tick counter [1 alarm]
   14.82   12.82   17.09    0.38   87%   9% Tick counter [many alarms]
    4.87    4.27    6.41    0.86   71%  71% Tick & fire counter [1 alarm]
   86.14   85.47   87.61    0.92   68%  68% Tick & fire counters [>1 together]
   16.89   14.96   19.23    0.48   84%  12% Tick & fire counters [>1 separately]
   10.77   10.68   14.96    0.16   96%  96% Alarm latency [0 threads]
   12.85   12.82   17.09    0.06   99%  99% Alarm latency [2 threads]
   14.06   10.68   17.09    1.21   46%   0% Alarm latency [many threads]
   21.45   21.37   32.05    0.16   99%  99% Alarm -> thread resume latency

    2.20    2.14    6.41    0.00            Clock/interrupt latency

    6.18    4.27   10.68    0.00            Clock DSR latency

   12       0     296  (main stack:  1392)  Thread stack used (1360 total)
           All done, main stack : stack used  1392 size  3920
            All done :  Interrupt stack used   208 size  4096
            All done : Idlethread stack used   276 size  2048

Timing complete - 30200 ms total

PASS:<Basic timing OK>
EXIT:<done>
```

Other Issues

The DNP/9200 with DNP/EVA9 platform HAL does not affect the implementation of other parts of the eCos HAL specification. The AT91RM9200 processor HAL, ARM9 variant HAL, and the ARM architectural HAL documentation should be consulted for further details.

Chapter 254. KwikByte KB920x Board Family Support

Name

eCos Support for the KB920x Board Family — Overview

Description

This document covers the KwikByte KB920x family of single board computers based on the Atmel AT91RM9200. The KB9200 contains the AT91RM9200 processor, 32Mb of SDRAM, 2MB of flash memory, an Intel LXT971 PHY and external connections for one serial channel, ethernet and the various other peripherals supported by the AT91RM9200. The KB9201 is similar but with an additional SPI dataflash. The KB9202 uses a different flash memory device and increases the flash memory capacity to 16MB. The KB9202B extends SDRAM to 64MB, and the KB9202C replaces the NOR flash with Dataflash.

For typical eCos development, a RedBoot image is programmed into the flash memory, and the board will boot into this image from reset. RedBoot provides gdb stub functionality so it is then possible to download and debug stand-alone and eCos applications via the gdb debugger. This can happen over either a serial line or over ethernet.

Supported Hardware

On the KB9200 and KB9201, the flash memory consists of 32 blocks of 64k bytes each. On the KB9202, the flash memory consists of 128 blocks of 128k bytes each. In a typical setup, the first two flash blocks are used for the ROMRAM RedBoot image. The topmost block is used to manage the flash and hold RedBoot **fconfig** values. The remaining blocks can be used by application code. For the KB9200/KB9201 these are 29 blocks between 0x60020000 and 0x601EFFFF; for the KB9202 these are 125 blocks between 0x60020000 and 0x60FDFFFF.

There is a serial driver `CYGPKG_IO_SERIAL_ARM_AT91` which supports both the Debug Unit and USART serial devices. These devices can be used by RedBoot for communication with the host. If either of these devices is needed by the application, either directly or via the serial driver, then it cannot also be used for RedBoot communication. Another communication channel such as ethernet should be used instead. The serial driver package is loaded automatically when configuring for the KB9200, KB9201 or KB9202 targets.

There is an ethernet driver `CYGPKG_DEVS_ETH_ARM_AT91RM9200` for the on-chip ethernet device. A second package `CYGPKG_DEVS_ETH_ARM_KB9200` is responsible for configuring this generic driver to the KB920x hardware. These drivers are also loaded automatically when configuring for the KB9200, KB9201 or KB9202 targets.

eCos manages the on-chip interrupt controller. Timer counter 0 is used to implement the eCos system clock and the microsecond delay function. Other on-chip devices (Caches, PIO, UARTs, EMAC) are initialized only as far as is necessary for eCos to run. Other devices (RTC, I²C, SPI, MCI etc.) are not touched unless the appropriate driver is loaded.

Tools

The KB920x port is intended to work with GNU tools configured for an arm-eabi target. The original port was undertaken using arm-elf-gcc version 3.3.3, arm-elf-gdb version 6.1, and binutils version 2.14, and subsequently retested with arm-elf-gcc version 3.4.3, arm-elf-gdb version 6.3 and binutils version 2.16.

Name

Setup — Preparing the KB920x boards for eCos Development

Overview

In a typical development environment, the KB920x boards boot from serial EEPROM into the KwikByte bootloader, which then boots the RedBoot ROM monitor from flash. eCos applications are configured for RAM startup and then downloaded and run on the board via the debugger **arm-eabi-gdb**. Preparing the board therefore usually involves programming a suitable RedBoot image into flash memory.

The following RedBoot configurations are supported:

Configuration	Description	Use	File
ROM	RedBoot running from ROM	`redboot_ROM.ecm`	`redboot_ROM.bin`
ROMRAM	RedBoot running from RAM, but contained in the board's flash boot sector.	`redboot_ROMRAM.ecm`	`redboot_ROMRAM.bin`
SRAM	RedBoot running from RAM, loaded by bootloader.	`redboot_SRAM.ecm`	`redboot_SRAM.bin`
UBOOT	RedBoot running from RAM, loaded from flash by U-Boot.	`redboot_UBOOT.ecm`	`redboot_UBOOT.bin`

For serial communications, all versions run with 8 bits, no parity, and 1 stop bit at 115200 baud. RedBoot also supports ethernet communication and flash management.

Generally only the ROMRAM version of RedBoot is used. The SRAM version is suitable only for RedBoot development and should not normally be used. On the KB9202C the UBOOT version is used.

Initial Installation -- KB9201, KB9202, KB9202B

The KB920x boards are shipped from KwikByte with the EEPROM bootloader installed and a version of Linux installed in the flash. The KB9202C board has a different installation method, described in the next section.

Installing RedBoot is a matter of downloading a new binary image and overwriting the existing Linux image. This is a two stage process, you must first download the KwikByte RAM Monitor, which is then used to download a RedBoot image and program it into the flash. To achieve this you will need a copy of the `ramMonitor.bin` image and, for Linux hosts, a copy of the KwikByte **download** tool. Both of these can be found on the CD-ROM that comes with the board. Note that the prebuilt version of the **download** tool is hardcoded to use `/dev/ttyS0` for downloads.

Connect a straight-through serial cable between the KB920x board serial port and a serial port of the host computer and start a terminal emulator such as **minicom** or HyperTerminal. Set the communication parameters to 115200 baud, 8 data bits, no parity bit and 1 stop bit. Press the Reset button on the board and when the bootloader starts you should see something similar to this if using the KB9200 or KB9201:

```
KB9200(www.kwikbyte.com)
        Default system configuration complete

Checking for input

>
```

When you see the "Checking for input" line, hit the Return key to put the bootloader into its command interpreter, otherwise it will proceed to boot Linux.

For the KB9202 there is less output, and the delay shorter, and so you will need to press the Return key several times immediately after pressing the Reset button before the bootloader banner is displayed:

```
KB9202(www.kwikbyte.com)

Auto boot..

>
```

The next step is to install a new startup command in the command table:

```
>s 1 e 0x10000000


>w


>
```

It is now necessary to download the Kwikbyte RAM Monitor. Start the download by giving the following command to the bootloader:

```
>x 0x20002000
C
```

You may get a sequence of C characters, which indicate that the bootloader is waiting for the download to start. If using HyperTerminal, you should now send the `ramMonitor.bin` file from the KwikByte CD-ROM using the Xmodem protocol. If using **minicom**, exit **minicom** and download the RAM Monitor using the following command:

```
$ download ramMonitor.bin
```

The **download** command produces a lot of output. When the transfer is finished, restart **minicom**.

Having downloaded the RAM monitor, issue the following command to start it:

```
>e 0x20002000

Entry: RAM Monitor

>
```

It is now necessary to prepare the flash for the RedBoot image by erasing the first 128K bytes. The RedBoot ROM image can then be downloaded:

```
>f e 0x10000000 0x1001ffff
Verifying sector erase
Flash sector erase at: 0x10000000 PASS
Verifying sector erase
Flash sector erase at: 0x10010000 PASS


>x 0x20100000
C
```

If using HyperTerminal, you should now send the `redboot_ROMRAM.bin` file using the Xmodem protocol. If using **minicom**, exit **minicom** and use the **download** command to send the file `redboot_ROMRAM.bin`. When the transfer is finished, restart **minicom**.

Having downloaded the RedBoot image, use the following command to write it to flash:

```
>f p 0x10000000 0x20100000 0x20000
..
Flash program status: PASS
```

```
>
```

The KB920x board may now be reset. It should then display the following startup sequence. Run the **fis init** and **fconfig -i** commands to initialize the flash, as shown. Note that KB9202 users will see output similar but not identical to that below.

```
KB9200(www.kwikbyte.com)
        Default system configuration complete

Checking for input
0x00 : [E]
0x01 : e 0x10000000[E]
0x02 : c 0x20210000 0x10100000 0x100000[E]
0x03 : m 0 0 0 0 0 0[E]
0x04 : t 0x20000100 console=ttyS0,115200 root=/dev/ram rw initrd=0x20210000,654]
0x05 : e 0x10000000[E]
0x06 : [E]
0x07 : [E]
0x08 : [E]
0x09 : [E]

>

>e 0x10000000
+... Read from 0x601f0000-0x601fffff to 0x01ff0000:
... Read from 0x601ff000-0x601fffff to 0x01fef000:
**Warning** FLASH configuration checksum error or invalid key
Use 'fconfig -i' to [re]initialize database
PHY: Intel LXT971
AT91RM9200 ETH: Waiting for link to come up.
AT91RM9200 ETH: 100Mb
... waiting for BOOTP information
Ethernet eth0: MAC address 00:23:31:37:00:1c
IP: 10.0.0.207/255.255.255.0, Gateway: 10.0.0.3
Default server: 10.0.0.1

RedBoot(tm) bootstrap and debug environment [ROMRAM]
Non-certified release, version UNKNOWN - built 12:01:32, Dec 13 2004

Platform: KwikByte KB9200 (ARM9)
Copyright (C) 2000, 2001, 2002, 2003, 2004 Free Software Foundation, Inc.
Copyright (C) 2003, 2004, eCosCentric Limited

RAM: 0x00000000-0x02000000, [0x0002ddf8-0x01fed000] available
FLASH: 0x60000000 - 0x601fffff 32 x 0x10000 blocks
RedBoot> fis init
About to initialize [format] FLASH image system - continue (y/n)? y
*** Initialize FLASH Image System
... Erase from 0x601f0000-0x601fffff: .
... Program from 0x01ff0000-0x02000000 to 0x601f0000: .
RedBoot> fconfig -i
Initialize non-volatile configuration - continue (y/n)? y
Run script at boot: false
Use BOOTP for network configuration: true
Default server IP address: 10.0.0.201
Network hardware address [MAC]: 0x00:0x23:0x31:0x37:0x00:0x1C
GDB connection port: 9000
Force console for special debug messages: false
Network debug at boot time: false
Default network device: at91rm9200_eth
Update RedBoot non-volatile configuration - continue (y/n)? y
... Read from 0x601f0000-0x601fefff to 0x01ff0000:
... Erase from 0x601f0000-0x601fffff: .
... Program from 0x01ff0000-0x02000000 to 0x601f0000: .
RedBoot>
```

If it proves necessary to re-install RedBoot, this may be achieved by repeating the above process. Alternatively, a new image may be downloaded and programmed into flash more directly using RedBoot's own commands. See the RedBoot documentation for details.

Initial Installation -- KB9202C

The KB9202C is not equipped with NOR flash, only Dataflash, so the installation method described above will not work. This board boots, via a second-stage loader into U-Boot. By default U-Boot then boots a Linux image that is also stored in Dataflash. The approach described here is to cause U-Boot to boot RedBoot instead of Linux. The following directions give basic instructions for installing RedBoot, for more details of how to configure U-Boot, please refer to the U-Boot documentation.

Connect a straight-through serial cable between the KB920C board serial port and a serial port of the host computer and start a terminal emulator such as **minicom** or HyperTerminal. Set the communication parameters to 115200 baud, 8 data bits, no parity bit and 1 stop bit. Power the board up and you should see the KwikByte loader and U-Boot startup messages. When it gets to the "Hit any key to stop autoboot" line, type any key. There is only a 1 second timeout here, so you need to be quick, or you can type ahead during the delay after the "NAND:" line. The final output should look like this:

```
KwikByte KB9202x Copy Loader v0.9
Loading boot loader. . . done


U-Boot 1.2.0 (Sep 26 2007 - 17:32:22)

DRAM:  64 MB
NAND:  NAND device: Manufacturer ID: 0x2c, Chip ID: 0xda (Micron NAND 256MiB 3,3V 8-bit)
256 MiB
DataFlash:AT45DB642
Nb pages:   8192
Page Size:   1056
Size= 8650752 bytes
Logical address: 0xC0000000
Area 0: C0000000 to C00020FF      Copy Loader (8kB)
Area 1: C0002100 to C00041FF      Environment (8kB)
Area 2: C0004200 to C003665F      U-Boot (195kB)
Area 3: C0036660 to C0041FFF      Secondary Boot (45kB)
Area 4: C0042000 to C0461FFF      OS (4MB)
Area 5: C0462000 to C083FFFF      DF_SPARE
In:    serial
Out:   serial
Err:   serial
Hit any key to stop autoboot:  0
KB920x >
```

Now we need to set up U-Boot to download the RedBoot image and program it into the DataFlash. First it is necessary to change U-Boot's environment to allow the image to be downloaded. In this example we will be downloading via tftp, so we need to set the ethernet and IP addresses for this board:

```
KB920x >setenv ethaddr 00:D0:93:00:05:B5
KB920x >setenv ipaddr 10.0.3.1
KB920x >setenv serverip 10.0.1.2
KB920x >saveenv
KB920x >printenv
bootargs=console=ttyS0,115200 noinitrd root=/dev/mtdblock0 rootfstype=jffs2 mem=
64M
bootdelay=1
baudrate=115200
hostname=KB9202C
kernel-size=31c8d0
bootcmd=cp.b c0042000 23000000 31c8d0; bootm 23000000
ethaddr=00:D0:93:00:05:B5
ipaddr=10.0.3.1
serverip=10.0.1.2
stdin=serial
stdout=serial
stderr=serial

Environment size: 307/8444 bytes
KB920x >
```

In the above example the `ipaddr` and `serverip` variables are set to example values. They should be set to the address of this board and the address of the TFTP server in your own network. The `redboot_UBOOT.bin` should be copied to the tftp server's root directory. It may now be downloaded:

```
KB920x >tftp 0x20100000 redboot_UBOOT.bin
TFTP from server 10.0.1.2; our IP address is 10.0.3.1
Filename 'redboot_UBOOT.bin'.
Load address: 0x20100000
Loading: T ###################
done
Bytes transferred = 94236 (1701c hex)
KB920x >
```

Now program the loaded binary into flash. In the following example, the new binary is installed into dataflash at offset 0x500000. This avoids overwriting the Linux image at offset 0x42000. If the Linux image is not required, RedBoot can be programmed in its place and the addresses in the following commands adjusted to match.

```
KB920x >cp.b 0x20100000 0xc0500000 0x20000
Copy to DataFlash... done
KB920x >
```

Finally, change the default boot command to load and execute RedBoot rather than Linux. Make sure to include the backslash before the semicolon.

```
KB920x >setenv bootcmd cp.b 0xc0500000 0x20100000 0x20000\; go 0x20100000
KB920x >saveenv
Saving Environment to dataflash...
```

Press the reset button on the board and the full boot sequence should be seen:

```
KwikByte KB9202x Copy Loader v0.9
Loading boot loader. . . done


U-Boot 1.2.0 (Sep 26 2007 - 17:32:22)

DRAM:  64 MB
NAND:  NAND device: Manufacturer ID: 0x2c, Chip ID: 0xda (Micron NAND 256MiB 3,3V 8-bit)
256 MiB
DataFlash:AT45DB642
Nb pages:   8192
Page Size:   1056
Size= 8650752 bytes
Logical address: 0xC0000000
Area 0: C0000000 to C00020FF      Copy Loader (8kB)
Area 1: C0002100 to C00041FF      Environment (8kB)
Area 2: C0004200 to C003665F      U-Boot (195kB)
Area 3: C0036660 to C0041FFF      Secondary Boot (45kB)
Area 4: C0042000 to C0461FFF      OS (4MB)
Area 5: C0462000 to C083FFFF      DF_SPARE
In:    serial
Out:   serial
Err:   serial
Hit any key to stop autoboot:  0
.O+AT91RM9200_ETH - Warning! ESA unknown.
AT91RM9200 ETH: Waiting for link to come up.
AT91RM9200 ETH: 100Mb
Ethernet eth0: MAC address 00:23:31:37:00:1c
IP: 10.0.2.6/255.0.0.0, Gateway: 10.0.0.3
Default server: 0.0.0.0, DNS server IP: 10.0.0.1

RedBoot(tm) bootstrap and debug environment [UBOOT]
Non-certified release, version UNKNOWN - built 17:30:00, Jan 11 2008

Platform: KwikByte KB9202C (ARM9)
Copyright (C) 2000, 2001, 2002, 2003, 2004 Free Software Foundation, Inc.
Copyright (C) 2003, 2004, 2005, 2006, 2007 eCosCentric Limited

RAM: 0x00000000-0x04000000, [0x00120840-0x04000000] available
RedBoot>
```

If it is necessary to reinstall RedBoot, the above steps for downloading and programming the new image should be repeated.

Rebuilding RedBoot

Should it prove necessary to rebuild a RedBoot binary, this is done most conveniently at the command line. The steps needed to rebuild the the ROMRAM version of RedBoot for the KB9200 are:

```
$ mkdir redboot_kb9200_romram
$ cd redboot_kb9200_romram
$ ecosconfig new kb9200 redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/arm/arm9/kb9200/current/misc/redboot_ROMRAM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

The steps needed to rebuild the the ROMRAM version of RedBoot for the KB9202 are:

```
$ mkdir redboot_kb9202_romram
$ cd redboot_kb9202_romram
$ ecosconfig new kb9202 redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/arm/arm9/kb9200/current/misc/redboot_ROMRAM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

The steps needed to rebuild the the UBOOT version of RedBoot for the KB9202C are:

```
$ mkdir redboot_kb9202c_uboot
$ cd redboot_kb9202c_uboot
$ ecosconfig new kb9202c redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/arm/arm9/kb9200/current/misc/redboot_UBOOT.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

At the end of the build the `install/bin` subdirectory should contain the file `redboot.bin`.

Name

Configuration — Platform-specific Configuration Options

Overview

The KB920x platform HAL package is loaded automatically when eCos is configured for a `kb9200` or `kb9202` targets. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The KB920x platform HAL package supports four separate startup types:

RAM
: This is the startup type which is normally used during application development. The board has RedBoot programmed into flash and boots into that initially. arm-eabi-gdb is then used to load a RAM startup application into memory and debug it. It is assumed that the hardware has already been initialized by RedBoot. By default the application will use the eCos virtual vectors mechanism to obtain certain services from RedBoot, including diagnostic output.

ROM
: This startup type can be used for finished applications which will be programmed into flash at physical address 0x10000000. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization.

ROM-RAM
: This startup type can be used for finished applications which will be programmed into flash at physical location 0x10000000. However, when it starts up the application will first copy itself to RAM at 0x00000000 and then run from there. RAM is generally faster than flash memory, so the program will run more quickly than a ROM-startup application. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization.

SRAM
: This startup type is used for applications that are downloaded via the KwikByte bootloader directly to RAM. The application is loaded into SDRAM at location 0x20000000 and started by executing from that address. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform limited hardware initialization since it is assumed that the machine has been set up already by the bootloader.

  This configuration is primarily present as a result of the development process. It has some limitations with regard to functionality since the MMU is not enabled and no exception vectors are installed at location zero, hence no interrupts can be handled.

RedBoot and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building RedBoot.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then the serial port will be claimed for HAL diagnostics.

Flash Driver

The KB9200 and KB9201 boards contain a 2Mb AMD Am29LV017D flash device. The `CYGPKG_DEVS_FLASH_AMD_AM29XXXXX` package contains all the code necessary to support this part and the `CYGPKG_DEVS_FLASH_KB9200` package contains definitions that customize the driver to the KB9200 board.

The KB9202 board contains a 16Mb Intel StrataFLASH 28F128J3 flash device. The `CYGPKG_DEVS_FLASH_STRATA_V2` package contains all the code necessary to support this part and the platform HAL contains definitions that customize the driver to the KB9202 board.

Ethernet Driver

The KB920x boards use the AT91RM9200's internal EMAC ethernet device attached to an external Intel LXT971 PHY. The `CYGPKG_DEVS_ETH_ARM_AT91RM9200` package contains all the code necessary to support this device and the `CYGPKG_DEVS_ETH_ARM_KB9200` package contains definitions that customize the driver to the KB920x boards.

System Clock

By default, the system clock interrupts once every 10ms, corresponding to a 100Hz clock. This can be changed by the configuration option `CYGNUM_HAL_RTC_DENOMINATOR` which corresponds to the clock frequency. Other clock-related settings are recalculated automatically if the denominator is changed.

Compiler Flags

The platform HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos. There is just one flag specific to this port:

`-mcpu=arm9`

The arm-eabi-gcc compiler supports many variants of the ARM architecture. A `-m` option should be used to select the specific variant in use, and with current tools `-mcpu=arm9` is the correct option for the ARM920T CPU in the AT91RM9200.

`-mthumb`

The arm-eabi-gcc compiler will compile C and C++ files into the Thumb instruction set when this option is used.

`-mthumb-interwork`

This option allows programs to be created that mix ARM and Thumb instruction sets. Without this option, some memory can be saved. This option should be used if -mthumb is used.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the KB920x hardware, and should be read in conjunction with that specification. The KB920x platform HAL package complements the ARM architectural HAL and the ARM9 variant HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset the HAL will initialize or reinitialize most of the on-chip peripherals. There is an exception for RAM startup applications which depend on a ROM monitor for certain services.

For ROM or ROMRAM startup, the HAL will perform additional initialization, setting up the external RAM and programming the various internal registers. This is all done in the `PLATFORM_SETUP1` macro in the assembler header file `hal_platform_setup.h`.

Linker Scripts and Memory Maps

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

Flash

This is located at address 0x10000000 of the physical memory space. The HAL uses the MMU to locate it at virtual address 0x60000000 after initialization. It remains accessible at address 0x1000000 but accesses to this address range are uncached.

SDRAM

This is located at address 0x20000000 of the physical memory space. However the HAL uses the MMU to relocate this to virtual address 0x00000000. The same memory is also accessible uncached and unbuffered at virtual location 0x20000000 for use by devices. The first 32 bytes are used for hardware exception vectors. The next 32 bytes are used for the VSR table and the next 256 bytes are normally used for the eCos virtual vectors, allowing RAM-based applications to use services provided by the ROM monitor. For ROM/ROMRAM startup, all remaining SDRAM is available. For RAM startup, available RAM starts at virtual location 0x00040000, with the bottom 256kB reserved for use by RedBoot.

On-chip SRAM

This is located at address 0x00200000 of the physical memory space. However the HAL uses the MMU to relocate this to virtual address 0x70000000. The same memory is also accessible uncached and unbuffered at virtual location 0x70100000 for use by devices. At present this memory is entirely reserved for use by the ethernet interface, since there are problems using external SDRAM for ethernet buffers.

On-chip Peripheral Registers

These are located at address 0xFF000000 in the physical memory space. When the MMU is enabled, it sets up a direct, uncached, unbuffered mapping so that these registers remain accessible at their physical locations.

Off-chip Peripherals

Apart from the SDRAM, flash and ethernet PHY, eCos does not currently make any use of the off-chip peripherals present on the KB920x boards.

Real-time characterization

The tm_basic kernel test gives statistics gathered about the real-time characterization and performance of the kernel. The sample output is shown here for information. The test was built in Thumb mode, which provided better performance than ARM mode.

Example 254.1. kb9200 Real-time characterization

```
            Startup, main stack : stack used   336 size  3920
             Startup :  Interrupt stack used   492 size  4096
             Startup : Idlethread stack used    88 size  2048

eCos Kernel Timings
Notes: all times are in microseconds (.000001) unless otherwise stated

Reading the hardware clock takes 0 'ticks' overhead
... this value will be factored out of all other measurements
Clock interrupt took   15.40 microseconds (7 raw clock ticks)

Testing parameters:
   Clock samples:            32
   Threads:                  64
   Thread switches:         128
   Mutexes:                  32
   Mailboxes:                32
   Semaphores:               32
   Scheduler operations:    128
   Counters:                 32
   Flags:                    32
   Alarms:                   32


                                 Confidence
     Ave     Min     Max     Var  Ave  Min  Function
  ======  ======  ======  ====== ========== ========
   16.70   10.67   23.47    3.01   51%  28% Create thread
    2.50    2.13    8.53    0.63   85%  85% Yield thread [all suspended]
    2.67    2.13    6.40    0.82   76%  76% Suspend [suspended] thread
    3.00    2.13    6.40    1.06   60%  60% Resume thread
    4.10    2.13   10.67    0.49   85%  12% Set priority
    1.23    0.00    2.13    1.04   57%  42% Get priority
    8.30    6.40   21.34    0.77   78%  20% Kill [suspended] thread
    2.73    2.13    8.53    0.90   75%  75% Yield [no other] thread
    4.50    4.27    8.53    0.42   90%  90% Resume [suspended low prio] thread
    2.70    2.13    6.40    0.85   75%  75% Resume [runnable low prio] thread
    4.33    2.13    8.53    0.32   89%   4% Suspend [runnable] thread
    2.70    2.13    4.27    0.83   73%  73% Yield [only low prio] thread
    2.77    2.13    4.27    0.89   70%  70% Suspend [runnable->not runnable]
    8.20    6.40   21.34    0.90   73%  25% Kill [runnable] thread
    7.40    6.40   14.93    1.15   98%  57% Destroy [dead] thread
   11.73   10.67   25.60    1.27   98%  59% Destroy [runnable] thread
   14.67   12.80   25.60    0.87   73%  23% Resume [high priority] thread
    6.72    6.40   10.67    0.54   85%  85% Thread switch

    0.55    0.00    2.13    0.82   74%  74% Scheduler lock
    1.80    0.00    2.13    0.56   84%  15% Scheduler unlock [0 threads]
    1.82    0.00    2.13    0.54   85%  14% Scheduler unlock [1 suspended]
    1.82    0.00    4.27    0.57   83%  15% Scheduler unlock [many suspended]
    1.82    0.00    4.27    0.57   83%  15% Scheduler unlock [many low prio]

    0.87    0.00    2.13    1.03   59%  59% Init mutex
    3.13    2.13    6.40    1.12   56%  56% Lock [unlocked] mutex
    3.53    2.13    6.40    1.05   59%  37% Unlock [locked] mutex
    3.13    2.13    6.40    1.12   56%  56% Trylock [unlocked] mutex
    2.60    2.13    6.40    0.76   81%  81% Trylock [locked] mutex
    0.87    0.00    2.13    1.03   59%  59% Destroy mutex
   15.13   14.93   21.34    0.38   96%  96% Unlock/Lock mutex

    1.27    0.00    4.27    1.11   53%  43% Create mbox
    0.93    0.00    2.13    1.05   56%  56% Peek [empty] mbox
    3.40    2.13    6.40    1.11   53%  43% Put [first] mbox
    0.87    0.00    2.13    1.03   59%  59% Peek [1 msg] mbox
    3.27    2.13    6.40    1.13   96%  50% Put [second] mbox
    1.07    0.00    2.13    1.07  100%  50% Peek [2 msgs] mbox
    3.40    2.13    6.40    1.11   53%  43% Get [first] mbox
    3.73    2.13    6.40    0.90   68%  28% Get [second] mbox
    3.53    2.13    6.40    1.05   59%  37% Tryput [first] mbox
    3.53    2.13    8.53    1.14   56%  40% Peek item [non-empty] mbox
    3.53    2.13    6.40    1.05   59%  37% Tryget [non-empty] mbox
    3.33    2.13    6.40    1.12   50%  46% Peek item [empty] mbox
    3.33    2.13    6.40    1.12   50%  46% Tryget [empty] mbox
    1.20    0.00    2.13    1.05   56%  43% Waiting to get mbox
    1.20    0.00    4.27    1.12   50%  46% Waiting to put mbox
    3.53    2.13    8.53    1.14   56%  40% Delete mbox
    7.60    6.40   17.07    1.42   93%  59% Put/Get mbox

    0.87    0.00    2.13    1.03   59%  59% Init semaphore
    2.33    2.13    4.27    0.36   90%  90% Post [0] semaphore
    2.87    2.13    6.40    1.01   68%  68% Wait [1] semaphore
    2.60    2.13    6.40    0.76   81%  81% Trywait [0] semaphore
    2.27    2.13    4.27    0.25   93%  93% Trywait [1] semaphore
    0.73    0.00    2.13    0.96   65%  65% Peek semaphore
    1.00    0.00    4.27    1.12   56%  56% Destroy semaphore
    9.07    8.53   17.07    0.90   84%  84% Post/Wait semaphore

    1.20    0.00    4.27    1.12   50%  46% Create counter
    0.87    0.00    2.13    1.03   59%  59% Get counter value
    0.60    0.00    2.13    0.86   71%  71% Set counter value
    3.33    2.13    6.40    1.12   50%  46% Tick counter
    1.27    0.00    2.13    1.03   59%  40% Delete counter

    0.73    0.00    2.13    0.96   65%  65% Init flag
    2.60    2.13    6.40    0.76   81%  81% Destroy flag
    2.47    2.13    4.27    0.56   84%  84% Mask bits in flag
    2.80    2.13    6.40    0.96   71%  71% Set bits in flag [no waiters]
    4.00    2.13    8.53    0.70   78%  18% Wait for flag [AND]
    3.67    2.13    8.53    1.05   62%  34% Wait for flag [OR]
    3.67    2.13    8.53    1.05   62%  34% Wait for flag [AND/CLR]
    3.67    2.13    8.53    1.05   62%  34% Wait for flag [OR/CLR]
    0.40    0.00    2.13    0.65   81%  81% Peek on flag

    0.67    0.00    4.27    0.96   71%  71% Create alarm
    4.67    4.27   10.67    0.70   87%  87% Initialize alarm
    2.73    2.13    6.40    0.90   75%  75% Disable alarm
    4.60    4.27   10.67    0.60   90%  90% Enable alarm
    3.13    2.13    6.40    1.12   56%  56% Delete alarm
    3.07    2.13    4.27    1.05   56%  56% Tick counter [1 alarm]
   16.60   14.93   17.07    0.73   78%  21% Tick counter [many alarms]
    5.07    4.27    8.53    1.05   65%  65% Tick & fire counter [1 alarm]
   87.67   87.48   89.61    0.36   90%  90% Tick & fire counters [>1 together]
   18.80   17.07   21.34    0.76   75%  21% Tick & fire counters [>1 separately]
   12.97   12.80   32.00    0.33   98%  98% Alarm latency [0 threads]
   15.03   12.80   36.27    0.97   65%  18% Alarm latency [2 threads]
   15.28   12.80   40.54    1.64   39%  27% Alarm latency [many threads]
   24.09   23.47   98.14    1.20   97%  97% Alarm -> thread resume latency

    2.27    2.13   10.67    0.00            Clock/interrupt latency

    5.40    4.27   12.80    0.00            Clock DSR latency

   10       0     764  (main stack:  1328)  Thread stack used (1360 total)
           All done, main stack : stack used  1328 size  3920
            All done :  Interrupt stack used   136 size  4096
            All done : Idlethread stack used   212 size  2048

Timing complete - 29880 ms total

PASS:<Basic timing OK>
EXIT:<done>
```

Other Issues

The KB920x platform HAL does not affect the implementation of other parts of the eCos HAL specification. The ARM9 variant HAL, and the ARM architectural HAL documentation should be consulted for further details.

Chapter 255. Motorola MX1ADS/A Board Support

Name

eCos Support for the MX1ADS/A Board — Overview

Description

This document covers the Motorola MX1ADS/A single board computer based on the Motorola MC9328MX1. This platform is both hardware and software compatible with the MXLADS/A, which contains a MC9328MXL; all references to the MX1ADS/A and MC9328MX1 should also be taken to refer to the MXLADS/A and MC9328MXL except where explicitly stated.

The MX1ADS/A contains the MC9328MX1 processor, 64Mb of SDRAM, 32MB of flash memory, a CS8900A ethernet MAC, connections for two serial channels and the various other peripherals supported by the MC9328MX1. The MX1ADS/A is identical to the MXLADS/A except that in addition to the devices supported by the MXLADS/A, it contains support for Bluetooth, analogue signal processing and SIM cards. However, since eCos does not use these devices, there is no difference in the support available.

For typical eCos development, a RedBoot image is programmed into the flash memory, and the board will boot this image from reset. RedBoot provides gdb stub functionality so it is then possible to download and debug stand-alone and eCos applications via the gdb debugger. This can happen over either a serial line or over ethernet.

Supported Hardware

The flash memory consists of two Am29PDL127H devices in parallel, giving 256 blocks of 128k bytes each. In a typical setup, the first flash block is used for the ROMRAM RedBoot image. The topmost block is used to manage the flash and hold RedBoot **fconfig** values. The remaining 254 blocks between 0x10020000 and 0x11FDFFFF can be used by application code.

There is a serial driver `CYGPKG_IO_SERIAL_ARM_MC9328MXL` which supports the two UART serial devices on the MC9328MX1. This is configured for the MX1ADS/A by the `CYGPKG_IO_SERIAL_ARM_MX1ADS_A` package. These devices can be used by RedBoot for communication with the host. If either of these devices is needed by the application, either directly or via the serial driver, then it cannot also be used for RedBoot communication. Another communication channel such as ethernet should be used instead. The serial driver package is loaded automatically when configuring for the MX1ADS/A target.

There is an ethernet driver `CYGPKG_DEVS_ETH_CL_CS8900A` for the Cirrus Logic CS8900A ethernet device. A second package `CYGPKG_DEVS_ETH_ARM_MX1ADS_A` is responsible for configuring this generic driver to the MX1ADS/A hardware. These drivers are also loaded automatically when configuring for the MX1ADS/A target.

eCos manages the on-chip interrupt controller. General Purpose Timer 1 is used to implement the eCos system clock and the microsecond delay function. The Watchdog Timer is also supported. Other on-chip devices (Caches, GPIO, UARTs, memory and interrupt controllers) are initialized only as far as is necessary for eCos to run. Other devices (SPI, I^{2}C, RTC etc.) are not touched.

Tools

The MX1ADS/A port is intended to work with GNU tools configured for an arm-eabi target. The original port was undertaken using arm-elf-gcc version 3.3.3, arm-elf-gdb version 6.1, and binutils version 2.14.

Name

Setup — Preparing the MX1ADS/A board for eCos Development

Overview

In a typical development environment, the MX1ADS/A board boots from flash into the RedBoot ROM monitor. eCos applications are configured for RAM startup and then downloaded and run on the board via the debugger **arm-eabi-gdb**. Preparing the board therefore usually involves programming a suitable RedBoot image into flash memory.

The following RedBoot configurations are supported:

Configuration	Description	Use	File
SRAM	RedBoot running from RAM, but loaded via the JTAG interface	`redboot_SRAM.ecm`	`redboot_SRAM.bin`
RAM	RedBoot running from RAM, usually loaded by another version of RedBoot	`redboot_RAM.ecm`	`redboot_RAM.bin`
ROM	RedBoot running from ROM	`redboot_ROM.ecm`	`redboot_ROM.bin`
ROMRAM	RedBoot running from RAM, but contained in the board's flash boot sector	`redboot_ROMRAM.ecm`	`redboot_ROMRAM.bin`

For serial communications, all versions run with 8 bits, no parity, and 1 stop bit at 38400 baud. RedBoot also supports ethernet communication and flash management.

Initial Installation

Installing RedBoot is a matter of downloading a new binary image and overwriting the existing Boot monitor ROM image.

There are two possible mechanisms for doing this, both via the JTAG interface. The first uses the ability of some JTAG debuggers to write directly to FLASH. The second merely uses the JTAG debugger to load a version of RedBoot and to then use that to program the FLASH.

Direct FLASH Programming

The following instructions describe how to install RedBoot by programming the FLASH directly from the JTAG debugger. At present this has only been tried using an Abatron BDI2000 debugger and these instruction apply to that device. However, it should be possible to adapt these instructions to any other device.

The BDI2000 configuration file needs to be set up to initialize the SDRAM and to tell it what type of FLASH device is present. The resulting config file is shown below:

```
; bdiGDB configuration for Motorola M9328MX1ADS board
; --------------------------------------------------
;
[INIT]
;Init SDRAM 16Mx16x2 IAM0 CS2 CL2
;
WM32    0x00221000  0x92120200  ;Set Precharge Command
WM32    0x08200000  0x00000000  ;Issue Precharge all Command
WM32    0x00221000  0xa2120200  ;Set AutoRefresh Command
WM32    0x08000000  0x00000000  ;Issue AutoRefresh Command
```

```
WM32    0x08000000  0x00000000
WM32    0x08000000  0x00000000
WM32    0x08000000  0x00000000
WM32    0x08000000  0x00000000
WM32    0x08000000  0x00000000
WM32    0x08000000  0x00000000
WM32    0x08000000  0x00000000
WM32    0x00221000  0xb2120200  ;Set Mode Register
WM32    0x08111800  0x00000000  ;Issue Mode Register Command, Burst Length = 8
WM32    0x00221000  0x82124200  ;Set to Normal Mode
;

[TARGET]
CPUTYPE     ARM920T
CLOCK       1                   ;JTAG clock (0=Adaptive, 1=8MHz, 2=4MHz, 3=2MHz)
WAKEUP      3000                ;because of slow rising reset line
RESET       HARD 1000           ;because of heavy capacitive load on reset line
ENDIAN      LITTLE              ;memory model (LITTLE | BIG)
BREAKMODE   HARD
VECTOR      CATCH 0x1f          ;catch D_Abort, P_Abort, SWI, Undef and Reset

[HOST]

[FLASH]
; Program RedBoot with:
; Core#0> erase
; Core#0> prog 0x10000000 MX1 BIN
WORKSPACE   0x08000000
CHIPTYPE    AM29DX32
CHIPSIZE    0x01000000
BUSWIDTH    32
ERASE       0x10000000
ERASE       0x10004000
ERASE       0x10008000
ERASE       0x1000c000
ERASE       0x10010000
ERASE       0x10014000
ERASE       0x10018000
ERASE       0x1001c000
ERASE       0x10020000


[REGS]
FILE    regMX1.def
```

The BDI2000 needs to be rebooted to cause it to reload this configuration file. Once this is done connect to the BDI2000 via its telnet port and issue a reset command:

```
Core#0>reset
- TARGET: processing reset request
- TARGET: BDI asserts TRST and RESET
- TARGET: BDI removes TRST
- TARGET: Bypass check 0x000000001 => 0x00000001
- TARGET: JTAG exists check passed
- Core#0: ID code is 0x1092001D
- TARGET: All ICEBreaker access checks passed
- TARGET: BDI removes RESET
- TARGET: BDI waits for RESET inactive
- TARGET: resetting target passed
- TARGET: processing target startup ....
- TARGET: processing target startup passed
Core#0>
```

Now ensure that the FLASH is erased. The following command uses the `ERASE` entries in the configuration file to erase the first 9 blocks in the FLASH.

```
Core#0>erase
```

```
Erasing flash at 0x10000000
Erasing flash at 0x10004000
Erasing flash at 0x10008000
Erasing flash at 0x1000c000
Erasing flash at 0x10010000
Erasing flash at 0x10014000
Erasing flash at 0x10018000
Erasing flash at 0x1001c000
Erasing flash at 0x10020000
Erasing flash passed
Core#0>
```

Copy `redboot_ROMRAM.bin` to the root directory of the same TFTP server used to fetch the configuration file and execute the following command:

```
Core#0>prog 0x10000000 redboot_ROMRAM.bin bin
Programming redboot_ROMRAM.bin , please wait ....
Programming flash passed
Core#0>
```

If this completes successfully then the FLASH has been programmed. You can start RedBoot by issuing the **go** command, or by detaching the BDI2000 and cycling the power switch of the board. You should see the RedBoot startup screen:

```
+... waiting for BOOTP information
Ethernet eth0: MAC address 0e:00:00:ea:18:f0
IP: 10.0.0.207/255.255.255.0, Gateway: 10.0.0.3
Default server: 10.0.0.1, DNS server IP: 10.0.0.1

RedBoot(tm) bootstrap and debug environment [ROMRAM]
Non-certified release, version UNKNOWN - built 15:51:18, Jul 19 2004

Platform: Motorola MX1ADS/A (ARM9)
Copyright (C) 2000, 2001, 2002, Free Software Foundation, Inc.
Copyright (C) 2003, 2004, eCosCentric Limited

RAM: 0x00000000-0x02000000, [0x0002f778-0x01fdd000] available
FLASH: 0x10000000 - 0x12000000, 256 blocks of 0x00020000 bytes each.
RedBoot>
```

Download RedBoot

The following instructions describe how to install RedBoot via the JTAG interface by downloading a version of RedBoot to program the FLASH. This is a two stage process, you must first download a RAM-resident version of RedBoot and then use that to download the ROM image to be programmed into the flash memory. The following directions are necessarily somewhat general since the specifics depend on the exact JTAG device available, and the software used to drive it.

Connect the JTAG device to the JTAG connector on the MX1ADS/A board and check that the device is functioning correctly. Using 32 bit memory writes, initialize the static memory controller so that the SDRAM and flash are accessible. The following assignments should be made:

```
*(long *)0x00221000 = 0x92120200;  // Set Precharge Command
*(long *)0x08200000 = 0x00000000;  // Issue Precharge all Command
*(long *)0x00221000 = 0xa2120200;  // Set AutoRefresh Command
*(long *)0x08000000 = 0x00000000;  // Issue AutoRefresh Command
*(long *)0x08000000 = 0x00000000;
*(long *)0x08000000 = 0x00000000;
*(long *)0x08000000 = 0x00000000;
*(long *)0x08000000 = 0x00000000;
*(long *)0x08000000 = 0x00000000;
*(long *)0x08000000 = 0x00000000;
*(long *)0x08000000 = 0x00000000;
*(long *)0x00221000 = 0xb2120200;  // Set Mode Register
*(long *)0x08111800 = 0x00000000;  // Issue Mode Register Command, Burst Length = 8
*(long *)0x00221000 = 0x82124200;  // Set to Normal Mode
```

Now load the SRAM redboot binary image from the file `redboot_SRAM.bin` into SDRAM at 0x08040000. Exactly how you do this depends on the JTAG driver software. Note that it may be easier to load the ELF or SREC files, if supported, since these contain the correct load addresses.

Connect the serial port of a host machine to UART 1 on the MX1ADS/A board and start a terminal emulator (for example **HyperTerminal** on Windows, **minicom** on Linux) set up to communicate at 38400 baud, 8 bits, one stop bit, no parity. Start RedBoot by executing from location 0x08040000, which should result in RedBoot starting up and emitting this message on the serial channel:

```
+... waiting for BOOTP information
Ethernet eth0: MAC address 0e:00:00:ea:18:f0
IP: 10.0.0.207/255.255.255.0, Gateway: 10.0.0.3
Default server: 10.0.0.1, DNS server IP: 10.0.0.1

RedBoot(tm) bootstrap and debug environment [SRAM]
Non-certified release, version UNKNOWN - built 15:50:10, Jul 19 2004

Platform: Motorola MX1ADS/A (ARM9)
Copyright (C) 2000, 2001, 2002, Free Software Foundation, Inc.
Copyright (C) 2003, 2004, eCosCentric Limited

RAM: 0x08000000-0x0c000000, [0x08065ea0-0x0bfdd000] available
FLASH: 0x10000000 - 0x12000000, 256 blocks of 0x00020000 bytes each.
RedBoot>
```

Now the ROM image can be downloaded using the following RedBoot command:

```
RedBoot> load -r -b %{FREEMEMLO} -m ymodem
```

Use the terminal emulator's Ymodem support to send the file `redboot_ROMRAM.bin`. This should result in something like the following output:

```
Raw file loaded 0x08066000-0x080b900d, assumed entry at 0x08066000
xyzModem - CRC mode, 2659(SOH)/0(STX)/0(CAN) packets, 5 retries
RedBoot>
```

Once the file has been uploaded, you can check that it has been transferred correctly using the **cksum** command. On the host (Linux or Cygwin) run the **cksum** program on the binary file:

```
$ cksum redboot_ROMRAM.bin
3848755608 118224 redboot_ROMRAM.bin
```

In RedBoot, run the **cksum** command on the data that has just been loaded:

```
RedBoot> cksum -b %{FREEMEMLO} -l 118224
POSIX cksum = 3848755608 118224 (0xe5675998 0x0001cdd0)
```

The second number in the output of the host **cksum** program is the file size, which should be used as the argument to the `-l` option in the RedBoot **cksum** command. The first numbers in each instance are the checksums, which should be equal.

If the program has downloaded successfully, then it can be programmed into the flash using the following commands:

```
RedBoot> fis init
About to initialize [format] FLASH image system - continue (y/n)?y
*** Initialize FLASH Image System
... Erase from 0x11fe0000-0x12000000: .
... Program from 0x0bfe0000-0x0c000000 at 0x11fe0000: .
RedBoot> fis create -b %{FREEMEMLO} RedBoot
An image named 'RedBoot' exists - continue (y/n)?y
... Erase from 0x10000000-0x10020000: .
... Program from 0x08066000-0x08086000 at 0x10000000: .
... Erase from 0x11fe0000-0x12000000: .
... Program from 0x0bfe0000-0x0c000000 at 0x11fe0000: .
RedBoot>
```

The MX1ADS/A board may now be disconnected from the JTAG device and reset by cycling the power. It should then display the startup screen for the ROMRAM version of RedBoot:

```
+... waiting for BOOTP information
Ethernet eth0: MAC address 0e:00:00:ea:18:f0
IP: 10.0.0.207/255.255.255.0, Gateway: 10.0.0.3
Default server: 10.0.0.1, DNS server IP: 10.0.0.1

RedBoot(tm) bootstrap and debug environment [ROMRAM]
Non-certified release, version UNKNOWN - built 15:51:18, Jul 19 2004

Platform: Motorola MX1ADS/A (ARM9)
Copyright (C) 2000, 2001, 2002, Free Software Foundation, Inc.
Copyright (C) 2003, 2004, eCosCentric Limited

RAM: 0x00000000-0x02000000, [0x0002f778-0x01fdd000] available
FLASH: 0x10000000 - 0x12000000, 256 blocks of 0x00020000 bytes each.
RedBoot>
```

Rebuilding RedBoot

Should it prove necessary to rebuild a RedBoot binary, this is done most conveniently at the command line. The steps needed to rebuild the the ROMRAM version of RedBoot for the MX1ADS/A are:

```
$ mkdir redboot_mx1ads_a_romram
$ cd redboot_mx1ads_a_romram
$ ecosconfig new mx1ads_a redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/arm/arm9/mx1ads_a/current/misc/redboot_ROMRAM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

To rebuild the SRAM version of RedBoot:

```
$ mkdir redboot_mx1ads_a_sram
$ cd redboot_mx1ads_a_sram
$ ecosconfig new mx1ads_a redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/arm/arm9/mx1ads_a/current/misc/redboot_SRAM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

At the end of the build the `install/bin` subdirectory should contain the file `redboot.bin`. This is the case for both the above builds, take care not to mix the two files up, since programming the SRAM RedBoot into the ROM will render the board unbootable.

Name

Configuration — Platform-specific Configuration Options

Overview

The MX1ADS/A platform HAL package is loaded automatically when eCos is configured for a `mx1ads_a` or `mxlads_a` target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

The MC9328MX1 SoC is supported by a separate HAL, `CYGPKG_HAL_ARM_ARM9_MC9328MXL`, which supports all the devices on the MC9328MX1/L that eCos uses.

Startup

The MX1ADS/A platform HAL package supports four separate startup types:

RAM

This is the startup type which is normally used during application development. The board has RedBoot programmed into flash and boots into that initially. arm-eabi-gdb is then used to load a RAM startup application into memory and debug it. It is assumed that the hardware has already been initialized by RedBoot. By default the application will use the eCos virtual vectors mechanism to obtain certain services from RedBoot, including diagnostic output.

ROM

This startup type can be used for finished applications which will be programmed into flash at physical address 0x10000000. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization.

ROMRAM

This startup type can be used for finished applications which will be programmed into flash at physical location 0x10000000. However, when it starts up the application will first copy itself to RAM at 0x00000000 and then run from there. RAM is generally faster than flash memory, so the program will run more quickly than a ROM-startup application. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization.

SRAM

This startup type is used for applications that are downloaded via the JTAG interface. The application is loaded into SRAM at location 0x08040000 and started by executing from that address. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization. However, it is assumed that the machine has been set up from the JTAG interface as described earlier for installing RedBoot.

This configuration is primarily present to provide support for installing RedBoot in the FLASH. It has some limitations with regard to functionality since the MMU is not enabled and no exception vectors are installed at location zero, hence no interrupts can be handled.

RedBoot and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building RedBoot.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for

a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then the serial port will be claimed for HAL diagnostics.

Flash Driver

The MX1ADS/A board contains two 16 bit AMD Am29PDL127H flash devices arranged in parallel to form a 32 bit wide interface. The `CYGPKG_DEVS_FLASH_AMD_AM29XXXXX` package contains all the code necessary to support these parts and the `CYGPKG_DEVS_FLASH_ARM_MX1ADS_A` package contains definitions that customize the driver to the MX1ADS/A board.

Ethernet Driver

The MX1ADS/A board contains a Cirrus Logic CS8900A ethernet MAC. The `CYGPKG_DEVS_ETH_CL_CS8900A` package contains all the code necessary to support this device and the `CYGPKG_DEVS_ETH_ARM_MX1ADS_A` package contains definitions that customize the driver to the MX1ADS/A board.

System Clock

By default, the system clock interrupts once every 10ms, corresponding to a 100Hz clock. This can be changed by the configuration option `CYGNUM_HAL_RTC_DENOMINATOR` which corresponds to the clock frequency. Other clock-related settings are recalculated automatically if the denominator is changed.

Compiler Flags

The platform HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos. There is just one flag specific to this port:

`-mcpu=arm9`
: The arm-eabi-gcc compiler supports many variants of the ARM architecture. A `-m` option should be used to select the specific variant in use, and with current tools `-mcpu=arm9` is the correct option for the ARM920T CPU in the MC9328MXL.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the MX1ADS/A hardware, and should be read in conjunction with that specification. The MX1ADS/A platform HAL package complements the ARM architectural HAL and the ARM9 variant HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset the HAL will initialize or reinitialize the on-chip peripherals that it uses. There is an exception for RAM startup applications which depend on a ROM monitor for certain services.

For ROM or ROMRAM startup, the HAL will perform additional initialization, setting up the external RAM and programming the various internal registers. This is all done in the `PLATFORM_SETUP1` macro in the assembler header file `hal_platform_setup.h`.

For SRAM startup, minimal initialization is performed, and it is assumed that the JTAG device has initialized the hardware as described earlier.

Linker Scripts and Memory Maps

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

SDRAM

This is located at address 0x08000000 of the physical memory space. However the HAL uses the MMU to relocate this to virtual address 0x00000000. The same memory is also accessible uncached and unbuffered at virtual location 0x04000000 for use by devices. The first 32 bytes are used for hardware exception vectors. The next 32 bytes are used for the VSR table and the next 256 bytes are normally used for the eCos virtual vectors, allowing RAM-based applications to use services provided by the ROM monitor. For ROM/ROMRAM startup, all remaining SDRAM is available. For RAM startup, available RAM starts at virtual location 0x00040000, with the bottom 256kB reserved for use by RedBoot.

In the SRAM startup configuration, the SDRAM remains at its physical address of 0x08000000, since the MMU is not enabled.

Flash

This is located at address 0x10000000 of the physical memory space. It is mapped by the HAL using the MMU to the same virtual address with caching and the write buffer enabled.

On-chip Peripheral Registers

These are located at address 0x00200000 in the physical memory space. When the MMU is enabled, it sets up a direct, uncached, unbuffered mapping for these at 0xA0000000 in the virtual address space.

All hardware addresses in `mc9328mxl.h` give the physical address. To use these, the macro `CYGARC_VIRTUAL_ADDRESS()` should be applied to these. This will yield the correct address depending on the startup type.

Ethernet MAC

The Cirrus Logic CS8900A is addressed by Chip Select 4, and is located at physical address 0x15000000. It is mapped uncached and unbuffered to the same virtual address when the MMU is enabled.

Other Issues

The MX1ADS/A platform HAL does not affect the implementation of other parts of the eCos HAL specification. The ARM9 variant HAL, and the ARM architectural HAL documentation should be consulted for further details.

Chapter 256. Texas Instruments OMAP L1xx Processor Support

Name

Support for the TI OMAP L1xx Processor — Overview

Description

This document covers the configuration and usage of the Hardware Abstraction Layer (HAL) for the TI OMAP L1xx processor family. It is expected to be read in conjunction with platform HAL-specific documentation, as well as the eCos HAL specification. This processor HAL package complements the ARM architectural HAL, ARM9 variant HAL and the platform HAL. It provides functionality common to all OMAP L1xx-based board implementations.

This support is found in the eCos package located at `packages/hal/arm/arm9/omap-l1xx` within the eCos source repository.

The OMAP L1xx processor HAL package is loaded automatically when eCos is configured for an OMAP-based platform. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Supported Hardware

Supported features of the TI OMAP processor within this processor HAL package include:

- OMAP L1xx-specific hardware definitions
- Interrupt controller
- Timers
- Serial UARTs
- MultiMedia Card Interface (MMC/SD)
- I²C two wire interface
- Pin Configuration and GPIO Support
- Peripheral Power Control
- DMA Support

Support for the on-chip SPI device, SPI NOR flash, interrupt-driven serial, watchdog and wallclock (RTC) features of the OMAP L1xx are also present and can be found in separate packages, outside of this processor HAL.

Name

OMAP L1xx Hardware Definitions — Details on obtaining hardware definitions for OMAP

Register definitions

The file `<cyg/hal/omap_l1xx.h>` can be included from application and eCos package sources to provide definitions related to OMAP subsystems. These include register definitions for the interrupt controller, power management controller, PLL clocks, memory controller, external bus interface, GPIO, USART, MMC/SD, Ethernet, timers RTC, and SPI subsystems. This file will normally be included automatically if `<cyg/hal/hal_io.h>` is included, which is the preferred way of getting these definitions.

Initialization Helper Macros

The file `<cyg/hal/omap_l1xx_init.inc>` contains definitions of helper macros which may be used by OMAP L1xx platform HALs in order to initialise common subsystems without excessive duplication between the platform HALs. Typically this file will be included by the `hal_platform_setup.h` header in the platform HAL, in turn included from the architectural HAL file `vectors.S`.

This file is solely intended to be used by platform HALs. At the same time, it is only present to assist initialization, and platform HALs are not obliged to use it if their startup requirements vary. NOTE: At present, the only extant OMAP L1xx port relies on either the TI-supplied User Boot Loader, or the JTAG initialization script, to initialize the PLLs and memory controller, so these macros currently largely contain ARM9-generic setup only.

Name

OMAP L1xx Interrupt Controller — Advanced Interrupt Controller Definitions And usage

Interrupt controller definitions

The file <cyg/hal/var_ints.h> (located at hal/arm/arm9/omap/*VERSION*/include/var_ints.h in the eCos source repository) contains interrupt vector number definitions for use with the eCos kernel and driver interrupt APIs.

The list of interrupt vectors may be augmented on a per-platform basis. Consult the platform HAL documentation for your platform for whether this is the case.

Interrupt Controller Functions

The source file src/omap_l1xx_misc.c within this package provides most of the support functions to manipulate the interrupt controller. The hal_IRQ_handler queries the IRQ status register to determine the interrupt cause. Functions hal_interrupt_mask and hal_interrupt_unmask enable or disable interrupts within the interrupt controller.

Interrupts are configured in the hal_interrupt_configure function. Only GPIO interrupts are configurable, and at present we do not support full decoding of these, so this function is empty.

The hal_interrupt_acknowledge function acknowledges an interrupt.

The hal_interrupt_set_level is used to set the priority level of the supplied interrupt within the Advanced Interrupt Controller. This is supported by assigning each vector to one of the 30 channels that map to the IRQ vector. The FIQ vector channels are currently ignored. The level value may range from 0 to 29, with 0 being the highest priority. This range is shifted into the 2..31 range, and any attempt to set a priority value greater than 29 is clamped to 29.

Note that in all the above, it is not recommended to call the described functions directly. Instead either the HAL macros (HAL_INTERRUPT_MASK et al) or preferably the kernel or driver APIs should be used to control interrupts.

Name

Timers — Use of on-chip timers

System Clock

The eCos kernel system clock is implemented using the 3:4 half of the 64-bit TimerPlus 0. By default, the system clock interrupts once every 10ms, corresponding to a 100Hz clock. This can be changed by the configuration option `CYGNUM_HAL_RTC_DENOMINATOR` which corresponds to the clock frequency. Other clock-related settings are recalculated automatically if the denominator is changed. If the desired frequency cannot be expressed accurately solely with changes to `CYGNUM_HAL_RTC_DENOMINATOR`, then the configuration option `CYGNUM_HAL_RTC_NUMERATOR` may also be adjusted, and again clock-related settings will automatically be recalculated.

The same Timer is also used to implement the HAL microsecond delay function, `HAL_DELAY_US`. This is used by some device drivers, and in non-kernel configurations such as with RedBoot where this timer is needed for loading program images via X/Y-modem protocols and debugging via TCP/IP. Standalone applications which require RedBoot services, such as debugging, should avoid use of this timer.

Timer-based profiling support

Timer-based profiling support is implemented using the 1:2 half of TimerPlus 0. If the gprof package, `CYGPKG_PROFILE_GPROF`, is included in the configuration, then TimerPlus0 1:2 is reserved for use by the profiler.

Name

Serial UARTs — Configuration and Implementation Details of Serial UART Support

Overview

Support is included in this processor HAL package for the OMAP's on-chip debug unit UART and up to four serial USART serial devices.

There are two forms of support: HAL diagnostic I/O; and a fully interrupt-driven serial driver. Unless otherwise specified in the platform HAL documentation, for all serial ports the default settings are 115200,8,N,1 with no flow control.

HAL Diagnostic I/O

This first form is polled mode HAL diagnostic output, intended primarily for use during debug and development. Operations are usually performed with global interrupts disabled, and thus this mode is not usually suitable for deployed systems. This can operate on any port, according to the configuration settings.

There are several configuration options usually found within a platform HAL which affect the use of this support in the OMAP processor HAL. The CDL option `CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL` selects the serial port channel to use as the console at startup time. This will be the channel that receives output from, for example, `diag_printf()`. The CDL option `CYGNUM_HAL_VIRTUAL_VECTOR_DEBUG_CHANNEL` selects the serial port channel to use for GDB communication by default. Note that when using RedBoot, these options are usually inactive as it is RedBoot that decides which channels are used. Applications may override RedBoot's selections by enabling the `CYGSEM_HAL_VIRTUAL_VECTOR_CLAIM_COMMS` CDL option in the HAL. Baud rates for each channel are set with the `CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL_BAUD` and `CYGNUM_HAL_VIRTUAL_VECTOR_DEBUG_CHANNEL_BAUD` options.

Interrupt-driven Serial Driver

The second form of support is an interrupt-driven serial driver, which is integrated into the eCos standard serial I/O infrastructure (`CYGPKG_IO_SERIAL`). This support can be enabled on any port.

Note that it is not recommended to share this driver when using the HAL diagnostic I/O on the same port. If the driver is shared with the GDB debugging port, it will prevent ctrl-c operation when debugging.

The main part of this driver is contained in the generic `CYGPKG_IO_SERIAL_GENERIC_16X5X` package. The package `CYGPKG_IO_SERIAL_ARM_OMAP_L1XX` contains definitions that configure the generic driver for the OMAP L1xx. That driver package should also be consulted for documentation and configuration options. The driver is not active until the `CYGPKG_IO_SERIAL_DEVICES` configuration option within the generic serial driver support package `CYGPKG_IO_SERIAL` is enabled in the configuration.

Support for hardware flow control and modem control lines is present in the driver, but will only be enabled if these control signals are brought out to the physical serial port.

Name

Multimedia Card Interface (MMC/SD) Driver — Using MMC/SD cards with block drivers and filesystems

Overview

The MultiMedia Card Interface (MMC/SD) driver in the OMAP processor HAL allows use of MultiMedia Cards (MMC cards) and Secure Digital (SD) flash storage cards within eCos, exported as block devices. This makes them suitable for use as the underlying devices for filesystems such as FAT.

Configuration

This driver provides the necessary support for the generic MMC bus layer within the `CYGPKG_DEVS_DISK_MMC` package to export a disk block device. The disk block device is only available if the generic disk I/O layer found in the package `CYGPKG_IO_DISK` is included in the configuration.

The block device may then be used as the device layer for a filesystem such as FAT. Example devices are "`/dev/mmcsd0/1`" to refer to the first partition on the card, or "`/dev/mmcsd0/0`" to address the whole device including potentially the partition table at the start.

The driver may be forcibly disabled within this processor HAL package with the configuration option `CYGPKG_HAL_ARM_ARM9_OMAP_L1XX_MMC`.

If the driver is enabled, the following options are available to control it:

`CYGPKG_HAL_ARM_ARM9_OMAP_L1XX_MMC_DEVICE`
The OMAP_L1XX has two MMC/SD devices. At present the generic MMC/SD code can only handle one device. This option selects which device that is.

`CYGIMP_HAL_ARM_ARM9_OMAP_L1XX_MMC_INTMODE`
This indicates that the driver should operate in interrupt-driven mode if possible. This is enabled by default if the eCos kernel is enabled. Note though that if the driver finds that global interrupts are off when running, then it will fall back to polled mode even if this option is enabled. This allows for use of the MMC/SD driver in an initialisation context.

`CYGNUM_HAL_ARM_ARM9_OMAP_L1XX_MMC_INT_PRI`
This is the MMC/SD bus interrupt priority. It may range from 1 to 29.

Usage Notes

The driver will detect the appropriate card sizes. Hotswapping of cards is supported by the driver, and in the case of eCosPro, the FAT filesystem. Although any cards removed before explicit unmounting or a `sync()` call to flush filesystem buffers will likely result in a corrupted filesystem on the removed card.

The MMC/SD bus layer will parse partition tables, although it can be configured to allow handling of cards with no partition table.

This driver implements multi-sector I/O operations. If you are using the FAT filesystem, see the generic MMC/SD driver documentation which describes how to exploit this feature when using FAT.

Name

I2C Two Wire Interface — Using I^{2}C devices

Overview

The I^{2}C driver in the OMAP processor HAL supports the use of I^{2}C devices within eCos. Access to the driver will be via the standard I^{2}C interface subsystem.

This driver provides support for both I^{2}C busses available on the OMAP L1XX.

Configuration

The HAL contains the following configuration options for the two I^{2}C busses:

`CYGINT_HAL_ARM_ARM9_OMAP_L1XX_I2C_BUSX`

This interface controls the inclusion of support for I^{2}C bus *X*. This will normally be implemented by the platform HAL to indicate that there are I^{2}C devices attached to the given bus, or that the SCL and SDA lines are routed to an external connector.

`CYGNUM_HAL_ARM_ARM9_OMAP_L1XX_I2C_BUSX_CLOCK`

This is the I^{2}C bus *X* clock speed in Hz. Usually frequencies of either 100kHz or 400kHz are chosen, the latter sometimes known as fast mode.

`CYGNUM_HAL_ARM_ARM9_OMAP_L1XX_I2C_BUSX_INTR_PRI`

This is the I^{2}C bus *X* interrupt priority. It may range from 1 to 29; the default of 15 places it in the centre of the priority range.

Additionally the HAL contains the following configuration option which applies to both I^{2}C buses:

`CYGNUM_HAL_ARM_ARM9_OMAP_L1XX_I2C_ALIGNED_RXBUF_SIZE`

When using DMA, transferred received data buffers must be aligned to the data cache line size, and its size must be a multiple of the cache line size. If the user does not pass in a suitable buffer, a bounce buffer must be used for the entire transfer. This option provides the size of that buffer. You should set this buffer to the maximum amount of data you can receive in a single transfer. If you can guarantee that all uses of this I^{2}C driver use appropriately aligned receive buffers, then you can disable this option entirely in order to remove the buffer. It defaults to 1024 bytes per I^{2}C bus.

Usage Notes

The design of the OMAP L1XX I^{2}C device does not make it possible to start a new bus transfer without also sending a START condition on the bus. This means that divided transactions are not possible. A divided transaction would look like this:

```
    cyg_i2c_transaction_begin(&cyg_aardvark_at24c02);
    cyg_i2c_transaction_tx(&cyg_aardvark_at24c02, 1, tx_buf1, 1, 0);
    cyg_i2c_transaction_tx(&cyg_aardvark_at24c02, 0, tx_buf2, 2, 0);
    cyg_i2c_transaction_tx(&cyg_aardvark_at24c02, 0, tx_buf3, 6, 1);
    cyg_i2c_transaction_end(&cyg_aardvark_at24c02);
```

In this transaction a START and one byte are sent from tx_buf1, then 2 bytes of data from tx_buf2, finishing with 6 bytes from tx_buf3 followed by a STOP. The OMAP L1XX will not allow the tx_buf2 and tx_buf3 transfers to happen without also sending a START. The only solution to this is to combine the data into a single buffer and perform a single transfer:

```
    memcpy( tx_buf, tx_buf1, 1);
    memcpy( tx_buf+1, tx_buf2, 2);
    memcpy( tx_buf+3, tx_buf3, 6);
    cyg_i2c_tx(&cyg_aardvark_at24c02, tx_buf, 9);
```

Name

Pin Configuration and GPIO Support — Use of pin configuration and GPIO

Synopsis

#include <cyg/hal/hal_io.h>

`pin = `**`CYGHWR_HAL_L1XX_PINMUX`**`(`*`reg`*`, `*`field`*`, `*`func`*`);`

`CYGHWR_HAL_L1XX_PINMUX_SET`` (`*`pin`*`);`

`pin = `**`CYGHWR_HAL_L1XX_GPIO`**`(`*`bank`*`, `*`bit`*`, `*`mode`*`);`

`CYGHWR_HAL_L1XX_GPIO_SET`` (`*`pin`*`);`

`CYGHWR_HAL_L1XX_GPIO_OUT`` (`*`pin`*`, `*`val`*`);`

`CYGHWR_HAL_L1XX_GPIO_IN`` (`*`pin`*`, `*`val`*`);`

`CYGHWR_HAL_L1XX_GPIO_INTCFG`` (`*`pin`*`, `*`mode`*`);`

`CYGHWR_HAL_L1XX_GPIO_INTSTAT`` (`*`pin`*`, `*`stat`*`);`

Description

The OMAP L1XX HAL provides a number of macros to support the encoding of pin multiplexing information and GPIO pin modes into 32 bit descriptors. This is useful to drivers and other packages that need to configure and use different lines for different devices. Because there is not a simple correspondence between pin multiplexing information and GPIO bank and pin identities, these two things are treated separately.

Pin Multiplexing

A pin multiplexing descriptor is created with `CYGHWR_HAL_L1XX_PINMUX( reg, field, func )` which takes the following arguments:

`reg`
: This identifies the PINMUX register which controls this pin. This is a value between 0 and 19.

`field`
: This gives the bit offset within the PINMUX register of the field that controls this pin. Fields are 4 bits wide, so this may only be 0, 4, 8, 12, 16, 20, 24 or 28.

`func`
: This defines the function code to program into the PINMUX field. There is no consistency between functions and function codes, and the same function for a pin may be represented by different codes in different PINMUX registers. You should refer to the OMAP L1xx documentation for the correct value to be used here.

The following examples show how this macro may be used:

```
// UART0 TX line is in PINMUX3, bits 20:23, function 2 = UART0_TXD
#define CYGHWR_HAL_L1XX_UART0_TX             CYGHWR_HAL_L1XX_PINMUX( 3, 20, 2 )

// MMCSS0 clock line is in PINMUX10, bits 0:3, function 2 = MMCSD0_CLK
#define CYGHWR_HAL_OMAP_MMCSD0_CLK           CYGHWR_HAL_L1XX_PINMUX( 10,  0, 2 )
```

The macro `CYGHWR_HAL_L1XX_PINMUX_SET( pin )` sets the pin multiplexing setting according to the descriptor passed in.

GPIO Support

A GPIO descriptor is created with `CYGHWR_HAL_L1XX_GPIO( bank, bit, mode)` which takes the following arguments:

`bank` This identifies the GPIO bank to which the pin is attached. This is a value between 0 and 7.

`bit` This gives the bit offset within the bank of the GPIO pin. This is a value between 0 and 15.

`mode` This defines whether this is an input or an output pin, and may take the values `INPUT` or `OUTPUT` respectively.

Additionally, the macro `CYGHWR_HAL_L1XX_GPIO_NONE` may be used in place of a pin descriptor and has a value that no valid descriptor can take. It may therefore be used as a placeholder where no GPIO pin is present or to be used.

The following examples show how this macro may be used:

```
// MMCSD0 card detect is attached to GP4[0] and is an input
#define CYGHWR_HAL_OMAP_MMCSD0_CD_GPIO                  CYGHWR_HAL_L1XX_GPIO( 4, 0, INPUT )

// MMCSD0 write protect is attached to GP4[1] and is an input
#define CYGHWR_HAL_OMAP_MMCSD0_WP_GPIO                  CYGHWR_HAL_L1XX_GPIO( 4, 1, INPUT )
```

The remaining macros all take a GPIO pin descriptor as an argument. `CYGHWR_HAL_L1XX_GPIO_SET` configures the pin according to the descriptor and must be called before any other macros. `CYGHWR_HAL_L1XX_GPIO_OUT` sets the output to the value of the least significant bit of the *`val`* argument. The *`val`* argument of `CYGHWR_HAL_L1XX_GPIO_IN` should be a pointer to an int, which will be set to 0 if the pin input is zero, and 1 otherwise.

There is also support for GPIO interrupts. `CYGHWR_HAL_L1XX_GPIO_INTCFG( pin, mode )` configures the interrupt mode of the pin. It may be either `FALL`, `RISE` or `FALLRISE` to configure the pin to interrupt on the falling edge, rising edge or both. The second argument to `CYGHWR_HAL_L1XX_GPIO_INTSTAT( pin, stat )` must be a pointer to an int, which will be set to 1 if an interrupt has be received on the given pin, and 0 otherwise. GPIO interrupts are currently not decoded into per-pin interrupt vectors, only the shared per-bank vectors are available. If an application needs to get interrupts from more than one pin on a bank, it needs to install a shared ISR and decode the specific pins itself.

Name

Peripheral Power Control — Description

Synopsis

#include <cyg/hal/hal_io.h>

pin = **CYGHWR_HAL_L1XX_POWER**(*device*);

CYGHWR_HAL_L1XX_POWER_ENABLE (*desc*);

CYGHWR_HAL_L1XX_POWER_DISABLE (*desc*);

Description

The OMAP L1XX HAL provides a number of macros to support the management of peripheral power. The macro `CYGHWR_HAL_L1XX_POWER( device )` encodes a power control descriptor into a 32 bit value. The argument is the name of the device to be described.

The remaining functions all take a peripheral power descriptor as an argument. `CYGHWR_HAL_L1XX_POWER_ENABLE( desc )` transitions the given device to enabled state in its PSC controller. Likewise `CYGHWR_HAL_L1XX_POWER_DISABLE( desc )` transitions the device to disabled state.

Name

DMA Support — Description

Synopsis

```
typedef struct hal_edma_channel hal_edma_channel;

typedef void hal_edma_callback( hal_edma_channel *edma_chan, cyg_uint32 event, CYG_ADDRWORD data );
```

#include <cyg/hal/hal_io.h>

pin = **CYGHWR_HAL_L1XX_EDMA_CHANNEL**(*controller*, *event*);

void hal_edma_channel_init (*hal_edma_channel *edma_chan*, *cyg_uint32 channel*, *hal_edma_callback *callback*, *CYG_ADDRWORD data*);

void hal_edma_channel_delete (*hal_edma_channel *edma_chan*);

void hal_edma_channel_source (*hal_edma_channel *edma_chan*, *void *src*, *cyg_int16 bidx*, *cyg_int16 cidx*);

void hal_edma_channel_dest (*hal_edma_channel *edma_chan*, *void *dest*, *cyg_int16 bidx*, *cyg_int16 cidx*);

void hal_edma_channel_burstsize (*hal_edma_channel *edma_chan*, *cyg_uint16 aburstsize*, *cyg_uint16 bburstsize*);

void hal_edma_channel_size (*hal_edma_channel *edma_chan*, *cyg_uint32 size*);

void hal_edma_channel_start (*hal_edma_channel *edma_chan*, *cyg_uint32 opt*);

void hal_edma_channel_stop (*hal_edma_channel *edma_chan*);

void hal_edma_poll (*void*);

Description

The HAL provides support for access to the EDMA3 DMA controllers. This support is not intended to expose the full functionality of these devices and is strictly limited to supporting peripheral DMA, in particular SPI and MMC/SD.

The user is referred to the TI EDMA3 documentation for a full description of the EDMA3 devices, and to the sources of the MMC and SPI drivers for examples of the use of this API. This documentation only gives a brief description of the functions available.

Each device transfer direction is described by a controller number (0 or 1) and an event numbers (0 to 31). These values are usually defined by the peripheral being accessed. The macro `CYGHWR_HAL_L1XX_EDMA_CHANNEL( controller, event)` combines these into a descriptor that may be used to initialize the channel. A channel is controlled by a hal_edma_channel object that must be allocated by the client. To initialize a channel, `hal_edma_channel_init()` is called, passing the channel object, the descriptor and an optional callback function and user-defined value. After use the channel can be deleted with `hal_edma_channel_delete()`.

Before starting a transfer, the channel must be initialized with the source, destination and size of the transfer to be done. `hal_edma_channel_source()` and `hal_edma_channel_dest()` describe the source and destination buffers. A memory buffer is described by its address plus the increments, or indexes, to step the read or write pointer through it. The sizes of these indexes depend on the size of the transfers that the peripheral is programmed to make, and its synchronisation mode. One of the source or destination will be the peripheral itself; the address should be the address of the peripheral's input or output data register, and the indexes should be zero to prevent the pointer incrementing.

`hal_edma_channel_size()` supplies the total transfer size in bytes. `hal_edma_channel_burstsize()` describes the A and B burst sizes for AB synchronized transfers and must be called before `hal_edma_channel_size()`. If `hal_edma_channel_burstsize()` is not called then A synchronization is assumed. The values set by `hal_edma_channel_burstsize()` remain set in the channel, and another call to this function is needed to reset them.

A transfer is started by calling `hal_edma_channel_start()`. The *`opt`* argument contains bits that will be merged with the PaRAM options field. When a transfer is complete, then `hal_edma_channel_stop()` should be called. This function may also be called to abort a transfer.

If a callback was passed to `hal_edma_channel_init()` then when the transfer finishes or encounters and error, the callback will be called from DSR context. The callback will be passed a pointer to the channel object, an event code and the user-defined value that was passed to `hal_edma_channel_init()`. There are two possible event codes: `CYGHWR_HAL_L1XX_EDMA_COMPLETE` indicates that the transfer finished successfully, and `CYGHWR_HAL_L1XX_EDMA_ERROR` indicates that an error condition was encountered.

In kernel configurations the DMA system will use interrupts to detect transfer completion or errors. In non-kernel configurations it will use polling. For the polling to happen, client code must call `hal_edma_poll()` on a regular basis. On DMA completion the callback function will be called, as before.

Chapter 257. Atmel SAM9 Processor Support

Name

Support for the Atmel SAM9 Processor — Overview

Description

This document covers the configuration and usage of the Hardware Abstraction Layer (HAL) for the Atmel AT91SAM9XXX processor family. It is expected to be read in conjunction with platform HAL-specific documentation, as well as the eCos HAL specification. This processor HAL package complements the ARM architectural HAL, ARM9 variant HAL and the platform HAL. It provides functionality common to all SAM9-based board implementations.

This support is found in the eCos package located at `packages/hal/arm/arm9/sam9` within the eCos source repository.

The SAM9 processor HAL package is loaded automatically when eCos is configured for an SAM9-based platform. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Supported Hardware

Supported features of the Atmel SAM9 processor within this processor HAL package include:

- SAM9-specific hardware definitions
- Interrupt controller
- Timer counters
- Serial UARTs
- Two-Wire Interface (TWI)
- Power saving

Support for the on-chip ethernet(AT91SAM9260 only), interrupt-driven serial, SPI, watchdog and wallclock (RTTC) features of the SAM9 are also present and can be found in separate packages, outside of this processor HAL.

Name

SAM9 hardware definitions — Details on obtaining hardware definitions for SAM9

Register definitions

The file `<cyg/hal/sam9.h>` can be included from application and eCos package sources to provide definitions related to SAM9 subsystems. These include register definitions for the interrupt controller, power management controller, clock generator, memory controller, external bus interface, GPIO, USART, MCI, TWI (I²C®), Ethernet, timer counter, RTTC, and SPI subsystems.

Initialization helper macros

The file `<cyg/hal/sam9_init.inc>` contains definitions of helper macros which may be used by SAM9 platform HALs in order to initialise common SAM9 subsystems without excessive duplication between the platform HALs. Typically this file will be included by the `hal_platform_setup.h` header in the platform HAL, in turn included from the architectural HAL file `vectors.S`.

This file is solely intended to be used by platform HALs. At the same time, it is only present to assist initialization, and platform HALs are not obliged to use it if their startup requirements vary.

Name

SAM9 interrupt controller — Advanced Interrupt Controller definitions and usage

Interrupt controller definitions

The file `<cyg/hal/var_ints.h>` (located at `hal/arm/arm9/sam9/`*`VERSION`*`/include/var_ints.h` in the eCos source repository) contains interrupt vector number definitions for use with the eCos kernel and driver interrupt APIs:

```
// The following are common to all SAM9 devices. Per-variant
// variations are supplied later.

#define CYGNUM_HAL_INTERRUPT_FIQ     0 // Advanced Interrupt Controller (FIQ)
#define CYGNUM_HAL_INTERRUPT_SYSTEM  1 // System Peripheral (debug unit, system timer)
#define CYGNUM_HAL_INTERRUPT_PIOA    2 // Parallel IO Controller A
#define CYGNUM_HAL_INTERRUPT_PIOB    3 // Parallel IO Controller B
#define CYGNUM_HAL_INTERRUPT_PIOC    4 // Parallel IO Controller C
                                       // Vector 5 variant specific
#define CYGNUM_HAL_INTERRUPT_US0     6 // USART 0
#define CYGNUM_HAL_INTERRUPT_US1     7 // USART 1
#define CYGNUM_HAL_INTERRUPT_US2     8 // USART 2
#define CYGNUM_HAL_INTERRUPT_MCI     9 // Multimedia Card Interface
#define CYGNUM_HAL_INTERRUPT_UDP    10 // USB Device Port
#define CYGNUM_HAL_INTERRUPT_TWI    11 // Two-Wire Interface
#define CYGNUM_HAL_INTERRUPT_SPI    12 // Serial Peripheral Interface 0
#define CYGNUM_HAL_INTERRUPT_SPI1   13 // Serial Peripheral Interface 1
#define CYGNUM_HAL_INTERRUPT_SSC0   14 // Serial Synchronous Controller 0
                                       // Vector 15 variant specific
                                       // Vector 16 variant specific
#define CYGNUM_HAL_INTERRUPT_TC0    17 // Timer Counter 0
#define CYGNUM_HAL_INTERRUPT_TC1    18 // Timer Counter 1
#define CYGNUM_HAL_INTERRUPT_TC2    19 // Timer Counter 2
#define CYGNUM_HAL_INTERRUPT_UHP    20 // USB Host port
                                       // Vectors 21..28 variant specific
#define CYGNUM_HAL_INTERRUPT_IRQ0   29 // Advanced Interrupt Controller (IRQ0)
#define CYGNUM_HAL_INTERRUPT_IRQ1   30 // Advanced Interrupt Controller (IRQ1)
#define CYGNUM_HAL_INTERRUPT_IRQ2   31 // Advanced Interrupt Controller (IRQ2)

// Variant specific vectors

#if defined(CYGHWR_HAL_ARM_ARM9_SAM9_SAM9260)

#define CYGNUM_HAL_INTERRUPT_ADC     5 // Analog to Digital Converter
                                       // Vector 15 unused
                                       // Vector 16 unused
#define CYGNUM_HAL_INTERRUPT_EMAC   21 // Ethernet MAC
#define CYGNUM_HAL_INTERRUPT_ISI    22 // Image Sensor Interface
#define CYGNUM_HAL_INTERRUPT_US3    23 // USART 3
#define CYGNUM_HAL_INTERRUPT_US4    24 // USART 4
#define CYGNUM_HAL_INTERRUPT_US5    25 // USART 5
#define CYGNUM_HAL_INTERRUPT_TC3    26 // Timer Counter 3
#define CYGNUM_HAL_INTERRUPT_TC4    27 // Timer Counter 4
#define CYGNUM_HAL_INTERRUPT_TC5    28 // Timer Counter 5

#elif defined(CYGHWR_HAL_ARM_ARM9_SAM9_SAM9261)

                                       // Vector 5 unused
#define CYGNUM_HAL_INTERRUPT_SSC1   15 // Serial Synchronous Controller 1
#define CYGNUM_HAL_INTERRUPT_SSC2   16 // Serial Synchronous Controller 2
#define CYGNUM_HAL_INTERRUPT_LCD    21 // LCD controller
                                       // Vectors 22..28 unused

#else
#error Unknown SAM9 variant
#endif
```

```
// The following interrupts are derived from the SYSTEM interrupt
#define CYGNUM_HAL_INTERRUPT_PITC   32 // System Timer Period Interval Timer
#define CYGNUM_HAL_INTERRUPT_DEBUG  33 // Debug unit
#define CYGNUM_HAL_INTERRUPT_WDTC   34 // Watchdog
#define CYGNUM_HAL_INTERRUPT_RTTC   35 // Real Time Clock
#define CYGNUM_HAL_INTERRUPT_PMC    36 // Power Management Controller
#define CYGNUM_HAL_INTERRUPT_RSTC   37 // Reset Controller
```

As indicated above, further decoding is performed on the SYSTEM interrupt to identify the cause more specifically. Note that as a result, placing an interrupt handler on the SYSTEM interrupt will not work as expected. Conversely, masking a decoded derivative of the SYSTEM interrupt will not work as this would mask other SYSTEM interrupts, but masking the SYSTEM interrupt itself will work. On the other hand, unmasking a decoded SYSTEM interrupt *will* unmask the SYSTEM interrupt as a whole, thus unmasking interrupts for the other units on this shared interrupt.

The list of interrupt vectors may be augmented on a per-platform basis. Consult the platform HAL documentation for your platform for whether this is the case.

Interrupt controller functions

The source file `src/sam9_misc.c` within this package provides most of the support functions to manipulate the interrupt controller. The `hal_IRQ_handler` queries the IRQ status register to determine the interrupt cause. Functions `hal_interrupt_mask` and `hal_interrupt_unmask` enable or disable interrupts within the interrupt controller.

Interrupts are configured in the `hal_interrupt_configure` function, where the `level` and `up` arguments are interpreted as follows:

level	up	interrupt on
0	0	Falling Edge
0	1	Rising Edge
1	0	Low Level
1	1	High Level

To fit into the eCos interrupt model, interrupts essentially must be acknowledged immediately once decoded, and as a result, the `hal_interrupt_acknowledge` function is empty.

The `hal_interrupt_set_level` is used to set the priority level of the supplied interrupt within the Advanced Interrupt Controller.

Note that in all the above, it is not recommended to call the described functions directly. Instead either the HAL macros (`HAL_INTERRUPT_MASK` et al) or preferably the kernel or driver APIs should be used to control interrupts.

Using the Advanced Interrupt Controller for VSRs

The SAM9 HAL has been designed to exploit benefits of the on-chip Advanced Interrupt Controller (AIC) on the SAM9. Support has been included for exploiting its ability to provide hardware vectoring for VSR interrupt handlers.

This support is dependent on definitions that may only be provided by the platform HAL and therefore is only enabled if the platform HAL package implements the `CYGINT_HAL_SAM9_AIC_VSR` CDL interface. The necessary definitions are available to all platform HALs which use the facilities of the `sam9_init.inc` header file.

The interrupt decoding path has been optimised by allowing the AIC to be interrogated for the interrupt handler VSR to use. These vectored interrupts are by default still configured to point to the default ARM architecture HAL IRQ and FIQ VSRs. However applications may set their own VSRs to override this default behaviour to allow optimised interrupt handling.

The VSR vector numbers to use when overriding are defined as follows:

```
// FIQ is already defined as vector 7 in the architecture hal_intr.h
#define CYGNUM_HAL_VECTOR_SYSTEM  8 // System Peripheral (debug unit, system timer)
#define CYGNUM_HAL_VECTOR_PIOA    9 // Parallel IO Controller A
#define CYGNUM_HAL_VECTOR_PIOB   10 // Parallel IO Controller B
#define CYGNUM_HAL_VECTOR_PIOC   11 // Parallel IO Controller C
                                    // VSR 12 variant specific
#define CYGNUM_HAL_VECTOR_US0    13 // USART 0
#define CYGNUM_HAL_VECTOR_US1    14 // USART 1
#define CYGNUM_HAL_VECTOR_US2    15 // USART 2
#define CYGNUM_HAL_VECTOR_MCI    16 // Multimedia Card Interface
#define CYGNUM_HAL_VECTOR_UDP    17 // USB Device Port
#define CYGNUM_HAL_VECTOR_TWI    18 // Two-Wire Interface
#define CYGNUM_HAL_VECTOR_SPI    19 // Serial Peripheral Interface
#define CYGNUM_HAL_VECTOR_SPI1   20 // Serial Peripheral Interface
#define CYGNUM_HAL_VECTOR_SSC0   21 // Serial Synchronous Controller 0
                                    // VSR 22 variant specific
                                    // VSR 23 variant specific
#define CYGNUM_HAL_VECTOR_TC0    24 // Timer Counter 0
#define CYGNUM_HAL_VECTOR_TC1    25 // Timer Counter 1
#define CYGNUM_HAL_VECTOR_TC2    26 // Timer Counter 2
#define CYGNUM_HAL_VECTOR_UHP    27 // USB Host port
                                    // VSRs 28..35 variant specific
#define CYGNUM_HAL_VECTOR_IRQ0   36 // Advanced Interrupt Controller (IRQ0)
#define CYGNUM_HAL_VECTOR_IRQ1   37 // Advanced Interrupt Controller (IRQ1)
#define CYGNUM_HAL_VECTOR_IRQ2   38 // Advanced Interrupt Controller (IRQ2)

// Variant specific vectors

#if defined(CYGHWR_HAL_ARM_ARM9_SAM9_SAM9260)

#define CYGNUM_HAL_VECTOR_ADC    12 // Analog to Digital Converter
                                    // Vector 22 unused
                                    // Vector 23 unused
#define CYGNUM_HAL_VECTOR_EMAC   28 // Ethernet MAC
#define CYGNUM_HAL_VECTOR_ISI    29 // Image Sensor Interface
#define CYGNUM_HAL_VECTOR_US3    30 // USART 3
#define CYGNUM_HAL_VECTOR_US4    31 // USART 4
#define CYGNUM_HAL_VECTOR_US5    32 // USART 5
#define CYGNUM_HAL_VECTOR_TC3    33 // Timer Counter 3
#define CYGNUM_HAL_VECTOR_TC4    34 // Timer Counter 4
#define CYGNUM_HAL_VECTOR_TC5    35 // Timer Counter 5


#elif defined(CYGHWR_HAL_ARM_ARM9_SAM9_SAM9261)

                                    // Vector 12 unused
#define CYGNUM_HAL_VECTOR_SSC1   22 // Serial Synchronous Controller 1
#define CYGNUM_HAL_VECTOR_SSC2   23 // Serial Synchronous Controller 2
#define CYGNUM_HAL_VECTOR_LCD    28 // LCD controller
                                    // Vectors 29..35 unused

#else

#error Unknown SAM9 variant

#endif
```

Consult the kernel and generic HAL documentation for more information on VSRs and how to set them.

Interrupt handling withing standalone applications

For non-eCos standalone applications running under RedBoot, it is possible to install an interrupt handler into the interrupt vector table manually. Memory mappings are platform-dependent and so the platform documentation should be consulted, but in general the address of the interrupt table can be determined by analyzing RedBoot's symbol table, and searching for the address of the symbol name `hal_interrupt_handlers`. Table slots correspond to the interrupt numbers above. Pointers inserted in this table should be pointers to a C/C++ function with the following prototype:

```
extern unsigned int isr( unsigned int vector, unsigned int data );
```

For non-eCos applications run from RedBoot, the return value can be ignored. The `vector` argument will also be the interrupt vector number. The `data` argument is extracted from a corresponding table named `hal_interrupt_data` which immediately follows the interrupt vector table. It is still the responsibility of the application to enable and configure the interrupt source appropriately if needed.

Name

Timers — Use of on-chip timers

Periodic Interval Timer

The eCos kernel system clock is implemented using the Periodic Interval Timer (PIT). By default, the system clock interrupts once every 10ms, corresponding to a 100Hz clock. This can be changed by the configuration option `CYGNUM_HAL_RTC_DENOMINATOR` which corresponds to the clock frequency. Other clock-related settings are recalculated automatically if the denominator is changed. If the desired frequency cannot be expressed accurately solely with changes to `CYGNUM_HAL_RTC_DENOMINATOR`, then the configuration option `CYGNUM_HAL_RTC_NUMERATOR` may also be adjusted, and again clock-related settings will automatically be recalculated.

The PIT is also used to implement the HAL microsecond delay function, `HAL_DELAY_US`. This is used by some device drivers, and in non-kernel configurations such as with RedBoot where this timer is needed for loading program images via X/Y-modem protocols and debugging via TCP/IP. Standalone applications which require RedBoot services, such as debugging, should avoid use of this timer.

Timer-based profiling support

Timer-based profiling support is implemented using timer counter 1 (TC1). If the gprof package, `CYGPKG_PROFILE_GPROF`, is included in the configuration, then TC1 is reserved for use by the profiler.

Name

Serial UARTs — Configuration and implementation details of serial UART support

Overview

Support is included in this processor HAL package for the SAM9's on-chip debug unit UART and up to four serial USART serial devices.

There are two forms of support: HAL diagnostic I/O; and a fully interrupt-driven serial driver. Unless otherwise specified in the platform HAL documentation, for all serial ports the default settings are 115200,8,N,1 with no flow control.

HAL diagnostic I/O

This first form is polled mode HAL diagnostic output, intended primarily for use during debug and development. Operations are usually performed with global interrupts disabled, and thus this mode is not usually suitable for deployed systems. This can operate on any port, according to the configuration settings.

There are several configuration options usually found within a platform HAL which affect the use of this support in the SAM9 processor HAL. The CDL option `CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL` selects the serial port channel to use as the console at startup time. This will be the channel that receives output from, for example, `diag_printf()`. The CDL option `CYGNUM_HAL_VIRTUAL_VECTOR_DEBUG_CHANNEL` selects the serial port channel to use for GDB communication by default. Note that when using RedBoot, these options are usually inactive as it is RedBoot that decides which channels are used. Applications may override RedBoot's selections by enabling the `CYGSEM_HAL_VIRTUAL_VECTOR_CLAIM_COMMS` CDL option in the HAL. Baud rates for each channel are set with the `CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL_BAUD` and `CYGNUM_HAL_VIRTUAL_VECTOR_DEBUG_CHANNEL_BAUD` options.

Interrupt-driven serial driver

The second form of support is an interrupt-driven serial driver, which is integrated into the eCos standard serial I/O infrastructure (`CYGPKG_IO_SERIAL`). This support can be enabled on any port.

Note that it is not recommended to share this driver when using the HAL diagnostic I/O on the same port. If the driver is shared with the GDB debugging port, it will prevent ctrl-c operation when debugging.

This driver is contained in the `CYGPKG_IO_SERIAL_ARM_AT91` package. That driver package should also be consulted for documentation and configuration options. The driver is not active until the `CYGPKG_IO_SERIAL_DEVICES` configuration option within the generic serial driver support package `CYGPKG_IO_SERIAL` is enabled in the configuration.

Note that unlike the USART devices, the serial debug port does not support modem control signals such as those used for hardware signals. In addition, USART devices for a particular platform may also not have these control signals brought out to the physical serial port.

Name

Two-Wire Interface (TWI) driver — Configuration and implementation details of TWI (I²C®) driver

Overview

The SAM9 processor HAL contains a driver for the Two-Wire Interface (TWI) controller on the SAM9. This type of bus is also known as I²C®. The API for this may be found within the `CYGPKG_IO_I2C` package.

I²C/TWI driver configuration

The I²C® driver uses the SAM9's internal Two-Wire Interface (TWI) support. This is controlled within the SAM9 processor HAL (`CYGPKG_HAL_SAM9`). The `CYGPKG_HAL_SAM9_TWI` CDL component controls whether the TWI driver is enabled. Within that component, there are two sub-options:

- `CYGNUM_HAL_SAM9_TWI_CLOCK` sets the speed of the TWI bus clock in Hz. This is usually 100kHz, but can be set up to 400kHz if the devices on the bus support this speed, also known as fast mode. However other values below 400kHz can also be chosen, subject to the accuracy of the clock waveform generation parameters.

- The second option within the `CYGPKG_HAL_SAM9_TWI` component is `CYGNUM_HAL_SAM9_TWI_CKDIV`. This is the clock divider used when configuring the TWI_CWGR register. Consult the SAM9 datasheet description of the TWI_CWGR register for the formula used to determine the clock frequency. Increasing the divider will decrease the accuracy in practice of the generated I²C bus clock compared to `CYGNUM_HAL_SAM9_TWI_CLOCK`. But the divider must also be sufficiently low that the relevant factors do not overflow valid values for CHDIV/CLDIV in TWI_CWGR. Note that when the SAM9 is using a 60MHz MCK, then for 100kHz operation, a value for this option of 1 is most appropriate. For 400kHz, a value for this option of 0 is most appropriate. The default value of this CDL is an appropriate value for CKDIV assuming a 60MHz MCK and a TWI clock between 29kHz and 400kHz.

  To be specific, the CLDIV/CHDIV fields of the TWI_CWGR are considered equal. The value of, for example, CLDIV, can be expressed as:

$$CLDIV = \frac{f_{MCK} - 6 f_{TWI}}{2^{CKDIV+1} \cdot f_{TWI}}$$

To use the I²C/TWI driver, the generic I²C driver package `CYGPKG_IO_I2C` must be used. Documentation for its API may be found elsewhere.

Usage notes

Due to the characteristics of the SAM9's operation, it is not possible to provide support for repeated starts with the I²C package API. Similarly indicating a NACK when performing a receive is equivalent to also sending a STOP.

A test application for use with the AT24C512 serial EEPROM fitted to the AT91SAM9260EK board is provided within the `tests` subdirectory of the `CYGPKG_HAL_SAM9` package. This test communicates with the I²C EEPROM on the board to perform read and write operations using I²C. This test is not built by default. It may be built by enabling the configuration option `CYGBLD_HAL_ARM_ARM9_SAM9_TEST_TWI_AT24C512` within the SAM9 processor HAL.

Name

Power saving support — Extensions for saving power

Overview

There is support in the SAM9 processor HAL for a simple power saving mechanism. This is provided by two functions:

```
#include <cyg/hal/hal_intr.h>

__externC void cyg_hal_sam9_powersave_init( cyg_uint32 ip_addr );

__externC void cyg_hal_sam9_powerdown( void );
```

The powersaving system is initialized by calling `cyg_hal_sam9_powersave_init()`. The argument should be the IP address of this machine in network order. This can usually be fetched from the bootp data for an interface after completion of the call to `init_all_network_interfaces()`. e.g. `eth0_bootp_data.bp_ciaddr.s_addr`.

A call to `cyg_hal_sam9_powerdown()` will put the machine into a low power mode. This will involve switching to a slower system clock speed, disabling all peripherals except those that are defined to cause the system to wake up and return from this function.

Configuration

The exact behaviour of the power saving system is controlled by the following configuration options:

CYGPKG_HAL_ARM_ARM9_SAM9_POWERSAVE

> This option controls the overall inclusion of the power saving system.
>
> Default value: on

CYGSEM_HAL_ARM_ARM9_SAM9_POWERSAVE_POLL_ETHERNET

> This option enables polling of the ethernet interface for relevant ARP packets and unicast IP packets. It is necessary for the CPU to run at a higher CPU speed for this option to work.
>
> Default value: off

CYGSEM_HAL_ARM_ARM9_SAM9_POWERSAVE_IDLE

> If this option is set, the CPU will go into idle mode, which will cause it to halt until an interrupt is delivered.
>
> Default value: off

CYGVAR_HAL_ARM_ARM9_SAM9_POWERSAVE_ACTIVE_DEVICES

> This option defines the devices that are to be kept running during power down mode. An interrupt from one of these devices is usually the only way of bringing the system out of idle mode. The value of this option is a bit mask with bits set for each device that is to be kept active. The bits correspond to the peripheral identifiers described in the SAM9 documentation.
>
> Default value: 0x00000000

CYGSEM_HAL_ARM_ARM9_SAM9_POWERSAVE_POLL_GPIO

> This option control whether the power saving system will poll GPIO pins during power saving. For this to work the CPU cannot be put into idle mode.

Default value: on

CYGVAR_HAL_ARM_ARM9_SAM9_POWERSAVE_PIO_HI

This is an array of bitmasks of the bits in the PIO PDSR registers. Within the array, index 0 corresponds to PIOA, index 1 to PIOB and so on. For each set bit in these masks, if the value is seen to be 1, then the low power mode will be terminated.

Default value: 0, 0, 0, 0

CYGVAR_HAL_ARM_ARM9_SAM9_POWERSAVE_PIO_LO

This is an array of bitmasks of the bits in the PIO PDSR registers. Within the array, index 0 corresponds to PIOA, index 1 to PIOB and so on. For each set bit in these masks, if the value is seen to be 0, then the low power mode will be terminated.

Default value: 0, 0, 0, 0

CYGVAR_HAL_ARM_ARM9_SAM9_POWERSAVE_PIO_CHANGE

This is an array of bitmasks of the bits in the PIO PDSR registers. Within the array, index 0 corresponds to PIOA, index 1 to PIOB and so on. For each set bit in these masks, if the value is seen to change between successive polls, then the low power mode will be terminated.

Default value: 0, 0, 0, 0

CYGBLD_HAL_ARM_ARM9_SAM9_TEST_POWERSAVE

This option controls whether a simple test is built to exercise power saving support. The test is not built by default as an external means is required to wake the processor up by one of the above configured mechanisms.

Default value: 0

Chapter 258. Atmel AT91SAM9260 Evaluation Kit Board Support

Name

eCos Support for the Atmel AT91SAM9260 Evaluation Kit — Overview

Description

This document covers the configuration and usage of eCos and RedBoot on the Atmel AT91SAM9260 Evaluation Kit. The AT91SAM9260 Evaluation Kit contains the AT91SAM9260 microprocessor, 64Mbytes of SDRAM, 256Mbytes of NAND flash memory, an Atmel Dataflash, an Atmel serial EEPROM, a Davicom DM9161A PHY, a SD/MMC/DataFlash socket, a DAC, external connections for three serial channels (one debug, one full modem, one flow controlled), ethernet, USB host/device, and the various other peripherals supported by the AT91SAM9260. eCos support for the many devices and peripherals on the boards and the AT91SAM9260 is described below.

For typical eCos development, a RedBoot image is programmed into the dataflash memory, and the board will load this image from reset. RedBoot provides gdb stub functionality so it is then possible to download and debug stand-alone and eCos applications via the gdb debugger. This can happen over either a serial line or over ethernet.

This documentation is expected to be read in conjunction with the SAM9 processor HAL documentation and further device support and subsystems are described and documented there.

Supported Hardware

The Dataflash consists of 8192 blocks of 1056 bytes each. In a typical setup, the first 33792 bytes are reserved for the second-level bootstrap, AT91Bootstrap. The following 164736 bytes are reserved for the use of the ROM RedBoot image (The odd size aligns the end of the RedBoot area to a 1056 block boundary). The topmost block is used to manage the flash and the next block down holds RedBoot **fconfig** values. The remaining blocks can be used by application code.

There is a serial driver `CYGPKG_IO_SERIAL_ARM_AT91` which supports both the Debug Unit and USART serial devices. The debug serial port at J17 and DTE port at J20 (connected to USART channel 0) and flow controlled port at J18 (connected to USART channel 1) can be used by RedBoot for communication with the host. If any of these devices is needed by the application, either directly or via the serial driver, then it cannot also be used for RedBoot communication. Another communication channel such as ethernet should be used instead. The serial driver package is loaded automatically when configuring for the AT91SAM9260-EK target.

There is an ethernet driver `CYGPKG_DEVS_ETH_ARM_AT91` for the on-chip ethernet device. The platform HAL package is responsible for configuring this generic driver to the hardware. This driver is also loaded automatically when configuring for the AT91SAM9260-EK board.

There is a watchdog driver `CYGPKG_DEVICES_WATCHDOG_ARM_AT91WDTC`. This driver is also loaded automatically when configuring for the board.

There is a driver for the on-chip real-time timer controller (RTTC) at `CYGPKG_DEVICES_WALLCLOCK_ARM_AT91RTTC`. This driver is also loaded automatically when configuring for the target.

The SAM9 processor HAL contains a driver for the Two-Wire Interface (TWI) controller on the AT91SAM9260. This type of bus is also known as I²C®. Further documentation may be found in the SAM9 processor HAL documentation.

There is a driver for the MultiMedia Card Interface (MCI) at `CYGPKG_DEVS_MMCSD_ATMEL_SAM_MCI`. This driver is loaded automatically when configuring for the AT91SAM9260-EK target and allows use of MMC and Secure Digital (SD) flash storage cards within eCos, exported as block devices. Further documentation may be found within that package.

The platform HAL provides definitions to allow access to devices on the SPI bus. The HAL provides information to the more general AT91 SPI driver (`CYGPKG_DEVS_SPI_ARM_AT91`) which in turn provides the underlying implementation for the SPI API layer in the `CYGPKG_IO_SPI` package. All these packages are automatically loaded when configuring for the board.

Furthermore, the platform HAL package contains support for SPI dataflash cards. The HAL support integrates with the CYGPKG_DEVS_FLASH_ATMEL_DATAFLASH package as well as the above SPI packages. That package is automatically loaded when configuring for the target. Dataflash media is then accessed as a Flash device, using the Flash I/O API within the CYGPKG_IO_FLASH package, if that package is loaded in the configuration.

It is also possible to configure the HAL to access MMC cards in SPI mode, instead of using the MCI interface.

In general, devices (Caches, PIO, UARTs, EMAC) are initialized only as far as is necessary for eCos to run. Other devices (RTC, I²C, SPI, MCI etc.) are not touched unless the appropriate driver is loaded, although in some cases, the HAL boot sequence will set up the appropriate PIO configuration.

Tools

The AT91SAM9260-EK support is intended to work with GNU tools configured for an arm-eabi target. The original port was undertaken using arm-elf-gcc version 3.4.4, arm-elf-gdb version 6.3, and binutils version 2.15.

Name

Setup — Preparing the AT91SAM9260-EK board for eCos Development

Overview

In a typical development environment, the AT91SAM9260-EK board boots from the DataFlash and runs the RedBoot ROM monitor from SDRAM. eCos applications are configured for RAM startup and then downloaded and run on the board via the debugger **arm-eabi-gdb**. Preparing the board therefore usually involves programming a suitable RedBoot image into flash memory.

The following RedBoot configurations are supported:

Configuration	Description	Use	File
ROM	RedBoot loaded from Dataflash to SDRAM	`redboot_ROM.ecm`	`redboot_ROM.bin`

For serial communications, all versions run with 8 bits, no parity, and 1 stop bit at 115200 baud. RedBoot also supports ethernet communication and flash management.

Note that the use of the term ROM for the initial RedBoot configuration is a historical accident. RedBoot actually runs from SDRAM after being loaded there from Dataflash by the second-level bootstrap. The use of ROM for this configuration is intended to indicate that it initializes the microprocessor and board peripherals, as opposed to the RAM configuration which assumes that this has already been done.

Initial Installation

The on-chip boot program on the AT91SAM9260 is only capable of loading programs from Dataflash or NAND flash into 4Kbytes of on-chip SRAM and is therefore quite restrictive. Consequently RedBoot cannot be booted directly and a second-level bootstrap must be used. Such a second-level bootstrap is supplied by Atmel in the form of AT91Bootstrap. This is therefore programmed into the start of Dataflash and is then responsible for initializing the SDRAM and loading RedBoot from Dataflash and executing it.

Caution

There is a size limit on the size of applications which the AT91Bootstrap second level bootstrap will load. Images larger than 320Kbytes will require the AT91Bootstrap application to be rebuilt with a larger `IMG_SIZE` definition in `AT91Bootstrap/board/at91sam9260ek/dataflash/at91sam9260ek.h` within the sam9260ek HAL package in the eCos source repository (`packages/hal/arm/arm9/sam9260ek/current/`).

There are basically two ways to write the second-level bootstrap and RedBoot to the Dataflash. The first is to use the Atmel-supplied SAM-BA program that interacts with the on-chip boot program. The second is to use a JTAG debugger that understands the microcontroller and can write to the dataflash (for example the Ronetix PEEDI). Since the availability of the latter cannot be guaranteed, only the first method will be described here.

Programming RedBoot into DataFlash using SAM-BA

The following gives the steps needed to program the second-level bootstrap and RedBoot into the DataFlash using SAM-BA. The user should refer to the SAM-BA documentation for full details of how to run the program.

1. Download the AT91 In-system Programmer software package from the Atmel website and install it. SAM9 series CPU's require the 2.1.x series version of SAM-BA. Atmel provide both Linux and Windows versions of SAM-BA so ensure you select the version appropriate to your host operating system. The remainder of this document describes the process according to a Windows installation. The steps for the Linux version of SAM-BA are similar and can easily be determined from the Windows process.

2. From the root directory of your eCosPro installation, copy the file `dataflash_at91sam9260ek.bin` from the sub-directory `packages/hal/arm/arm9/sam9260ek/current/AT91Bootstrap/board/at91sam9260ek` and the file `redboot_ROM.bin` from the sub-directory `loaders/sam9260ek` to a suitable location on the Windows PC.

3. Connect a null-modem serial cable between the DEBUG serial port of the board and a serial port on a convenient host (which need not be the PC running SAM-BA). Run a terminal emulator (Hyperterm or minicom) at 115200 baud. Connect a USB cable between the PC and the AT91SAM9260-EK board.

4. Power up or reset the board and Windows should recognize the USB device. If it does not, hold down switch BP4 while powering up or resetting the board. This will erase any previous bootloader from the dataflash. Reset the board again and it should be recognized now. Windows may ask you to install a new driver, in which case follow the instructions. If the USB device does not appear, check the USB cable.

   ### Note

   Under Linux this device may appear as `/dev/ACM0` or `/dev/ttyUSB0`. SAM-BA currently only recognises the latter so you may have to create a symbolic link from `/dev/ACM0` to `/dev/ttyUSB0`.

5. Start SAM-BA. Select "\usb\ARM0" for the communication interface, and "AT91SAM9260-EK" for the board. If the USB option does not appear, look in the Windows Device Manager for the new USB COM device. If all is well, click on "Connect".

6. In the SAM-BA main window, select the "DataFlash AT45DB/DBC" tab and in the "Scripts" dropdown menu select "Enable Dataflash (SPIO CS1)", to program the on-board Dataflash device. Click Execute and SAM-BA should emit the following in the message area:

   ```
   (AT91-ISP v1.13) 1 % DATAFLASH::Init 1
   -I- DATAFLASH::Init 1 (trace level : 4)
   -I- Loading applet isp-dataflash-at91sam9260.bin at address 0x20000000
   -I- Memory Size : 0x840000 bytes
   -I- Buffer address : 0x20002A70
   -I- Buffer size: 0x80E80 bytes
   -I- Applet initialization done
   ```

   The actual options and output of SAM-BA may vary according to the version you are using. The behaviour documented here is that of SAM-BA 2.9.

7. Now select the DataFlash tab again, "Send BootFile" from the "Scripts" menu and "Execute" it. When the file open dialog appears, select the `dataflash_at91sam9260ek.bin` file and click "Open". The following output should be seen:

   ```
   (AT91-ISP v1.13) 1 % GENERIC::SendBootFileGUI
   GENERIC::SendFile dataflash_at91sam9260ek.bin at address 0x0
   -I- File size : 0xF82 byte(s)
   -I-     Writing: 0xF82 bytes at 0x0 (buffer addr : 0x20002A70)
   -I-     0xF82 bytes written by applet
   ```

8. The second-level bootstrap has now been written to DataFlash, we must now write RedBoot.

9. In the "Send File Name" box type in the path name to the `redboot_ROM.bin` file, or use the Open Folder button and browse to it.

10. In the Address field set the value to 0x8400.

11. Click the "Send File" button. SAM-BA will put up a dialog box while it is writing the file to the DataFlash, and will output something similar to the following in the message area:

    ```
    (AT91-ISP v1.13) 1 % send_file {DataFlash AT45DB/DCB} "redboot_ROM.bin" 0x8400 0
    -I- Send File //bert/Shared/Releng/sam9260ek/redboot_ROM.bin at address 0x8400
    GENERIC::SendFile //bert/Shared/Releng/sam9260ek/redboot_ROM.bin at address 0x8400
    -I- File size : 0x243D0 byte(s)
    -I-     Writing: 0x243D0 bytes at 0x8400 (buffer addr : 0x20002A70)
    ```

```
-I-     0x243D0 bytes written by applet
```

12. Shut down SAM-BA and disconnect the USB cable. Press the reset button on the board and something similar to the following should be output on the DEBUG serial line.

```
RomBOOT
>Start AT91Bootstrap...
+**Warning** FLASH configuration checksum error or invalid key
Use 'fconfig -i' to [re]initialize database
No space to add 'net_device'
AT91_ETH: Waiting for PHY to reset.
AT91_ETH: Waiting for link to come up..
Ethernet eth0: MAC address 12:34:56:78:9a:bc
No IP info for device!

RedBoot(tm) bootstrap and debug environment [ROM]
eCosCentric certified release, version v3_0_3 - built 12:50:09, Sep 24 2009

Copyright (C) 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009
Free Software Foundation, Inc.
Copyright (C) 2003, 2004, 2005, 2006, 2007, 2008 eCosCentric Limited
RedBoot is free software, covered by the eCos license, derived from the
GNU General Public License. You are welcome to change it and/or distribute
copies of it under certain conditions. Under the license terms, RedBoot's
source code and full license terms must have been made available to you.
Redboot comes with ABSOLUTELY NO WARRANTY.

Platform: AT91SAM9260-EK (ARM9)
RAM: 0x20000000-0x24000000 [0x20035d08-0x23ffef80 available]
FLASH: 0x40000000-0x4083ffff, 8192 x 0x420 blocks
RedBoot>
```

RedBoot Flash configuration

The following steps describe how to initialize RedBoot's Flash configuration.

1. Use the following command to initialize RedBoot's Flash Information System (FIS):

```
RedBoot> fis init
About to initialize [format] FLASH image system - continue (y/n)? y
*** Initialize FLASH Image System
... Erase from 0x4083fbe0-0x4083ffff: .
... Program from 0x23fffbe0-0x24000000 to 0x4083fbe0: .
RedBoot>
```

2. Now configure RedBoot's Flash configuration with the command:

```
RedBoot> fconfig -i
```

Remember to substitute the appropriate MAC address for this board at the appropriate step. If a BOOTP/DHCP server is not available, then IP configuration may be set manually. The default server IP address can be set to a PC that will act as a TFTP host for future RedBoot load operations, or may be left unset. The following gives an example configuration:

```
RedBoot> fconfig -i
Initialize non-volatile configuration - continue (y/n)? y
Run script at boot: false
Use BOOTP for network configuration: false
Gateway IP address: 192.168.7.11
Local IP address: 192.168.7.83
Local IP address mask: 255.255.255.0
Default server IP address: 192.168.7.11
Console baud rate: 115200
DNS domain name: farm.ecoscentric.com
DNS server IP address: 192.168.7.11
Network hardware address [MAC]: 0x0E:0x00:0x00:0xEA:0x18:0xF0
```

```
GDB connection port: 9000
Force console for special debug messages: false
Update RedBoot non-volatile configuration - continue (y/n)? y
... Erase from 0x4083f7c0-0x4083fbdf: .
... Program from 0x23fff7c0-0x23fffbe0 to 0x4083f7c0: .
RedBoot>
```

The RedBoot installation is now complete. This can be tested by powering off the board, and then powering on the board again. Output similar to the following should be seen on the DEBUG serial port. Verify the IP settings are as expected.

```
Ethernet eth0: MAC address 0e:00:00:ea:18:a2
IP: 192.168.7.83/255.255.255.0, Gateway: 192.168.7.11
Default server: 192.168.7.11
DNS server IP: 192.168.7.11, DNS domain name: <null>

RedBoot(tm) bootstrap and debug environment [ROM]
eCosCentric certified release, version v3_0_3 - built 12:50:09, Sep 24 2009

Copyright (C) 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009
Free Software Foundation, Inc.
Copyright (C) 2003, 2004, 2005, 2006, 2007, 2008 eCosCentric Limited
RedBoot is free software, covered by the eCos license, derived from the
GNU General Public License. You are welcome to change it and/or distribute
copies of it under certain conditions. Under the license terms, RedBoot's
source code and full license terms must have been made available to you.
Redboot comes with ABSOLUTELY NO WARRANTY.

Platform: AT91SAM9260-EK (ARM9)
RAM: 0x20000000-0x24000000 [0x20035d08-0x23ffef80 available]
FLASH: 0x40000000-0x4083ffff, 8192 x 0x420 blocks
RedBoot>
```

If it proves necessary to re-install RedBoot, this may be achieved by repeating the above process. Alternatively, a new image may be downloaded and programmed into flash more directly using RedBoot's own commands. See the RedBoot documentation for details.

Rebuilding RedBoot

Should it prove necessary to rebuild a RedBoot binary, this is done most conveniently at the command line. The steps needed to rebuild the ROM version of RedBoot for the AT91SAM9260-EK are:

```
$ mkdir redboot_at91sam9260ek_rom
$ cd redboot_at91sam9260ek_rom
$ ecosconfig new at91sam9260ek redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/arm/arm9/sam9260ek/current/misc/redboot_ROM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

At the end of the build the `install/bin` subdirectory should contain the file `redboot.bin`.

Rebuilding AT91Bootstrap

The sources of AT91Bootstrap are found in the `AT91Bootstrap` directory of the sam9260ek package. This is a copy of the software as supplied by Atmel with some slight modifications to permit it to be built with the same tools as eCos.

To rebuild the second-level bootstrap for the AT91SAM9260-EK execute the following commands:

```
$ cd $ECOS_REPOSITORY/hal/arm/arm9/sam9260ek/current/AT91Bootstrap/board/at91sam9260ek/dataflash
$ make
```

This should result in the creation of a number of files, including `dataflash_at91sam9260ek.bin` which can be copied out.

Name

Configuration — Platform-specific Configuration Options

Overview

The AT91SAM9260-EK platform HAL package is loaded automatically when eCos is configured for the `sam9260ek` target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The platform HAL package supports two separate startup types:

RAM
: This is the startup type which is normally used during application development. The board has RedBoot programmed into flash and boots into that initially. arm-eabi-gdb is then used to load a RAM startup application into memory and debug it. It is assumed that the hardware has already been initialized by RedBoot. By default the application will use the eCos virtual vectors mechanism to obtain certain services from RedBoot, including diagnostic output.

ROM
: This startup type can be used for finished applications which will be programmed into DataFlash. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization.

JTAG
: This is the startup type which can be used during application development via a JTAG device such as the PEEDI. arm-eabi-gdb is used to load a JTAG startup application into memory and debug it. Hardware setup is divided between the initialization section of the PEEDI configuration file and software in the loaded application.

RedBoot and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building RedBoot.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then the serial port will be claimed for HAL diagnostics.

Flash Driver

The AT91SAM9260-EK board contains an 8Mbyte Atmel AT45DB DataFlash device. The `CYGPKG_DEVS_FLASH_ATMEL_DATAFLASH` package contains all the code necessary to support this part and the platform HAL package contains definitions that customize the driver to the AT91SAM9260-EK board. This driver is not active until the generic Flash support package, `CYGPKG_IO_FLASH`, is included in the configuration.

Ethernet Driver

The AT91SAM9260-EK board uses the AT91SAM9260's internal EMAC ethernet device attached to an external Davicom DM9161A PHY. The `CYGPKG_DEVS_ETH_ARM_AT91` package contains all the code necessary to support this device and the platform HAL package contains definitions that customize the driver to the AT91SAM9260-EK board. This driver is not active until the generic Ethernet support package, `CYGPKG_IO_ETH_DRIVERS`, is included in the configuration.

RTC Driver

The AT91SAM9260-EK board uses the AT91SAM9260's internal RTTC support. The `CYGPKG_DEVICES_WALLCLOCK_ARM_AT91RTTC` package contains all the code necessary to support this device. This driver is not active until the generic wallclock device support package, `CYGPKG_IO_WALLCLOCK`, is included in the configuration.

I^{2}C Driver

Warning

While the I^{2}C® driver uses the SAM9's internal Two-Wire Interface (TWI) support (see Two-Wire Interface (TWI) driver), you may experience problems using the I^{2}C device on the AT91SAM9260-EK board. This is because the Ethernet MII interface and I^{2}C bus 0 share GPIO lines PA23 and PA24. By default the Ethernet MAC uses the RMII interface and these shared lines are not, in theory, used. However, these lines are connected to the PHY and interference from the PHY has been observed, which prevents I^{2}C functioning correctly.

The solution is to unsolder resistors R121 and R122 to isolate the PHY from the I^{2}C lines.

Watchdog Driver

The AT91SAM9260-EK board uses the AT91SAM9260's internal watchdog support. The `CYGPKG_DEVICES_WATCHDOG_ARM_AT91WDTC` package contains all the code necessary to support this device. Within that package the `CYGNUM_DEVS_WATCHDOG_ARM_AT91WDTC_DESIRED_TIMEOUT_MS` configuration option controls the watchdog timeout, and by default will force a reset of the board upon timeout. This driver is not active until the generic watchdog device support package, `CYGPKG_IO_WATCHDOG`, is included in the configuration.

Warning

The ATSAM926x processor will boot with watchdog support enabled, and the watchdog configuration is write-once. That is, if it is disabled, it cannot be re-enabled. Due to its nature, RedBoot disables the watchdog when it starts so any eCos applications with watchdog support enabled that are run by RedBoot will not function correctly.

USART Serial Driver

The AT91SAM9260-EK board uses the AT91SAM9260's internal USART serial support as described in the SAM9 processor HAL documentation. Three serial ports are available: the serial debug port which is mapped to virtual vector channel 0 in the HAL diagnostic driver or "`/dev/dbg`" in the interrupt-driven driver; USART 0 which is mapped to virtual vector channel 1 and "`/dev/ser0`"; and USART 1 which is mapped to virtual vector channel 2 and "`/dev/ser1`". Only USART 0 supports full modem control signals but USART 1 supports RTS/CTS.

MCI Driver

As the SAM MCI driver is included in the hardware-specific configuration for this target, nothing is required to load it. Similarly the MMC/SD bus driver layer (`CYGPKG_DEVS_DISK_MMC`) is automatically included as part of the hardware-specific configuration for this target. All that is required to enable the support is to include the generic disk I/O infrastructure package (`CYGPKG_IO_DISK`), along with the intended filesystem, typically, the FAT filesystem (`CYGPKG_FS_FAT`) and any of its package dependencies (including `CYGPKG_LIBC_STRING` and `CYGPKG_LINUX_COMPAT` for FAT).

If the generic disk I/O infrastructure is needed for some other reason, and you do not wish to also include the MCI driver, then the configuration option within this platform HAL `CYGPKG_HAL_ARM_ARM9_SAM9260EK_MMCSD` can be used to forcibly disable it.

Various options can be used to control specifics of the SAM MCI driver. Consult the SAM MCI driver documentation for information on its configuration.

On this target, it is not possible to detect from the MMC/SD socket whether cards have been inserted or removed. Thus the disk I/O layer's removeable media support will not detect when cards have been inserted or removed, and therefore the only way to detect if a card has been inserted is to attempt mounts.

The MMC/SD socket also does not permit detection of the write-protect (or "lock") switch present on SD cards. "Locked" cards will therefore not be detected which means that despite the switch position, it is still possible to write to them since the lock switch does not physically enforce write protection.

Compiler Flags

The platform HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos. There are just three flags specific to this port:

`-mcpu=arm9`
The arm-eabi-gcc compiler supports many variants of the ARM architecture. A `-m` option should be used to select the specific variant in use, and with current tools `-mcpu=arm9` is the correct option for the ARM926EJ CPU in the AT91SAM9260.

`-mthumb`
The arm-eabi-gcc compiler will compile C and C++ files into the Thumb instruction set when this option is used. The best way to build eCos in Thumb mode is to enable the configuration option `CYGHWR_THUMB`.

`-mthumb-interwork`
This option allows programs to be created that mix ARM and Thumb instruction sets. Without this option, some memory can be saved. This option should be used if -mthumb is used. The best way to build eCos with Thumb interworking is to enable the configuration option `CYGBLD_ARM_ENABLE_THUMB_INTERWORK`.

Onboard NAND

The HAL port includes a low-level driver to access the on-board Samsung K9F2G08U08 NAND flash memory chip. To enable the driver, activate the CDL option `CYGPKG_HAL_SAM9260EK_NAND` and ensure that the `CYGPKG_DEVS_NAND_SAMSUNG_K9` package is present in your eCos configuration.

`CYGHWR_HAL_SAM9260EK_NAND_USE_STATUS_LINE`
If set, this option configures the driver to wait for NAND operations to complete by waiting for the chip to deassert its Busy line. This is the default behaviour and is recommended, but may be disabled if you need to use the line (PIO C13) for some other purpose. (If disabled, the memory controller is configured to stall NAND accesses until they complete, which will interfere with multi-threading.)

`CYGNUM_HAL_SAM9260EK_NAND_POLL_INTERVAL`
The number of microseconds delay in the polling loops which wait for NAND operations to complete.

Partitioning the NAND chip

The NAND chip must be partitioned before it can become available to applications.

A CDL script which allows the chip to be manually partitioned is provided (see `CYGSEM_DEVS_NAND_SAM9260EK_PARTITION_MANUAL_CONFIG`); if you choose to use this, the relevant data structures will automatically be set up for you when the device is initialised. By default, the manual config CDL script sets up a single partition (number 0) encompassing the entire device.

It is possible to configure the partitions in some other way, should it be appropriate for your setup, for example to read a Linux-style partition table from the chip. To do so you will have to add appropriate code to `sam9260ek_nand.c`.

Name

JTAG support — Usage

Use of JTAG for debugging

JTAG can be used to single-step and debug loaded applications, including RedBoot. Only JTAG configuration applications should be debugged using JTAG, RAM applications assume the presence of RedBoot.

Ronetix PEEDI notes

On the Ronetix PEEDI, the `peedi.at91sam9260ek.cfg` file should be used to setup and configure the hardware to an appropriate state to load programs. This includes setting up the SDRAM controller.

The `peedi.at91sam9260ek.cfg` file also contains an option to define whether hardware or software breakpoints are used by default, using the `CORE0_BREAKMODE` directive in the `[PLATFORM_ARM]` section. Edit this file if you wish to use hardware break points, and remember to restart the PEEDI to make the changes take effect.

On the PEEDI, debugging can be performed either via the telnet interface or using **arm-eabi-gdb** and the GDB interface. In the case of the latter, **arm-eabi-gdb** needs to connect to TCP port 2000 on the PEEDI's IP address. For example:

```
(gdb) target remote 111.222.333.444:2000
```

By default when the PEEDI is powered up, the target will always run the initialization section of the `peedi.at91sam9260ek.cfg` file (which configures the SDRAM among other things), and halts the target. This behavior is repeated with the **reset** command.

If the board is reset (either with the '**reset**', or by pressing the reset button) and the '**go**' command is then given, then the board will boot as normal. If a second-level bootstrap and ROM RedBoot is resident in DataFlash, it will be run.

An issue occurs when the AT91 Ethernet driver is included in your configuration. In order to work around a board hardware design issue, the CPU generates an external reset in order to reset the Ethernet PHY. However this can be interpreted by the PEEDI as an indication that the CPU itself has reset, and if the PEEDI configuration file option `CORE0_STARTUP_MODE` is set to `RESET` then the CPU will be halted at this point. To avoid this issue, the `CORE0_STARTUP_MODE` can be set to `RUN`.

Consult the PEEDI documentation for information on other features.

Running JTAG applications

Applications configured for JTAG startup can be run directly under a JTAG debugger. Once loaded and running via JTAG, HAL diagnostic output will appear by default on the serial debug port. USARTs 0 or 1 can be chosen instead by setting the `CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL` configuration option in the platform HAL to channel 1 or 2.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the AT91SAM9260-EK hardware, and should be read in conjunction with that specification. The AT91SAM9260-EK platform HAL package complements the ARM architectural HAL, the ARM9 variant HAL and the SAM9 processor HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset, the HAL will initialize or reinitialize most of the on-chip peripherals. There is an exception for RAM startup applications which depend on a ROM monitor for certain services.

For ROM startup, the HAL will perform additional initialization. This is all done in the `PLATFORM_SETUP1` macro in the assembler header file `hal_platform_setup.h`.

Linker Scripts and Memory Maps

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

SDRAM

This is located at address 0x20000000 of the physical memory space. The HAL configures the MMU to retain the SDRAM at virtual address 0x20000000, but in order to assign hardware exception vectors vectors at address 0x00000000, the HAL also uses the MMU to create a clone of this memory at virtual address 0x00000000. The same memory is also accessible uncached and unbuffered at virtual location 0x30000000 for use by devices. The first 32 bytes are used for hardware exception vectors. The next 32 bytes are used for the VSR table and the next 256 bytes are normally used for the eCos virtual vectors, allowing RAM-based applications to use services provided by the ROM monitor. Memory is required for the MMU tables, and must be aligned on a 16Kbyte boundary. These therefore occupy memory from 0x4000 to 0x8000. For ROM startup, all remaining SDRAM is available. For RAM startup, available RAM starts at virtual location 0x20040000, with the bottom 256kB reserved for use by RedBoot.

On-chip SRAM

This is located at address 0x00200000 of the physical memory space. However the HAL uses the MMU to relocate this to virtual address 0x70000000. The same memory is also accessible uncached and unbuffered at virtual location 0x70100000 for use by devices. At present this memory is unused by eCos and is available for application use.

On-chip ROM

This is located at address 0x00100000 of the physical memory space. However the HAL uses the MMU to relocate this to virtual address 0x71000000. The same memory is also accessible uncached and unbuffered at virtual location 0x71800000.

USB host port

The USB host port (UHP) registers are located at address 0x00300000 of the physical memory space. However the HAL uses the MMU to relocate this to virtual address 0x72800000. Memory accessed at this address is uncached and unbuffered. There is no cached variant.

SPI dataflash

SPI Dataflash media can only be accessed with the Flash API. For the purposes of this API a placeholder address range has been allocated as if the Flash is present at this address. The base of this address range is 0x40000000 for the on-board flash and 0x50000000 for the dataflash slot, the extent will clearly depend on the Dataflash capacity. This reserved range is not real memory and any attempt to access it directly by the processor other than via the Flash API will result in a memory address exception.

On-chip Peripheral Registers

These are located at address 0xFF000000 in the physical memory space. When the MMU is enabled, it sets up a direct, uncached, unbuffered mapping so that these registers remain accessible at their physical locations.

Off-chip Peripherals

eCos uses the SDRAM, ethernet PHY, MCI, and SPI dataflash facilities on the AT91SAM9260-EK board. eCos does not currently make any use of any other off-chip peripherals present on this board.

Advanced Interrupt Controller

This port has been designed to exploit benefits of the Advanced Interrupt Controller of the AT91SAM9260, using the facilities of the SAM9 processor HAL. Consult the documentation in that package for details.

SPI Dataflash

eCos supports SPI access to Dataflash on the AT91SAM9260-EK. An on-board device and an external card slot are provided on the board. The on-chip device is typically used to contain RedBoot and flash configuration data. The external slot is available for application use.

Accesses to Dataflash are performed via the Flash API, using 0x40000000 or 0x50000000 as the nominal address of the device, although it does not truly exist in the processor address space. For the external card slot, on driver initialisation, eCos and RedBoot can detect the presence of a card in the socket. In particular, on reset RedBoot will indicate the presence of Flash at the 0x40000000 address range in its startup banner if it has been successfully detected. Hot swapping is not possible.

Since Dataflash is not directly addressable, access from RedBoot is only possible using **fis** command operations.

The MCI driver cannot be enabled simultaneously with the SPI driver, as the drivers need differing pin configurations for the same pins on this board due to the shared socket.

Real-time characterization

The tm_basic kernel test gives statistics gathered about the real-time characterization and performance of the kernel. The sample output is shown here for information. The test was built in ARM mode, which provided better performance than Thumb mode.

Example 258.1. sam9260ek Real-time characterization

```
            Startup, main stack : stack used   420 size  3920
             Startup :  Interrupt stack used   528 size  4096
             Startup : Idlethread stack used    80 size  2048

eCos Kernel Timings
Notes: all times are in microseconds (.000001) unless otherwise stated

Reading the hardware clock takes 1 'ticks' overhead
... this value will be factored out of all other measurements
Clock interrupt took    6.28 microseconds (38 raw clock ticks)

Testing parameters:
   Clock samples:            32
   Threads:                  64
   Thread switches:         128
   Mutexes:                  32
   Mailboxes:                32
   Semaphores:               32
   Scheduler operations:    128
   Counters:                 32
   Flags:                    32
   Alarms:                   32
```

```
                                 Confidence
     Ave     Min     Max     Var  Ave  Min  Function
  ======  ======  ======  ====== ========== ========
    5.07    3.38    8.70    0.87   46%  29% Create thread
    0.86    0.81    2.26    0.08   81%  81% Yield thread [all suspended]
    1.06    0.97    2.58    0.11   95%  56% Suspend [suspended] thread
    1.05    0.97    2.26    0.11   92%  67% Resume thread
    1.47    1.29    3.87    0.12   62%  25% Set priority
    0.41    0.16    0.81    0.10   92%   1% Get priority
    3.18    2.90    9.02    0.25   64%  62% Kill [suspended] thread
    0.85    0.81    1.93    0.08   79%  79% Yield [no other] thread
    1.65    1.45    3.54    0.13   73%  23% Resume [suspended low prio] thread
    1.06    0.97    2.42    0.11   95%  60% Resume [runnable low prio] thread
    1.45    1.29    3.70    0.09   64%  28% Suspend [runnable] thread
    0.87    0.81    2.09    0.09   73%  73% Yield [only low prio] thread
    1.06    0.97    2.42    0.11   92%  60% Suspend [runnable->not runnable]
    3.09    2.74    9.18    0.26   84%  54% Kill [runnable] thread
    2.42    2.09    5.80    0.15   39%   1% Destroy [dead] thread
    4.27    3.87   10.47    0.29   84%  42% Destroy [runnable] thread
    6.09    5.48   11.44    0.35   75%  45% Resume [high priority] thread
    2.00    1.93    4.51    0.09   73%  73% Thread switch

    0.16    0.16    0.64    0.01   99%  99% Scheduler lock
    0.68    0.64    0.97    0.05   80%  80% Scheduler unlock [0 threads]
    0.68    0.64    0.97    0.05   80%  80% Scheduler unlock [1 suspended]
    0.68    0.64    1.13    0.05   80%  80% Scheduler unlock [many suspended]
    0.68    0.64    0.97    0.05   80%  80% Scheduler unlock [many low prio]

    0.29    0.16    1.13    0.09   62%  34% Init mutex
    1.05    0.81    2.09    0.13   84%   9% Lock [unlocked] mutex
    1.20    0.97    2.90    0.15   81%  15% Unlock [locked] mutex
    1.00    0.81    2.42    0.12   62%  21% Trylock [unlocked] mutex
    0.93    0.81    1.93    0.11   53%  43% Trylock [locked] mutex
    0.20    0.16    0.64    0.07   81%  81% Destroy mutex
    5.72    5.32    7.73    0.21   68%  21% Unlock/Lock mutex

    0.43    0.32    1.61    0.12   96%  53% Create mbox
    0.33    0.16    0.64    0.07   65%  18% Peek [empty] mbox
    1.26    1.13    2.42    0.16   81%  81% Put [first] mbox
    0.32    0.16    0.64    0.06   65%  18% Peek [1 msg] mbox
    1.27    1.13    2.74    0.13   43%  46% Put [second] mbox
    0.32    0.16    0.64    0.07   65%  21% Peek [2 msgs] mbox
    1.32    1.13    3.22    0.16   62%  34% Get [first] mbox
    1.33    1.13    3.22    0.15   75%  21% Get [second] mbox
    1.11    0.97    2.09    0.12   43%  40% Tryput [first] mbox
    1.12    0.97    2.09    0.13   37%  43% Peek item [non-empty] mbox
    1.21    0.97    2.42    0.14   84%   6% Tryget [non-empty] mbox
    0.99    0.81    1.77    0.08   71%  12% Peek item [empty] mbox
    1.04    0.97    2.26    0.11   96%  75% Tryget [empty] mbox
    0.33    0.16    0.81    0.10   56%  25% Waiting to get mbox
    0.35    0.16    0.64    0.07   71%   9% Waiting to put mbox
    0.60    0.48    1.93    0.13   93%  56% Delete mbox
    4.33    3.87    8.70    0.37   84%  71% Put/Get mbox

    0.31    0.16    1.29    0.08   68%  28% Init semaphore
    0.87    0.81    2.09    0.10   96%  84% Post [0] semaphore
    0.95    0.81    2.09    0.11   56%  37% Wait [1] semaphore
    0.85    0.64    2.09    0.10   78%   9% Trywait [0] semaphore
    0.82    0.64    1.29    0.03   93%   3% Trywait [1] semaphore
    0.31    0.16    0.97    0.08   65%  25% Peek semaphore
    0.22    0.16    0.81    0.09   75%  75% Destroy semaphore
    3.43    3.06    5.96    0.22   87%  37% Post/Wait semaphore

    0.42    0.32    1.61    0.12   96%  62% Create counter
    0.29    0.16    0.97    0.09   62%  34% Get counter value
    0.24    0.16    0.81    0.11   90%  65% Set counter value
    1.06    0.81    2.09    0.13   84%   9% Tick counter
    0.30    0.16    1.45    0.11   56%  37% Delete counter

    0.28    0.16    1.29    0.11   50%  46% Init flag
    0.96    0.81    2.74    0.13   50%  43% Destroy flag
    0.82    0.64    1.93    0.12   53%  28% Mask bits in flag
    0.97    0.81    2.09    0.10   56%  31% Set bits in flag [no waiters]
    1.28    1.13    2.74    0.10   62%  34% Wait for flag [AND]
    1.28    1.13    2.90    0.11   62%  34% Wait for flag [OR]
    1.28    1.13    2.90    0.12   56%  37% Wait for flag [AND/CLR]
    1.26    1.13    2.90    0.13   96%  50% Wait for flag [OR/CLR]
    0.17    0.16    0.32    0.01   96%  96% Peek on flag

    0.60    0.48    1.93    0.13   93%  53% Create alarm
    1.70    1.45    4.35    0.20   75%  71% Initialize alarm
    0.98    0.81    2.42    0.10   68%  25% Disable alarm
    1.65    1.45    4.99    0.22   96%  90% Enable alarm
    1.16    0.97    2.74    0.13   71%  21% Delete alarm
    1.07    0.97    1.45    0.09   56%  40% Tick counter [1 alarm]
    4.89    4.83    5.96    0.10   81%  81% Tick counter [many alarms]
    1.89    1.77    3.06    0.13   93%  56% Tick & fire counter [1 alarm]
   29.89   29.80   30.93    0.11   96%  62% Tick & fire counters [>1 together]
    5.70    5.64    7.25    0.11   96%  87% Tick & fire counters [>1 separately]
    5.66    5.64    7.73    0.04   97%  97% Alarm latency [0 threads]
    6.43    5.80    8.54    0.37   54%  37% Alarm latency [2 threads]
   14.05   12.56   15.79    0.81   46%  33% Alarm latency [many threads]
    8.96    8.86   15.14    0.15   96%  78% Alarm -> thread resume latency

    2.26    1.45    6.12    0.00            Clock/interrupt latency

    2.69    1.77    6.93    0.00            Clock DSR latency

   33      0     292  (main stack:  1388)  Thread stack used (8016 total)
           All done, main stack : stack used  1388 size  3920
            All done :  Interrupt stack used   208 size  4096
            All done : Idlethread stack used   268 size  2048

Timing complete - 30140 ms total

PASS:<Basic timing OK>
EXIT:<done>
```

Other Issues

The AT91SAM9260-EK platform HAL does not affect the implementation of other parts of the eCos HAL specification. The SAM9 processor HAL, ARM9 variant HAL, and the ARM architectural HAL documentation should be consulted for further details.

Chapter 259. Atmel AT91SAM9261 Evaluation Kit Board Support

Name

eCos Support for the Atmel AT91SAM9261 Evaluation Kit — Overview

Description

This document covers the configuration and usage of eCos and RedBoot on the Atmel AT91SAM9261 Evaluation Kit. The AT91SAM9261 Evaluation Kit contains the AT91SAM9261 microprocessor, 64Mbytes of SDRAM, 256Mbytes of NAND flash memory, an Atmel Dataflash, a Davicom DM9000 MAC+PHY, a SD/MMC/DataFlash socket, a DAC, an LCD display, external connections for a DEBUG serial channel, ethernet, USB host/device, and the various other peripherals supported by the AT91SAM9261. eCos support for the many devices and peripherals on the boards and the AT91SAM9261 is described below.

For typical eCos development, a RedBoot image is programmed into the on-board dataflash memory, and the board will load this image from reset. RedBoot provides gdb stub functionality so it is then possible to download and debug stand-alone and eCos applications via the gdb debugger. This can happen over either a serial line or over ethernet.

This documentation is expected to be read in conjunction with the SAM9 processor HAL documentation and further device support and subsystems are described and documented there.

Supported Hardware

The on-board Dataflash consists of 8192 blocks of 1056 bytes each. In a typical setup, the first 32K bytes are reserved for the second-level bootstrap, AT91Bootstrap. The following 164736 bytes are reserved for the use of the ROM RedBoot image (The odd size aligns the end of the RedBoot area to a 1056 block boundary). The topmost block is used to manage the flash and the next block down holds RedBoot **fconfig** values. The remaining blocks can be used by application code.

There is a serial driver `CYGPKG_IO_SERIAL_ARM_AT91` which supports both the Debug Unit and USART serial devices. The debug serial port at J15 can be used by RedBoot for communication with the host. If this device is needed by the application, either directly or via the serial driver, then it cannot also be used for RedBoot communication. Another communication channel such as ethernet should be used instead. The serial driver package is loaded automatically when configuring for the AT91SAM9261EK target.

There is an ethernet driver `CYGPKG_DEVS_ETH_DAVICOM_DM9000` for the DM9000 ethernet device. The platform HAL package is responsible for configuring this generic driver to the hardware. This driver is also loaded automatically when configuring for the AT91SAM9261EK board.

There is a watchdog driver `CYGPKG_DEVICES_WATCHDOG_ARM_AT91WDTC`. This driver is also loaded automatically when configuring for the board.

There is a driver for the on-chip real-time timer controller (RTTC) at `CYGPKG_DEVICES_WALLCLOCK_ARM_AT91RTTC`. This driver is also loaded automatically when configuring for the target.

The SAM9 processor HAL contains a driver for the Two-Wire Interface (TWI) controller on the AT91SAM9261. This type of bus is also known as I²C®. Further documentation may be found in the SAM9 processor HAL documentation.

There is a driver for the MultiMedia Card Interface (MCI) at `CYGPKG_DEVS_MMCSD_ATMEL_SAM_MCI`. This driver is loaded automatically when configuring for the SAM9261-EK target and allows use of MMC and Secure Digital (SD) flash storage cards within eCos, exported as block devices. Further documentation may be found within that package.

The platform HAL provides definitions to allow access to devices on the SPI bus. The HAL provides information to the more general AT91 SPI driver (`CYGPKG_DEVS_SPI_ARM_AT91`) which in turn provides the underlying implementation for the SPI API layer in the `CYGPKG_IO_SPI` package. All these packages are automatically loaded when configuring for the board..

Furthermore, the platform HAL package contains support for SPI dataflash cards. The HAL support integrates with the `CYGPKG_DEVS_FLASH_ATMEL_DATAFLASH` package as well as the above SPI packages. That package is automatically loaded when configuring for the target. Dataflash media is then accessed as a Flash device, using the Flash I/O API within the `CYGPKG_IO_FLASH` package, if that package is loaded in the configuration.

It is also possible to configure the HAL to access MMC cards in SPI mode, instead of using the MCI interface.

In general, devices (Caches, PIO, UARTs) are initialized only as far as is necessary for eCos to run. Other devices (RTC, I²C, SPI, MCI etc.) are not touched unless the appropriate driver is loaded, although in some cases, the HAL boot sequence will set up the appropriate PIO configuration.

Tools

The AT91SAM9261-EK support is intended to work with GNU tools configured for an arm-eabi target. The original port was undertaken using arm-elf-gcc version 3.4.4, arm-elf-gdb version 6.3, and binutils version 2.15.

Name

Setup — Preparing the AT91SAM9261-EK board for eCos Development

Overview

In a typical development environment, the AT91SAM9261-EK board boots from the DataFlash and run the RedBoot ROM monitor from SDRAM. eCos applications are configured for RAM startup and then downloaded and run on the board via the debugger **arm-eabi-gdb**. Preparing the board therefore usually involves programming a suitable RedBoot image into flash memory.

The following RedBoot configurations are supported:

Configuration	Description	Use	File
ROM	RedBoot loaded from Dataflash to SDRAM	`redboot_ROM.ecm`	`redboot_ROM.bin`

For serial communications, all versions run with 8 bits, no parity, and 1 stop bit at 115200 baud. RedBoot also supports ethernet communication and flash management.

Note that the use of the term ROM for the initial RedBoot configuration is an historical accident. RedBoot actually runs from SDRAM after being loaded there from Dataflash by the second-level bootstrap. The use of ROM for this configuration is intended to indicate that it initializes the microprocessor and board peripherals, as opposed to the RAM configuration which assumes that this has already been done.

Initial Installation

The on-chip boot program on the AT91SAM9261 is only capable of loading programs from Dataflash or NAND flash into on-chip SRAM and is therefore quite restrictive. Consequently RedBoot cannot be booted directly and a second-level bootstrap must be used. Such a second-level bootstrap is supplied by Atmel in the form of AT91Bootstrap. This is therefore programmed into the start of Dataflash and is then responsible for initializing the SDRAM and loading RedBoot from Dataflash and executing it.

Caution

There is a size limit on the size of applications which the AT91Bootstrap second level bootstrap will load. Images larger than 320Kbytes will require the AT91Bootstrap application to be rebuilt with a larger `IMG_SIZE` definition in `AT91Bootstrap/board/at91sam9261ek/dataflash/at91sam9261ek.h` within the sam9260ek HAL package in the eCos source repository (`packages/hal/arm/arm9/sam9260ek/current/`).

There are basically two ways to write the second-level bootstrap and RedBoot to the Dataflash. The first is to use the Atmel-supplied SAM-BA program that interacts with the on-chip boot program. The second is to use a JTAG debugger that understands the microcontroller and can write to the dataflash (for example the Ronetix PEEDI). Since the availability of the latter cannot be guaranteed, only the first method will be described here.

Programming RedBoot into DataFlash using SAM-BA

The following gives the steps needed to program the second-level bootstrap and RedBoot into the DataFlash using SAM-BA. The user should refer to the SAM-BA documentation for full details of how to run the program.

1. Download the AT91 In-system Programmer software package from the Atmel website. SAM9 series CPU's require the 2.1.x series version of SAM-BA. Install it on a suitable PC running Windows.

2. From the root directory of your eCosPro installation, copy the file `dataflash_at91sam9261ek.bin` from the sub-directory `packages/hal/arm/arm9/sam9260ek/current/AT91Bootstrap/board/at91sam9261ek` and `redboot_ROM.bin` from the sub-directory `loaders/sam9261ek` to a suitable location on the Windows PC.

3. Connect a null-modem serial cable between the DEBUG serial port of the board and a serial port on a convenient host (which need not be the PC running SAM-BA). Run a terminal emulator (Hyperterm or minicom) at 115200 baud. Connect a USB cable between the PC and the AT91SAM9261-EK board.

4. Power up or reset the board and Windows should recognize the USB device. If it does not, then move J21 to the 2-3 position and reset the board, it should be recognized now. Windows may ask you to install a new driver, in which case follow the instructions.

5. Start SAM-BA. Select "\usb\ARM0" for the communication interface, and "AT91SAM9261-EK" for the board. If the USB option does not appear, check the cable and look in the Windows Device Manager for the new USB COM device. If all is well, click on "Connect".

6. If you moved J21 to 2-3, move it back to the default 1-2 position.

7. In the SAM-BA main window, select the "DataFlash AT45DB/DBC" tab and in the "Scripts" dropdown menu select "Enable Dataflash (SPI0 CS0)", to program the on-board Dataflash device. Click Execute and SAM-BA should emit the following in the message area:

   ```
   (AT91-ISP v1.13) 1 % DATAFLASH::Init 0
   -I- DATAFLASH::Init 0 (trace level : 4)
   -I- Loading applet isp-dataflash-at91sam9261.bin at address 0x20000000
   -I- Memory Size : 0x840000 bytes
   -I- Buffer address : 0x20002A40
   -I- Buffer size: 0x80E80 bytes
   -I- Applet initialization done
   ```

8. Select "Send BootFile" from the "Scripts" menu and "Execute" it. When the file open dialog appears, select the `dataflash_at91sam9261ek.bin` file and click "Open". The following output should be seen:

   ```
   (AT91-ISP v1.13) 1 % GENERIC::SendBootFileGUI
   GENERIC::SendFile dataflash_at91sam9261ek.bin at address 0x0
   -I- File size : 0x10A2 byte(s)
   -I-     Writing: 0x10A2 bytes at 0x0 (buffer addr : 0x20002A40)
   -I-     0x10A2 bytes written by applet
   ```

9. The second-level bootstrap has now been written to DataFlash, we must now write RedBoot.

10. In the "Send File Name" box type in the path name to the `redboot_ROM.bin` file, or use the Open Folder button and browse to it.

11. In the Address field set the value to 0x8400.

12. Click the "Send File" button. SAM-BA will put up a dialog box while it is writing the file to the DataFlash, and will output something similar to the following in the message area:

    ```
    (AT91-ISP v1.13) 1 % send_file {DataFlash AT45DB/DCB} "redboot_ROM.bin" 0x8400 0
    -I- Send File //bert/Shared/Releng/sam9261ek/redboot_ROM.bin at address 0x8400
    GENERIC::SendFile //bert/Shared/Releng/sam9261ek/redboot_ROM.bin at address 0x8400
    -I- File size : 0x1F928 byte(s)
    -I-     Writing: 0x1F928 bytes at 0x8400 (buffer addr : 0x20002A40)
    -I-     0x1F928 bytes written by applet
    ```

13. Shut down SAM-BA and disconnect the USB cable. Press the reset button on the board and something similar to the following should be output on the DEBUG serial line.

    ```
    RomBOOT
    >Start AT91Bootstrap...
    +**Warning** FLASH configuration checksum error or invalid key
    Use 'fconfig -i' to [re]initialize database
    >Start AT91Bootstrap...
    No network interfaces found

    RedBoot(tm) bootstrap and debug environment [ROM]
    ```

```
eCosCentric certified release, version v3_0_3 - built 12:53:09, Sep 24 2009

Copyright (C) 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009
Free Software Foundation, Inc.
Copyright (C) 2003, 2004, 2005, 2006, 2007, 2008 eCosCentric Limited
RedBoot is free software, covered by the eCos license, derived from the
GNU General Public License. You are welcome to change it and/or distribute
copies of it under certain conditions. Under the license terms, RedBoot's
source code and full license terms must have been made available to you.
Redboot comes with ABSOLUTELY NO WARRANTY.

Platform: SAM9261-EK (ARM9)
RAM: 0x20000000-0x24000000 [0x20031230-0x23ffef80 available]
FLASH: 0x40000000-0x4083ffff, 8192 x 0x420 blocks
RedBoot>
```

RedBoot Flash configuration

The following steps describe how to initialize RedBoot's Flash configuration.

1. Use the following command to initialize RedBoot's Flash Information System (FIS):

```
RedBoot> fis init
About to initialize [format] FLASH image system - continue (y/n)? y
*** Initialize FLASH Image System
... Erase from 0x4083fbe0-0x4083ffff: .
... Program from 0x23fffbe0-0x24000000 to 0x4083fbe0: .
RedBoot> fis list
Name              FLASH addr  Mem addr    Length      Entry point
(reserved)        0x40000000  0x40000000  0x00008000  0x00000000
RedBoot           0x40008000  0x40008000  0x00028380  0x00000000
RedBoot config    0x4083F7C0  0x4083F7C0  0x00000420  0x00000000
FIS directory     0x4083FBE0  0x4083FBE0  0x00000420  0x00000000
RedBoot>
```

2. Now configure RedBoot's Flash configuration with the command:

```
RedBoot> fconfig -i
```

Remember to substitute the appropriate MAC address for this board at the appropriate step. If a BOOTP/DHCP server is not available, then IP configuration may be set manually. The default server IP address can be set to a PC that will act as a TFTP host for future RedBoot load operations, or may be left unset. The following gives an example configuration:

```
RedBoot> fconfig -i
Initialize non-volatile configuration - continue (y/n)? y
Run script at boot: false
Use BOOTP for network configuration: false
Gateway IP address: 192.168.7.11
Local IP address: 192.168.7.222
Local IP address mask: 255.255.255.0
Default server IP address: 192.168.7.9
Console baud rate: 115200
DNS server IP address: 192.168.7.11
Network hardware address [MAC]: 0x00:0x23:0x31:0x37:0x00:0x4e
GDB connection port: 9000
Force console for special debug messages: false
Update RedBoot non-volatile configuration - continue (y/n)? y
... Erase from 0x4083f7c0-0x4083fbdf: .
... Program from 0x23fff7c0-0x23fffbe0 to 0x4083f7c0: .
RedBoot>
```

The RedBoot installation is now complete. This can be tested by powering off the board, and then powering on the board again. Output similar to the following should be seen on the DEBUG serial port. Verify the IP settings are as expected.

```
RomBOOT
```

```
>Start AT91Bootstrap...
+Ethernet eth0: MAC address 00:03:47:df:32:a8
IP: 192.168.7.85/255.255.255.0, Gateway: 192.168.7.11
Default server: 192.168.7.11
DNS server IP: 192.168.7.11, DNS domain name: <null%gt;

RedBoot(tm) bootstrap and debug environment [ROM]
eCosCentric certified release, version v3_0_3 - built 12:53:09, Sep 24 2009

Copyright (C) 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009
Free Software Foundation, Inc.
Copyright (C) 2003, 2004, 2005, 2006, 2007, 2008 eCosCentric Limited
RedBoot is free software, covered by the eCos license, derived from the
GNU General Public License. You are welcome to change it and/or distribute
copies of it under certain conditions. Under the license terms, RedBoot's
source code and full license terms must have been made available to you.
Redboot comes with ABSOLUTELY NO WARRANTY.

Platform: SAM9261-EK (ARM9)
RAM: 0x20000000-0x24000000 [0x20031230-0x23ffef80 available]
FLASH: 0x40000000-0x4083ffff, 8192 x 0x420 blocks
RedBoot>
```

If it proves necessary to re-install RedBoot, this may be achieved by repeating the above process. Alternatively, a new image may be downloaded and programmed into flash more directly using RedBoot's own commands. See the RedBoot documentation for details.

Rebuilding RedBoot

Should it prove necessary to rebuild a RedBoot binary, this is done most conveniently at the command line. The steps needed to rebuild the ROM version of RedBoot for the AT91SAM9261-EK are:

```
$ mkdir redboot_at91sam9261ek_rom
$ cd redboot_at91sam9261ek_rom
$ ecosconfig new at91sam9261ek redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/arm/arm9/sam9261ek/current/misc/redboot_ROM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

At the end of the build the `install/bin` subdirectory should contain the file `redboot.bin`.

Rebuilding AT91Bootstrap

The sources of AT91Bootstrap are found in the `AT91Bootstrap` directory of the sam9260ek package. This is a copy of the software as supplied by Atmel with some slight modifications to permit it to be built with the same tools as eCos.

To rebuild the second-level bootstrap for the AT91SAM9261EK execute the following commands:

```
$ cd $ECOS_REPOSITORY/hal/arm/arm9/sam9260ek/current/AT91Bootstrap/board/at91sam9261ek/dataflash
$ make
```

This should result in the creation of a number of files, including `dataflash_at91sam9261ek.bin` which can be copied out.

Name

Configuration — Platform-specific Configuration Options

Overview

The AT91SAM9261-EK platform HAL package is loaded automatically when eCos is configured for the `sam9261ek` target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The platform HAL package supports two separate startup types:

RAM
: This is the startup type which is normally used during application development. The board has RedBoot programmed into flash and boots into that initially. arm-eabi-gdb is then used to load a RAM startup application into memory and debug it. It is assumed that the hardware has already been initialized by RedBoot. By default the application will use the eCos virtual vectors mechanism to obtain certain services from RedBoot, including diagnostic output.

ROM
: This startup type can be used for finished applications which will be programmed into DataFlash. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization.

JTAG
: This is the startup type which can be used during application development via a JTAG device such as the PEEDI. arm-eabi-gdb is used to load a JTAG startup application into memory and debug it. Hardware setup is divided between the initialization section of the PEEDI configuration file and software in the loaded application.

RedBoot and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building RedBoot.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then the serial port will be claimed for HAL diagnostics.

Flash Driver

The AT91SAM9261-EK board contains an 8Mbyte Atmel AT45DB DataFlash device. The `CYGPKG_DEVS_FLASH_ATMEL_DATAFLASH` package contains all the code necessary to support this part and the platform HAL package contains definitions that customize the driver to the AT91SAM9261-EK board. This driver is not active until the generic Flash support package, `CYGPKG_IO_FLASH`, is included in the configuration.

Ethernet Driver

The AT91SAM9261-EK board uses a Davicom DM9000 ethernet MAC and PHY chip for ethernet connectivity. The `CYGPKG_DEVS_ETH_DAVICOM_DM9000` package contains all the code necessary to support this device and the platform HAL package contains definitions that customize the driver to the AT91SAM9261-EK board. This driver is not active until the generic Ethernet support package, CYGPKG_IO_ETH_DRIVERS, is included in the configuration.

RTC Driver

The AT91SAM9261-EK board uses the AT91SAM9261's internal RTTC support. The CYGPKG_DEVICES_WALLCLOCK_ARM_AT91RTTC package contains all the code necessary to support this device. This driver is not active until the generic wallclock device support package, CYGPKG_IO_WALLCLOCK, is included in the configuration.

Watchdog Driver

The AT91SAM9261-EK board uses the AT91SAM9261's internal watchdog support. The CYGPKG_DEVICES_WATCHDOG_ARM_AT91WDTC package contains all the code necessary to support this device. Within that package the CYGNUM_DEVS_WATCHDOG_ARM_AT91WDTC_DESIRED_TIMEOUT_MS configuration option controls the watchdog timeout, and by default will force a reset of the board upon timeout. This driver is not active until the generic watchdog device support package, CYGPKG_IO_WATCHDOG, is included in the configuration.

Warning

The ATSAM926x processor will boot with watchdog support enabled, and the watchdog configuration is write-once. That is, if it is disabled, it cannot be re-enabled. Due to its nature, RedBoot disables the watchdog when it starts so any eCos applications with watchdog support enabled that are run by RedBoot will not function correctly.

USART Serial Driver

The AT91SAM9261-EK board uses the AT91SAM9261's internal USART serial support as described in the SAM9 processor HAL documentation. Just one serial port is available: the serial debug port which is mapped to virtual vector channel 0 in the HAL diagnostic driver or "/dev/dbg" in the interrupt-driven driver. This serial port only provides basic TX and RX lines, with no modem control or flow control lines. RTS/CTS.

MCI Driver

As the SAM MCI driver is included in the hardware-specific configuration for this target, nothing is required to load it. Similarly the MMC/SD bus driver layer (CYGPKG_DEVS_DISK_MMC) is automatically included as part of the hardware-specific configuration for this target. All that is required to enable the support is to include the generic disk I/O infrastructure package (CYGPKG_IO_DISK), along with the intended filesystem, typically, the FAT filesystem (CYGPKG_FS_FAT) and any of its package dependencies (including CYGPKG_LIBC_STRING and CYGPKG_LINUX_COMPAT for FAT).

If the generic disk I/O infrastructure is needed for some other reason, and you do not wish to also include the MCI driver, then the configuration option within this platform HAL CYGPKG_HAL_ARM_ARM9_SAM9261EK_MMCSD can be used to forcibly disable it.

Various options can be used to control specifics of the SAM MCI driver. Consult the SAM MCI driver documentation for information on its configuration.

On this target, it is not possible to detect from the MMC/SD socket whether cards have been inserted or removed. Thus the disk I/O layer's removeable media support will not detect when cards have been inserted or removed, and therefore the only way to detect if a card has been inserted is to attempt mounts.

The MMC/SD socket also does not permit detection of the write-protect (or "lock") switch present on SD cards. "Locked" cards will therefore not be detected which means that despite the switch position, it is still possible to write to them since the lock switch does not physically enforce write protection.

Compiler Flags

The platform HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos. There are just three flags specific to this port:

`-mcpu=arm9`
: The arm-eabi-gcc compiler supports many variants of the ARM architecture. A `-m` option should be used to select the specific variant in use, and with current tools `-mcpu=arm9` is the correct option for the ARM926EJ CPU in the AT91SAM9261.

`-mthumb`
: The arm-eabi-gcc compiler will compile C and C++ files into the Thumb instruction set when this option is used. The best way to build eCos in Thumb mode is to enable the configuration option `CYGHWR_THUMB`.

`-mthumb-interwork`
: This option allows programs to be created that mix ARM and Thumb instruction sets. Without this option, some memory can be saved. This option should be used if -mthumb is used. The best way to build eCos with Thumb interworking is to enable the configuration option `CYGBLD_ARM_ENABLE_THUMB_INTERWORK`.

Onboard NAND

The HAL port includes a low-level driver to access the on-board Samsung K9F1208U08 NAND flash memory chip. To enable the driver, activate the CDL option `CYGPKG_HAL_SAM9261EK_NAND` and ensure that the `CYGPKG_DEVS_NAND_SAMSUNG_K9` package is present in your eCos configuration.

`CYGHWR_HAL_SAM9261EK_NAND_USE_STATUS_LINE`
: If set, this option configures the driver to wait for NAND operations to complete by waiting for the chip to deassert its Busy line. This is the default behaviour and is recommended, but may be disabled if you need to use the line (PIO C13) for some other purpose. (If disabled, the memory controller is configured to stall NAND accesses until they complete, which will interfere with multi-threading.)

`CYGNUM_HAL_SAM9261EK_NAND_POLL_INTERVAL`
: The number of microseconds delay in the polling loops which wait for NAND operations to complete.

Partitioning the NAND chip

The NAND chip must be partitioned before it can become available to applications.

A CDL script which allows the chip to be manually partitioned is provided (see `CYGSEM_DEVS_NAND_SAM9261EK_PARTITION_MANUAL_CONFIG`); if you choose to use this, the relevant data structures will automatically be set up for you when the device is initialised. By default, the manual config CDL script sets up a single partition (number 0) encompassing the entire device.

It is possible to configure the partitions in some other way, should it be appropriate for your setup, for example to read a Linux-style partition table from the chip. To do so you will have to add appropriate code to `sam9261ek_nand.c`.

Name

JTAG support — Usage

Use of JTAG for debugging

JTAG can be used to single-step and debug loaded applications, including RedBoot. Only JTAG configuration applications should be debugged using JTAG, RAM applications assume the presence of RedBoot.

Ronetix PEEDI notes

On the Ronetix PEEDI, the `peedi.at91sam9261ek.cfg` file should be used to setup and configure the hardware to an appropriate state to load programs. This includes setting up the SDRAM controller.

The `peedi.at91sam9261ek.cfg` file also contains an option to define whether hardware or software breakpoints are used by default, using the `CORE0_BREAKMODE` directive in the `[PLATFORM_ARM]` section. Edit this file if you wish to use hardware break points, and remember to restart the PEEDI to make the changes take effect.

On the PEEDI, debugging can be performed either via the telnet interface or using **arm-eabi-gdb** and the GDB interface. In the case of the latter, **arm-eabi-gdb** needs to connect to TCP port 2000 on the PEEDI's IP address. For example:

```
(gdb) target remote 111.222.333.444:2000
```

By default when the PEEDI is powered up, the target will always run the initialization section of the `peedi.at91sam9261ek.cfg` file (which configures the SDRAM among other things), and halts the target. This behavior is repeated with the **reset** command.

If the board is reset (either with the '**reset**', or by pressing the reset button) and the '**go**' command is then given, then the board will boot as normal. If a second-level bootstrap and ROM RedBoot is resident in DataFlash, it will be run.

Consult the PEEDI documentation for information on other features.

Running JTAG applications

Applications configured for JTAG startup can be run directly under a JTAG debugger. Once loaded and running via JTAG, HAL diagnostic output will appear by default on the serial debug port.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the AT91SAM9261-EK hardware, and should be read in conjunction with that specification. The AT91SAM9261-EK platform HAL package complements the ARM architectural HAL, the ARM9 variant HAL and the SAM9 processor HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset, the HAL will initialize or reinitialize most of the on-chip peripherals. There is an exception for RAM startup applications which depend on a ROM monitor for certain services.

For ROM startup, the HAL will perform additional initialization. This is all done in the `PLATFORM_SETUP1` macro in the assembler header file `hal_platform_setup.h`.

Linker Scripts and Memory Maps

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

SDRAM

This is located at address 0x20000000 of the physical memory space. The HAL configures the MMU to retain the SDRAM at virtual address 0x20000000, but in order to assign hardware exception vectors at address 0x00000000, the HAL also uses the MMU to create a clone of this memory at virtual address 0x00000000. The same memory is also accessible uncached and unbuffered at virtual location 0x30000000 for use by devices. The first 32 bytes are used for hardware exception vectors. The next 32 bytes are used for the VSR table and the next 256 bytes are normally used for the eCos virtual vectors, allowing RAM-based applications to use services provided by the ROM monitor. Memory is required for the MMU tables, and must be aligned on a 16Kbyte boundary. These therefore occupy memory from 0x4000 to 0x8000. For ROM startup, all remaining SDRAM is available. For RAM startup, available RAM starts at virtual location 0x20040000, with the bottom 256kB reserved for use by RedBoot.

On-chip SRAM

This is located at address 0x00200000 of the physical memory space. However the HAL uses the MMU to relocate this to virtual address 0x70000000. The same memory is also accessible uncached and unbuffered at virtual location 0x70100000 for use by devices. At present this memory is unused by eCos and is available for application use.

On-chip ROM

This is located at address 0x00100000 of the physical memory space. However the HAL uses the MMU to relocate this to virtual address 0x71000000. The same memory is also accessible uncached and unbuffered at virtual location 0x71800000.

USB host port

The USB host port (UHP) registers are located at address 0x00300000 of the physical memory space. However the HAL uses the MMU to relocate this to virtual address 0x72800000. Memory accessed at this address is uncached and unbuffered. There is no cached variant.

SPI dataflash

SPI Dataflash media can only be accessed with the Flash API. For the purposes of this API a placeholder address range has been allocated as if the Flash is present at this address. The base of this address range is 0x40000000 for the on-board flash and 0x50000000 for the dataflash slot, the extent will clearly depend on the Dataflash capacity. This reserved range is not real memory and any attempt to access it directly by the processor other than via the Flash API will result in a memory address exception.

On-chip Peripheral Registers
: These are located at address 0xFF000000 in the physical memory space. When the MMU is enabled, it sets up a direct, uncached, unbuffered mapping so that these registers remain accessible at their physical locations.

DM9000 Ethernet Device
: This is located on chip select 2 of the static memory controller and is visible at physical address 0x30000000. The HAL uses the MMU to relocate this to 0x80000000 in the virtual memory space.

Off-chip Peripherals
: eCos uses the SDRAM, MCI, and SPI dataflash facilities on the AT91SAM9261-EK board. eCos does not currently make any use of any other off-chip peripherals present on this board.

Advanced Interrupt Controller

This port has been designed to exploit benefits of the Advanced Interrupt Controller of the AT91SAM9261, using the facilities of the SAM9 processor HAL. Consult the documentation in that package for details.

SPI Dataflash

eCos supports SPI access to Dataflash on the AT91SAM9261. An on-board device and an external card slot are provided on the board. The on-chip device is typically used to contain RedBoot and flash configuration data. The external slot is available for application use.

Accesses to Dataflash are performed via the Flash API, using 0x40000000 or 0x50000000 as the nominal address of the device, although it does not truly exist in the processor address space. For the external card slot, on driver initialisation, eCos and RedBoot can detect the presence of a card in the socket. In particular, on reset RedBoot will indicate the presence of Flash at the 0x40000000 address range in its startup banner if it has been successfully detected. Hot swapping is not possible.

Since Dataflash is not directly addressable, access from RedBoot is only possible using **fis** command operations.

RedBoot or applications can also be booted from the Dataflash card socket, as well as from the on-board Dataflash device. Booting from the Dataflash card socket can be performed by switching jumper J21 from pins 1-2 closed, to pins 2-3 closed. When booting RedBoot or applications in this way, you *must* enable the SAM9261-EK platform HAL configuration option "SPI chip select #0 is socket" (`CYGHWR_HAL_ARM_ARM9_SAM9261EK_DATAFLASH_NPCS0_SOCKET`) found within the "External Atmel AT49xxxx DataFlash memory support" component. Failure to do so will not just render the Dataflash card inaccessible after booting, but is likely to cause permanent damage to the AT91SAM9261.

Once an appropriately configured "ROM" startup image has been built, it can be converted to raw binary format using **arm-eabi-objcopy**. You must then copy an AT91Bootstrap second stage boot loader binary image to the beginning (offset 0) of the card, and then the RedBoot/application image at offset 0x8000 on the card.

Programming on the card can be performed with SAM-BA, an external programmer, or with a standard build of RedBoot booted from the on-board Dataflash. An example of using an installed RedBoot loaded from on-board Dataflash to program the card would be as follows. Note the two Flash memories detected by RedBoot, as shown in the **version** output. The first is the on-board Dataflash part, the second is the Dataflash located in the card socket.

```
RedBoot> version

RedBoot(tm) bootstrap and debug environment [ROM]
Non-certified release, version UNKNOWN - built 15:10:18, Nov  2 2007

Platform: SAM9261-EK (ARM9)
Copyright (C) 2000, 2001, 2002, 2003, 2004 Free Software Foundation, Inc.
Copyright (C) 2003, 2004, 2005, 2006, 2007 eCosCentric Limited

RAM: 0x20000000-0x24000000, [0x200305d0-0x23ffef80] available
FLASH: 0x40000000-0x4083ffff, 8192 x 0x420 blocks
FLASH: 0x50000000-0x5083ffff, 8192 x 0x420 blocks
RedBoot> fis list
Name              FLASH addr  Mem addr    Length      Entry point
```

```
(reserved)        0x40000000  0x40000000  0x00008000  0x00000000
RedBoot           0x40008000  0x20030800  0x00028380  0x20030800
RedBoot config    0x4083F7C0  0x4083F7C0  0x00000420  0x00000000
FIS directory     0x4083FBE0  0x4083FBE0  0x00000420  0x00000000
RedBoot> fis load -b %{freememlo} (reserved)
RedBoot> fis write -b %{freememlo} -f 0x50000000 -l 0x8000
* CAUTION * about to program FLASH
            at 0x50000000..0x500083ff from 0x20030800 - continue (y/n)? y
... Erase from 0x50000000-0x500083ff: ................................
... Program from 0x20030800-0x20038c00 to 0x50000000: ................................
RedBoot> load -r -m tftp -b %{freememlo} /sam9261ek/redboot.bin
Raw file loaded 0x20030800-0x2004f3a3, assumed entry at 0x20030800
RedBoot> fis write -b %{freememlo} -f 0x50008000 -l 0x28380
* CAUTION * about to program FLASH
            at 0x50008000..0x5003037f from 0x20030800 - continue (y/n)? y
... Erase from 0x50007fe0-0x5003037f: ................................................................................
... Program from 0x20030800-0x20058b80 to 0x50008000: ................................................................
RedBoot>
```

After this, the board can be powered off, jumper J21 switched to pins 2-3, and the board powered up again. The application or RedBoot will then boot from the Dataflash card.

The MCI driver cannot be enabled simultaneously with the SPI driver, as the drivers need differing pin configurations for the same pins on this board due to the shared socket.

Real-time characterization

The tm_basic kernel test gives statistics gathered about the real-time characterization and performance of the kernel. The sample output is shown here for information. The test was built in ARM mode, which provided better performance than Thumb mode.

Example 259.1. sam9261ek Real-time characterization

```
            Startup, main stack : stack used   420 size  3920
             Startup :  Interrupt stack used   536 size  4096
             Startup : Idlethread stack used    80 size  2048

eCos Kernel Timings
Notes: all times are in microseconds (.000001) unless otherwise stated

Reading the hardware clock takes 1 'ticks' overhead
... this value will be factored out of all other measurements
Clock interrupt took    6.37 microseconds (39 raw clock ticks)

Testing parameters:
   Clock samples:            32
   Threads:                  64
   Thread switches:         128
   Mutexes:                  32
   Mailboxes:                32
   Semaphores:               32
   Scheduler operations:    128
   Counters:                 32
   Flags:                    32
   Alarms:                   32


                                 Confidence
     Ave     Min     Max     Var  Ave  Min  Function
  ======  ======  ======  ====== ========== ========
    4.95    3.38    6.44    0.81   48%  26% Create thread
    0.84    0.81    1.93    0.06   85%  85% Yield thread [all suspended]
    1.04    0.97    2.26    0.10   98%  64% Suspend [suspended] thread
    0.95    0.81    1.61    0.07   65%  23% Resume thread
    1.35    1.13    3.87    0.12   92%   6% Set priority
    0.26    0.16    0.64    0.10   92%  50% Get priority
    2.98    2.58    8.05    0.24   87%  39% Kill [suspended] thread
    0.85    0.81    1.61    0.07   79%  79% Yield [no other] thread
    1.48    1.29    2.42    0.09   64%  12% Resume [suspended low prio] thread
    0.93    0.81    1.13    0.07   65%  29% Resume [runnable low prio] thread
    1.17    0.97    1.93    0.09   62%  10% Suspend [runnable] thread
    0.84    0.81    1.61    0.05   87%  87% Yield [only low prio] thread
    0.88    0.81    1.29    0.09   93%  59% Suspend [runnable->not runnable]
    2.85    2.58    6.60    0.18   81%  68% Kill [runnable] thread
    2.16    1.93    4.67    0.15   75%  17% Destroy [dead] thread
    3.92    3.70    6.77    0.16   68%  70% Destroy [runnable] thread
    5.76    5.32   10.15    0.29   81%  32% Resume [high priority] thread
    2.02    1.93    4.19    0.10   98%  54% Thread switch

    0.21    0.16    0.64    0.07   74%  74% Scheduler lock
    0.69    0.64    1.29    0.07   75%  75% Scheduler unlock [0 threads]
    0.69    0.64    0.97    0.06   74%  74% Scheduler unlock [1 suspended]
    0.65    0.64    0.81    0.00   99%  99% Scheduler unlock [many suspended]
    0.68    0.64    0.97    0.05   79%  79% Scheduler unlock [many low prio]

    0.31    0.16    1.29    0.08   71%  25% Init mutex
    1.03    0.81    2.58    0.14   84%  12% Lock [unlocked] mutex
    1.13    0.97    2.58    0.09   68%  28% Unlock [locked] mutex
    0.92    0.81    1.93    0.11   50%  46% Trylock [unlocked] mutex
    0.82    0.81    0.97    0.02   93%  93% Trylock [locked] mutex
    0.20    0.16    0.64    0.07   81%  81% Destroy mutex
    5.06    4.99    6.44    0.12   93%  90% Unlock/Lock mutex

    0.42    0.32    1.61    0.12   96%  62% Create mbox
    0.33    0.16    1.13    0.11   50%  28% Peek [empty] mbox
    1.08    0.97    2.26    0.12   93%  53% Put [first] mbox
    0.19    0.16    0.64    0.05   90%  90% Peek [1 msg] mbox
    1.03    0.97    1.93    0.10   96%  75% Put [second] mbox
    0.18    0.16    0.32    0.04   87%  87% Peek [2 msgs] mbox
    1.08    0.97    2.26    0.12   93%  56% Get [first] mbox
    1.03    0.97    1.45    0.09   65%  65% Get [second] mbox
    0.89    0.81    1.61    0.10   96%  59% Tryput [first] mbox
    0.88    0.81    1.93    0.11   96%  71% Peek item [non-empty] mbox
    0.97    0.81    1.77    0.06   78%  15% Tryget [non-empty] mbox
    0.85    0.81    1.13    0.07   75%  75% Peek item [empty] mbox
    0.91    0.81    1.77    0.11   96%  53% Tryget [empty] mbox
    0.19    0.16    0.32    0.04   84%  84% Waiting to get mbox
    0.19    0.16    0.48    0.05   84%  84% Waiting to put mbox
    0.41    0.32    1.29    0.11   96%  62% Delete mbox
    3.59    3.38    6.60    0.19   65%  96% Put/Get mbox

    0.26    0.16    0.81    0.09   50%  46% Init semaphore
    0.84    0.64    1.77    0.07   84%   6% Post [0] semaphore
    0.92    0.81    1.93    0.11   50%  46% Wait [1] semaphore
    0.82    0.64    1.45    0.05   87%   6% Trywait [0] semaphore
    0.82    0.81    1.29    0.03   96%  96% Trywait [1] semaphore
    0.28    0.16    0.81    0.09   59%  37% Peek semaphore
    0.20    0.16    0.48    0.06   78%  78% Destroy semaphore
    3.29    3.06    5.15    0.13   90%   3% Post/Wait semaphore

    0.43    0.32    1.61    0.12   96%  53% Create counter
    0.29    0.16    0.64    0.11   40%  43% Get counter value
    0.20    0.16    0.64    0.07   81%  81% Set counter value
    0.97    0.81    1.61    0.05   81%  12% Tick counter
    0.20    0.16    0.48    0.06   81%  81% Delete counter

    0.26    0.16    0.97    0.10   96%  50% Init flag
    0.91    0.81    2.09    0.12   96%  59% Destroy flag
    0.76    0.64    1.29    0.09   59%  37% Mask bits in flag
    0.91    0.81    1.45    0.10   46%  46% Set bits in flag [no waiters]
    1.28    1.13    2.58    0.09   65%  31% Wait for flag [AND]
    1.23    1.13    1.61    0.09   46%  46% Wait for flag [OR]
    1.26    1.13    1.77    0.08   65%  28% Wait for flag [AND/CLR]
    1.24    1.13    1.77    0.09   56%  40% Wait for flag [OR/CLR]
    0.17    0.16    0.32    0.02   93%  93% Peek on flag

    0.60    0.48    2.09    0.13   96%  53% Create alarm
    1.67    1.45    3.87    0.25   84%  65% Initialize alarm
    0.87    0.81    1.93    0.10   81%  81% Disable alarm
    1.41    1.29    3.22    0.15   93%  59% Enable alarm
    0.97    0.81    1.93    0.07   75%  18% Delete alarm
    1.09    0.97    1.61    0.08   62%  34% Tick counter [1 alarm]
    4.89    4.83    5.32    0.08   71%  71% Tick counter [many alarms]
    1.87    1.77    3.06    0.12   96%  62% Tick & fire counter [1 alarm]
   29.72   29.64   30.44    0.10   96%  59% Tick & fire counters [>1 together]
    5.67    5.64    6.28    0.05   90%  90% Tick & fire counters [>1 separately]
    5.81    5.80    7.25    0.02   98%  98% Alarm latency [0 threads]
    5.87    5.80    7.89    0.13   91%  88% Alarm latency [2 threads]
    9.28    8.22   11.28    0.58   49%  30% Alarm latency [many threads]
    8.91    8.86   13.05    0.09   96%  96% Alarm -> thread resume latency

    1.66    1.45    3.54    0.00            Clock/interrupt latency

    2.40    1.93    5.15    0.00            Clock DSR latency

   33       0     312  (main stack:  1416)  Thread stack used (8016 total)
           All done, main stack : stack used  1416 size  3920
            All done :  Interrupt stack used   208 size  4096
            All done : Idlethread stack used   804 size  2048

Timing complete - 30040 ms total

PASS:<Basic timing OK>
EXIT:<done>
```

Other Issues

The AT91SAM9261-EK platform HAL does not affect the implementation of other parts of the eCos HAL specification. The SAM9 processor HAL, ARM9 variant HAL, and the ARM architectural HAL documentation should be consulted for further details.

Chapter 260. Atmel AT91SAM9263 Evaluation Kit Board Support

Name

eCos Support for the Atmel AT91SAM9263 Evaluation Kit — Overview

Description

This document covers the configuration and usage of eCos and RedBoot on the Atmel AT91SAM9263 Evaluation Kit. The AT91SAM9263 Evaluation Kit contains the AT91SAM9263 microprocessor, 64Mbytes of SDRAM, 4MBytes of PSRAM, 256Mbytes of NAND flash memory, an Atmel serial EEPROM, a Davicom DM9161A PHY, one SD/MMC/DataFlash socket and a SD/MMC socket, a DAC, external connections for two serial channels (one debug channel and one flow controlled), ethernet, USB host/device, and the various other peripherals supported by the AT91SAM9263. eCos support for the many devices and peripherals on the boards and the AT91SAM9263 is described below.

For typical eCos development, a RedBoot image is programmed onto a dataflash card in the SD/MMC/DataFlash socket, and the board will load this image from reset. RedBoot provides gdb stub functionality so it is then possible to download and debug stand-alone and eCos applications via the gdb debugger. This can happen over either a serial line or over ethernet.

This documentation is expected to be read in conjunction with the SAM9 processor HAL documentation and further device support and subsystems are described and documented there.

Supported Hardware

Booting is from a DataFlash card in the SD/MMC/DataFlash socket. In a typical setup, the first 32K bytes are reserved for the second-level bootstrap, AT91Bootstrap. The following 164736 bytes are reserved for the use of the ROM RedBoot image (The odd size aligns the end of the RedBoot area to a block boundary). The topmost block is used to manage the flash and the next block down holds RedBoot **fconfig** values. The remaining blocks can be used by application code.

There is a serial driver `CYGPKG_IO_SERIAL_ARM_AT91` which supports both the Debug Unit and USART serial devices. The debug serial port at J14 and flow controlled port at J18 (connected to USART channel 0) can be used by RedBoot for communication with the host. If any of these devices is needed by the application, either directly or via the serial driver, then it cannot also be used for RedBoot communication. Another communication channel such as ethernet should be used instead. The serial driver package is loaded automatically when configuring for the AT91SAM9263EK target.

There is an ethernet driver `CYGPKG_DEVS_ETH_ARM_AT91` for the on-chip ethernet device. The platform HAL package is responsible for configuring this generic driver to the hardware. This driver is also loaded automatically when configuring for the AT91SAM9263EK board.

There is a watchdog driver `CYGPKG_DEVICES_WATCHDOG_ARM_AT91WDTC`. This driver is also loaded automatically when configuring for the board.

There is a driver for the on-chip real-time timer controller (RTTC) at `CYGPKG_DEVICES_WALLCLOCK_ARM_AT91RTTC`. This driver is also loaded automatically when configuring for the target.

The SAM9 processor HAL contains a driver for the Two-Wire Interface (TWI) controller on the AT91SAM9263. This type of bus is also known as I²C®. Further documentation may be found in the SAM9 processor HAL documentation.

There is a driver for the MultiMedia Card Interface (MCI) at `CYGPKG_DEVS_MMCSD_ATMEL_SAM_MCI`. This driver is loaded automatically when configuring for the SAM9263-EK target and allows use of MMC and Secure Digital (SD) flash storage cards within eCos, exported as block devices. Further documentation may be found within that package. The driver can be configured to use either the SD/MMC/DataFlash socket at J19 or the SD/MMC socket at J10. By default it uses J10, leaving J9 for the bootstrap Dataflash.

The platform HAL provides definitions to allow access to devices on the SPI bus. The HAL provides information to the more general AT91 SPI driver (`CYGPKG_DEVS_SPI_ARM_AT91`) which in turn provides the underlying implementation for the SPI API layer in the `CYGPKG_IO_SPI` package. All these packages are automatically loaded when configuring for the board.

Furthermore, the platform HAL package contains support for SPI dataflash cards. The HAL support integrates with the CYGPKG_DEVS_FLASH_ATMEL_DATAFLASH package as well as the above SPI packages. That package is automatically loaded when configuring for the target. Dataflash media is then accessed as a Flash device, using the Flash I/O API within the CYGPKG_IO_FLASH package, if that package is loaded in the configuration.

It is also possible to configure the HAL to access MMC cards in SPI mode, instead of using the MCI interface.

The on-board NAND interface is supported. At the time of writing, this has been tested with the Micron MT29F2G08 part fitted to kit BOM revision 007 and later.

Note

Revision 007 of this board has a known issue with accessing the NAND flash device. Refer to section 1.5, "NAND Flash Access Issue", of the Rev.B User Guide (Atmel document no. 6341) for more details. During testing with a rev 007 board, eCosCentric found it necessary to tie the CS line as per that document in order to ensure reliable access to the device.

In general, devices (Caches, PIO, UARTs, EMAC) are initialized only as far as is necessary for eCos to run. Other devices (RTC, I²C, SPI, MCI etc.) are not touched unless the appropriate driver is loaded, although in some cases, the HAL boot sequence will set up the appropriate PIO configuration.

Tools

The AT91SAM9263-EK support is intended to work with GNU tools configured for an arm-eabi target. The original port was undertaken using arm-elf-gcc version 3.4.4, arm-elf-gdb version 6.3, and binutils version 2.15.

Name

Setup — Preparing the AT91SAM9263-EK board for eCos Development

Overview

In a typical development environment, the AT91SAM9263-EK board boots from a 4MiB DataFlash card and run the RedBoot ROM monitor from SDRAM. eCos applications are configured for RAM startup and then downloaded and run on the board via the debugger **arm-eabi-gdb**. Preparing the board therefore usually involves programming a suitable RedBoot image into flash memory.

The following RedBoot configurations are supported:

Configuration	Description	Use	File
ROM	RedBoot loaded from 4MiB Dataflash to SDRAM	`redboot_ROM.ecm`	`redboot_ROM.bin`

For serial communications, all versions run with 8 bits, no parity, and 1 stop bit at 115200 baud. RedBoot also supports ethernet communication and flash management.

Note that the use of the term ROM for the initial RedBoot configuration is an historical accident. RedBoot actually runs from SDRAM after being loaded there from Dataflash by the second-level bootstrap. The use of ROM for this configuration is intended to indicate that it initializes the microprocessor and board peripherals, as opposed to the RAM configuration which assumes that this has already been done.

Initial Installation

The on-chip boot program on the AT91SAM9263 is only capable of loading programs from DataFlash, SD card or NAND flash into on-chip SRAM and is therefore quite restrictive. Consequently RedBoot cannot be booted directly and a second-level bootstrap must be used. Such a second-level bootstrap is supplied by Atmel in the form of AT91Bootstrap. This is therefore programmed into the start of Dataflash and is then responsible for initializing the SDRAM and loading RedBoot from Dataflash and executing it.

Caution

There is a size limit on the size of applications which the AT91Bootstrap second level bootstrap will load. Images larger than 320Kbytes will require the AT91Bootstrap application to be rebuilt with a larger `IMG_SIZE` definition in `AT91Bootstrap/board/at91sam9263ek/dataflash/at91sam9263ek.h` within the sam9260ek HAL package in the eCos source repository (`packages/hal/arm/arm9/sam9260ek/current/`).

There are basically two ways to write the second-level bootstrap and RedBoot to the Dataflash. The first is to use the Atmel-supplied SAM-BA program that interacts with the on-chip boot program. The second is to use a JTAG debugger that understands the microcontroller and can write to the dataflash (for example the Ronetix PEEDI). Since the availability of the latter cannot be guaranteed, only the first method will be described here.

Programming RedBoot into DataFlash using SAM-BA

The following gives the steps needed to program the second-level bootstrap and RedBoot into the DataFlash card using SAM-BA. The user should refer to the SAM-BA documentation for full details of how to run the program.

1. Download the AT91 In-system Programmer software package from the Atmel website. SAM9 series CPU's require the 2.1.x series version of SAM-BA. Install it on a suitable PC running Windows.
2. From the root directory of your eCosPro installation, copy the file `dataflash_at91sam9263ek.bin` from the sub-directory `packages/hal/arm/arm9/sam9260ek/current/AT91Bootstrap/board/at91sam9263ek` and `redboot_ROM.bin` from the sub-directory `loaders/sam9263ek` to a suitable location on the Windows PC.

3. Connect a null-modem serial cable between the DEBUG serial port of the board and a serial port on a convenient host (which need not be the PC running SAM-BA). Run a terminal emulator (Hyperterm or minicom) at 115200 baud. Connect a USB cable between the PC and the AT91SAM9263-EK board. Windows may ask you to install a new driver, in which case follow the instructions.

4. Power up the board without the DataFlash card inserted. This will force the bootstrap to enter ISP mode.

5. Start SAM-BA. Select "\usb\ARM0" for the communication interface, and "AT91SAM9263-EK" for the board. If the USB option does not appear, check the cable and look in the Windows Device Manager for the new USB COM device. If all is well, click on "Connect".

6. In the SAM-BA main window, select the "SDRAM" tab, select the "Enable SDRAM 100MHz" script from the dropdown menu and click Execute. SAM-BA should emit the following messages:

   ```
   (AT91-ISP v1.10) 34 % SDRAM::initSDRAM_100
   -I- Configure PIOD as peripheral (D16/D31)
   -I- Init MATRIX to support EBI0 CS1 for SDRAM
   -I- Init SDRAM
   -I- 1. A minimum pause of 200us is provided to precede any signal toggle
   -I- 2. A Precharge All command is issued to the SDRAM
   -I- *pSDRAM = 0;
   -I- 3. Eight Auto-refresh are provided
   -I- *pSDRAM = 0;
   -I- *pSDRAM = 0;
   -I- *pSDRAM = 0;
   -I- *pSDRAM = 0;
   -I- *pSDRAM = 0;
   -I- *pSDRAM = 0;
   -I- *pSDRAM = 0;
   -I- *pSDRAM = 0;
   -I- 4. A mode register cycle is issued to program the SDRAM parameters
   -I- *(pSDRAM+0x20) = 0;
   -I- 5. Write refresh rate into SDRAMC refresh timer COUNT register
   -I- 6. A Normal Mode Command is provided, 3 clocks after tMRD is set
   -I- *pSDRAM = 0;
   -I- End of Init_SDRAM_100
   (AT91-ISP v1.10) 34 %
   ```

7. Now insert the DataFlash card into the socket at J9. In the SAM-BA main window, select the "DataFlash AT45DB/DBC" tab and in the "Scripts" dropdown menu select "Enable Dataflash (SPI0 CS0)", to program the Dataflash card. Click Execute and SAM-BA should emit the following in the message area:

   ```
   (AT91-ISP v1.13) 1 % DATAFLASH::Init 0
   -I- DATAFLASH::Init 0 (trace level : 4)
   -I- Loading applet isp-dataflash-at91sam9263.bin at address 0x20000000
   -I- Memory Size : 0x840000 bytes
   -I- Buffer address : 0x20002A40
   -I- Buffer size: 0x80E80 bytes
   -I- Applet initialization done
   ```

   The actual options and output of SAM-BA may vary according to the version you are using. The behaviour documented here is that of SAM-BA 2.9.

8. Select "Send BootFile" from the "Scripts" menu and "Execute" it. When the file open dialog appears, select the `dataflash_at91sam9263ek.bin` file and click "Open". The following output should be seen:

   ```
   (AT91-ISP v1.13) 1 % GENERIC::SendBootFileGUI
   GENERIC::SendFile dataflash_at91sam9263ek.bin at address 0x0
   -I- File size : 0x106E byte(s)
   -I-     Writing: 0x106E bytes at 0x0 (buffer addr : 0x20002A40)
   -I-     0x106E bytes written by applet
   ```

9. The second-level bootstrap has now been written to DataFlash, we must now write RedBoot.

10. In the "Send File Name" box type in the path name to the `redboot_ROM.bin` file, or use the Open Folder button and browse to it.

11. In the Address field set the value to 0x8400.

12. Click the "Send File" button. SAM-BA will put up a dialog box while it is writing the file to the DataFlash, and will output something similar to the following in the message area:

    ```
    (AT91-ISP v1.13) 1 % send_file {DataFlash AT45DB/DCB} "redboot_ROM.bin" 0x8400 0
    -I- Send File //bert/Shared/Releng/sam9263ek/redboot_ROM.bin at address 0x8400
    GENERIC::SendFile //bert/Shared/Releng/sam9263ek/redboot_ROM.bin at address 0x8400
    -I- File size : 0x24290 byte(s)
    -I-     Writing: 0x24290 bytes at 0x8400 (buffer addr : 0x20002A40)
    -I-     0x24290 bytes written by applet
    ```

13. Shut down SAM-BA and disconnect the USB cable. Press the reset button on the board and something similar to the following should be output on the DEBUG serial line.

    ```
    RomBOOT
    >Start AT91Bootstrap...
    +**Warning** FLASH configuration checksum error or invalid key
    Use 'fconfig -i' to [re]initialize database
    No space to add 'net_device'
    AT91_ETH: Waiting for PHY to reset.
    AT91_ETH: Waiting for link to come up..
    No network interfaces found

    RedBoot(tm) bootstrap and debug environment [ROM]
    eCosCentric certified release, version v3_0_3 - built 12:56:09, Sep 24 2009

    Copyright (C) 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009
    Free Software Foundation, Inc.
    Copyright (C) 2003, 2004, 2005, 2006, 2007, 2008 eCosCentric Limited
    RedBoot is free software, covered by the eCos license, derived from the
    GNU General Public License. You are welcome to change it and/or distribute
    copies of it under certain conditions. Under the license terms, RedBoot's
    source code and full license terms must have been made available to you.
    Redboot comes with ABSOLUTELY NO WARRANTY.

    Platform: AT91SAM9263-EK (ARM9)
    RAM: 0x20000000-0x24000000 [0x20035ba8-0x23fff5b0 available]
    FLASH: 0x40000000-0x4083ffff, 8192 x 0x420 blocks
    RedBoot>
    ```

RedBoot Flash configuration

The following steps describe how to initialize RedBoot's Flash configuration.

1. Use the following command to initialize RedBoot's Flash Information System (FIS):

    ```
    RedBoot> fis init
    About to initialize [format] FLASH image system - continue (y/n)? y
    *** Initialize FLASH Image System
    ... Erase from 0x4083fbe0-0x4083ffff: .
    ... Program from 0x23fffbe0-0x24000000 to 0x4083fbe0: .
    RedBoot>
    ```

2. Now configure RedBoot's Flash configuration with the command:

    ```
    RedBoot> fconfig -i
    ```

    Remember to substitute the appropriate MAC address for this board at the appropriate step. If a BOOTP/DHCP server is not available, then IP configuration may be set manually. The default server IP address can be set to a PC that will act as a TFTP host for future RedBoot load operations, or may be left unset. The following gives an example configuration:

```
RedBoot> fconfig -i
Initialize non-volatile configuration - continue (y/n)? y
Run script at boot: false
Use BOOTP for network configuration: false
Gateway IP address: 192.168.7.11
Local IP address: 192.168.7.222
Local IP address mask: 255.255.255.0
Default server IP address: 192.168.7.9
Console baud rate: 115200
DNS domain name: ecoscentric.com
DNS server IP address: 192.168.7.11
Network hardware address [MAC]: 0x00:0x23:0x31:0x37:0x00:0x4e
Update RedBoot non-volatile configuration - continue (y/n)? y
... Erase from 0x4083f7c0-0x4083fbdf: .
... Program from 0x23fff7c0-0x23fffbe0 to 0x4083f7c0: .
RedBoot>
```

The RedBoot installation is now complete. This can be tested by powering off the board, and then powering on the board again. Output similar to the following should be seen on the DEBUG serial port. Verify the IP settings are as expected.

```
RomBOOT
>Start AT91Bootstrap...
Ethernet eth0: MAC address 0e:00:00:ea:18:f0
IP: 192.168.7.222/255.255.255.0, Gateway: 192.168.7.11
Default server: 192.168.7.11
DNS server IP: 192.168.7.11, DNS domain name: <null>

RedBoot(tm) bootstrap and debug environment [ROM]
eCosCentric certified release, version v3_0_3 - built 12:56:09, Sep 24 2009

Copyright (C) 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009
Free Software Foundation, Inc.
Copyright (C) 2003, 2004, 2005, 2006, 2007, 2008 eCosCentric Limited
RedBoot is free software, covered by the eCos license, derived from the
GNU General Public License. You are welcome to change it and/or distribute
copies of it under certain conditions. Under the license terms, RedBoot's
source code and full license terms must have been made available to you.
Redboot comes with ABSOLUTELY NO WARRANTY.

Platform: AT91SAM9263-EK (ARM9)
RAM: 0x20000000-0x24000000 [0x20035ba8-0x23fff5b0 available]
FLASH: 0x40000000-0x4083ffff, 8192 x 0x420 blocks
RedBoot>
```

If it proves necessary to re-install RedBoot, this may be achieved by repeating the above process. Alternatively, a new image may be downloaded and programmed into flash more directly using RedBoot's own commands. See the RedBoot documentation for details.

Rebuilding RedBoot

Should it prove necessary to rebuild a RedBoot binary, this is done most conveniently at the command line. The steps needed to rebuild the ROM version of RedBoot for the AT91SAM9263-EK are:

```
$ mkdir redboot_at91sam9263ek_rom
$ cd redboot_at91sam9263ek_rom
$ ecosconfig new at91sam9263ek redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/arm/arm9/sam9263ek/current/misc/redboot_ROM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

At the end of the build the `install/bin` subdirectory should contain the file `redboot.bin`.

Rebuilding AT91Bootstrap

The sources of AT91Bootstrap are found in the `AT91Bootstrap` directory of the sam9260ek package. This is a copy of the software as supplied by Atmel with some slight modifications to permit it to be built with the same tools as eCos.

To rebuild the second-level bootstrap for the AT91SAM9263EK execute the following commands:

```
$ cd $ECOS_REPOSITORY/hal/arm/arm9/sam9260ek/current/AT91Bootstrap/board/at91sam9263ek/dataflash
$ make
```

This should result in the creation of a number of files, including `dataflash_at91sam9263ek.bin` which can be copied out.

Name

Configuration — Platform-specific Configuration Options

Overview

The AT91SAM9263-EK platform HAL package is loaded automatically when eCos is configured for the `sam9263ek` target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The platform HAL package supports two separate startup types:

RAM
: This is the startup type which is normally used during application development. The board has RedBoot programmed into flash and boots into that initially. arm-eabi-gdb is then used to load a RAM startup application into memory and debug it. It is assumed that the hardware has already been initialized by RedBoot. By default the application will use the eCos virtual vectors mechanism to obtain certain services from RedBoot, including diagnostic output.

ROM
: This startup type can be used for finished applications which will be programmed into DataFlash. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization.

JTAG
: This is the startup type which can be used during application development via a JTAG device such as the PEEDI. arm-eabi-gdb is used to load a JTAG startup application into memory and debug it. Hardware setup is divided between the initialization section of the PEEDI configuration file and software in the loaded application.

RedBoot and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building RedBoot.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then the serial port will be claimed for HAL diagnostics.

Flash Driver

The board has an SD/MMC/DataFlash socket into which a dataflash card may be inserted. The `CYGPKG_DEVS_FLASH_ATMEL_DATAFLASH` package contains all the code necessary to support this part and the platform HAL package contains definitions that customize the driver to the AT91SAM9263-EK board. This driver is not active until the generic Flash support package, `CYGPKG_IO_FLASH`, is included in the configuration.

Ethernet Driver

The AT91SAM9263-EK board uses the AT91SAM9263's internal EMAC ethernet device attached to an external Davicom DM9161A PHY. The `CYGPKG_DEVS_ETH_ARM_AT91` package contains all the code necessary to support this device and the platform HAL package contains definitions that customize the driver to the AT91SAM9263-EK board. This driver is not active until the generic Ethernet support package, `CYGPKG_IO_ETH_DRIVERS`, is included in the configuration.

RTC Driver

The AT91SAM9263-EK board uses the AT91SAM9263's internal RTTC support. The CYGPKG_DEVICES_WALLCLOCK_ARM_AT91RTTC package contains all the code necessary to support this device. This driver is not active until the generic wallclock device support package, CYGPKG_IO_WALLCLOCK, is included in the configuration.

Watchdog Driver

The AT91SAM9263-EK board uses the AT91SAM9263's internal watchdog support. The CYGPKG_DEVICES_WATCHDOG_ARM_AT91WDTC package contains all the code necessary to support this device. Within that package the CYGNUM_DEVS_WATCHDOG_ARM_AT91WDTC_DESIRED_TIMEOUT_MS configuration option controls the watchdog timeout, and by default will force a reset of the board upon timeout. This driver is not active until the generic watchdog device support package, CYGPKG_IO_WATCHDOG, is included in the configuration.

Warning

The ATSAM926x processor will boot with watchdog support enabled, and the watchdog configuration is write-once. That is, if it is disabled, it cannot be re-enabled. Due to its nature, RedBoot disables the watchdog when it starts so any eCos applications with watchdog support enabled that are run by RedBoot will not function correctly.

USART Serial Driver

The AT91SAM9263-EK board uses the AT91SAM9263's internal USART serial support as described in the SAM9 processor HAL documentation. Two serial ports are available: the serial debug port which is mapped to virtual vector channel 0 in the HAL diagnostic driver or "/dev/dbg" in the interrupt-driven driver; and USART 0 which is mapped to virtual vector channel 1 and "/dev/ser0". The debug port is two wires only. but USART 0 supports RTS/CTS.

MCI Driver

As the SAM MCI driver is included in the hardware-specific configuration for this target, nothing is required to load it. Similarly the MMC/SD bus driver layer (CYGPKG_DEVS_DISK_MMC) is automatically included as part of the hardware-specific configuration for this target. All that is required to enable the support is to include the generic disk I/O infrastructure package (CYGPKG_IO_DISK), along with the intended filesystem, typically, the FAT filesystem (CYGPKG_FS_FAT) and any of its package dependencies (including CYGPKG_LIBC_STRING and CYGPKG_LINUX_COMPAT for FAT).

If the generic disk I/O infrastructure is needed for some other reason, and you do not wish to also include the MCI driver, then the configuration option within this platform HAL CYGPKG_HAL_ARM_ARM9_SAM9263EK_MMCSD can be used to forcibly disable it.

Various options can be used to control specifics of the SAM MCI driver. Consult the SAM MCI driver documentation for information on its configuration. The option CYGHWR_DEVS_MMCSD_ATMEL_SAM_MCI_DEVICE controls which of the two MCI interfaces will be used by the driver.

The MMC/SD socket does not permit detection of the write-protect (or "lock") switch present on SD cards. "Locked" cards will therefore not be detected which means that despite the switch position, it is still possible to write to them since the lock switch does not physically enforce write protection.

Compiler Flags

The platform HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos. There are just three flags specific to this port:

`-mcpu=arm9`
: The arm-eabi-gcc compiler supports many variants of the ARM architecture. A `-m` option should be used to select the specific variant in use, and with current tools `-mcpu=arm9` is the correct option for the ARM926EJ CPU in the AT91SAM9263.

`-mthumb`
: The arm-eabi-gcc compiler will compile C and C++ files into the Thumb instruction set when this option is used. The best way to build eCos in Thumb mode is to enable the configuration option `CYGHWR_THUMB`.

`-mthumb-interwork`
: This option allows programs to be created that mix ARM and Thumb instruction sets. Without this option, some memory can be saved. This option should be used if -mthumb is used. The best way to build eCos with Thumb interworking is to enable the configuration option `CYGBLD_ARM_ENABLE_THUMB_INTERWORK`.

Onboard NAND

The HAL port includes a low-level driver to access the on-board Micron MT29F2G08 NAND flash memory chip found on revision B of the board. To enable the driver, add the CYGPKG_IO_NAND package to your eCos configuration.

`CYGHWR_HAL_SAM9263EK_NAND_USE_STATUS_LINE`
: If set, this option configures the driver to wait for NAND operations to complete by waiting for the chip to deassert its Busy line. This is the default behaviour and is recommended, but may be disabled if you need to use the line for some other purpose or on derived hardware. (If disabled, the driver falls back to a combination of delay loops and polling the chip's Read Status function.)

`CYGNUM_HAL_SAM9263EK_NAND_POLL_INTERVAL`
: The number of microseconds delay in the polling loops which wait for NAND operations to complete.

Partitioning the NAND chip

The NAND chip must be partitioned before it can become available to applications.

A CDL script which allows the chip to be manually partitioned is provided (see `CYGSEM_DEVS_NAND_SAM9263EK_PARTITION_MANUAL_CONFIG`); if you choose to use this, the relevant data structures will automatically be set up for you when the device is initialised. By default, the manual config CDL script sets up a single partition (number 0) encompassing the entire device.

It is possible to configure the partitions in some other way, should it be appropriate for your setup, for example to read a Linux-style partition table from the chip. To do so you will have to add appropriate code to `sam9263ek_nand.c`.

Name

JTAG support — Usage

Use of JTAG for debugging

JTAG can be used to single-step and debug loaded applications, including RedBoot. Only JTAG configuration applications should be debugged using JTAG, RAM applications assume the presence of RedBoot.

Ronetix PEEDI notes

On the Ronetix PEEDI, the `peedi.at91sam9263ek.cfg` file should be used to setup and configure the hardware to an appropriate state to load programs. This includes setting up the SDRAM controller.

The `peedi.at91sam9263ek.cfg` file also contains an option to define whether hardware or software breakpoints are used by default, using the `CORE0_BREAKMODE` directive in the `[PLATFORM_ARM]` section. Edit this file if you wish to use hardware break points, and remember to restart the PEEDI to make the changes take effect.

On the PEEDI, debugging can be performed either via the telnet interface or using **arm-eabi-gdb** and the GDB interface. In the case of the latter, **arm-eabi-gdb** needs to connect to TCP port 2000 on the PEEDI's IP address. For example:

```
(gdb) target remote 111.222.333.444:2000
```

By default when the PEEDI is powered up, the target will always run the initialization section of the `peedi.at91sam9263ek.cfg` file (which configures the SDRAM among other things), and halts the target. This behavior is repeated with the **reset** command.

If the board is reset (either with the '**reset**', or by pressing the reset button) and the '**go**' command is then given, then the board will boot as normal. If a second-level bootstrap and ROM RedBoot is resident in DataFlash, it will be run.

An issue occurs when the AT91 Ethernet driver is included in your configuration. In order to work around a board hardware design issue, the CPU generates an external reset in order to reset the Ethernet PHY. However this can be interpreted by the PEEDI as an indication that the CPU itself has reset, and if the PEEDI configuration file option `CORE0_STARTUP_MODE` is set to `RESET` then the CPU will be halted at this point. To avoid this issue, the `CORE0_STARTUP_MODE` can be set to `RUN`.

Consult the PEEDI documentation for information on other features.

Running JTAG applications

Applications configured for JTAG startup can be run directly under a JTAG debugger. Once loaded and running via JTAG, HAL diagnostic output will appear by default on the serial debug port. USARTs 0 or 1 can be chosen instead by setting the `CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL` configuration option in the platform HAL to channel 1 or 2.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the AT91SAM9263-EK hardware, and should be read in conjunction with that specification. The AT91SAM9263-EK platform HAL package complements the ARM architectural HAL, the ARM9 variant HAL and the SAM9 processor HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset, the HAL will initialize or reinitialize most of the on-chip peripherals. There is an exception for RAM startup applications which depend on a ROM monitor for certain services.

For ROM startup, the HAL will perform additional initialization. This is all done in the `PLATFORM_SETUP1` macro in the assembler header file `hal_platform_setup.h`.

Linker Scripts and Memory Maps

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

SDRAM

This is located at address 0x20000000 of the physical memory space. The HAL configures the MMU to retain the SDRAM at virtual address 0x20000000, but in order to assign hardware exception vectors vectors at address 0x00000000, the HAL also uses the MMU to create a clone of this memory at virtual address 0x00000000. The same memory is also accessible uncached and unbuffered at virtual location 0x30000000 for use by devices. The first 32 bytes are used for hardware exception vectors. The next 32 bytes are used for the VSR table and the next 256 bytes are normally used for the eCos virtual vectors, allowing RAM-based applications to use services provided by the ROM monitor. Memory is required for the MMU tables, and must be aligned on a 16Kbyte boundary. These therefore occupy memory from 0x4000 to 0x8000. For ROM startup, all remaining SDRAM is available. For RAM startup, available RAM starts at virtual location 0x20040000, with the bottom 256kB reserved for use by RedBoot.

On-chip SRAM

This is located at address 0x00200000 of the physical memory space. However the HAL uses the MMU to relocate this to virtual address 0x70000000. The same memory is also accessible uncached and unbuffered at virtual location 0x70100000 for use by devices. At present this memory is unused by eCos and is available for application use.

On-chip ROM

This is located at address 0x00100000 of the physical memory space. However the HAL uses the MMU to relocate this to virtual address 0x71000000. The same memory is also accessible uncached and unbuffered at virtual location 0x71800000.

USB host port

The USB host port (UHP) registers are located at address 0x00300000 of the physical memory space. However the HAL uses the MMU to relocate this to virtual address 0x72800000. Memory accessed at this address is uncached and unbuffered. There is no cached variant.

SPI dataflash

SPI Dataflash media can only be accessed with the Flash API. For the purposes of this API a placeholder address range has been allocated as if the Flash is present at this address. The base of this address range is 0x40000000 for the dataflash slot, the extent will clearly depend on the Dataflash capacity. This reserved range is not real memory and any attempt to access it directly by the processor other than via the Flash API will result in a memory address exception.

On-chip Peripheral Registers

These are located at address 0xFF000000 in the physical memory space. When the MMU is enabled, it sets up a direct, uncached, unbuffered mapping so that these registers remain accessible at their physical locations.

Off-chip Peripherals

eCos uses the SDRAM, ethernet PHY, MCI, and SPI dataflash facilities on the AT91SAM9263-EK board. eCos does not currently make any use of any other off-chip peripherals present on this board.

Advanced Interrupt Controller

This port has been designed to exploit benefits of the Advanced Interrupt Controller of the AT91SAM9263, using the facilities of the SAM9 processor HAL. Consult the documentation in that package for details.

SPI Dataflash

eCos supports SPI access to Dataflash on the AT91SAM9263. An external card slot are provided on the board which is typically used to contain RedBoot and flash configuration data.

Accesses to Dataflash are performed via the Flash API, using 0x40000000 as the nominal address of the device, although it does not truly exist in the processor address space. For the external card slot, on driver initialisation, eCos and RedBoot can detect the presence of a card in the socket. In particular, on reset RedBoot will indicate the presence of Flash at the 0x40000000 address range in its startup banner if it has been successfully detected. Hot swapping is not possible.

Since Dataflash is not directly addressable, access from RedBoot is only possible using **fis** command operations.

Real-time characterization

The tm_basic kernel test gives statistics gathered about the real-time characterization and performance of the kernel. The sample output is shown here for information. The test was built in ARM mode, which provided better performance than Thumb mode.

Example 260.1. sam9263ek Real-time characterization

```
            Startup, main stack : stack used   420 size  3920
             Startup :  Interrupt stack used   528 size  4096
             Startup : Idlethread stack used    96 size  2048

eCos Kernel Timings
Notes: all times are in microseconds (.000001) unless otherwise stated

Reading the hardware clock takes 1 'ticks' overhead
... this value will be factored out of all other measurements
Clock interrupt took    5.95 microseconds (37 raw clock ticks)

Testing parameters:
   Clock samples:            32
   Threads:                  64
   Thread switches:         128
   Mutexes:                  32
   Mailboxes:                32
   Semaphores:               32
   Scheduler operations:    128
   Counters:                 32
   Flags:                    32
   Alarms:                   32


                                 Confidence
     Ave     Min     Max     Var  Ave  Min  Function
  ======  ======  ======  ====== ========== ========
    4.89    3.20    7.84    0.85   48%  25% Create thread
    0.73     0.64     1.92     0.10    98%  54% Yield thread [all suspended]
    0.93     0.80     2.08     0.11    46%  42% Suspend [suspended] thread
    0.91     0.80     1.92     0.10    45%  46% Resume thread
    1.32     1.12     3.36     0.11    67%  14% Set priority
    0.27     0.16     0.64     0.10    42%  43% Get priority
    2.97     2.72     7.20     0.19    71%  65% Kill [suspended] thread
    0.71     0.64     0.96     0.08    56%  56% Yield [no other] thread
    1.51     1.28     3.20     0.12    87%   7% Resume [suspended low prio] thread
    0.89     0.80     1.44     0.09    95%  53% Resume [runnable low prio] thread
    1.27     1.12     2.24     0.08    60%  28% Suspend [runnable] thread
    0.72     0.64     1.44     0.09    98%  56% Yield [only low prio] thread
    0.93     0.80     1.60     0.09    57%  34% Suspend [runnable->not runnable]
    2.83     2.56     6.08     0.15    90%   4% Kill [runnable] thread
    2.27     2.08     5.44     0.14    70%  28% Destroy [dead] thread
    3.92     3.52     6.72     0.16    79%  15% Destroy [runnable] thread
    5.79     5.44     9.44     0.24    81%  48% Resume [high priority] thread
    1.88     1.76     3.68     0.09    61%  36% Thread switch

    0.08     0.00     0.32     0.08    50%  49% Scheduler lock
    0.56     0.48     1.12     0.08    99%  50% Scheduler unlock [0 threads]
    0.57     0.48     1.12     0.08    50%  49% Scheduler unlock [1 suspended]
    0.56     0.48     1.12     0.08    98%  55% Scheduler unlock [many suspended]
    0.57     0.48     1.28     0.08    99%  50% Scheduler unlock [many low prio]

    0.19     0.00     1.12     0.07    87%   6% Init mutex
    0.96     0.80     2.72     0.12    59%  37% Lock [unlocked] mutex
    1.08     0.80     2.88     0.15    87%   9% Unlock [locked] mutex
    0.86     0.64     1.92     0.12    84%  12% Trylock [unlocked] mutex
    0.76     0.64     1.28     0.08    62%  34% Trylock [locked] mutex
    0.08     0.00     0.32     0.08    96%  50% Destroy mutex
    4.95     4.80     7.68     0.23    90%  90% Unlock/Lock mutex

    0.28     0.16     1.28     0.11    53%  43% Create mbox
    0.19     0.00     0.48     0.07    71%   6% Peek [empty] mbox
    1.12     0.96     2.40     0.09    65%  28% Put [first] mbox
    0.19     0.00     0.48     0.05    81%   3% Peek [1 msg] mbox
    1.10     0.96     1.44     0.07    68%  25% Put [second] mbox
    0.21     0.00     0.64     0.08    68%   3% Peek [2 msgs] mbox
    1.17     0.96     2.56     0.12    81%  15% Get [first] mbox
    1.16     0.96     1.44     0.07    68%   6% Get [second] mbox
    0.97     0.80     1.92     0.08    71%  18% Tryput [first] mbox
    1.00     0.80     1.92     0.09    78%   6% Peek item [non-empty] mbox
    1.08     0.96     2.40     0.12    96%  50% Tryget [non-empty] mbox
    0.88     0.80     2.08     0.11    96%  71% Peek item [empty] mbox
    0.90     0.80     1.60     0.10    96%  53% Tryget [empty] mbox
    0.21     0.16     0.48     0.07    75%  75% Waiting to get mbox
    0.21     0.16     0.48     0.07    78%  78% Waiting to put mbox
    0.45     0.32     1.44     0.10    59%  37% Delete mbox
    3.50     3.36     6.56     0.20    96%  96% Put/Get mbox

    0.19     0.16     1.12     0.06    96%  96% Init semaphore
    0.73     0.64     1.76     0.12    93%  65% Post [0] semaphore
    0.81     0.64     1.76     0.08    71%  18% Wait [1] semaphore
    0.72     0.64     1.76     0.11    96%  71% Trywait [0] semaphore
    0.70     0.64     1.28     0.09    71%  71% Trywait [1] semaphore
    0.17     0.00     0.64     0.06    75%  12% Peek semaphore
    0.11     0.00     0.96     0.11    46%  46% Destroy semaphore
    3.15     3.04     5.60     0.18    93%  93% Post/Wait semaphore

    0.30     0.16     1.60     0.13    40%  46% Create counter
    0.16     0.00     0.80     0.10    46%  31% Get counter value
    0.11     0.00     0.80     0.09    56%  40% Set counter value
    0.91     0.80     1.76     0.10    50%  46% Tick counter
    0.15     0.00     0.96     0.11    46%  34% Delete counter

    0.20     0.16     1.28     0.07    93%  93% Init flag
    0.82     0.64     2.08     0.11    62%  25% Destroy flag
    0.68    0.48    1.44    0.09   68%   9% Mask bits in flag
    0.81    0.64    1.92    0.10   59%  28% Set bits in flag [no waiters]
    1.15    0.96    2.56    0.09   87%   9% Wait for flag [AND]
    1.15    1.12    1.92    0.05   96%  96% Wait for flag [OR]
    1.14    1.12    1.60    0.03   96%  96% Wait for flag [AND/CLR]
    1.14    0.96    1.76    0.06   81%   9% Wait for flag [OR/CLR]
    0.07    0.00    0.16    0.08   53%  53% Peek on flag

    0.50    0.32    1.92    0.11   65%  25% Create alarm
    1.60    1.44    4.32    0.21   96%  84% Initialize alarm
    0.85    0.64    1.92    0.11   84%   9% Disable alarm
    1.47    1.28    3.36    0.17   56%  81% Enable alarm
    1.05    0.80    2.56    0.14   87%   6% Delete alarm
    0.98    0.80    1.92    0.07   84%   9% Tick counter [1 alarm]
    4.74    4.64    5.28    0.09   96%  50% Tick counter [many alarms]
    1.74    1.60    3.04    0.11   59%  37% Tick & fire counter [1 alarm]
   29.67   29.61   30.09    0.08   71%  71% Tick & fire counters [>1 together]
    5.52    5.44    5.92    0.09   96%  59% Tick & fire counters [>1 separately]
    5.13    5.12    6.08    0.01   99%  99% Alarm latency [0 threads]
    5.64    5.12    6.24    0.36   46%  53% Alarm latency [2 threads]
    8.07    6.88    9.76    0.68   43%  37% Alarm latency [many threads]
    8.20    8.16   12.49    0.07   98%  98% Alarm -> thread resume latency

    1.45    1.28    4.00    0.00            Clock/interrupt latency

    2.13    1.60    5.28    0.00            Clock DSR latency

    6      0     312  (main stack:  1388)  Thread stack used (1360 total)
           All done, main stack : stack used  1388 size  3920
            All done :  Interrupt stack used   204 size  4096
            All done : Idlethread stack used   796 size  2048

Timing complete - 29980 ms total

PASS:<Basic timing OK>
EXIT:<done>
```

Other Issues

The AT91SAM9263-EK platform HAL does not affect the implementation of other parts of the eCos HAL specification. The SAM9 processor HAL, ARM9 variant HAL, and the ARM architectural HAL documentation should be consulted for further details.

Chapter 261. Atmel AT91SAM9G20 Evaluation Kit Board Support

Name

eCos Support for the Atmel AT91SAM9G20 Evaluation Kit — Overview

Description

This document covers the configuration and usage of eCos and RedBoot on the Atmel AT91SAM9G20 Evaluation Kit. The AT91SAM9G20 Evaluation Kit contains the AT91SAM9G20 microprocessor, 64Mbytes of SDRAM, 256Mbytes of NAND flash memory, an Atmel Dataflash, an Atmel serial EEPROM, a Davicom DM9161A PHY, a SD/MMC/DataFlash socket, a DAC, external connections for three serial channels (one debug, one full modem, one flow controlled), ethernet, USB host/device, and the various other peripherals supported by the AT91SAM9G20. eCos support for the many devices and peripherals on the boards and the AT91SAM9G20 is described below.

For typical eCos development, a RedBoot image is programmed into the dataflash memory, and the board will load this image from reset. RedBoot provides gdb stub functionality so it is then possible to download and debug stand-alone and eCos applications via the gdb debugger. This can happen over either a serial line or over ethernet.

This documentation is expected to be read in conjunction with the SAM9 processor HAL documentation and further device support and subsystems are described and documented there.

Supported Hardware

The Dataflash consists of 8192 blocks of 1056 bytes each. In a typical setup, the first 33792 bytes are reserved for the second-level bootstrap, AT91Bootstrap. The following 164736 bytes are reserved for the use of the ROM RedBoot image (The odd size aligns the end of the RedBoot area to a 1056 block boundary). The topmost block is used to manage the flash and the next block down holds RedBoot **fconfig** values. The remaining blocks can be used by application code.

There is a serial driver `CYGPKG_IO_SERIAL_ARM_AT91` which supports both the Debug Unit and USART serial devices. The debug serial port at J17 and DTE port at J20 (connected to USART channel 0) and flow controlled port at J18 (connected to USART channel 1) can be used by RedBoot for communication with the host. If any of these devices is needed by the application, either directly or via the serial driver, then it cannot also be used for RedBoot communication. Another communication channel such as ethernet should be used instead. The serial driver package is loaded automatically when configuring for the AT91SAM9G20-EK target.

There is an ethernet driver `CYGPKG_DEVS_ETH_ARM_AT91` for the on-chip ethernet device. The platform HAL package is responsible for configuring this generic driver to the hardware. This driver is also loaded automatically when configuring for the AT91SAM9G20-EK board.

There is a watchdog driver `CYGPKG_DEVICES_WATCHDOG_ARM_AT91WDTC`. This driver is also loaded automatically when configuring for the board.

There is a driver for the on-chip real-time timer controller (RTTC) at `CYGPKG_DEVICES_WALLCLOCK_ARM_AT91RTTC`. This driver is also loaded automatically when configuring for the target.

The SAM9 processor HAL contains a driver for the Two-Wire Interface (TWI) controller on the AT91SAM9G20. This type of bus is also known as I²C®. Further documentation may be found in the SAM9 processor HAL documentation.

There is a driver for the MultiMedia Card Interface (MCI) at `CYGPKG_DEVS_MMCSD_ATMEL_SAM_MCI`. This driver is loaded automatically when configuring for the AT91SAM9G20-EK target and allows use of MMC and Secure Digital (SD) flash storage cards within eCos, exported as block devices. Further documentation may be found within that package.

The platform HAL provides definitions to allow access to devices on the SPI bus. The HAL provides information to the more general AT91 SPI driver (`CYGPKG_DEVS_SPI_ARM_AT91`) which in turn provides the underlying implementation for the SPI API layer in the `CYGPKG_IO_SPI` package. All these packages are automatically loaded when configuring for the board.

Furthermore, the platform HAL package contains support for SPI dataflash cards. The HAL support integrates with the CYGPKG_DEVS_FLASH_ATMEL_DATAFLASH package as well as the above SPI packages. That package is automatically loaded when configuring for the target. Dataflash media is then accessed as a Flash device, using the Flash I/O API within the CYGPKG_IO_FLASH package, if that package is loaded in the configuration.

It is also possible to configure the HAL to access MMC cards in SPI mode, instead of using the MCI interface.

In general, devices (Caches, PIO, UARTs, EMAC) are initialized only as far as is necessary for eCos to run. Other devices (RTC, I²C, SPI, MCI etc.) are not touched unless the appropriate driver is loaded, although in some cases, the HAL boot sequence will set up the appropriate PIO configuration.

Tools

The AT91SAM9G20-EK support is intended to work with GNU tools configured for an arm-eabi target. The original port was undertaken using arm-elf-gcc version 3.4.4, arm-elf-gdb version 6.3, and binutils version 2.15.

Name

Setup — Preparing the AT91SAM9G20-EK board for eCos Development

Overview

In a typical development environment, the AT91SAM9G20-EK board boots from the DataFlash and runs the RedBoot ROM monitor from SDRAM. eCos applications are configured for RAM startup and then downloaded and run on the board via the debugger **arm-eabi-gdb**. Preparing the board therefore usually involves programming a suitable RedBoot image into flash memory.

The following RedBoot configurations are supported:

Configuration	Description	Use	File
ROM	RedBoot loaded from Dataflash to SDRAM	`redboot_ROM.ecm`	`redboot_ROM.bin`

For serial communications, all versions run with 8 bits, no parity, and 1 stop bit at 115200 baud. RedBoot also supports ethernet communication and flash management.

Note that the use of the term ROM for the initial RedBoot configuration is a historical accident. RedBoot actually runs from SDRAM after being loaded there from Dataflash by the second-level bootstrap. The use of ROM for this configuration is intended to indicate that it initializes the microprocessor and board peripherals, as opposed to the RAM configuration which assumes that this has already been done.

Initial Installation

The on-chip boot program on the AT91SAM9G20 is only capable of loading programs from Dataflash or NAND flash into 4Kbytes of on-chip SRAM and is therefore quite restrictive. Consequently RedBoot cannot be booted directly and a second-level bootstrap must be used. Such a second-level bootstrap is supplied by Atmel in the form of AT91Bootstrap. This is therefore programmed into the start of Dataflash and is then responsible for initializing the SDRAM and loading RedBoot from Dataflash and executing it.

Caution

There is a size limit on the size of applications which the AT91Bootstrap second level bootstrap will load. Images larger than 320Kbytes will require the AT91Bootstrap application to be rebuilt with a larger `IMG_SIZE` definition in `AT91Bootstrap/board/at91sam9g20ek/dataflash/at91sam9g20ek.h` within the sam9g20ek HAL package in the eCos source repository (`packages/hal/arm/arm9/sam9260ek/current/`).

There are basically two ways to write the second-level bootstrap and RedBoot to the Dataflash. The first is to use the Atmel-supplied SAM-BA program that interacts with the on-chip boot program. The second is to use a JTAG debugger that understands the microcontroller and can write to the dataflash (for example the Ronetix PEEDI). Since the availability of the latter cannot be guaranteed, only the first method will be described here.

Programming RedBoot into DataFlash using SAM-BA

The following gives the steps needed to program the second-level bootstrap and RedBoot into the DataFlash using SAM-BA. The user should refer to the SAM-BA documentation for full details of how to run the program.

1. Download the corresponding AT91 In-system Programmer software package from the Atmel website according to your host operating system and install it on your PC (SAM9 series CPU's require the 2.1.x series version of SAM-BA).
2. From the root directory of your eCosPro installation, copy the file `dataflash_at91sam9g20ek.bin` from the sub-directory `packages/hal/arm/arm9/sam9260ek/current/AT91Bootstrap/board/at91sam9g20ek/`

`dataflash` and `redboot_ROM.bin` from the sub-directory `loaders/sam9g20ek` to a suitable location on the Windows PC.

3. Connect a null-modem serial cable between the DEBUG serial port of the board and a serial port on a convenient host (which need not be the PC running SAM-BA). Run a terminal emulator (Hyperterm or minicom) at 115200 baud. Connect a USB cable between the PC and the AT91SAM9G20-EK board. Windows may ask you to install a new driver, in which case follow the instructions.

4. Remove Jumper J33 from the board and press the reset button. You should see the following output on the serial line:

   ```
   RomBoot
   >
   ```

   Now reinsert the jumper.

5. Start SAM-BA. Select the appropriate COM port for the communication interface (on Windows hosts this will be of the form "COMx" and on Linux hosts this will be of the form "/dev/ttyUSBx"), and "AT91SAM9G20-EK" for the board. Click on "Connect".

6. In the SAM-BA main window, select the "DataFlash AT45DB/DBC" tab and in the "Scripts" dropdown menu select "Enable Dataflash (SPIO CS1)", to program the on-board Dataflash device. Click Execute and SAM-BA should emit the following in the message area:

   ```
   sam-ba_cdc_linux) 1 % DATAFLASH::Init 1
   -I- DATAFLASH::Init 1 (trace level : 4)
   -I- Loading applet isp-dataflash-at91sam9g20.bin at address 0x200000
   -I- Memory Size : 0x840000 bytes
   -I- Buffer address : 0x202D38
   -I- Buffer size: 0xC60 bytes
   -I- Applet initialization done
   ```

   The actual options and output of SAM-BA may vary according to the version you are using. The behaviour documented here is that of SAM-BA CDC 2.10 on Linux.

7. Now select "Send BootFile" from the "Scripts" menu and "Execute" it. When the file open dialog appears, select the `dataflash_at91sam9g20ek.bin` file and click "Open". The following output should be seen:

   ```
   (sam-ba_cdc_linux) 1 % GENERIC::SendBootFileGUI
   GENERIC::SendFile /tmp/dataflash_at91sam9g20ek.bin at address 0x0
   -I- File size : 0xE7C byte(s)
   -I-     Writing: 0xC60 bytes at 0x0 (buffer addr : 0x202D38)
   -I-     0xC60 bytes written by applet
   -I-     Writing: 0x21C bytes at 0xC60 (buffer addr : 0x202D38)
   -I-     0x21C bytes written by applet
   ```

8. The second-level bootstrap has now been written to DataFlash, we must now write RedBoot.

9. In the "Send File Name" box type in the path name to the `redboot_ROM.bin` file, or use the Open Folder button and browse to it.

10. In the Address field set the value to 0x8400.

11. Click the "Send File" button. SAM-BA will put up a dialog box while it is writing the file to the DataFlash, and will output something similar to the following in the message area:

    ```
    (sam-ba_cdc_linux) 1 % send_file {DataFlash AT45DB/DCB} "/tmp/redboot_ROM.bin" 0x8400 0
    -I- Send File /tmp/redboot_ROM.bin at address 0x8400
    GENERIC::SendFile /tmp/redboot_ROM.bin at address 0x8400
    -I- File size : 0x248D8 byte(s)
    -I-     Writing: 0xC60 bytes at 0x8400 (buffer addr : 0x202D38)
    -I-     0xC60 bytes written by applet
    -I-     Writing: 0xC60 bytes at 0x9060 (buffer addr : 0x202D38)
    ```

```
-I-     0xC60 bytes written by applet
...
-I-     Writing: 0xC60 bytes at 0x2BD40 (buffer addr : 0x202D38)
-I-     0xC60 bytes written by applet
-I-     Writing: 0x338 bytes at 0x2C9A0 (buffer addr : 0x202D38)
-I-     0x338 bytes written by applet
```

12. Shut down SAM-BA and disconnect the USB cable. Press the reset button on the board and something similar to the following should be output on the DEBUG serial line.

```
RomBOOT
>Start AT91Bootstrap...
+**Warning** FLASH configuration checksum error or invalid key
Use 'fconfig -i' to [re]initialize database
No space to add 'net_device'
AT91_ETH: Waiting for PHY to reset.
AT91_ETH: Waiting for link to come up..
Ethernet eth0: MAC address 12:34:56:78:9a:bc
No IP info for device!

RedBoot(tm) bootstrap and debug environment [ROM]
eCosCentric certified release, version v3_0_22 - built 14:12:12, Sep  7 2010

Copyright (C) 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009
Free Software Foundation, Inc.
Copyright (C) 2003, 2004, 2005, 2006, 2007, 2008 eCosCentric Limited
RedBoot is free software, covered by the eCos license, derived from the
GNU General Public License. You are welcome to change it and/or distribute
copies of it under certain conditions. Under the license terms, RedBoot's
source code and full license terms must have been made available to you.
Redboot comes with ABSOLUTELY NO WARRANTY.

Platform: AT91SAM9G20-EK (ARM9)
RAM: 0x20000000-0x24000000 [0x200362e8-0x23ffef80 available]
FLASH: 0x40000000-0x4083ffff, 8192 x 0x420 blocks
RedBoot>
```

RedBoot Flash configuration

The following steps describe how to initialize RedBoot's Flash configuration.

1. Use the following command to initialize RedBoot's Flash Information System (FIS):

```
RedBoot> fis init
About to initialize [format] FLASH image system - continue (y/n)? y
*** Initialize FLASH Image System
... Erase from 0x4083fbe0-0x4083ffff: .
... Program from 0x23fffbe0-0x24000000 to 0x4083fbe0: .
RedBoot>
```

2. Now configure RedBoot's Flash configuration with the command:

```
RedBoot> fconfig -i
```

Remember to substitute the appropriate MAC address for this board at the appropriate step. If a BOOTP/DHCP server is not available, then IP configuration may be set manually. The default server IP address can be set to a PC that will act as a TFTP host for future RedBoot load operations, or may be left unset. The following gives an example configuration:

```
RedBoot> fconfig -i
Initialize non-volatile configuration - continue (y/n)? y
Run script at boot: false
Use BOOTP for network configuration: false
Gateway IP address: 192.168.7.11
Local IP address: 192.168.7.83
Local IP address mask: 255.255.255.0
```

```
Default server IP address: 192.168.7.11
Console baud rate: 115200
DNS domain name: farm.ecoscentric.com
DNS server IP address: 192.168.7.11
Network hardware address [MAC]: 0x0E:0x00:0x00:0xEA:0x18:0xF0
GDB connection port: 9000
Force console for special debug messages: false
Update RedBoot non-volatile configuration - continue (y/n)? y
... Erase from 0x4083f7c0-0x4083fbdf: .
... Program from 0x23fff7c0-0x23fffbe0 to 0x4083f7c0: .
RedBoot>
```

The RedBoot installation is now complete. This can be tested by powering off the board, and then powering on the board again. Output similar to the following should be seen on the DEBUG serial port. Verify the IP settings are as expected.

```
Ethernet eth0: MAC address 0e:00:00:ea:18:e3
IP: 192.168.7.222/255.255.255.0, Gateway: 192.168.7.11
Default server: 192.168.7.11
DNS server IP: 192.168.7.11, DNS domain name: <null>

RedBoot(tm) bootstrap and debug environment [ROM]
eCosCentric certified release, version v3_0_22 - built 14:12:12, Sep  7 2010

Copyright (C) 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009
Free Software Foundation, Inc.
Copyright (C) 2003, 2004, 2005, 2006, 2007, 2008 eCosCentric Limited
RedBoot is free software, covered by the eCos license, derived from the
GNU General Public License. You are welcome to change it and/or distribute
copies of it under certain conditions. Under the license terms, RedBoot's
source code and full license terms must have been made available to you.
Redboot comes with ABSOLUTELY NO WARRANTY.

Platform: AT91SAM9G20-EK (ARM9)
RAM: 0x20000000-0x24000000 [0x20036218-0x23ffef80 available]
FLASH: 0x40000000-0x4083ffff, 8192 x 0x420 blocks
RedBoot>
```

If it proves necessary to re-install RedBoot, this may be achieved by repeating the above process. Alternatively, a new image may be downloaded and programmed into flash more directly using RedBoot's own commands. See the RedBoot documentation for details.

Rebuilding RedBoot

Should it prove necessary to rebuild a RedBoot binary, this is done most conveniently at the command line. The steps needed to rebuild the ROM version of RedBoot for the AT91SAM9G20-EK are:

```
$ mkdir redboot_at91sam9g20ek_rom
$ cd redboot_at91sam9g20ek_rom
$ ecosconfig new at91sam9g20ek redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/arm/arm9/sam9g20ek/current/misc/redboot_ROM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

At the end of the build the `install/bin` subdirectory should contain the file `redboot.bin`.

Rebuilding AT91Bootstrap

The sources of AT91Bootstrap are found in the `AT91Bootstrap` directory of the sam9260ek package. This is a copy of the software as supplied by Atmel with some slight modifications to permit it to be built with the same tools as eCos.

To rebuild the second-level bootstrap for the AT91SAM9G20-EK execute the following commands:

```
$ cd $ECOS_REPOSITORY/hal/arm/arm9/sam9260ek/current/AT91Bootstrap/board/at91sam9g20ek/dataflash
$ make
```

This should result in the creation of a number of files, including `dataflash_at91sam9g20ek.bin` which can be copied out.

Name

Configuration — Platform-specific Configuration Options

Overview

The AT91SAM9G20-EK platform HAL package is loaded automatically when eCos is configured for the `sam9g20ek` target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The platform HAL package supports two separate startup types:

RAM
: This is the startup type which is normally used during application development. The board has RedBoot programmed into flash and boots into that initially. arm-eabi-gdb is then used to load a RAM startup application into memory and debug it. It is assumed that the hardware has already been initialized by RedBoot. By default the application will use the eCos virtual vectors mechanism to obtain certain services from RedBoot, including diagnostic output.

ROM
: This startup type can be used for finished applications which will be programmed into DataFlash. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization.

JTAG
: This is the startup type which can be used during application development via a JTAG device such as the PEEDI. arm-eabi-gdb is used to load a JTAG startup application into memory and debug it. Hardware setup is divided between the initialization section of the PEEDI configuration file and software in the loaded application.

RedBoot and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building RedBoot.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then the serial port will be claimed for HAL diagnostics.

Flash Driver

The AT91SAM9G20-EK board contains an 8Mbyte Atmel AT45DB DataFlash device. The `CYGPKG_DEVS_FLASH_ATMEL_DATAFLASH` package contains all the code necessary to support this part and the platform HAL package contains definitions that customize the driver to the AT91SAM9G20-EK board. This driver is not active until the generic Flash support package, `CYGPKG_IO_FLASH`, is included in the configuration.

Ethernet Driver

The AT91SAM9G20-EK board uses the AT91SAM9G20's internal EMAC ethernet device attached to an external Davicom DM9161A PHY. The `CYGPKG_DEVS_ETH_ARM_AT91` package contains all the code necessary to support this device and the platform HAL package contains definitions that customize the driver to the AT91SAM9G20-EK board. This driver is not active until the generic Ethernet support package, `CYGPKG_IO_ETH_DRIVERS`, is included in the configuration.

RTC Driver

The AT91SAM9G20-EK board uses the AT91SAM9G20's internal RTTC support. The `CYGPKG_DEVICES_WALLCLOCK_ARM_AT91RTTC` package contains all the code necessary to support this device. This driver is not active until the generic wallclock device support package, `CYGPKG_IO_WALLCLOCK`, is included in the configuration.

Watchdog Driver

The AT91SAM9G20-EK board uses the AT91SAM9G20's internal watchdog support. The `CYGPKG_DEVICES_WATCHDOG_ARM_AT91WDTC` package contains all the code necessary to support this device. Within that package the `CYGNUM_DEVS_WATCHDOG_ARM_AT91WDTC_DESIRED_TIMEOUT_MS` configuration option controls the watchdog timeout, and by default will force a reset of the board upon timeout. This driver is not active until the generic watchdog device support package, `CYGPKG_IO_WATCHDOG`, is included in the configuration.

Warning

The ATSAM926x processor will boot with watchdog support enabled, and the watchdog configuration is write-once. That is, if it is disabled, it cannot be re-enabled. Due to its nature, RedBoot disables the watchdog when it starts so any eCos applications with watchdog support enabled that are run by RedBoot will not function correctly.

USART Serial Driver

The AT91SAM9G20-EK board uses the AT91SAM9G20's internal USART serial support as described in the SAM9 processor HAL documentation. Three serial ports are available: the serial debug port which is mapped to virtual vector channel 0 in the HAL diagnostic driver or "`/dev/dbg`" in the interrupt-driven driver; USART 0 which is mapped to virtual vector channel 1 and "`/dev/ser0`"; and USART 1 which is mapped to virtual vector channel 2 and "`/dev/ser1`". Only USART 0 supports full modem control signals but USART 1 supports RTS/CTS.

MCI Driver

As the SAM MCI driver is included in the hardware-specific configuration for this target, nothing is required to load it. Similarly the MMC/SD bus driver layer (`CYGPKG_DEVS_DISK_MMC`) is automatically included as part of the hardware-specific configuration for this target. All that is required to enable the support is to include the generic disk I/O infrastructure package (`CYGPKG_IO_DISK`), along with the intended filesystem, typically, the FAT filesystem (`CYGPKG_FS_FAT`) and any of its package dependencies (including `CYGPKG_LIBC_STRING` and `CYGPKG_LINUX_COMPAT` for FAT).

If the generic disk I/O infrastructure is needed for some other reason, and you do not wish to also include the MCI driver, then the configuration option within this platform HAL `CYGPKG_HAL_ARM_ARM9_SAM9G20EK_MMCSD` can be used to forcibly disable it.

Various options can be used to control specifics of the SAM MCI driver. Consult the SAM MCI driver documentation for information on its configuration.

On this target, it is not possible to detect from the MMC/SD socket whether cards have been inserted or removed. Thus the disk I/O layer's removeable media support will not detect when cards have been inserted or removed, and therefore the only way to detect if a card has been inserted is to attempt mounts.

The MMC/SD socket also does not permit detection of the write-protect (or "lock") switch present on SD cards. "Locked" cards will therefore not be detected which means that despite the switch position, it is still possible to write to them since the lock switch does not physically enforce write protection.

Compiler Flags

The platform HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos. There are just three flags specific to this port:

`-mcpu=arm9`
The arm-eabi-gcc compiler supports many variants of the ARM architecture. A `-m` option should be used to select the specific variant in use, and with current tools `-mcpu=arm9` is the correct option for the ARM926EJ CPU in the AT91SAM9G20.

`-mthumb`
The arm-eabi-gcc compiler will compile C and C++ files into the Thumb instruction set when this option is used. The best way to build eCos in Thumb mode is to enable the configuration option `CYGHWR_THUMB`.

`-mthumb-interwork`
This option allows programs to be created that mix ARM and Thumb instruction sets. Without this option, some memory can be saved. This option should be used if -mthumb is used. The best way to build eCos with Thumb interworking is to enable the configuration option `CYGBLD_ARM_ENABLE_THUMB_INTERWORK`.

Onboard NAND

The HAL port includes a low-level driver to access the on-board Samsung K9F2G08U08 NAND flash memory chip. To enable the driver, activate the CDL option `CYGPKG_HAL_SAM9G20EK_NAND` and ensure that the `CYGPKG_DEVS_NAND_SAMSUNG_K9` package is present in your eCos configuration.

`CYGHWR_HAL_SAM9G20EK_NAND_USE_STATUS_LINE`
If set, this option configures the driver to wait for NAND operations to complete by waiting for the chip to deassert its Busy line. This is the default behaviour and is recommended, but may be disabled if you need to use the line (PIO C13) for some other purpose. (If disabled, the memory controller is configured to stall NAND accesses until they complete, which will interfere with multi-threading.)

`CYGNUM_HAL_SAM9G20EK_NAND_POLL_INTERVAL`
The number of microseconds delay in the polling loops which wait for NAND operations to complete.

Partitioning the NAND chip

The NAND chip must be partitioned before it can become available to applications.

A CDL script which allows the chip to be manually partitioned is provided (see `CYGSEM_DEVS_NAND_SAM9G20EK_PARTITION_MANUAL_CONFIG`); if you choose to use this, the relevant data structures will automatically be set up for you when the device is initialised. By default, the manual config CDL script sets up a single partition (number 0) encompassing the entire device.

It is possible to configure the partitions in some other way, should it be appropriate for your setup, for example to read a Linux-style partition table from the chip. To do so you will have to add appropriate code to `sam9g20ek_nand.c`.

Name

JTAG support — Usage

Use of JTAG for debugging

JTAG can be used to single-step and debug loaded applications, including RedBoot. Only JTAG configuration applications should be debugged using JTAG, RAM applications assume the presence of RedBoot.

Ronetix PEEDI notes

On the Ronetix PEEDI, the `peedi.at91sam9g20ek.cfg` file should be used to setup and configure the hardware to an appropriate state to load programs. This includes setting up the SDRAM controller.

The `peedi.at91sam9g20ek.cfg` file also contains an option to define whether hardware or software breakpoints are used by default, using the `CORE0_BREAKMODE` directive in the `[PLATFORM_ARM]` section. Edit this file if you wish to use hardware break points, and remember to restart the PEEDI to make the changes take effect.

On the PEEDI, debugging can be performed either via the telnet interface or using **arm-eabi-gdb** and the GDB interface. In the case of the latter, **arm-eabi-gdb** needs to connect to TCP port 2000 on the PEEDI's IP address. For example:

```
(gdb) target remote 111.222.333.444:2000
```

By default when the PEEDI is powered up, the target will always run the initialization section of the `peedi.at91sam9g20ek.cfg` file (which configures the SDRAM among other things), and halts the target. This behavior is repeated with the **reset** command.

If the board is reset (either with the '**reset**', or by pressing the reset button) and the '**go**' command is then given, then the board will boot as normal. If a second-level bootstrap and ROM RedBoot is resident in DataFlash, it will be run.

An issue occurs when the AT91 Ethernet driver is included in your configuration. In order to work around a board hardware design issue, the CPU generates an external reset in order to reset the Ethernet PHY. However this can be interpreted by the PEEDI as an indication that the CPU itself has reset, and if the PEEDI configuration file option `CORE0_STARTUP_MODE` is set to `RESET` then the CPU will be halted at this point. To avoid this issue, the `CORE0_STARTUP_MODE` can be set to `RUN`.

Consult the PEEDI documentation for information on other features.

Running JTAG applications

Applications configured for JTAG startup can be run directly under a JTAG debugger. Once loaded and running via JTAG, HAL diagnostic output will appear by default on the serial debug port. USARTs 0 or 1 can be chosen instead by setting the `CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL` configuration option in the platform HAL to channel 1 or 2.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the AT91SAM9G20-EK hardware, and should be read in conjunction with that specification. The AT91SAM9G20-EK platform HAL package complements the ARM architectural HAL, the ARM9 variant HAL and the SAM9 processor HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset, the HAL will initialize or reinitialize most of the on-chip peripherals. There is an exception for RAM startup applications which depend on a ROM monitor for certain services.

For ROM startup, the HAL will perform additional initialization. This is all done in the `PLATFORM_SETUP1` macro in the assembler header file `hal_platform_setup.h`.

Linker Scripts and Memory Maps

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

SDRAM

This is located at address 0x20000000 of the physical memory space. The HAL configures the MMU to retain the SDRAM at virtual address 0x20000000, but in order to assign hardware exception vectors vectors at address 0x00000000, the HAL also uses the MMU to create a clone of this memory at virtual address 0x00000000. The same memory is also accessible uncached and unbuffered at virtual location 0x30000000 for use by devices. The first 32 bytes are used for hardware exception vectors. The next 32 bytes are used for the VSR table and the next 256 bytes are normally used for the eCos virtual vectors, allowing RAM-based applications to use services provided by the ROM monitor. Memory is required for the MMU tables, and must be aligned on a 16Kbyte boundary. These therefore occupy memory from 0x4000 to 0x8000. For ROM startup, all remaining SDRAM is available. For RAM startup, available RAM starts at virtual location 0x20040000, with the bottom 256kB reserved for use by RedBoot.

On-chip SRAM

This is located at address 0x00200000 of the physical memory space. However the HAL uses the MMU to relocate this to virtual address 0x70000000. The same memory is also accessible uncached and unbuffered at virtual location 0x70100000 for use by devices. At present this memory is unused by eCos and is available for application use.

On-chip ROM

This is located at address 0x00100000 of the physical memory space. However the HAL uses the MMU to relocate this to virtual address 0x71000000. The same memory is also accessible uncached and unbuffered at virtual location 0x71800000.

USB host port

The USB host port (UHP) registers are located at address 0x00300000 of the physical memory space. However the HAL uses the MMU to relocate this to virtual address 0x72800000. Memory accessed at this address is uncached and unbuffered. There is no cached variant.

SPI dataflash

SPI Dataflash media can only be accessed with the Flash API. For the purposes of this API a placeholder address range has been allocated as if the Flash is present at this address. The base of this address range is 0x40000000 for the on-board flash and 0x50000000 for the dataflash slot, the extent will clearly depend on the Dataflash capacity. This reserved range is not real memory and any attempt to access it directly by the processor other than via the Flash API will result in a memory address exception.

On-chip Peripheral Registers
These are located at address 0xFF000000 in the physical memory space. When the MMU is enabled, it sets up a direct, uncached, unbuffered mapping so that these registers remain accessible at their physical locations.

Off-chip Peripherals
eCos uses the SDRAM, ethernet PHY, MCI, and SPI dataflash facilities on the AT91SAM9G20-EK board. eCos does not currently make any use of any other off-chip peripherals present on this board.

Advanced Interrupt Controller

This port has been designed to exploit benefits of the Advanced Interrupt Controller of the AT91SAM9G20, using the facilities of the SAM9 processor HAL. Consult the documentation in that package for details.

SPI Dataflash

eCos supports SPI access to Dataflash on the AT91SAM9G20-EK. An on-board device and an external card slot are provided on the board. The on-chip device is typically used to contain RedBoot and flash configuration data. The external slot is available for application use.

Accesses to Dataflash are performed via the Flash API, using 0x40000000 or 0x50000000 as the nominal address of the device, although it does not truly exist in the processor address space. For the external card slot, on driver initialisation, eCos and RedBoot can detect the presence of a card in the socket. In particular, on reset RedBoot will indicate the presence of Flash at the 0x40000000 address range in its startup banner if it has been successfully detected. Hot swapping is not possible.

Since Dataflash is not directly addressable, access from RedBoot is only possible using **fis** command operations.

The MCI driver cannot be enabled simultaneously with the SPI driver, as the drivers need differing pin configurations for the same pins on this board due to the shared socket.

Real-time characterization

The tm_basic kernel test gives statistics gathered about the real-time characterization and performance of the kernel. The sample output is shown here for information. The test was built in ARM mode, which provided better performance than Thumb mode.

Example 261.1. sam9g20ek Real-time characterization

```
INFO:<code from 0x20040040 -> 0x2004bad4, CRC eb0a>
            Startup, main stack : stack used   388 size  3920
             Startup :  Interrupt stack used   524 size  4096
             Startup : Idlethread stack used    96 size  2048

eCos Kernel Timings
Notes: all times are in microseconds (.000001) unless otherwise stated

Reading the hardware clock takes 1 'ticks' overhead
... this value will be factored out of all other measurements
Clock interrupt took    2.93 microseconds (24 raw clock ticks)

Testing parameters:
   Clock samples:            32
   Threads:                  64
   Thread switches:         128
   Mutexes:                  32
   Mailboxes:                32
   Semaphores:               32
   Scheduler operations:    128
   Counters:                 32
   Flags:                    32
   Alarms:                   32


                                 Confidence
     Ave     Min     Max     Var  Ave  Min  Function
  ======  ======  ======  ====== ========== ========
    2.60    1.82    4.84    0.40   46%  28% Create thread
    0.32    0.24    1.33    0.08   98%  50% Yield thread [all suspended]
    0.48    0.36    1.09    0.07   56%  28% Suspend [suspended] thread
    0.43    0.36    0.85    0.09   85%  68% Resume thread
    0.66    0.48    1.70    0.08   95%   1% Set priority
    0.21    0.00    0.48    0.09   79%   1% Get priority
    1.37    1.21    3.88    0.14   54%  78% Kill [suspended] thread
    0.32    0.24    0.73    0.06   54%  43% Yield [no other] thread
    0.67    0.48    1.33    0.10   79%   7% Resume [suspended low prio] thread
    0.43    0.36    0.73    0.09   85%  68% Resume [runnable low prio] thread
    0.71    0.61    1.21    0.07   54%  31% Suspend [runnable] thread
    0.31    0.24    0.85    0.07   50%  48% Yield [only low prio] thread
    0.48    0.36    0.97    0.07   53%  28% Suspend [runnable->not runnable]
    1.23    1.09    2.54    0.09   60%  20% Kill [runnable] thread
    1.25    0.97    3.39    0.14   54%  34% Destroy [dead] thread
    1.90    1.70    3.15    0.11   81%   4% Destroy [runnable] thread
    2.88    2.54    5.21    0.21   64%  29% Resume [high priority] thread
    0.90    0.85    1.94    0.07   98%  61% Thread switch

    0.02    0.00    0.61    0.03   86%  86% Scheduler lock
    0.25    0.24    0.73    0.01   98%  98% Scheduler unlock [0 threads]
    0.25    0.24    0.48    0.01   98%  98% Scheduler unlock [1 suspended]
    0.25    0.24    0.48    0.02   92%  92% Scheduler unlock [many suspended]
    0.25    0.24    0.48    0.02   92%  92% Scheduler unlock [many low prio]

    0.11    0.00    0.97    0.06   68%  28% Init mutex
    0.51    0.36    1.45    0.10   71%  25% Lock [unlocked] mutex
    0.54    0.36    1.94    0.13   75%  68% Unlock [locked] mutex
    0.44    0.36    1.21    0.09   96%  56% Trylock [unlocked] mutex
    0.39    0.24    0.61    0.07   53%  15% Trylock [locked] mutex
    0.03    0.00    0.24    0.04   81%  81% Destroy mutex
    2.08    2.06    2.66    0.04   96%  96% Unlock/Lock mutex

    0.19    0.12    1.09    0.09   96%  65% Create mbox
    0.16    0.12    0.61    0.06   75%  75% Peek [empty] mbox
    0.60    0.48    1.57    0.10   43%  40% Put [first] mbox
    0.16    0.12    0.48    0.06   81%  81% Peek [1 msg] mbox
    0.59    0.48    0.97    0.08   40%  40% Put [second] mbox
    0.15    0.00    0.36    0.08   62%  12% Peek [2 msgs] mbox
    0.63    0.48    1.70    0.10   68%  25% Get [first] mbox
    0.60    0.48    0.97    0.07   50%  31% Get [second] mbox
    0.51    0.36    1.33    0.10   65%  21% Tryput [first] mbox
    0.56    0.36    1.33    0.10   84%   6% Peek item [non-empty] mbox
    0.59    0.48    1.45    0.09   43%  43% Tryget [non-empty] mbox
    0.46    0.36    0.73    0.07   56%  34% Peek item [empty] mbox
    0.45    0.36    0.97    0.08   46%  43% Tryget [empty] mbox
    0.14    0.00    0.36    0.07   65%  15% Waiting to get mbox
    0.15    0.12    0.48    0.04   84%  84% Waiting to put mbox
    0.22    0.12    0.73    0.08   46%  40% Delete mbox
    1.48    1.45    2.30    0.05   96%  96% Put/Get mbox

    0.02    0.00    0.24    0.03   87%  87% Init semaphore
    0.31    0.24    0.85    0.08   96%  59% Post [0] semaphore
    0.42    0.36    1.09    0.08   96%  71% Wait [1] semaphore
    0.32    0.24    0.85    0.08   96%  50% Trywait [0] semaphore
    0.29    0.24    0.48    0.06   62%  62% Trywait [1] semaphore
    0.08    0.00    0.48    0.09   87%  53% Peek semaphore
    0.05    0.00    0.85    0.07   81%  81% Destroy semaphore
    1.37    1.33    2.18    0.07   93%  93% Post/Wait semaphore
    0.22    0.12    1.45    0.10   50%  46% Create counter
    0.13    0.00    0.48    0.04   75%  12% Get counter value
    0.07    0.00    0.36    0.08   90%  56% Set counter value
    0.48    0.36    0.73    0.07   46%  28% Tick counter
    0.14    0.00    0.61    0.08   59%  21% Delete counter

    0.06    0.00    0.48    0.08   93%  59% Init flag
    0.40    0.24    1.33    0.07   81%   6% Destroy flag
    0.33    0.24    1.09    0.10   87%  59% Mask bits in flag
    0.39    0.24    0.97    0.07   62%  15% Set bits in flag [no waiters]
    0.53    0.48    1.82    0.09   96%  90% Wait for flag [AND]
    0.47    0.36    0.97    0.05   75%  21% Wait for flag [OR]
    0.50    0.48    0.85    0.03   90%  90% Wait for flag [AND/CLR]
    0.47    0.36    0.85    0.05   71%  25% Wait for flag [OR/CLR]
    0.00    0.00    0.12    0.01   96%  96% Peek on flag

    0.34    0.24    1.21    0.08   53%  43% Create alarm
    0.77    0.61    2.42    0.12   75%  18% Initialize alarm
    0.42    0.36    1.09    0.08   93%  68% Disable alarm
    0.75    0.61    2.18    0.10   71%  25% Enable alarm
    0.51    0.36    1.21    0.07   65%  15% Delete alarm
    0.44    0.36    0.97    0.07   50%  46% Tick counter [1 alarm]
    2.67    2.66    2.91    0.01   96%  96% Tick counter [many alarms]
    0.82    0.73    1.33    0.07   59%  37% Tick & fire counter [1 alarm]
   14.87   14.78   15.14    0.06   65%  31% Tick & fire counters [>1 together]
    3.06    3.03    3.39    0.05   81%  81% Tick & fire counters [>1 separately]
    2.43    2.42    3.51    0.02   99%  99% Alarm latency [0 threads]
    2.56    2.42    2.91    0.07   60%  17% Alarm latency [2 threads]
    3.64    3.15    4.36    0.26   67%  39% Alarm latency [many threads]
    3.83    3.75    6.30    0.08   98%  50% Alarm -> thread resume latency

    0.81    0.73    1.94    0.00            Clock/interrupt latency

    0.87    0.61    2.18    0.00            Clock DSR latency

    4      0     260  (main stack:  1356)  Thread stack used (1360 total)
           All done, main stack : stack used  1356 size  3920
            All done :  Interrupt stack used   664 size  4096
            All done : Idlethread stack used   240 size  2048

Timing complete - 29880 ms total

PASS:<Basic timing OK>
EXIT:<done>
```

Other Issues

The AT91SAM9G20-EK platform HAL does not affect the implementation of other parts of the eCos HAL specification. The SAM9 processor HAL, ARM9 variant HAL, and the ARM architectural HAL documentation should be consulted for further details.

Chapter 262. Atmel AT91SAM9G45-EKES Evaluation Kit Board Support

Name

eCos Support for the Atmel AT91SAM9G45-EKES Evaluation Kit — Overview

Description

This document covers the configuration and usage of eCos and RedBoot on the Atmel AT91SAM9G45-EKES Evaluation Kit. The AT91SAM9G45-EKES contains the AT91SAM9G45 microprocessor, 128Mbytes of SDRAM, 256Mbytes of NAND flash memory, an Atmel Dataflash, an Atmel serial EEPROM, a Davicom DM9161A PHY, two SD/MMC sockets, a DAC, external connections for two serial channels (one debug, one flow controlled), ethernet, USB host/device, and the various other peripherals supported by the AT91SAM9G45. eCos support for the devices and peripherals on the boards and the AT91SAM9G45 is described below.

For typical eCos development, a RedBoot image is programmed into the start of NAND, and the board will load this image from reset. RedBoot provides gdb stub functionality so it is then possible to download and debug stand-alone and eCos applications via the gdb debugger. This can happen over either a serial line or over ethernet.

This documentation is expected to be read in conjunction with the SAM9 processor HAL documentation and further device support and subsystems are described and documented there.

Supported Hardware

The NAND flash contains the second level bootstrap, AT91Bootstrap, in the first block. This initializes the system clocks and the SDRAM. It then loads a 320KiB program image from offset 0x20000 in the NAND into SDRAM at 0x20008000 and then jumps into it. Typically this image will be a ROM version of RedBoot.

The Dataflash consists of 8192 blocks of 528 bytes each. The topmost block is used to manage the flash and the next four block holds RedBoot **fconfig** values. The remaining blocks can be used by application code.

There is a serial driver `CYGPKG_IO_SERIAL_ARM_AT91` which supports both the Debug Unit and USART serial devices. The debug serial port at J10 and flow controlled port at J11 (connected to USART channel 1) can be used by RedBoot for communication with the host. If any of these devices is needed by the application, either directly or via the serial driver, then it cannot also be used for RedBoot communication. Another communication channel such as ethernet should be used instead. The serial driver package is loaded automatically when configuring for the AT91SAM9G45-EKES target.

There is an ethernet driver `CYGPKG_DEVS_ETH_ARM_AT91` for the on-chip ethernet device. The platform HAL package is responsible for configuring this generic driver to the hardware. This driver is also loaded automatically when configuring for the AT91SAM9G45-EK board.

There is a watchdog driver `CYGPKG_DEVICES_WATCHDOG_ARM_AT91WDTC`. This driver is also loaded automatically when configuring for the board.

There is a driver for the on-chip real-time timer controller (RTTC) at `CYGPKG_DEVICES_WALLCLOCK_ARM_AT91RTC`. This driver is also loaded automatically when configuring for the target.

The SAM9 processor HAL contains a driver for the Two-Wire Interface (TWI) controller on the AT91SAM9G45. This type of bus is also known as I²C®. Further documentation may be found in the SAM9 processor HAL documentation. Note that the implementation of the TWI device on this part appears to have changed from earlier parts. This device does not now appear to be able to handle a repeat start sequence.

There is a driver for the MultiMedia Card Interface (MCI) at `CYGPKG_DEVS_MMCSD_ATMEL_SAM_MCI`. This driver is loaded automatically when configuring for the AT91SAM9G45-EKES target and allows use of MMC and Secure Digital (SD) flash storage cards within eCos, exported as block devices. Further documentation may be found within that package.

The platform HAL provides definitions to allow access to devices on the SPI bus. The HAL provides information to the general AT91 SPI driver (`CYGPKG_DEVS_SPI_ARM_AT91`) which in turn provides the underlying implementation for the SPI API layer in the `CYGPKG_IO_SPI` package. All these packages are automatically loaded when configuring for the board.

Furthermore, the platform HAL package contains support for SPI dataflash cards. The HAL support integrates with the CYGPKG_DEVS_FLASH_ATMEL_DATAFLASH package as well as the above SPI packages. That package is automatically loaded when configuring for the target. Dataflash media is then accessed as a Flash device, using the Flash I/O API within the CYGPKG_IO_FLASH package, if that package is loaded in the configuration.

In general, devices (Caches, PIO, UARTs, EMAC) are initialized only as far as is necessary for eCos to run. Other devices (RTC, I²C, SPI, MCI etc.) are not touched unless the appropriate driver is loaded, although in some cases, the HAL boot sequence will set up the appropriate PIO configuration.

Tools

The AT91SAM9G45-EKES support is intended to work with GNU tools configured for an arm-eabi target. The original port was undertaken using arm-elf-gcc version 4.3.2, arm-elf-gdb version 6.8, and binutils version 2.18.

Name

Setup — Preparing the AT91SAM9G45-EKES board for eCos Development

Overview

In a typical development environment, the AT91SAM9G45-EKES board boots from the NAND flash and runs the RedBoot ROM monitor from SDRAM. eCos applications are configured for RAM startup and then downloaded and run on the board via the debugger **arm-eabi-gdb**. Preparing the board therefore usually involves programming a suitable RedBoot image into flash memory.

The following RedBoot configurations are supported:

Configuration	Description	Use	File
ROM	RedBoot loaded from NAND to SDRAM	`redboot_ROM.ecm`	`redboot_ROM.bin`

For serial communications, all versions run with 8 bits, no parity, and 1 stop bit at 115200 baud. RedBoot also supports ethernet communication and flash management.

Note that the use of the term ROM for the initial RedBoot configuration is a historical accident. RedBoot actually runs from SDRAM after being loaded there from NAND by the second-level bootstrap. The use of ROM for this configuration is intended to indicate that it initializes the microprocessor and board peripherals, as opposed to the RAM configuration which assumes that this has already been done.

Initial Installation

The on-chip boot program on the AT91SAM9G45 is only capable of loading programs from Dataflash or NAND flash into on-chip SRAM and is therefore quite restrictive. Consequently RedBoot cannot be booted directly and a second-level bootstrap must be used. Such a second-level bootstrap is supplied by Atmel in the form of AT91Bootstrap. This is therefore programmed into the start of NAND and is then responsible for initializing the SDRAM and loading RedBoot from NAND and executing it.

Caution

There is a size limit on the size of applications which the AT91Bootstrap second level bootstrap will load. Images larger than 320Kbytes will require the AT91Bootstrap application to be rebuilt with a larger `IMG_SIZE` definition in `AT91Bootstrap/board/at91sam9g45ek/nandflash/at91sam9g45ek.h` within the SAM9260 HAL package in the eCos source repository (`packages/hal/arm/arm9/sam9260ek/current/`). A pre-built AT91SAM9G45 specific AT91Bootstrap binary can be found here: `AT91Bootstrap/board/at91sam9g45ekes/nandflash/nandflash_at91sam9g45ekes.bin`. This will required during the SAM-BA based board setup process described below.

There are basically two ways to write the second-level bootstrap and RedBoot to the NAND. The first is to use the Atmel-supplied SAM-BA program that interacts with the on-chip boot program. The second is to use a JTAG debugger that understands the microcontroller and can write to the NAND flash (for example the Ronetix PEEDI). Since the availability of the latter cannot be guaranteed, only the first method will be described here.

Programming RedBoot into NAND Flash using SAM-BA

The following gives the steps needed to program the second-level bootstrap and RedBoot into the NAND Flash using SAM-BA. The user should refer to the SAM-BA documentation for full details of how to run the program. SAM-BA can communicate with the boot program via either USB or serial. The steps are essentially similar for both since USB operates through a driver that simulates a serial port.

1. Download the AT91 SAM-BA software package from the Atmel website and install it. SAM9 series CPU's require the 2.1.x series version of SAM-BA. Atmel provide both Linux and Windows versions of SAM-BA so ensure you select the version appropriate to your host operating system. The remainder of this document describes the process according to a Windows installation. The steps for the Linux version of SAM-BA are similar and can easily be determined from the Windows process.

2. From the root directory of your eCosPro installation, copy the file `nandflash_at91sam9g45ekes.bin` from the sub-directory `packages/hal/arm/arm9/sam9260ek/current/AT91Bootstrap/board/at91sam9g45ekes/nandflash` and `redboot_ROM.bin` from the sub-directory `loaders/sam9g45ekes` to a suitable location on the Windows PC.

3. Connect a null-modem serial cable between the J10 DEBUG serial port of the board and a serial port on the host running SAM-BA. If using USB download also connect a USB cable between the host and the J14 HOST/DEV socket.

4. Unplug the power supply from the board and remove JP10(NANDCS) and JP12(NPCS0). Reapply power to the board. If USB is being used follow the instructions in the SAM-BA documentation to set up the Windows device driver for the first time if you are running a Microsoft Windows-based operating system.

5. Start SAM-BA. Select the appropriate COM port for the communication interface (on Windows hosts this will be of the form "COMx" and on Linux hosts this will be of the form "/dev/ttyUSBx"), and "at91sam9g45-ekes" for the board. Click on "Connect".

6. Re-insert JP10 and JP12.

7. In the SAM-BA main window, select the "NandFlash" tab and in the "Scripts" dropdown menu select "Enable NandFlash", to program the on-board NandFlash device. Click Execute and SAM-BA should emit the following in the message area:

   ```
   (SAM-BA v2.10) 1 % NANDFLASH::Init
   -I- NANDFLASH::Init (trace level : 0)
   -I- Loading applet isp-nandflash-at91sam9g45.bin at address 0x70000000
   -I- Memory Size : 0x10000000 bytes
   -I- Buffer address : 0x70003E34
   -I- Buffer size: 0x20000 bytes
   -I- Applet initialization done
   (SAM-BA v2.10) 1 %
   ```

   The actual options and output of SAM-BA may vary according to the version you are using. The behaviour documented here is that of SAM-BA 2.10.

8. Now select "Send BootFile" from the "Scripts" menu and "Execute" it. When the file open dialog appears, select the `nandflash_at91sam9g45ekes.bin` file and click "Open". The following output should be seen:

   ```
   (SAM-BA v2.10) 1 % GENERIC::SendBootFileGUI
   GENERIC::SendFile Z:/eCos/SAM-BA/nandflash_at91sam9g45ekes.bin at address 0x0
   -I- File size : 0x1334 byte(s)
   -I-     Writing: 0x1334 bytes at 0x0 (buffer addr : 0x70003E34)
   -I-     0x1334 bytes written by applet
   (SAM-BA v2.10) 1 %
   ```

9. The second-level bootstrap has now been written to NAND Flash, we must now write RedBoot.

10. In the "Send File Name" box type in the path name to the `redboot_ROM.bin` file, or use the Open Folder button and browse to it.

11. In the Address field set the value to 0x020000.

12. Click the "Send File" button. SAM-BA will put up a dialog box while it is writing the file to the NAND Flash, and will output something similar to the following in the message area:

    ```
    (SAM-BA v2.10) 1 % send_file {NandFlash} "Z:/eCos/SAM-BA/redboot_ROM.bin" 0x020000 0
    -I- Send File Z:/eCos/SAM-BA/redboot_ROM.bin at address 0x020000
    ```

```
GENERIC::SendFile Z:/eCos/SAM-BA/redboot_ROM.bin at address 0x20000
-I- File size : 0x24658 byte(s)
-I-     Writing: 0x20000 bytes at 0x20000 (buffer addr : 0x70003E34)
-I-     0x20000 bytes written by applet
-I-     Writing: 0x4658 bytes at 0x40000 (buffer addr : 0x70003E34)
-I-     0x4658 bytes written by applet
(SAM-BA v2.10) 1 %
```

13. Shut down SAM-BA and start up Hyperterm or similar on the real COM port, not the USB port, configured for 115200 baud 8-N-1 with no flow control. Attach an ethernet cable between the board and a switch on your network. Press the reset button on the board and something similar to the following should be output on the DEBUG serial line.

```
Start AT91Bootstrap...
+**Warning** FLASH configuration checksum error or invalid key
Use 'fconfig -i' to [re]initialize database
Ethernet eth0: MAC address 0e:00:00:ea:18:f0
IP: 10.0.2.1/255.0.0.0, Gateway: 10.0.0.3
Default server: 0.0.0.0
DNS server IP: 10.0.0.1, DNS domain name: <null>

RedBoot(tm) bootstrap and debug environment [ROM]
Non-certified release, version UNKNOWN - built 14:54:15, Aug 20 2010

Copyright (C) 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009
Free Software Foundation, Inc.
Copyright (C) 2003, 2004, 2005, 2006, 2007, 2008 eCosCentric Limited
RedBoot is free software, covered by the eCos license, derived from the
GNU General Public License. You are welcome to change it and/or distribute
copies of it under certain conditions. Under the license terms, RedBoot's
source code and full license terms must have been made available to you.
Redboot comes with ABSOLUTELY NO WARRANTY.

Platform: AT91SAM9G45-EK (ARM9)
RAM: 0x20000000-0x28000000 [0x20035f78-0x27ffe530 available]
FLASH: 0x50000000-0x5041ffff, 8192 x 0x210 blocks
RedBoot>
```

The board is now running and the flash can be configured.

RedBoot Flash configuration

The following steps describe how to initialize RedBoot's Flash configuration.

1. Use the following command to initialize RedBoot's Flash Information System (FIS):

```
RedBoot> fis init
About to initialize [format] FLASH image system - continue (y/n)? y
*** Initialize FLASH Image System
... Erase from 0x5041fdf0-0x5041ffff: .
... Program from 0x27fffdf0-0x28000000 to 0x5041fdf0: .
RedBoot>
```

2. Now configure RedBoot's Flash configuration with the **fconfig** command. Remember to substitute the appropriate MAC address for this board at the appropriate step. If a BOOTP/DHCP server is not available, then IP configuration may be set manually. The default server IP address can be set to a PC that will act as a TFTP host for future RedBoot load operations, or may be left unset. The following gives an example configuration:

```
RedBoot> fconfig -i
Initialize non-volatile configuration - continue (y/n)? y
Run script at boot: false
Use BOOTP for network configuration: false
Gateway IP address: 192.168.7.11
Local IP address: 192.168.7.83
Local IP address mask: 255.255.255.0
```

```
   Default server IP address: 192.168.7.11
   Console baud rate: 115200
   DNS domain name: farm.ecoscentric.com
   DNS server IP address: 192.168.7.11
   Network hardware address [MAC]: 0x0E:0x00:0x00:0xEA:0x18:0xF0
   GDB connection port: 9000
   Force console for special debug messages: false
   Update RedBoot non-volatile configuration - continue (y/n)? y
   ... Erase from 0x5041ef80-0x5041f7bf: ....
   ... Program from 0x27fff5b0-0x27fffdf0 to 0x5041ef80: ....
   RedBoot>
```

The RedBoot installation is now complete. This can be tested by power cycling the board. Output similar to the following should be seen on the DEBUG serial port. Verify the IP settings are as expected.

```
Ethernet eth0: MAC address 0e:00:00:ea:18:f0
IP: 192.168.7.83/255.255.255.0, Gateway: 192.168.7.11
Default server: 192.168.7.11
DNS server IP: 192.168.7.11, DNS domain name: <null>

RedBoot(tm) bootstrap and debug environment [ROM]
Non-certified release, version UNKNOWN - built 14:54:15, Aug 20 2010

Copyright (C) 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009
Free Software Foundation, Inc.
Copyright (C) 2003, 2004, 2005, 2006, 2007, 2008 eCosCentric Limited
RedBoot is free software, covered by the eCos license, derived from the
GNU General Public License. You are welcome to change it and/or distribute
copies of it under certain conditions. Under the license terms, RedBoot's
source code and full license terms must have been made available to you.
Redboot comes with ABSOLUTELY NO WARRANTY.

Platform: AT91SAM9G45-EK (ARM9)
RAM: 0x20000000-0x28000000 [0x20035f78-0x27ffe530 available]
FLASH: 0x50000000-0x5041ffff, 8192 x 0x210 blocks
RedBoot>
```

If it proves necessary to re-install RedBoot, this may be achieved by repeating the above process.

Rebuilding RedBoot

Should it prove necessary to rebuild a RedBoot binary, this is done most conveniently at the command line. The steps needed to rebuild the ROM version of RedBoot for the AT91SAM9G45-EKES are:

```
$ mkdir redboot_at91sam9g45ek_rom
$ cd redboot_at91sam9g45ek_rom
$ ecosconfig new at91sam9g45ekes redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/arm/arm9/sam9g45ek/current/misc/redboot_ROM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

At the end of the build the `install/bin` subdirectory should contain the file `redboot.bin`.

Rebuilding AT91Bootstrap

The sources of AT91Bootstrap are found in the `AT91Bootstrap` directory of the sam9260ek package. This is a copy of the software as supplied by Atmel with some slight modifications to permit it to be built with the same tools as eCos.

To rebuild the second-level bootstrap for the AT91SAM9G45-EKES execute the following commands:

```
    $ cd $ECOS_REPOSITORY/hal/arm/arm9/sam9260ek/current/AT91Bootstrap/board/at91sam9g45ekes/nandflash
    $ make
```

This should result in the creation of a number of files, including `nandflash_at91sam9g45ekes.bin` which can be copied out.

Name

Configuration — Platform-specific Configuration Options

Overview

The AT91SAM9G45-EKES platform HAL package is loaded automatically when eCos is configured for the `sam9g45ek` target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The platform HAL package supports two separate startup types:

RAM
: This is the startup type which is normally used during application development. The board has RedBoot programmed into flash and boots into that initially. arm-eabi-gdb is then used to load a RAM startup application into memory and debug it. It is assumed that the hardware has already been initialized by RedBoot. By default the application will use the eCos virtual vectors mechanism to obtain certain services from RedBoot, including diagnostic output.

ROM
: This startup type can be used for finished applications which will be programmed into NAND Flash. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization.

JTAG
: This is the startup type which can be used during application development via a JTAG device such as the PEEDI. arm-eabi-gdb is used to load a JTAG startup application into memory and debug it. Hardware setup is divided between the initialization section of the PEEDI configuration file and software in the loaded application.

RedBoot and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building RedBoot.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then the serial port will be claimed for HAL diagnostics.

Flash Driver

The AT91SAM9G45-EKES board contains a 4Mbyte Atmel AT45 DataFlash device. The `CYGPKG_DEVS_FLASH_ATMEL_DATAFLASH` package contains all the code necessary to support this part and the platform HAL package contains definitions that customize the driver to the AT91SAM9G45-EKES board. This driver is not active until the generic Flash support package, `CYGPKG_IO_FLASH`, is included in the configuration.

Ethernet Driver

The AT91SAM9G45-EKES board uses the AT91SAM9G45's internal EMAC ethernet device attached to an external Davicom DM9161A PHY. The `CYGPKG_DEVS_ETH_ARM_AT91` package contains all the code necessary to support this device and the platform HAL package contains definitions that customize the driver to the AT91SAM9G45-EKES board. This driver is not active until the generic Ethernet support package, `CYGPKG_IO_ETH_DRIVERS`, is included in the configuration.

RTC Driver

The AT91SAM9G45-EKES board uses the AT91SAM9G45's internal RTC support. The `CYGPKG_DEVICES_WALLCLOCK_ARM_AT91RTC` package contains all the code necessary to support this device. This driver is not active until the generic wallclock device support package, `CYGPKG_IO_WALLCLOCK`, is included in the configuration.

Watchdog Driver

The AT91SAM9G45-EKES board uses the AT91SAM9G45's internal watchdog support. The `CYGPKG_DEVICES_WATCHDOG_ARM_AT91WDTC` package contains all the code necessary to support this device. Within that package the `CYGNUM_DEVS_WATCHDOG_ARM_AT91WDTC_DESIRED_TIMEOUT_MS` configuration option controls the watchdog timeout, and by default will force a reset of the board upon timeout. This driver is not active until the generic watchdog device support package, `CYGPKG_IO_WATCHDOG`, is included in the configuration.

Warning

The ATSAM926x processor will boot with watchdog support enabled, and the watchdog configuration is write-once. That is, if it is disabled, it cannot be re-enabled. Due to its nature, RedBoot disables the watchdog when it starts so any eCos applications with watchdog support enabled that are run by RedBoot will not function correctly.

USART Serial Driver

The AT91SAM9G45-EKES board uses the AT91SAM9G45's internal USART serial support as described in the SAM9 processor HAL documentation. Two serial ports are available: the serial debug port which is mapped to virtual vector channel 0 in the HAL diagnostic driver or "`/dev/dbg`" in the interrupt-driven driver and USART 1 which is mapped to virtual vector channel 1 and "`/dev/ser1`". The debug port has no additional signals, but USART 1 supports RTS/CTS.

MCI Driver

As the SAM MCI driver is included in the hardware-specific configuration for this target, nothing is required to load it. Similarly the MMC/SD bus driver layer (`CYGPKG_DEVS_DISK_MMC`) is automatically included as part of the hardware-specific configuration for this target. All that is required to enable the support is to include the generic disk I/O infrastructure package (`CYGPKG_IO_DISK`), along with the intended filesystem, typically, the FAT filesystem (`CYGPKG_FS_FAT`) and any of its package dependencies (including `CYGPKG_LIBC_STRING` and `CYGPKG_LINUX_COMPAT` for FAT).

If the generic disk I/O infrastructure is needed for some other reason, and you do not wish to also include the MCI driver, then the configuration option within this platform HAL `CYGPKG_HAL_ARM_ARM9_SAM9G45EK_MMCSD` can be used to forcibly disable it.

Various options can be used to control specifics of the SAM MCI driver. Consult the SAM MCI driver documentation for information on its configuration.

The MCI driver at present can only handle a single MCI interface, the configuration option `CYGHWR_DEVS_MMCSD_ATMEL_SAM_MCI_DEVICE` selects between them.

Only MMC/SD socket 1 permits detection of the write-protect (or "lock") switch present on SD cards. "Locked" cards will therefore not be detected on socket 0 which means that despite the switch position, it is still possible to write to them since the lock switch does not physically enforce write protection.

Compiler Flags

The platform HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos. There are just three flags specific to this port:

`-mcpu=arm9`
: The arm-eabi-gcc compiler supports many variants of the ARM architecture. A `-m` option should be used to select the specific variant in use, and with current tools `-mcpu=arm9` is the correct option for the ARM926EJ CPU in the AT91SAM9G45.

`-mthumb`
: The arm-eabi-gcc compiler will compile C and C++ files into the Thumb instruction set when this option is used. The best way to build eCos in Thumb mode is to enable the configuration option `CYGHWR_THUMB`.

`-mthumb-interwork`
: This option allows programs to be created that mix ARM and Thumb instruction sets. Without this option, some memory can be saved. This option should be used if -mthumb is used. The best way to build eCos with Thumb interworking is to enable the configuration option `CYGBLD_ARM_ENABLE_THUMB_INTERWORK`.

Onboard NAND

The HAL port includes a low-level driver to access the on-board Micron MT29F2G08ABD NAND flash memory chip. To enable the driver, activate the CDL option `CYGPKG_HAL_SAM9G45EK_NAND` and ensure that the `CYGPKG_DEVS_NAND_MICRON_MT29F` package is present in your eCos configuration.

`CYGHWR_HAL_SAM9G45EK_NAND_USE_STATUS_LINE`
: If set, this option configures the driver to wait for NAND operations to complete by waiting for the chip to deassert its Busy line. This is the default behaviour and is recommended, but may be disabled if you need to use the line (PIO C8) for some other purpose. (If disabled, the memory controller is configured to stall NAND accesses until they complete, which will interfere with multi-threading.)

`CYGNUM_HAL_SAM9G45EK_NAND_POLL_INTERVAL`
: The number of microseconds delay in the polling loops which wait for NAND operations to complete.

Partitioning the NAND chip

The NAND chip must be partitioned before it can become available to applications.

A CDL script which allows the chip to be manually partitioned is provided (see `CYGSEM_DEVS_NAND_SAM9G45EK_PARTITION_MANUAL_CONFIG`); if you choose to use this, the relevant data structures will automatically be set up for you when the device is initialised. By default, the manual config CDL script sets up a single partition (number 0) encompassing almost the entire device. The first block of the device contains AT91Bootstrap and the second and subsequent blocks contain RedBoot, or a ROM application. The first partition should therefore start above this. The default is set to block 8, leaving 1MiB at the base of NAND for bootstrapping.

It is possible to configure the partitions in some other way, should it be appropriate for your setup, for example to read a Linux-style partition table from the chip. To do so you will have to add appropriate code to `sam9g45ek_nand.c`.

Name

JTAG support — Usage

Use of JTAG for debugging

JTAG can be used to single-step and debug loaded applications, including RedBoot. Only JTAG configuration applications should be debugged using JTAG, RAM applications assume the presence of RedBoot and ROM applications reset the CPU clock, which may cause the debugger to disconnect.

Ronetix PEEDI notes

On the Ronetix PEEDI, the `peedi.at91sam9g45ek.cfg` file should be used to setup and configure the hardware to an appropriate state to load programs. This includes setting up the SDRAM controller.

The `peedi.at91sam9g45ek.cfg` file also contains an option to define whether hardware or software breakpoints are used by default, using the `CORE0_BREAKMODE` directive in the `[PLATFORM_ARM]` section. Edit this file if you wish to use hardware break points, and remember to restart the PEEDI to make the changes take effect.

On the PEEDI, debugging can be performed either via the telnet interface or using **arm-eabi-gdb** and the GDB interface. In the case of the latter, **arm-eabi-gdb** needs to connect to TCP port 2000 on the PEEDI's IP address. For example:

```
(gdb) target remote 111.222.333.444:2000
```

By default when the PEEDI is powered up, the target will always run the initialization section of the `peedi.at91sam9g45ek.cfg` file (which configures the SDRAM among other things), and halts the target. This behavior is repeated with the **reset** command.

If the board is reset (either with the '**reset**', or by pressing the reset button) and the '**go**' command is then given, then the board will boot as normal. If a second-level bootstrap and ROM RedBoot is resident in NANDFlash, it will be run.

Consult the PEEDI documentation for information on other features.

Running JTAG applications

Applications configured for JTAG startup can be run directly under JTAG. Once loaded and running via JTAG, HAL diagnostic output will appear by default on the serial debug port. USART 1 can be chosen instead by setting the `CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL` configuration option in the platform HAL to channel 1.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the AT91SAM9G45-EKES hardware, and should be read in conjunction with that specification. The AT91SAM9G45-EK platform HAL package complements the ARM architectural HAL, the ARM9 variant HAL and the SAM9 processor HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset, the HAL will initialize or reinitialize most of the on-chip peripherals. There is an exception for RAM startup applications which depend on a ROM monitor for certain services.

For ROM startup, the HAL will perform additional initialization. This is all done in the `PLATFORM_SETUP1` macro in the assembler header file `hal_platform_setup.h`.

Memory Map

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

SDRAM

This is located at address 0x20000000 of the physical memory space. The HAL configures the MMU to retain the SDRAM at virtual address 0x20000000, but in order to assign hardware exception vectors vectors at address 0x00000000, the HAL also uses the MMU to create a clone of this memory at virtual address 0x00000000. The same memory is also accessible uncached and unbuffered at virtual location 0x30000000 for use by devices. The first 32 bytes are used for hardware exception vectors. The next 32 bytes are used for the VSR table and the next 256 bytes are normally used for the eCos virtual vectors, allowing RAM-based applications to use services provided by the ROM monitor. Memory is required for the MMU tables, and must be aligned on a 16Kbyte boundary. These therefore occupy memory from 0x4000 to 0x8000. For ROM startup, all remaining SDRAM is available. For RAM startup, available RAM starts at virtual location 0x20040000, with the bottom 256kB reserved for use by RedBoot.

On-chip SRAM

This is located at address 0x00300000 of the physical memory space. However the HAL uses the MMU to relocate this to virtual address 0x70000000. The same memory is also accessible uncached and unbuffered at virtual location 0x70100000 for use by devices. At present this memory is unused by eCos and is available for application use.

On-chip ROM

This is located at address 0x00400000 of the physical memory space. However the HAL uses the MMU to relocate this to virtual address 0x71000000. The same memory is also accessible uncached and unbuffered at virtual location 0x70300000.

NAND

This is located at address 0x40000000 of the physical memory space. The HAL maps this uncached and unbuffered to 0x40000000 in the virtual address space. This mapping is only enabled if the NAND driver is enabled, otherwise it will generate an address fault.

USB

The USB device registers are located at addresses 0x00600000. 0x00700000, 0x00800000 of the physical memory space. However the HAL uses the MMU to relocate this to virtual addresses 0x71000000, 0x71100000 and 0x71200000 respectively. Memory accessed at these addresses is uncached and unbuffered. There is no cached variant.

SPI dataflash

SPI Dataflash media can only be accessed with the Flash API. For the purposes of this API a placeholder address range has been allocated as if the Flash is present at this address. The base of this address range is 0x50000000 This reserved range is not real memory and any attempt to access it directly by the processor other than via the Flash API will result in a memory address exception.

RedBoot manages the SPI dataflash using its FIS subsystem and stores system configuration data in the top 5 blocks of the device.

On-chip Peripheral Registers

These are located at address 0xFF000000 in the physical memory space. When the MMU is enabled, it sets up a direct, uncached, unbuffered mapping so that these registers remain accessible at their physical locations.

Off-chip Peripherals

eCos uses the SDRAM, ethernet PHY, MCI, NAND flash and SPI dataflash facilities on the AT91SAM9G45-EKES board. eCos does not currently make any use of any other off-chip peripherals present on this board.

Advanced Interrupt Controller

This port has been designed to exploit benefits of the Advanced Interrupt Controller of the AT91SAM9G45, using the facilities of the SAM9 processor HAL. Consult the documentation in that package for details.

SPI Dataflash

eCos supports SPI access to Dataflash on the AT91SAM9G45-EKES board. The device is typically used to contain flash configuration data.

Accesses to Dataflash are performed via the Flash API, using 0x50000000 as the nominal address of the device, although it does not truly exist in the processor address space.

Since Dataflash is not directly addressable, access from RedBoot is only possible using **fis** command operations.

Real-time characterization

The tm_basic kernel test gives statistics gathered about the real-time characterization and performance of the kernel. The sample output is shown here for information. The test was built in ARM mode, which provided better performance than Thumb mode.

Example 262.1. sam9g45ek Real-time characterization

```
INFO:<code from 0x20040040 -> 0x2004bad4, CRC eb0a>
            Startup, main stack : stack used   387 size  3920
             Startup :  Interrupt stack used   524 size  4096
             Startup : Idlethread stack used    88 size  2048

eCos Kernel Timings
Notes: all times are in microseconds (.000001) unless otherwise stated

Reading the hardware clock takes 0 'ticks' overhead
... this value will be factored out of all other measurements
Clock interrupt took    2.95 microseconds (24 raw clock ticks)

Testing parameters:
   Clock samples:            32
   Threads:                  64
   Thread switches:         128
   Mutexes:                  32
   Mailboxes:                32
   Semaphores:               32
   Scheduler operations:    128
```

```
 Counters:                 32
 Flags:                    32
 Alarms:                   32


                               Confidence
   Ave     Min     Max     Var  Ave  Min  Function
======  ======  ======  ====== ========== ========
  3.26    2.40    5.16    0.38   64%  18% Create thread
  0.44    0.36    1.56    0.08   98%  50% Yield thread [all suspended]
  0.62    0.48    1.32    0.08   59%  21% Suspend [suspended] thread
  0.60    0.48    1.20    0.09   42%  35% Resume thread
  0.85    0.72    2.40    0.08   60%  25% Set priority
  0.36    0.24    0.60    0.04   73%  12% Get priority
  1.55    1.32    4.08    0.16   81%  59% Kill [suspended] thread
  0.42    0.36    0.72    0.06   98%  50% Yield [no other] thread
  0.86    0.72    1.56    0.08   48%  25% Resume [suspended low prio] thread
  0.59    0.48    0.84    0.08   42%  35% Resume [runnable low prio] thread
  0.85    0.72    1.68    0.08   51%  28% Suspend [runnable] thread
  0.43    0.36    0.84    0.07   50%  48% Yield [only low prio] thread
  0.60    0.48    1.08    0.08   48%  32% Suspend [runnable->not runnable]
  1.37    1.20    2.40    0.08   89%   3% Kill [runnable] thread
  1.44    1.20    3.36    0.13   90%  48% Destroy [dead] thread
  2.00    1.80    3.00    0.09   84%   4% Destroy [runnable] thread
  2.97    2.64    5.76    0.24   79%  68% Resume [high priority] thread
  0.99    0.96    2.04    0.04   85%  85% Thread switch

  0.14    0.12    0.60    0.03   85%  85% Scheduler lock
  0.36    0.36    0.60    0.00   99%  99% Scheduler unlock [0 threads]
  0.36    0.36    0.48    0.00   99%  99% Scheduler unlock [1 suspended]
  0.37    0.36    0.84    0.02   94%  94% Scheduler unlock [many suspended]
  0.36    0.36    0.60    0.00   99%  99% Scheduler unlock [many low prio]

  0.27    0.24    1.20    0.06   96%  96% Init mutex
  0.63    0.48    1.68    0.10   75%  21% Lock [unlocked] mutex
  0.67    0.48    2.16    0.14   65%  56% Unlock [locked] mutex
  0.57    0.36    1.68    0.11   87%   6% Trylock [unlocked] mutex
  0.50    0.36    0.84    0.07   53%  18% Trylock [locked] mutex
  0.19    0.12    0.48    0.07   96%  50% Destroy mutex
  2.21    2.16    3.12    0.08   96%  78% Unlock/Lock mutex

  0.36    0.24    1.08    0.07   59%  28% Create mbox
  0.30    0.24    0.60    0.08   90%  62% Peek [empty] mbox
  0.74    0.60    2.04    0.09   65%  25% Put [first] mbox
  0.31    0.24    0.60    0.08   84%  62% Peek [1 msg] mbox
  0.72    0.60    1.08    0.05   68%  21% Put [second] mbox
  0.30    0.24    0.60    0.08   87%  68% Peek [2 msgs] mbox
  0.74    0.60    1.80    0.10   50%  31% Get [first] mbox
  0.74    0.60    1.20    0.08   50%  21% Get [second] mbox
  0.67    0.48    1.68    0.10   87%   6% Tryput [first] mbox
  0.69    0.60    1.32    0.08   53%  40% Peek item [non-empty] mbox
  0.72    0.60    1.44    0.06   65%  25% Tryget [non-empty] mbox
  0.58    0.48    0.96    0.08   43%  40% Peek item [empty] mbox
  0.57    0.48    1.08    0.10   81%  56% Tryget [empty] mbox
  0.30    0.24    0.60    0.08   84%  68% Waiting to get mbox
  0.28    0.24    0.48    0.06   75%  75% Waiting to put mbox
  0.40    0.24    1.08    0.08   87%   9% Delete mbox
  1.56    1.44    2.16    0.04   78%  18% Put/Get mbox

  0.16    0.12    0.48    0.06   78%  78% Init semaphore
  0.44    0.36    0.96    0.08   96%  50% Post [0] semaphore
  0.51    0.48    0.96    0.05   81%  81% Wait [1] semaphore
  0.43    0.36    0.96    0.08   96%  56% Trywait [0] semaphore
  0.41    0.36    0.60    0.06   62%  62% Trywait [1] semaphore
  0.20    0.12    0.48    0.08   87%  53% Peek semaphore
  0.17    0.12    0.60    0.07   96%  65% Destroy semaphore
  1.48    1.44    2.40    0.07   87%  87% Post/Wait semaphore
```

```
    0.37    0.24    1.44    0.09   59%  28% Create counter
    0.28    0.12    0.60    0.10   65%  21% Get counter value
    0.22    0.12    0.48    0.07   53%  34% Set counter value
    0.60    0.48    0.96    0.06   56%  25% Tick counter
    0.28    0.12    0.72    0.08   65%   9% Delete counter

    0.21    0.12    0.72    0.08   43%  46% Init flag
    0.51    0.36    1.44    0.09   75%  18% Destroy flag
    0.45    0.36    1.32    0.09   90%  50% Mask bits in flag
    0.50    0.36    1.20    0.06   71%  15% Set bits in flag [no waiters]
    0.62    0.48    1.68    0.07   87%   9% Wait for flag [AND]
    0.59    0.48    1.08    0.05   71%  25% Wait for flag [OR]
    0.61    0.48    0.96    0.02   93%   3% Wait for flag [AND/CLR]
    0.57    0.48    0.96    0.06   59%  37% Wait for flag [OR/CLR]
    0.12    0.00    0.12    0.01   96%   3% Peek on flag

    0.57    0.48    1.56    0.11   87%  56% Create alarm
    0.97    0.72    3.00    0.20   68%  50% Initialize alarm
    0.53    0.48    0.84    0.07   71%  71% Disable alarm
    0.87    0.72    2.04    0.12   62%  78% Enable alarm
    0.61    0.48    1.32    0.08   53%  28% Delete alarm
    0.57    0.48    1.44    0.08   53%  43% Tick counter [1 alarm]
    2.77    2.76    3.12    0.02   96%  96% Tick counter [many alarms]
    0.92    0.84    1.44    0.08   96%  50% Tick & fire counter [1 alarm]
   14.77   14.76   15.12    0.02   96%  96% Tick & fire counters [>1 together]
    3.16    3.12    3.72    0.06   78%  78% Tick & fire counters [>1 separately]
    2.53    2.52    3.48    0.02   96%  96% Alarm latency [0 threads]
    2.84    2.52    3.84    0.20   61%  34% Alarm latency [2 threads]
    4.02    3.36    5.04    0.31   65%  16% Alarm latency [many threads]
    3.87    3.84    6.36    0.05   97%  97% Alarm -> thread resume latency

    0.80    0.72    1.56    0.00            Clock/interrupt latency

    0.93    0.60    2.28    0.00            Clock DSR latency

    2      0     260  (main stack:  1364)  Thread stack used (1360 total)
           All done, main stack : stack used  1364 size  3920
            All done :  Interrupt stack used   664 size  4096
            All done : Idlethread stack used   232 size  2048

Timing complete - 30000 ms total

PASS:<Basic timing OK>
EXIT:<done>
```

Other Issues

The AT91SAM9G45-EKES platform HAL does not affect the implementation of other parts of the eCos HAL specification. The SAM9 processor HAL, ARM9 variant HAL, and the ARM architectural HAL documentation should be consulted for further details.

Chapter 263. ARM Versatile 926EJ-S Board Support

Name

eCos Support for the Versatile 926EJ-S Board — Overview

Description

This document covers the ARM Versatile Platform Baseboard for the ARM926EJ-S development chip, hereafter referred to as the VPB926EJS. The VPB926EJS contains the ARM926EJ-S processor, 128Mb of SDRAM, 64MB of Intel Strataflash memory, 2Mb of static RAM, a SMSC LAN91C111 Ethernet MAC, and external connections for the three on-chip and one off-chip serial channels, ethernet and the various other peripherals supported by the ARM926EJ-S.

For typical eCos development, a RedBoot image is programmed into the flash memory, and the board will boot this image from reset. RedBoot provides gdb stub functionality so it is then possible to download and debug stand-alone and eCos applications via the gdb debugger. This can happen over either a serial line or over ethernet.

Supported Hardware

The flash memory consists of 256 blocks of 256k bytes each. In a typical setup, the first flash block is used for the ROMRAM RedBoot image. The topmost block is used to manage the flash and hold RedBoot **fconfig** values. The remaining 254 blocks between 0x34040000 and 0x37FBFFFF can be used by application code.

There is a serial driver `CYGPKG_IO_SERIAL_ARM_PL011` which supports the ARM PL011 PrimeCell UARTs used by the VPB926EJS. The `CYGPKG_IO_SERIAL_ARM_VPB926EJS` package provides customization of this generic driver to the VPB926EJS hardware. These devices can be used by RedBoot for communication with the host. If any of these devices is needed by the application, either directly or via the serial driver, then it cannot also be used for RedBoot communication. Another communication channel such as ethernet should be used instead. The serial driver packages are loaded automatically when configuring for the VPB926EJS target.

There is an ethernet driver `CYGPKG_DEVS_ETH_SMSC_LAN91CXX` for the SMSC LAN91C111 ethernet device. A second package `CYGPKG_DEVS_ETH_ARM_VPB926EJS` is responsible for configuring this generic driver to the VPB926EJS hardware. These drivers are also loaded automatically when configuring for the VPB926EJS target.

eCos manages the on-chip interrupt controller. Timer 0 is used to implement the eCos system clock and the microsecond delay function. Other on-chip devices (Caches, UARTs, MPMC, SSMC, I²C etc.) are initialized only as far as is necessary for eCos to run. Other devices (PCI, SSP, SCI, GPIO etc.) are not touched.

Tools

The VPB926EJS port is intended to work with GNU tools configured for an arm-eabi target. The original port was undertaken using arm-elf-gcc version 3.2.1, arm-elf-gdb version 5.3, and binutils version 2.13.1.

Name

Setup — Preparing the VPB926EJS board for eCos Development

Overview

In a typical development environment, the VPB926EJS board boots from flash into the RedBoot ROM monitor. eCos applications are configured for RAM startup and then downloaded and run on the board via the debugger **arm-eabi-gdb**. Preparing the board therefore usually involves programming a suitable RedBoot image into flash memory.

The following RedBoot configurations are supported:

Configuration	Description	Use	File
RAM	RedBoot running from RAM	`redboot_RAM.ecm`	`redboot_RAM.bin`
ROM	RedBoot running from flash ROM	`redboot_ROM.ecm`	`redboot_ROM.bin`
ROMRAM	RedBoot running from RAM, but contained in the board's flash boot sector	`redboot_ROMRAM.ecm`	`redboot_ROMRAM.bin`
SRAM	RedBoot running from static RAM	`redboot_SRAM.ecm`	`redboot_SRAM.bin`

For serial communications, all versions run with 8 bits, no parity, and 1 stop bit at 38400 baud. RedBoot also supports ethernet communication and flash management.

Initial Installation

Flash Installation

Installing RedBoot is a matter of downloading a new binary image and overwriting the existing Boot monitor ROM image. This is a two stage process, you must first download an SRAM-resident version of RedBoot and then use that to download the ROM image to be programmed into the flash memory.

The VPB926EJS boards are shipped from ARM with a version of ARM's boot monitor installed. Unfortunately this boot monitor only works in conjunction with ARM's tools, which are not supplied with the board. Hence the only viable approach is to install RedBoot via the JTAG interface. The following directions are necessarily somewhat general since the specifics depend on the exact JTAG device available, and the software used to drive it.

Connect the JTAG device to the JTAG connector on the Versatile board and check that the device is functioning correctly. Using 32 bit memory writes, initialize the static memory controller so that the SRAM and flash are accessible. The following assignments should be made:

```
*(long *)0x10100034 = 0x00303021;
*(long *)0x10100054 = 0x00303021;
*(long *)0x10100074 = 0x00303021;
*(long *)0x10100094 = 0x00303021;
```

Now load the SRAM redboot binary image from the file `redboot_SRAM.bin` into the base of SRAM at 0x38000000. Exactly how you do this depends on the JTAG driver software.

Start RedBoot by executing from location 0x38000040, which should result in RedBoot starting up.

```
+... waiting for BOOTP information
Ethernet eth0: MAC address 00:02:f7:00:0b:34
IP: 10.0.0.208/255.255.255.0, Gateway: 10.0.0.1
Default server: 10.0.0.201
```

```
RedBoot(tm) bootstrap and debug environment [SRAM]
Non-certified release, version UNKNOWN - built 16:03:09, May  5 2004

Platform: ARM Versatile VPB926EJS (ARM9)
Copyright (C) 2000, 2001, 2002, Free Software Foundation, Inc.
Copyright (C) 2003, 2004, eCosCentric Limited

RAM: 0x00000000-0x02000000, [0x00013b20-0x01fbd000] available
FLASH: 0x34000000 - 0x38000000, 256 blocks of 0x00040000 bytes each.
RedBoot>
```

Now the ROM image can be downloaded using the following RedBoot command:

```
RedBoot> load -r -b %{FREEMEMLO} -m ymodem
```

Use HyperTerminal's Ymodem support to send the file `redboot_ROMRAM.bin`. This should result in something like the following output:

```
Raw file loaded 0x0002f800-0x0004be6b, assumed entry at 0x0002f800
xyzModem - CRC mode, 911(SOH)/0(STX)/0(CAN) packets, 4 retries
RedBoot>
```

Once the file has been uploaded, you can check that it has been transferred correctly using the **cksum** command. On the host (Linux or Cygwin) run the **cksum** program on the binary file:

```
$ cksum redboot_ROMRAM.bin
140216855 116332 redboot_ROMRAM.bin
```

In RedBoot, run the **cksum** command on the data that has just been loaded:

```
RedBoot> cksum -b %{FREEMEMLO} -l 116332
POSIX cksum = 140216855 116332 (0x085b8a17 0x0001c66c)
```

The second number in the output of the host **cksum** program is the file size, which should be used as the argument to the `-l` option in the RedBoot **cksum** command. The first numbers in each instance are the checksums, which should be equal.

If the program has downloaded successfully, then it can be programmed into the flash using the following commands:

```
RedBoot> fis init
About to initialize [format] FLASH image system - continue (y/n)?y
*** Initialize FLASH Image System
... Unlock from 0x37fc0000-0x38000000: .
... Erase from 0x37fc0000-0x38000000: .
... Program from 0x03fc0000-0x04000000 at 0x37fc0000: .
... Lock from 0x37fc0000-0x38000000: .
RedBoot> fis create -b %{FREEMEMLO} RedBoot
An image named 'RedBoot' exists - continue (y/n)?y
... Unlock from 0x34000000-0x34040000: .
... Erase from 0x34000000-0x34040000: .
... Program from 0x0002f800-0x0006f800 at 0x34000000: .
... Lock from 0x34000000-0x34040000: .
... Unlock from 0x37fc0000-0x38000000: .
... Erase from 0x37fc0000-0x38000000: .
... Program from 0x03fc0000-0x04000000 at 0x37fc0000: .
... Lock from 0x37fc0000-0x38000000: .
RedBoot>
```

The VPB926EJS board may now be reset either by cycling the power, pressing the reset switch, or with the **reset** command. It should then display the startup screen for the ROMRAM version of RedBoot.

Rebuilding RedBoot

Should it prove necessary to rebuild a RedBoot binary, this is done most conveniently at the command line. The steps needed to rebuild the the ROMRAM version of RedBoot for the VPB926EJS are:

```
$ mkdir redboot_vpb926ejs_romram
$ cd redboot_vpb926ejs_romram
$ ecosconfig new vpb926ejs redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/arm/arm9/vpb926ejs/current/misc/redboot_ROMRAM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

To rebuild the SRAM version of RedBoot:

```
$ mkdir redboot_vpb926ejs_sram
$ cd redboot_vpb926ejs_sram
$ ecosconfig new vpb926ejs redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/arm/arm9/vpb926ejs/current/misc/redboot_SRAM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

At the end of the build the `install/bin` subdirectory should contain the file `redboot.bin`. This is the case for both the above builds, take care not to mix the two files up, since programming the SRAM RedBoot into the ROM will render the board unbootable.

Name

Configuration — Platform-specific Configuration Options

Overview

The VPB926EJS platform HAL package is loaded automatically when eCos is configured for a `vpb926ejs` target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The VPB926EJS platform HAL package supports four separate startup types:

RAM
: This is the startup type which is normally used during application development. The board has RedBoot programmed into flash and boots into that initially. arm-eabi-gdb is then used to load a RAM startup application into memory and debug it. It is assumed that the hardware has already been initialized by RedBoot. By default the application will use the eCos virtual vectors mechanism to obtain certain services from RedBoot, including diagnostic output.

ROM
: This startup type can be used for finished applications which will be programmed into flash at physical address 0x34000000. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization.

ROM-RAM
: This startup type can be used for finished applications which will be programmed into flash at physical location 0x34000000. However, when it starts up, the application will first copy itself to RAM at 0x00000000 and then run from there. RAM is generally faster than flash memory, so the program will run more quickly than a ROM-startup application. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization.

SRAM
: This startup type exists only to support the installation of RedBoot via the JTAG interface. Functionally, it is equivalent to the ROM startup type except that the executable image is loaded into the static RAM at 0x38000000 rather than the flash ROM. It is of little use for other applications.

RedBoot and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building RedBoot.

If the application is required to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for RAM startup, disabled otherwise. It can be manually disabled for RAM startup, making the application self-contained, as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then the serial port will be claimed for HAL diagnostics.

Flash Driver

The VPB926EJS board contains two 32Mb 28F256K3 Intel StrataFlash flash devices, giving 64MB of flash in total. The `CYGPKG_DEVS_FLASH_STRATA_V2` package contains all the code necessary to support these parts and the VPB926EJS platform HAL package contains definitions that customize the driver to the VPB926EJS board.

Ethernet Driver

The VPB926EJS board contains a SMSC LAN91C111 ethernet controller. The `CYGPKG_DEVS_ETH_SMSC_LAN91CXX` package contains all the code necessary to support this device and the `CYGPKG_DEVS_ETH_ARM_VPB926EJS` package contains definitions that customize the driver to the VPB926EJS board.

System Clock

By default, the system clock interrupts once every 10ms, corresponding to a 100Hz clock. This can be changed by the configuration option `CYGNUM_HAL_RTC_DENOMINATOR` which corresponds to the clock frequency. Other clock-related settings are recalculated automatically if the denominator is changed.

Compiler Flags

The platform HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos. There is just one flag specific to this port:

`-mcpu=arm9`
: The arm-eabi-gcc compiler supports many variants of the ARM architecture. A `-m` option should be used to select the specific variant in use, and with current tools `-mcpu=arm9` is the correct option for the ARM926E CPU.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the VPB926EJS hardware, and should be read in conjunction with that specification. The VPB926EJS platform HAL package complements the ARM architectural HAL and the ARM9 variant HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset the HAL will initialize or reinitialize the on-chip peripherals that are used by eCos. There is an exception for RAM startup applications which depend on a ROM monitor for certain services.

For ROM, ROMRAM or SRAM startup, the HAL will perform additional initialization, setting up the external SDRAM and programming the various internal registers. This is all done in the `PLATFORM_SETUP1` macro in the assembler header file `hal_platform_setup.h`.

Linker Scripts and Memory Maps

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

Flash

This is located at address 0x34000000 of the physical memory space. The HAL uses the MMU to map it cached and buffered at the same address.

SDRAM

There are two blocks of SDRAM in the system. One block of 64Mb is present at physical address 0x00000000 and is echoed at 0x04000000. The second block is present at physical address 0x08000000. The HAL uses the MMU to map the first block to virtual address 0x00000000 and the second to virtual address 0x04000000, forming a single contiguous 128Mb block of SDRAM starting at 0x00000000. The same memory is also accessible uncached and unbuffered at virtual location 0x70000000 for use by devices. The first 32 bytes are used for hardware exception vectors. The next 32 bytes are used for the VSR table and the next 256 bytes are normally used for the eCos virtual vectors, allowing RAM-based applications to use services provided by the ROM monitor. For ROM/ROMRAM startup, all remaining SDRAM is available. For RAM startup, available RAM starts at virtual location 0x00040000, with the bottom 256kB reserved for use by RedBoot.

On-chip SRAM

This is located at address 0x38000000 of the physical memory space. The HAL uses the MMU to map this to the same virtual address, but cached and buffered. The same memory is also accessible uncached and unbuffered at virtual location 0x78000000 for use by devices. This memory is not used by eCos for any purpose except in the SRAM startup mode, when it contains the system image.

On-chip Peripheral Registers

These are located at address 0x10000000 in the physical memory space. When the MMU is enabled, it sets up a direct, uncached, unbuffered mapping so that these registers remain accessible at their physical locations.

Off-chip Peripherals

All off-chip peripherals are also visible in the 0x10000000 address space.

Other Issues

The VPB926EJS platform HAL does not affect the implementation of other parts of the eCos HAL specification. The ARM9 variant HAL, and the ARM architectural HAL documentation should be consulted for further details.

Chapter 264. Spectrum Digital OMAP-L137 Board Support

Name

eCos Support for the Spectrum Digital OMAP-L137 Evaluation Module — Overview

Description

This document covers the configuration and usage of eCos and RedBoot on the Spectrum Digital OMAP-L137 Evaluation Module. This board is fitted with an OMAP L137 processor, 64 MBytes of SDRAM, four megabytes of serial NOR flash attached to the SPI0 bus, 32K of EEPROM attached to the I2C0 bus, a DB9 serial connector for UART2, a KSZ8893MQL ethernet switch/phy, a jtag connector, and a number of other peripherals and expansion sockets. eCos support for the devices and peripherals on the board is described below.

For typical eCos development, a RedBoot image is programmed into the SPI NOR flash memory, and the board will load this image from reset. RedBoot provides gdb stub functionality so it is then possible to download and debug stand-alone and eCos applications via the gdb debugger. This can happen over either a serial line or over ethernet.

This documentation is expected to be read in conjunction with the OMAP L1xx processor HAL documentation and further device support and subsystems are described and documented there.

Supported Hardware

Bootstrap on the OMAP-L137 is complicated. The chip has two processors, a DSP and an ARM. The chip powers up with the DSP running a primary bootloader from on-chip memory, and with the ARM powered down. The primary bootloader checks the state of a number of GPIO pins to determine how to proceed. In a typical setup it will proceed to read in an AIS script from the serial NOR flash on the SPI0 bus and execute the script's instructions. The AIS script will load a secondary bootloader into on-chip memory and transfer control to that. The secondary bootloader will load a tertiary bootloader elsewhere in on-chip memory, activate the ARM processor, and put the DSP to sleep. The tertiary bootloader will initialize more of the hardware, in particular the external SDRAM, load RedBoot or another ROM startup eCos application, and transfer control. RedBoot can then be used to download and debug a RAM startup application, or in production systems it can load such an application from flash or other storage and start it.

There are 4 MBytes of SPI NOR flash, arranged in 64 64K blocks. In a typical setup the first four blocks are used to hold a boot image containing the AIS script for the primary bootloader, the secondary and tertiary bootloaders, and RedBoot. The topmost block is used to manage the flash and also holds RedBoot **fconfig** values. The remaining blocks can be used by application code.

On this board only uart2 is connected, and this is normally used by RedBoot for communication with the host. If the device is needed by the application, either directly or via the serial driver, then it cannot also be used for RedBoot communication and another communication channel such as ethernet should be used instead. The port also includes support for the ethernet, watchdog, and real-time clock devices and for the SPI and I²C buses. Details of this support can be found below.

In general, devices (Caches, GPIO, UARTs) are initialized only as far as is necessary for eCos to run. Other devices (RTC, SPI, I²C, etc.) are not touched unless the appropriate driver is loaded, although in some cases, the HAL boot sequence will set up the appropriate power control and pin multiplexing configuration.

Tools

The board support is intended to work with GNU tools configured for an arm-eabi target. The original port was undertaken using arm-eabi-gcc version 4.4.5, arm-eabi-gdb version 7.2, and binutils version 2.20.

Name

Setup — Preparing the SD-L137 board for eCos Development

Overview

In a typical development environment, the SD-L137 board boots from the SPI NOR and runs the RedBoot ROM monitor from SDRAM. eCos applications are configured for RAM startup and then downloaded and run on the board via the debugger **arm-eabi-gdb**. Preparing the board therefore usually involves programming a suitable RedBoot image into flash memory.

The following RedBoot configurations are supported:

Configuration	Description	Use	File
ROM	RedBoot loaded from SPI NOR flash to SDRAM	`redboot_ROM.ecm`	`redboot_ROM.img`

For serial communications, all versions run with 8 bits, no parity, and 1 stop bit at 115200 baud. RedBoot also supports ethernet communication and flash management.

Note that the use of the term ROM for the initial RedBoot configuration is a historical accident. RedBoot actually runs from SDRAM after being loaded there from NOR flash by the first three bootloaders. The use of ROM for this configuration is intended to indicate that it initializes the microprocessor and board peripherals, as opposed to the RAM configuration which assumes that this has already been done.

Initial Installation

Installation involves writing an image to the start of the serial flash. This image contains the AIS script interpreted by the primary bootloader, the secondary and tertiary bootloaders, and a RedBoot executable. A suitable image `redboot.img` is provided with the release or can be rebuilt using the instructions below.

There are two ways of writing the image. The first is to use a dedicated utility such as TI's Serial Boot and Flash Loading Utility for OMAP-L137. The second is to set up a JTAG emulator. This can then be used to run a RAM-resident RedBoot which will allow the flash to be programmed. The recommended JTAG emulator is the Ronetix PEEDI, but other JTAG emulators will involve a similar setup.

⚠ Caution

The L137 powers up with the ARM processor disabled, and the ARM does not start up until the primary bootloader has executed the AIS script and the secondary bootloader has run to completion. Until that time it will not be possible to attach a PEEDI or other ARM JTAG emulator. If a bogus image is programmed into the flash such that the AIS script or secondary bootloader are missing or corrupt, it will not be possible to recover the board via JTAG. Recovery should still be possible by using the TI utilities and uart2.

Programming RedBoot into SPI NOR flash using the TI Flash Utility

The following gives the steps needed to program RedBoot into the SPI NOR Flash using TI's Serial Boot and Flash Loading Utility for OMAP-L137. This tool and it's associated documentation can be found on the TI wiki http://processors.wiki.ti.com/index.php/Serial_Boot_and_Flash_Loading_Utility_for_OMAP-L137 page.

A RedBoot image file has been provided that combines the required L137 AIS script, DSP and ARM bootloaders, along with RedBoot itself. The `redboot_ROM.img` file can be found in the `loaders/sd_l137` subdirectory of your eCosPro installation.

The basic bootloader installation process is to set the board into a bootstrap update mode, use the TI utility to download and write the provided image to the flash, and then restore the board to normal operational mode.

1. Install the TI tools on your workstation.

2. Make a note of the boards current SW2 BOOT switch pin configuration and then configure SW2 to enable bootstrap update mode. The required pin configuration for your specific board revision can be found in the TI documentation. For example, on revision "C" and later boards: pin 7 ON, pin 2 OFF, pin 1 ON, pin 0 OFF, pin 3 OFF.

3. Connect a serial cable between the board's DB9 serial connector and the host PC. Run a serial terminal emulator (Putty, Hyperterm or minicom) on the host, connecting at 115200 baud 8N1 with no flow control. When the board is reset you should see "BOOTME" output on the serial line.

4. Use the TI **sfh_OMAP-L137** utility to install the `redboot_ROM.img` on the board. You will need to modify the example command **sfh_OMAP-L137 -p com2 -flash_noubl redboot_ROM.img** used in the example output below, substituting "com2" with the serial port corresponding to your setup, and by adding appropriate path prefixes to **sfh_OMAP-L137** and `redboot_ROM.img` corresponding to their installed location on your workstation.

   Once the flash utility starts running you will need to reset the board almost immediately as directed by the utility. Until you do this the benign message "(Serial Port): Read error! (The operation has timed out.)" will be output continously.

   You will also need to be patient as the initial "Loading section..." phase can take many minutes to complete with no obvious output or other signs of life.

```
C:>sfh_OMAP-L137 -p com2 -flash_noubl redboot_ROM.img
-----------------------------------------------------
   TI Serial Flasher Host Program for OMAP-L137
   (C) 2010, Texas Instruments, Inc.
   Ver. 1.67
-----------------------------------------------------



      [TYPE] Single boot image
[BOOT IMAGE] redboot_ROM.img
    [TARGET] OMAPL137_v2
    [DEVICE] SPI_MEM

Attempting to connect to device com4...
Press any key to end this program at any time.

(AIS Parse): Read magic word 0x41504954.
(AIS Parse): Waiting for BOOTME... (power on or reset target now)
(Serial Port): Read error! (The operation has timed out.)
(AIS Parse): BOOTME received!
(AIS Parse): Performing Start-Word Sync...
(AIS Parse): Performing Ping Opcode Sync...
(AIS Parse): Processing command 0: 0x58535901.
(AIS Parse): Performing Opcode Sync...
(AIS Parse): Loading section...
(AIS Parse): Loaded 14912-Byte section to address 0x80000000.
(AIS Parse): Processing command 1: 0x58535901.
(AIS Parse): Performing Opcode Sync...
(AIS Parse): Loading section...
(AIS Parse): Loaded 784-Byte section to address 0x80004240.
(AIS Parse): Processing command 2: 0x58535901.
(AIS Parse): Performing Opcode Sync...
(AIS Parse): Loading section...
(AIS Parse): Loaded 32-Byte section to address 0x80004550.
(AIS Parse): Processing command 3: 0x58535901.
(AIS Parse): Performing Opcode Sync...
(AIS Parse): Loading section...
(AIS Parse): Loaded 20-Byte section to address 0x80004590.
(AIS Parse): Processing command 4: 0x58535906.
(AIS Parse): Performing Opcode Sync...
```

```
(AIS Parse): Performing jump and close...
(AIS Parse): AIS complete. Jump to address 0x800034C0.
(AIS Parse): Waiting for DONE...
(AIS Parse): Boot completed successfully.

Waiting for SFT on the OMAP-L137...

Flashing application redboot_ROM.img (206288 bytes)

 100% [ ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦ ]
                  Image data transmitted over UART.

 100% [ ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦ ]
                   Application programming complete


Operation completed successfully.
```

5. Power off the board and restore the SW2 BOOT switch to it's normal "RUN" configuration. Power on the board again and you should see the following output on the serial port from RedBoot.

```
+**Warning** FLASH configuration checksum error or invalid key
Use 'fconfig -i' to [re]initialize database
Ethernet eth0: MAC address 00:0e:99:03:13:ee
IP: 10.1.1.147/255.255.255.0, Gateway: 10.1.1.241
Default server: 0.0.0.0

RedBoot(tm) bootstrap and debug environment [ROM]
eCosCentric certified release, version v3_1_10 - built 11:59:59, Jun 23 2011

Copyright (C) 2000-2009 Free Software Foundation, Inc.
Copyright (C) 2003-2011 eCosCentric Limited
RedBoot is free software, covered by the eCos license, derived from the
GNU General Public License. You are welcome to change it and/or distribute
copies of it under certain conditions. Under the license terms, RedBoot's
source code and full license terms must have been made available to you.
Redboot comes with ABSOLUTELY NO WARRANTY.

Platform: Spectrum Digital OMAP-L137 Evaluation Module (ARM9)
RAM: 0xc0000000-0xc4000000 [0xc003be80-0xc3fed000 available]
FLASH: 0x70000000-0x703fffff, 64 x 0x10000 blocks

RedBoot>
```

The flash configuration warning is expected. The ethernet MAC address will have come from the serial EEPROM on the I2C0 bus. The IP and gateway addresses will have been provided by a BOOTP (DHCP) server.

6. Run the following command to initialize RedBoot's flash file system and flash configuration:

```
RedBoot> fis init
About to initialize [format] FLASH image system - continue (y/n)? y
*** Initialize FLASH Image System
... Erase from 0x703f0000-0x703fffff: .
... Program from 0xc3ff0000-0xc4000000 to 0x703f0000: .
RedBoot> fis list
Name              FLASH addr  Mem addr    Length      Entry point
RedBoot           0x70000000  0x70000000  0x00040000  0x00000000
FIS directory     0x703F0000  0x703F0000  0x0000F000  0x00000000
RedBoot config    0x703FF000  0x703FF000  0x00001000  0x00000000
RedBoot>
```

If desired the fconfig settings can be initialized at this time using the **fconfig -i** command. Most of the settings relate to ethernet, for example the IP address that RedBoot should use.

7. The RedBoot installation is now complete. The board is now ready for development using **arm-eabi-gdb**, RedBoot's gdb stubs, and eCos applications configured for RAM startup.

Programming RedBoot into NOR flash using the PEEDI

The following gives the steps needed to program RedBoot into the SPI NOR Flash using the PEEDI. The basic process is to load and run a copy of RedBoot, then use that to initialize the flash, download the full image including the AIS script and bootloaders, and write this image to the flash.

1. Set up the PEEDI as described in the Ronetix documentation. The `peedi.sd_l137.cfg` file in the platform HAL's `misc` subdirectory contains the required hardware initialization support. Other parts of this file will need to be edited, for example the license key details.

2. Connect a serial cable between the boards DB9 serial connector and the host PC. Run a serial terminal emulator (Hyperterm or minicom) on the host, connecting at 115200 baud 8N1 with no flow control.

3. Use **arm-eabi-gdb** to run the `redboot.elf` executable from the `loaders/sd_l137` subdirectory of your eCosPro installation. Substitute the appropriate TCP/IP address and port number corresponding to your PEEDI setup.

   ```
   $ arm-eabi-gdb --quiet <path>/redboot.elf
   (gdb) target remote peedi:9000
   Remote debugging using peedi:9000
   0xffff0000 in ?? ()
   (gdb) load
   Loading section .rom_vectors, size 0x40 lma 0xc0008000
   Loading section .text, size 0x1a078 lma 0xc0008040
   Loading section .rodata, size 0x3fac lma 0xc00220b8
   Loading section .data, size 0xd2b4 lma 0xc0026064
   Start address 0xc0008040, load size 176920
   Transfer rate: 34 KB/sec, 13609 bytes/write.
   (gdb) continue
   Continuing.
   ```

   Address 0xffff0000 corresponds to the ARM reset vector. At this point the primary and secondary bootloaders have run and the processor would be about to start the tertiary bootloaders, but the PEEDI will have halted the processor and run its hardware initialization macro instead. RedBoot will start running after the **continue** command, and the following should be sent out of the serial line.

   ```
   +**Warning** FLASH configuration checksum error or invalid key
   Use 'fconfig -i' to [re]initialize database
   Ethernet eth0: MAC address 00:0e:99:03:13:ee
   IP: 10.1.1.147/255.255.255.0, Gateway: 10.1.1.241
   Default server: 0.0.0.0

   RedBoot(tm) bootstrap and debug environment [ROM]
   eCosCentric certified release, version v3_1_10 - built 11:59:59, Jun 23 2011

   Copyright (C) 2000-2009 Free Software Foundation, Inc.
   Copyright (C) 2003-2011 eCosCentric Limited
   RedBoot is free software, covered by the eCos license, derived from the
   GNU General Public License. You are welcome to change it and/or distribute
   copies of it under certain conditions. Under the license terms, RedBoot's
   source code and full license terms must have been made available to you.
   Redboot comes with ABSOLUTELY NO WARRANTY.

   Platform: Spectrum Digital OMAP-L137 Evaluation Module (ARM9)
   RAM: 0xc0000000-0xc4000000 [0xc003cbd8-0xc3fed000 available]
   FLASH: 0x70000000-0x703fffff, 64 x 0x10000 blocks
   RedBoot>
   ```

   The flash configuration warning is expected at this stage. The ethernet MAC address will have come from the serial EEPROM on the I2C0 bus. The IP and gateway addresses will have been provided by a BOOTP server.

4. Run the following command to initialize RedBoot's flash file system and flash configuration:

   ```
   RedBoot> fis init
   ```

```
About to initialize [format] FLASH image system - continue (y/n)? y
*** Initialize FLASH Image System
... Erase from 0x703f0000-0x703fffff: .
... Program from 0xc3ff0000-0xc4000000 to 0x703f0000: .
RedBoot> fis list
Name              FLASH addr  Mem addr    Length      Entry point
RedBoot           0x70000000  0x70000000  0x00040000  0x00000000
FIS directory     0x703F0000  0x703F0000  0x0000F000  0x00000000
RedBoot config    0x703FF000  0x703FF000  0x00001000  0x00000000
RedBoot>
```

If desired the fconfig settings can be initialized at this time using the **fconfig -i** command. Most of the settings relate to ethernet, for example the IP address that RedBoot should use.

5. Next the full RedBoot image should be loaded into RAM.

```
       RedBoot> load -r -m y -b %{freememlo}
       C
```

From the terminal emulator upload the `redboot.img` file from the `loaders/sd_l137` directory using Y-Modem protocol. When the upload is complete you should see something similar to the following output.

```
CRaw file loaded 0xc003cc00-0xc006ff27, assumed entry at 0xc003cc00
xyzModem - CRC mode, 1641(SOH)/0(STX)/0(CAN) packets, 3 retries
RedBoot>
```

6. Now program the image to flash:

```
RedBoot> fis create RedBoot
An image named 'RedBoot' exists - continue (y/n)? y
... Erase from 0x70000000-0x7003ffff: .....
... Program from 0xc003cc00-0xc006ff27 to 0x70000000: .....
... Erase from 0x703f0000-0x703fffff: .
... Program from 0xc39f0000-0xc3a00000 to 0x703f0000: .
RedBoot>
```

7. The RedBoot installation is now complete. Terminate the **arm-eabi-gdb** session by hitting ctrl-C and then running the **quit** command. Detach the PEEDI and power cycle the board. RedBoot should now start running after the primary, secondary and tertiary bootloaders and output a banner similar to the one above. The board is now ready for development using **arm-eabi-gdb**, RedBoot's gdb stubs, and eCos applications configured for RAM startup.

If it proves necessary to re-install RedBoot, this may be achieved by repeating the serial download and **fis create** parts of the above process. It is not necessary to reinitialize the FIS and fconfig.

Rebuilding RedBoot

Should it prove necessary to rebuild a RedBoot binary, this is done most conveniently at the command line. The steps needed to rebuild the ROM version of RedBoot for this board are:

```
$ mkdir redboot_sdl137_rom
$ cd redboot_sdl137_rom
$ ecosconfig new sd_l137 redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/arm/arm9/sd_l137/VERSION/misc/redboot_ROM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

At the end of the build the `install/bin` subdirectory should contain the files `redboot.elf` and `redboot.img`. `redboot.elf` can be executed on the board using **arm-eabi-gdb** and a JTAG emulator. `redboot.img` is an image containing the AIS script for the primary bootloader, the secondary and tertiary bootloaders, and RedBoot. It is this image which should be programmed into flash to install or update RedBoot.

Name

Configuration — Platform-specific Configuration Options

Overview

The SD-L137 platform HAL package is loaded automatically when eCos is configured for the `sd_l137` target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The platform HAL package supports two separate startup types:

RAM This is the startup type which is normally used during application development. The board has RedBoot programmed into flash and boots into that initially. arm-eabi-gdb is then used to load a RAM startup application into memory and debug it. It is assumed that the hardware has already been initialized by RedBoot. By default the application will use the eCos virtual vectors mechanism to obtain certain services from RedBoot, including diagnostic output.

ROM This startup type can be used for applications which will be programmed into flash, or which will be loaded and debugged via JTAG. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization.

Application binaries cannot just be programmed into flash. Instead it is necessary to incorporate them into a larger image file containing the AIS boot script for the primary bootloader as well as the secondary and tertiary bootloaders. The platform HAL's `misc` subdirectory contains a script `gensdl137aisimg.tcl` which can be used. Given an ELF executable `test` produced by the linker, the steps required are:

```
$ arm-eabi-objcopy -O binary test test.bin
$ tclsh <path>/misc/gensdl137aisimg.tcl test.bin test.img
```

The resulting image file can be programmed into flash instead of RedBoot. The build process for RedBoot will invoke these commands automatically to generate the `redboot.img` file.

RedBoot and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building RedBoot.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then the serial port will be claimed for HAL diagnostics.

Flash Driver

The board contains a 4 Mbyte serial NOR flash device. The `CYGPKG_DEVS_FLASH_SPI_M25PXX` package contains all the code necessary to support this part and the platform HAL package contains definitions that customize the driver to the board. This driver is not active until the generic Flash support package, `CYGPKG_IO_FLASH`, is included in the configuration. The driver only supports 64K block erase operations, not the smaller 4K sector erase operations.

This driver is capable of supporting the JFFS2 filesystem. However, note that the SPI interface means that this file system has reduced bandwidth and increased latency compared with other implementations. All that is required to enable the support is to include the filesystem (CYGPKG_FS_JFFS2) and any of its package dependencies (including CYGPKG_IO_FILEIO and CYGPKG_LINUX_COMPAT) together with the flash infrastructure (CYGPKG_IO_FLASH).

Ethernet Driver

The board uses the OMAP L137's internal EMAC ethernet device attached to an external Micrel KSZ8893 PHY. The CYGPKG_DEVS_ETH_ARM_OMAP package contains all the code necessary to support this device and the platform HAL package contains definitions that customize the driver to the board. This driver is not active until the generic Ethernet support package, CYGPKG_IO_ETH_DRIVERS, is included in the configuration. The ethernet MAC address is held in the last 256-byte page of the serial EEPROM attached to the I2C0 bus.

RTC Driver

The board uses the OMAP L137's internal RTC support. The CYGPKG_DEVICES_WALLCLOCK_ARM_OMAP_L1XX package contains all the code necessary to support this device. This driver is not active until the generic wallclock device support package, CYGPKG_IO_WALLCLOCK, is included in the configuration.

Watchdog Driver

The board uses the OMAP L137's internal watchdog support. The CYGPKG_DEVICES_WATCHDOG_ARM_OMAP_L1XX package contains all the code necessary to support this device. Within that package the CYGNUM_DEVS_WATCHDOG_ARM_OMAP_L1XX_DESIRED_TIMEOUT_MS configuration option controls the watchdog timeout, and by default will force a reset of the board upon timeout. This driver is not active until the generic watchdog device support package, CYGPKG_IO_WATCHDOG, is included in the configuration.

UART Serial Driver

The board uses the OMAP L137's internal UART serial support as described in the OMAP L1xx processor HAL documentation. Only uart2 has a connector and only the tx and rx lines are connected, so hardware flow control and modem support signals are not available. The uart is normally used by RedBoot for communication with the host. If the device is needed by the application, either directly or via the serial driver, then it cannot be used for RedBoot communication and another channel such as ethernet should be used instead.

Device driver support is through the CYGPKG_IO_SERIAL_GENERIC_16X5X generic driver package which is modified by the CYGPKG_IO_SERIAL_ARM_OMAP_L1XX driver package for the OMAP L1xx. The packages are loaded automatically when configuring for the sd-l137 target but the option CYGPKG_IO_SERIAL_DEVICES has to be enabled to instantiate the device. The default device name is /dev/ser2.

I2C Bus Driver

The OMAP L1XX processor HAL contains a driver for the I²C two wire interface. This driver is loaded automatically when configuring for this target. The platform HAL enables bus 0 by default but leaves bus 1 disabled since there are no attached devices. This can be changed in the configuration if devices are attached via an expansion socket using option CYGHWR_HAL_ARM_SD_L137_I2C1. The bus names are hal_omap_l1xx_i2c_bus0 and hal_omap_l1xx_i2c_bus1.

The platform HAL also instantiates three I²C device objects: hal_i2c_eeprom for the serial EEPROM; hal_i2c_codec for the audio codec; and hal_i2c_eth_switch for the ethernet switch/phy. These devices can be accessed via the generic I²C API. The last 256-byte page of the eeprom is used to hold the ethernet MAC address but the remaining pages are available for use by the application. The ethernet switch is accessed only during ethernet driver initialization. The codec is not used by any eCos code.

The bus and device objects do not have to be enabled explicitly. If they are not used by the application, directly or indirectly, then they will be removed by link-time garbage collection.

SPI Bus Driver

The SPI buses are supported via the OMAP SPI driver CYGPKG_DEVS_SPI_ARM_OMAP. On this board only bus 0 is enabled by default since bus 1 does not ordinarily have any attached devices, but this can be changed via the configuration option CYGHWR_HAL_ARM_SD_L137_SPI1. Additionally, if CYGHWR_HAL_ARM_SD_L137_SPI_MMCSD is set, that will ensure the appropriate bus it is configured to use is enabled (see below). The bus names are cyg_spi_omap_bus0 and cyg_spi_omap_bus1 respectively. The platform HAL also provides an SPI device object cyg_m25pxx_spi_device for the serial flash. Normally this device is used only the flash device driver, not directly by the application.

The bus and device objects do not have to be enabled explicitly. If they are not used by the application, directly or indirectly, then they will be removed by link-time garbage collection.

MMC/SD cards over SPI

The HAL can be configured to support an MMC/SD card socket connected by SPI. Ensure CYGHWR_HAL_ARM_SD_L137_SPI_MMCSD is enabled to include this support, which it is by default if the disk driver package (CYGPKG_IO_DISK) is added to the eCos configuration.

An MMC/SD socket via SPI is not a standard feature of the Spectrum Digital OMAP-L137 platform, however it is possible to add one using a daughterboard connected via an expansion bus. As a result of this there are a number of associated configuration options as it could be connected in a variety of ways:

CYGHWR_HAL_ARM_SD_L137_SPI_MMCSD_BUS

> This option selects which SPI bus number is connected to the socket. It has been observed that some MMC/SD cards do not behave correctly if a second SPI device is present on the same SPI bus, so it is not recommended to share the MMC/SD socket's SPI bus with any other device. Given the dataflash on the SPI0 bus, that would only leave SPI1, which is the default.

CYGHWR_HAL_ARM_SD_L137_SPI_MMCSD_CS

> This component allows configuration of the chip select line used for communicating with the SPI-connected card. With this component enabled, this chip select line will be managed using GPIO. Alternatively, with this component disabled, the default chip select 0 will be used.

CYGHWR_HAL_ARM_SD_L137_SPI_MMCSD_CS_PINMUX_REG

> This option configures the PINMUX register used for the chip select line used to communicate with the card. The SPI1 chip select 0 is multiplexed with a UART2 line, and since that is the UART used for HAL diagnostic output the default is instead to use GPIO3[10] (shared with the unused UART1_TXD). Its PINMUX function is set within PINMUX register 11, which is the default for this option. Consult the OMAP-L137 documentation for values for alternative pins.

CYGHWR_HAL_ARM_SD_L137_SPI_MMCSD_CS_PINMUX_FIELD

> This option configures the field within the PINMUX register used for the chip select line used to communicate with the card. The value of this field is the amount of left bitshifting required in the register to correspond to the correct pin. SPI1 chip select 0 is multiplexed with a UART2 line, and since that is the UART used for HAL diagnostic output the default is instead to use GPIO3[10] (shared with the unused UART1_TXD). Its PINMUX function is set within PINMUX register 11 at bit position 12, which is the default for this option. Consult the OMAP-L137 documentation for values for alternative pins.

CYGHWR_HAL_ARM_SD_L137_SPI_MMCSD_CS_GPIO_BANK

> This option configures the GPIO bank used for the chip select line used to communicate with the card. The SPI1 chip select 0 is multiplexed with a UART2 line, and since that is the UART used for HAL diagnostic output the default is instead to use GPIO3[10] (shared with the unused UART1_TXD), so bank 3 is the default for this option. Consult the OMAP-L137 documentation for values for alternative pins.

`CYGHWR_HAL_ARM_SD_L137_SPI_MMCSD_CS_GPIO_BIT`

> This option configures the GPIO bit used for the chip select line used to communicate with the card. The SPI1 chip select 0 is multiplexed with a UART2 line, and since that is the UART used for HAL diagnostic output the default is instead to use GPIO3[10] (shared with the unused UART1_TXD), so bit 10 is the default for this option. Consult the OMAP-L137 documentation for values for alternative pins.

If using the default chip select via GPIO3[10], then you must ensure the SEL_EXP2_UART1 pin is set high. If you are using the Spectrum Digital prototype daughterboard, you can do this on SW2 by setting switch 4 to the ON position.

Similarly, if using the SPI1 bus, and expansion connector 2, then you must ensure the SEL_SPI1_EXP2_B pin is set high. If you are using the Spectrum Digital prototype daughterboard and have connected your card socket to J4, you can set SEL_SPI1_EXP2_B high by setting switch 3 of SW2 to the ON position.

Compiler Flags

The platform HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos. There are just three flags specific to this port:

`-mcpu=arm9`

> The arm-eabi-gcc compiler supports many variants of the ARM architecture. A `-m` option should be used to select the specific variant in use, and with current tools `-mcpu=arm9` is the correct option for the ARM926EJ CPU in the OMAP L1xx.

`-mthumb`

> The arm-eabi-gcc compiler will compile C and C++ files into the Thumb instruction set when this option is used. The best way to build eCos in Thumb mode is to enable the configuration option `CYGHWR_THUMB`.

`-mthumb-interwork`

> This option allows programs to be created that mix ARM and Thumb instruction sets. Without this option, some memory can be saved. This option should be used if -mthumb is used. The best way to build eCos with Thumb interworking is to enable the configuration option `CYGBLD_ARM_ENABLE_THUMB_INTERWORK`.

Name

JTAG support — Usage

Use of JTAG for debugging

JTAG can be used to single-step and debug loaded applications, including RedBoot. Only ROM configuration applications should be debugged using JTAG, RAM applications assume the presence of RedBoot.

Ronetix PEEDI notes

On the Ronetix PEEDI, the `peedi.sd_l137.cfg` file should be used to setup and configure the hardware to an appropriate state to load programs. This includes setting up the PLLs and SDRAM controller.

The `peedi.sd_l137.cfg` file also contains an option to define whether hardware or software breakpoints are used by default, using the `CORE0_BREAKMODE` directive in the `[PLATFORM_ARM]` section. Edit this file if you wish to use hardware break points, and remember to restart the PEEDI to make the changes take effect.

On the PEEDI, debugging can be performed either via the telnet interface or using **arm-eabi-gdb** and the GDB interface. In the case of the latter, **arm-eabi-gdb** needs to connect to TCP port 9000 on the PEEDI's IP address. For example:

```
(gdb) target remote 111.222.333.444:9000
```

The target will always run the primary and secondary bootloaders because the ARM is not powered up until the secondary bootloader has activated the ARM and put the DSP to sleep. The PEEDI will then run the initialization section of the `peedi.sd_l137.cfg` file, and halt the target. This behaviour is repeated whenever the board is powercycled. If the PEEDI is given a '**go**' command then the board will continue booting as normal.

Consult the PEEDI documentation for information on other features.

Running ROM applications

Applications configured for ROM startup can be run directly under JTAG. Once loaded and running via JTAG, HAL diagnostic output will appear by default on the serial debug port.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the hardware, and should be read in conjunction with that specification. The platform HAL package complements the ARM architectural HAL, the ARM9 variant HAL and the OMAP L1xx processor HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset, the HAL will initialize or reinitialize most of the on-chip peripherals. There is an exception for RAM startup applications which depend on a ROM monitor for certain services.

For ROM startup, the HAL will perform additional initialization. This is all done in the `PLATFORM_SETUP1` macro in the assembler header file `hal_platform_setup.h`.

Linker Scripts and Memory Maps

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

SDRAM

This is located at address 0xC0000000 of the physical memory space. The HAL configures the MMU to retain the SDRAM at virtual address 0xC0000000, but in order to assign hardware exception vectors at address 0x00000000, the HAL also uses the MMU to create a clone of this memory at virtual address 0x00000000. The same memory is also accessible uncached and unbuffered at virtual location 0xD0000000 for use by devices. The first 32 bytes are used for hardware exception vectors. The next 32 bytes are used for the VSR table and the next 256 bytes are normally used for the eCos virtual vectors, allowing RAM-based applications to use services provided by the ROM monitor. Memory is required for the MMU tables, and must be aligned on a 16Kbyte boundary. These therefore occupy memory from 0x4000 to 0x8000. For ROM startup, all remaining SDRAM is available. For RAM startup, available RAM starts at virtual location 0xC0040000, with the bottom 256kB reserved for use by RedBoot.

On-chip SRAM

There are a number of on-chip SRAM areas. These are identity mapped unbuffered and uncached with their physical addresses. eCos does not use any of these areas so they are all available to the application.

SPI NOR Flash

SPI NOR flash media can only be accessed with the Flash API. For the purposes of this API a placeholder address range has been allocated as if the Flash is present at this address. The base of this address range is 0x70000000. This reserved range is not real memory and any attempt to access it directly by the processor other than via the Flash API will result in a memory address exception.

On-chip Peripheral Registers

These are located at various addresses in the physical memory space. When the MMU is enabled, it sets up a direct, uncached, unbuffered mapping so that these registers remain accessible at their physical locations.

Off-chip Peripherals

eCos uses the SDRAM, ethernet PHY, SPI flash, and I^{2}C EEPROM facilities on the board. eCos does not currently make any use of any other off-chip peripherals present on this board.

SPI NOR Flash

eCos supports SPI access to the NOR flash on the board. The device is typically used to contain RedBoot and flash configuration data.

Accesses to SPI flash are performed via the Flash API, using 0x70000000 or as the nominal address of the device, although it does not truly exist in the processor address space.

Since SPI flash is not directly addressable, access from RedBoot is only possible using **fis** command operations.

Other Issues

The platform HAL does not affect the implementation of other parts of the eCos HAL specification. The OMAP L1xx processor HAL, ARM9 variant HAL, and the ARM architectural HAL documentation should be consulted for further details.

Chapter 265. Logic Zoom Board Support

Name

eCos Support for the Logic Zoom Board — Overview

Description

This document covers the configuration and usage of eCos and RedBoot on the Logic Zoom eXperimenter board. The board comes in two versions, one fitted with an TI OMAP L138 processor and the other with a TI Sitara AM1808 processor. The boards and processors are functionally identical as far as eCos and this documentation are concerned. To simplify the text below the board is generally referred to as the Zoom and where the text doesn't differentiate between the two processors, all references to the OMAP-L138 should be taken to refer to either processor. In addition, all OMAP-L138 specific processor configuration and target names referred to below the should also be used for the Sitara AM1808 prcoessor. The Zoom board consists of a base board and a CPU module. The CPU module containing either the OMAP L138 or Sitara AM1808 processors. The base board contains the CPU module, 64Mbytes of SDRAM, 8Mbytes of serial NOR flash memory on SPI0, a LAN87810A PHY, a MMC/SD socket, external connections for one serial channel, ethernet, USB host/device, and the various other peripherals supported by the CPUs. eCos support for the devices and peripherals on the board and the CPU is described below.

For typical eCos development, a RedBoot image is programmed into the SPI NOR flash memory, and the board will load this image from reset. RedBoot provides gdb stub functionality so it is then possible to download and debug stand-alone and eCos applications via the gdb debugger. This can happen over either a serial line or over ethernet.

This documentation is expected to be read in conjunction with the OMAP L1xx processor HAL documentation and further device support and subsystems are described and documented there.

Supported Hardware

The SPI NOR flash consists of 128 blocks of 64Ki bytes each. In a typical setup, the first block is reserved for the second-level bootstrap, the User Boot Loader. The following two blocks are reserved for the use of the ROM RedBoot image. The topmost block is used to manage the flash and the next block down holds RedBoot **fconfig** values. The remaining blocks can be used by application code.

Serial support is through the `CYGPKG_IO_SERIAL_GENERIC_16X5X` generic driver package which is modified by the `CYGPKG_IO_SERIAL_ARM_OMAP_L1XX` driver package for the OMAP L1xx. These packages can support all the serial devices on the OMAP L138. However, this board only has UART2 connected to an external connector which this HAL indicates by implementing the `CYGINT_HAL_L1XX_UART2` interface. This serial channel is used by RedBoot for communication with the host. If this device is needed by the application, either directly or via the serial driver, then it cannot also be used for RedBoot communication. Another communication channel such as ethernet should be used instead. The serial driver package is loaded automatically when configuring for the Zoom-L138 target.

There is an ethernet driver `CYGPKG_DEVS_ETH_ARM_OMAP` for the on-chip ethernet device. The platform HAL package is responsible for configuring this generic driver to the hardware. This driver is also loaded automatically when configuring for the Zoom-l138 board.

There is a watchdog driver `CYGPKG_DEVICES_WATCHDOG_ARM_OMAP_L1XX`. This driver is also loaded automatically when configuring for the board.

There is a driver for the on-chip real-time clock (RTC) at `CYGPKG_DEVS_WALLCLOCK_ARM_OMAP_L1XX`. This driver is also loaded automatically when configuring for the target.

The OMAP L1XX processor HAL contains a driver for the MultiMedia Card Interface (MMC/SD). This driver is loaded automatically when configuring for this target and allows use of MMC and Secure Digital (SD) flash storage cards within eCos, exported as block devices. Further documentation on the driver may be found in the OMAP L1XX processor HAL documentation.

The platform HAL provides definitions to allow access to devices on the SPI bus. The HAL provides information to the more general OMAP SPI driver (`CYGPKG_DEVS_SPI_ARM_OMAP`) which in turn provides the underlying implementation for the SPI API layer in the `CYGPKG_IO_SPI` package. All these packages are automatically loaded when configuring for the board.

Furthermore, the platform HAL package contains support for the SPI NOR flash. The HAL support integrates with the CYGPKG_DEVS_FLASH_SPI_M25PXX package as well as the above SPI packages. That package is automatically loaded when configuring for the target. This driver is capable of supporting the JFFS2 filesystem.

In general, devices (Caches, GPIO, UARTs) are initialized only as far as is necessary for eCos to run. Other devices (RTC, SPI, MMC/SD etc.) are not touched unless the appropriate driver is loaded, although in some cases, the HAL boot sequence will set up the appropriate power control and pin multiplexing configuration.

Tools

The board support is intended to work with GNU tools configured for an arm-eabi target. The original port was undertaken using arm-eabi-gcc version 4.3.2, arm-eabi-gdb version 6.8, and binutils version 2.18.

Name

Setup — Preparing the Zoom board for eCos Development

Overview

In a typical development environment, the Zoom board boots from the SPI NOR and runs the RedBoot ROM monitor from SDRAM. eCos applications are configured for RAM startup and then downloaded and run on the board via the debugger **arm-eabi-gdb**. Preparing the board therefore usually involves programming a suitable RedBoot image into flash memory.

The following RedBoot configurations are supported:

Configuration	Description	Use	File
ROM	RedBoot loaded from SPI NOR flash to SDRAM	`redboot_ROM.ecm`	`redboot_ROM.bin`

For serial communications, all versions run with 8 bits, no parity, and 1 stop bit at 115200 baud. RedBoot also supports ethernet communication and flash management.

Note that the use of the term ROM for the initial RedBoot configuration is a historical accident. RedBoot actually runs from SDRAM after being loaded there from NOR flash by the User Boot Loader. The use of ROM for this configuration is intended to indicate that it initializes the microprocessor and board peripherals, as opposed to the RAM configuration which assumes that this has already been done.

Initial Installation

The Zoom board comes with U-Boot installed by default. The booting mechanism is that the on-chip firmware loads a small User Boot Loader from the start of NOR flash which then loads U-Boot from later in the flash. Our strategy is to leave the User Boot Loader in place and replace U-Boot with RedBoot.

To write RedBoot to the SPI NOR flash, there are two possibilities: either run a RAM-resident RedBoot using a JTAG emulator and use that to program RedBoot into the SPI NOR flash; or use a Serial Boot and Flash Utility for OMAP-L138.

The following section describes this process using the Ronetix PEEDI; other JTAG emulators will have similar steps.

Programming RedBoot into NOR flash using the PEEDI

The following gives the steps needed to program RedBoot into the SPI NOR Flash using the PEEDI. The basic process is to load and run a copy of RedBoot, then use that to initialize the flash, download a new copy of RedBoot and write that to the flash.

1. Set up the PEEDI as described in the Ronetix documentation. The `peedi.zoom.cfg` file should be used to setup and configure the hardware.
2. Connect a null-modem serial cable between the serial port of the board and a serial port on a convenient host. Run a terminal emulator (Hyperterm or minicom) at 115200 baud.
3. Copy `redboot_ROM.srec` to to a TFTP server that the PEEDI can access. Copy `redboot_ROM.img` to the machine running the terminal emulator.
4. Connect a telnet session to the PEEDI and issue the following command, substituting your own TFTP server address:

```
        zoom>> mem load tftp://10.0.1.1/redboot.srec srec
        ++ info: Loading image file:  tftp://10.0.1.1/redboot.srec
        ++ info: At absolute address: 0xC0008000
        loading at 0xC0008000
        loading at 0xC000C000
```

```
       loading at 0xC0010000
       loading at 0xC0014000
       loading at 0xC0018000
       loading at 0xC001C000

       Successfully loaded 88KB (90336 bytes) in 20.4s
       zoom>
```

5. Now issue the go command:

   ```
          zoom> go 0xC0008000
   ```

   You should see the following output on the Zoom board serial line:

   ```
   +M25PXX : Init device with JEDEC ID 0x202017.
   **Warning** FLASH configuration checksum error or invalid key
   Use 'fconfig -i' to [re]initialize database

   RedBoot(tm) bootstrap and debug environment [ROM]
   Non-certified release, version UNKNOWN - built 10:27:53, Jan 26 2010

   Copyright (C) 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009
   Free Software Foundation, Inc.
   Copyright (C) 2003, 2004, 2005, 2006, 2007, 2008 eCosCentric Limited
   RedBoot is free software, covered by the eCos license, derived from the
   GNU General Public License. You are welcome to change it and/or distribute
   copies of it under certain conditions. Under the license terms, RedBoot's
   source code and full license terms must have been made available to you.
   Redboot comes with ABSOLUTELY NO WARRANTY.

   Platform: Logic Zoom OMAP L138 eXperimenter (ARM9)
   RAM: 0xc0000000-0xc4000000 [0xc0022128-0xc3fed000 available]
   FLASH: 0x70000000-0x707fffff, 128 x 0x10000 blocks
   RedBoot>
   ```

6. Run the following command to initialize RedBoot's flash file system and flash configuration:

   ```
   RedBoot> fis init
   About to initialize [format] FLASH image system - continue (y/n)? y
   *** Initialize FLASH Image System
   ... Erase from 0x707f0000-0x707fffff: .
   ... Program from 0xc3ff0000-0xc4000000 to 0x707f0000: .
   RedBoot> fconfig -i
   Initialize non-volatile configuration - continue (y/n)? y
   Run script at boot: false
   Console baud rate: 115200
   Update RedBoot non-volatile configuration - continue (y/n)? y
   ... Erase from 0x707e0000-0x707e0fff: .
   ... Program from 0xc3fef000-0xc3ff0000 to 0x707e0000: .
   RedBoot>
   ```

7. We now need to download a copy of RedBoot and program it into the flash. Give the following command to RedBoot:

   ```
          RedBoot> load -r -m y -b %{freememlo}
          C
   ```

   From the terminal emulator upload the `redboot_ROM.img` file using Y-Modem protocol. When the upload is complete you should see something similar to the following output.

   ```
   RedBoot> load -r -m y -b %{freememlo}
   CRaw file loaded 0xc0022400-0xc00384ef, assumed entry at 0xc0022400
   xyzModem - CRC mode, 708(SOH)/0(STX)/0(CAN) packets, 4 retries
   RedBoot>
   ```

8. Now program the RedBoot image to flash:

   ```
   RedBoot> fis cre RedBoot
   ```

```
An image named 'RedBoot' exists - continue (y/n)? y
... Erase from 0x70010000-0x7002ffff: ..
... Program from 0xc0022400-0xc00384f0 to 0x70010000: ..
... Erase from 0x707f0000-0x707fffff: .
... Program from 0xc3ff0000-0xc4000000 to 0x707f0000: .
RedBoot>
```

The RedBoot installation is now complete. This can be tested by issuing the **reset run** command to the PEEDI, or by detaching the PEEDI and power cycling the board. Output similar to the following should be seen on the serial port.

```
+M25PXX : Init device with JEDEC ID 0x202017.
RedBoot(tm) bootstrap and debug environment [ROM]
Non-certified release, version UNKNOWN - built 10:27:53, Jan 26 2010

Copyright (C) 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009
Free Software Foundation, Inc.
Copyright (C) 2003, 2004, 2005, 2006, 2007, 2008 eCosCentric Limited
RedBoot is free software, covered by the eCos license, derived from the
GNU General Public License. You are welcome to change it and/or distribute
copies of it under certain conditions. Under the license terms, RedBoot's
source code and full license terms must have been made available to you.
Redboot comes with ABSOLUTELY NO WARRANTY.

Platform: Logic Zoom OMAP L138 eXperimenter (ARM9)
RAM: 0xc0000000-0xc4000000 [0xc0022128-0xc3fed000 available]
FLASH: 0x70000000-0x707fffff, 128 x 0x10000 blocks
RedBoot>
```

If it proves necessary to re-install RedBoot, this may be achieved by repeating the serial download and **fis create** parts of the above process. It is not necessary to reinitialize the FIS and fconfig.

Note

If the board has been supplied with a TI User Boot Loader (UBL) version prior to 1.65, then on startup the board may output a string on the serial port saying "No magic number found". Earlier Zoom boards were known to be supplied with version 1.30. In this situation you will need to locate the script `flashimg.tcl` in the `misc` subdirectory of the Zoom platform HAL (i.e. `packages/hal/arm/arm9/zoom_l138/`*`VERSION`*`/misc/flashimg.tcl`) along with the `redboot_ROM.bin` file in the `loaders` subdirectory of your eCosPro installation, and run the following command at a command prompt:

```
flashimg.tcl -oldubl redboot_ROM.bin redboot-oldubl.img
```

You can then follow the above instructions for installing RedBoot, but use `redboot-oldubl.img` in place of uses of `redboot_ROM.img`.

You will also need to perform this step on any ROM startup user applications to be programmed into Flash using a JTAG device and booted by UBL.

Programming RedBoot into NOR flash using the TI Serial Boot and Flash utility

Texas Instruments have made available a command-line Serial Boot and Flash Loading Utility for OMAP-L138. More information including download and usage instructions is available here on their website.

With this utility, you can program a ROM startup version of RedBoot in raw binary format (a prebuilt version of which may be found at `loaders/zoom_l138/redboot_ROM.bin` within your eCos installation). On Windows, we advise running the utility from a Command Prompt, rather than from a Cygwin bash shell.

Before running the utility, first you need to connect your PC to the board's serial port using a null-modem RS232 serial cable. Secondly, (with the board powered off) you must set DIP switch bank S7 so that switches 7, and 8 are set to the ON position and the rest are set to the OFF position. If you wish to confirm the board is configured correctly, then you can start a terminal emulator application (such as Hyperterminal on some versions of Windows) and connect to the serial port at 115200 baud, 8-N-1 with no hardware flow control, then you should see a "BOOTME" prompt when you power on the board.

To program the `redboot_ROM.bin` image, change directory to the directory containing the UBL binary file (which is included with the Serial Boot and Flash utility download), copy the `redboot_ROM.bin` into that directory, and then run the utility as follows:

```
sfh_OMAP-L138.exe -flash ubl_OMAPL138_SPI_MEM.bin redboot_ROM.bin -APPStartAddr 0xC0008040 -APPLoadAddr 0xC0008000
```

After successful completion, RedBoot will be resident in SPI NOR Flash. To return to the normal boot mode, you must reset the SW7 DIP switches to their default position allowing booting from SPI NOR Flash. To do so, set all S7 switches to their OFF positions.

Rebuilding RedBoot

Should it prove necessary to rebuild a RedBoot binary, this is done most conveniently at the command line. The steps needed to rebuild the ROM version of RedBoot for the Zoom L138 are:

```
$ mkdir redboot_zoom_l138_rom
$ cd redboot_zoom_l138_rom
$ ecosconfig new zoom_l138 redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/arm/arm9/zoom_l138/VERSION/misc/redboot_ROM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

At the end of the build the `install/bin` subdirectory should contain the files `redboot.srec` and `redboot.img`. `redboot.img` is a binary file that includes a 16 byte header needed by the User Boot Loader to load RedBoot successfully.

Name

Configuration — Platform-specific Configuration Options

Overview

The Zoom L138 platform HAL package is loaded automatically when eCos is configured for the `zoom-l138` target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The platform HAL package supports two separate startup types:

RAM This is the startup type which is normally used during application development. The board has RedBoot programmed into flash and boots into that initially. arm-eabi-gdb is then used to load a RAM startup application into memory and debug it. It is assumed that the hardware has already been initialized by RedBoot. By default the application will use the eCos virtual vectors mechanism to obtain certain services from RedBoot, including diagnostic output.

ROM This startup type can be used for finished applications which will be programmed into Flash. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization. This startup type can also be used for applications loaded via JTAG.

RedBoot and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building RedBoot.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then the serial port will be claimed for HAL diagnostics.

Flash Driver

The Zoom board contains an 8Mbyte Numonyx M25P64 SPI serial NOR flash device. The `CYGPKG_DEVS_FLASH_SPI_M25PXX` package contains all the code necessary to support this part and the platform HAL package contains definitions that customize the driver to the Zoom L138 board. This driver is not active until the generic Flash support package, `CYGPKG_IO_FLASH`, is included in the configuration.

This driver is capable of supporting the JFFS2 filesystem. However, note that the SPI interface means that this file system has reduced bandwidth and increased latency compared with other implementations. All that is required to enable the support is to include the filesystem (`CYGPKG_FS_JFFS2`) and any of its package dependencies (including `CYGPKG_IO_FILEIO` and `CYGPKG_LINUX_COMPAT`) together with the flash infrastructure (`CYGPKG_IO_FLASH`).

Ethernet Driver

The Zoom L138 board uses the OMAP L138's internal EMAC ethernet device attached to an external SMSC LAN8710A PHY. The `CYGPKG_DEVS_ETH_ARM_OMAP` package contains all the code necessary to support this device and the platform HAL package contains definitions that customize the driver to the ZOOM L138 board. This driver is not active until the generic Ethernet support package, `CYGPKG_IO_ETH_DRIVERS`, is included in the configuration.

RTC Driver

The ZOOM L138 board uses the OMAP L138's internal RTC support. The `CYGPKG_DEVICES_WALLCLOCK_ARM_OMAP_L1XX` package contains all the code necessary to support this device. This driver is not active until the generic wallclock device support package, `CYGPKG_IO_WALLCLOCK`, is included in the configuration.

Watchdog Driver

The Zoom L138 board uses the OMAP L138's internal watchdog support. The `CYGPKG_DEVICES_WATCHDOG_ARM_OMAP_L1XX` package contains all the code necessary to support this device. Within that package the `CYGNUM_DEVS_WATCHDOG_ARM_OMAP_L1XX_DESIRED_TIMEOUT_MS` configuration option controls the watchdog timeout, and by default will force a reset of the board upon timeout. This driver is not active until the generic watchdog device support package, `CYGPKG_IO_WATCHDOG`, is included in the configuration.

UART Serial Driver

The Zoom L138 board uses the OMAP L138's internal UART serial support as described in the OMAP L1xx processor HAL documentation. Only one serial connector is available on the board, which is connected to UART2. This connector has the RTS/CTS hardware flow control lines connected in addition to the data lines.

MMC/SD Driver

As the OMAP L1xx MMC/SD driver is part of the OMAP L1xx HAL, nothing is required to load it. Similarly the MMC/SD bus driver layer (`CYGPKG_DEVS_DISK_MMC`) is automatically included as part of the hardware-specific configuration for this target. All that is required to enable the support is to include the generic disk I/O infrastructure package (`CYGPKG_IO_DISK`), along with the intended filesystem, typically, the FAT filesystem (`CYGPKG_FS_FAT`) and any of its package dependencies (including `CYGPKG_LIBC_STRING` and `CYGPKG_LINUX_COMPAT` for FAT).

Various options can be used to control specifics of the MMC/SD driver. Consult the OMAP L1xx HAL documentation for information on its configuration.

This board has the MMC/SD socket's card detect and write protect lines connected to GPIO lines. The card detect line is additionally monitored by an interrupt handler. Thus the disk I/O layer's removeable media support will detect when cards have been inserted or removed, and the FILEIO layer's automounter can be used.

Compiler Flags

The platform HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos. There are just three flags specific to this port:

`-mcpu=arm926ej-s`
: The arm-eabi-gcc compiler supports many variants of the ARM architecture. A `-m` option should be used to select the specific variant in use, and with current tools `-mcpu=arm926ej-s` is the correct option for the ARM926EJ-S CPU in the OMAP L1xx.

`-mthumb`
: The arm-eabi-gcc compiler will compile C and C++ files into the Thumb instruction set when this option is used. The best way to build eCos in Thumb mode is to enable the configuration option `CYGHWR_THUMB`.

`-mthumb-interwork`
: This option allows programs to be created that mix ARM and Thumb instruction sets. Without this option, some memory can be saved. This option should be used if -mthumb is used. The best way to build eCos with Thumb interworking is to enable the configuration option `CYGBLD_ARM_ENABLE_THUMB_INTERWORK`.

Name

JTAG support — Usage

Use of JTAG for debugging

JTAG can be used to single-step and debug loaded applications, including RedBoot. Only ROM configuration applications should be debugged using JTAG, RAM applications assume the presence of RedBoot.

Ronetix PEEDI notes

On the Ronetix PEEDI, the `peedi.zoom.cfg` file should be used to setup and configure the hardware to an appropriate state to load programs. This includes setting up the PLLs and SDRAM controller.

The `peedi.zoom.cfg` file also contains an option to define whether hardware or software breakpoints are used by default, using the `CORE0_BREAKMODE` directive in the `[PLATFORM_ARM]` section. Edit this file if you wish to use hardware break points, and remember to restart the PEEDI to make the changes take effect.

On the PEEDI, debugging can be performed either via the telnet interface or using **arm-eabi-gdb** and the GDB interface. In the case of the latter, **arm-eabi-gdb** needs to connect to TCP port 2000 on the PEEDI's IP address. For example:

```
(gdb) target remote 111.222.333.444:2000
```

By default when the PEEDI is powered up, the target will always run the initialization section of the `peedi.zoom.cfg` file, and halts the target. This behaviour is repeated with the **reset** command.

If the board is reset (either with the '**reset**', or by pressing the reset button) and the '**go**' command is then given, then the board will boot as normal. If a second-level bootstrap and ROM RedBoot is resident in flash, it will be run.

Consult the PEEDI documentation for information on other features.

Running ROM applications

Applications configured for ROM startup can be run directly under JTAG. Once loaded and running via JTAG, HAL diagnostic output will appear by default on the serial debug port.

Installing user applications into Flash with JTAG

If you wish to install a ROM startup application into Flash to be automatically booted, you can follow a similar procedure to installing RedBoot into Flash. However before you can do so, you must first prepend a header to your application image in order for the TI User Boot Loader (UBL) to recognise it as a valid application.

You will need to locate the script **flashimg.tcl** in the `misc` subdirectory of the Zoom platform HAL (i.e. `packages/hal/arm/arm9/zoom_l138/`*`VERSION`*`/misc/flashimg.tcl`) and generate a binary image of your program using the **arm-eabi-objcopy** command. The following gives an example simplified command sequence which can be run at a command shell prompt:

```
arm-eabi-objcopy -O binary myapp myapp.bin
flashimg.tcl myapp.bin myapp.img
```

You will need to subsitute your own paths and filenames where applicable. Additionally, if your board is installed with a UBL version earlier than 1.65, you are likely to need to use the extra option *`-oldubl`* to **flashimg.tcl**, otherwise you may receive errors about missing magic numbers from UBL at boot time.

Once you have the `.img` file, you can follow the same process as installing RedBoot via JTAG.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the Zoom L138 hardware, and should be read in conjunction with that specification. The platform HAL package complements the ARM architectural HAL, the ARM9 variant HAL and the OMAP L1xx processor HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset, the HAL will initialize or reinitialize most of the on-chip peripherals. There is an exception for RAM startup applications which depend on a ROM monitor for certain services.

For ROM startup, the HAL will perform additional initialization. This is all done in the `PLATFORM_SETUP1` macro in the assembler header file `hal_platform_setup.h`.

Linker Scripts and Memory Maps

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

SDRAM

This is located at address 0xC0000000 of the physical memory space. The HAL configures the MMU to retain the SDRAM at virtual address 0xC0000000, but in order to assign hardware exception vectors vectors at address 0x00000000, the HAL also uses the MMU to create a clone of this memory at virtual address 0x00000000. The same memory is also accessible uncached and unbuffered at virtual location 0xD0000000 for use by devices. The first 32 bytes are used for hardware exception vectors. The next 32 bytes are used for the VSR table and the next 256 bytes are normally used for the eCos virtual vectors, allowing RAM-based applications to use services provided by the ROM monitor. Memory is required for the MMU tables, and must be aligned on a 16Kbyte boundary. These therefore occupy memory from 0x4000 to 0x8000. For ROM startup, all remaining SDRAM is available. For RAM startup, available RAM starts at virtual location 0xC0040000, with the bottom 256kB reserved for use by RedBoot. ROM applications are loaded starting at 0xC0008000, which leaves space for the User Boot Loader.

On-chip SRAM

There are a number of on-chip SRAM areas. These are identity mapped unbuffered and uncached with their physical addresses.

SPI NOR Flash

SPI NOR flash media can only be accessed with the Flash API. For the purposes of this API a placeholder address range has been allocated as if the Flash is present at this address. The base of this address range is 0x70000000. This reserved range is not real memory and any attempt to access it directly by the processor other than via the Flash API will result in a memory address exception.

On-chip Peripheral Registers

These are located at various addresses in the physical memory space. When the MMU is enabled, it sets up a direct, uncached, unbuffered mapping so that these registers remain accessible at their physical locations.

Off-chip Peripherals

eCos uses the SDRAM, ethernet PHY, MMC/SD, and SPI flash facilities on the Zoom L138 board. eCos does not currently make any use of any other off-chip peripherals present on this board.

SPI NOR Flash

eCos supports SPI access to the NOR flash on the board. The device is typically used to contain RedBoot and flash configuration data.

Accesses to SPI flash are performed via the Flash API, using 0x70000000 or as the nominal address of the device, although it does not truly exist in the processor address space.

Since SPI flash is not directly addressable, access from RedBoot is only possible using **fis** command operations.

Real-time characterization

The tm_basic kernel test gives statistics gathered about the real-time characterization and performance of the kernel. The sample output is shown here for information. The test was built in ARM mode, which provides better performance than Thumb mode.

Example 265.1. zoom_l138 Real-time characterization

```
            Startup, main stack : stack used   392 size  3920
             Startup :  Interrupt stack used   504 size  4096
             Startup : Idlethread stack used    88 size  2048

eCos Kernel Timings
Notes: all times are in microseconds (.000001) unless otherwise stated

Reading the hardware clock takes 0 'ticks' overhead
... this value will be factored out of all other measurements
Clock interrupt took    4.52 microseconds (9 raw clock ticks)

Testing parameters:
   Clock samples:            32
   Threads:                  64
   Thread switches:         128
   Mutexes:                  32
   Mailboxes:                32
   Semaphores:               32
   Scheduler operations:    128
   Counters:                 32
   Flags:                    32
   Alarms:                   32


                                 Confidence
     Ave     Min     Max     Var  Ave  Min  Function
  ======  ======  ======  ====== ========== ========
    3.21    2.00    8.50    0.53   60%  25% Create thread
    0.80    0.50    2.50    0.27   54%  43% Yield thread [all suspended]
    1.08    0.50    3.00    0.15   85%   1% Suspend [suspended] thread
    1.07    0.50    2.50    0.18   76%   6% Resume thread
    1.51    1.00    5.00    0.19   73%  17% Set priority
    0.90    0.50    1.50    0.22   64%  28% Get priority
    2.42    2.00   10.50    0.36   54%  42% Kill [suspended] thread
    0.79    0.50    2.00    0.26   53%  45% Yield [no other] thread
    1.45    1.00    4.00    0.22   67%  25% Resume [suspended low prio] thread
    1.05    0.50    2.50    0.14   81%   6% Resume [runnable low prio] thread
    1.50    1.00    4.00    0.17   71%  17% Suspend [runnable] thread
    0.80    0.50    2.50    0.27   53%  45% Yield [only low prio] thread
    1.08    0.50    2.50    0.16   81%   3% Suspend [runnable->not runnable]
    2.14    1.50    7.50    0.28   78%   4% Kill [runnable] thread
    2.33    2.00    6.00    0.31   50%  46% Destroy [dead] thread
    2.96    2.50    7.00    0.26   62%  28% Destroy [runnable] thread
    4.34    3.50    9.50    0.43   82%   6% Resume [high priority] thread
    1.48    1.00    4.00    0.08   90%   7% Thread switch
    0.43    0.00    1.00    0.13   84%  14% Scheduler lock
    0.70    0.50    1.00    0.24   60%  60% Scheduler unlock [0 threads]
    0.69    0.50    1.00    0.24   61%  61% Scheduler unlock [1 suspended]
    0.70    0.50    1.50    0.24   60%  60% Scheduler unlock [many suspended]
    0.70    0.50    1.00    0.24   60%  60% Scheduler unlock [many low prio]

    0.45    0.00    2.00    0.17   78%  18% Init mutex
    1.02    0.50    2.50    0.12   84%   9% Lock [unlocked] mutex
    1.13    0.50    3.50    0.29   68%   9% Unlock [locked] mutex
    0.98    0.50    2.50    0.12   84%  12% Trylock [unlocked] mutex
    0.88    0.50    1.50    0.21   68%  28% Trylock [locked] mutex
    0.50    0.00    1.50    0.06   90%   6% Destroy mutex
    3.14    3.00    6.50    0.25   90%  90% Unlock/Lock mutex

    0.52    0.50    1.00    0.03   96%  96% Create mbox
    0.53    0.50    1.00    0.06   93%  93% Peek [empty] mbox
    1.17    1.00    3.00    0.26   75%  75% Put [first] mbox
    0.73    0.50    1.00    0.25   53%  53% Peek [1 msg] mbox
    1.17    1.00    3.00    0.26   75%  75% Put [second] mbox
    0.55    0.50    1.00    0.08   90%  90% Peek [2 msgs] mbox
    1.20    1.00    3.00    0.28   68%  68% Get [first] mbox
    1.30    1.00    2.00    0.26   53%  43% Get [second] mbox
    1.09    1.00    2.50    0.16   87%  87% Tryput [first] mbox
    1.11    1.00    2.50    0.18   84%  84% Peek item [non-empty] mbox
    1.16    1.00    2.50    0.23   75%  75% Tryget [non-empty] mbox
    1.06    0.50    2.50    0.14   87%   3% Peek item [empty] mbox
    0.98    0.50    2.50    0.12   84%  12% Tryget [empty] mbox
    0.75    0.50    1.00    0.25  100%  50% Waiting to get mbox
    0.55    0.50    1.00    0.08   90%  90% Waiting to put mbox
    0.81    0.50    2.50    0.31   93%  50% Delete mbox
    2.42    2.00    7.00    0.42   93%  50% Put/Get mbox

    0.44    0.00    0.50    0.11   87%  12% Init semaphore
    0.88    0.50    2.00    0.26   59%  34% Post [0] semaphore
    0.88    0.50    2.00    0.23   65%  31% Wait [1] semaphore
    0.84    0.50    2.00    0.26   59%  37% Trywait [0] semaphore
    0.77    0.50    1.00    0.25   53%  46% Trywait [1] semaphore
    0.55    0.00    1.50    0.17   75%   9% Peek semaphore
    0.53    0.00    1.50    0.12   84%   6% Destroy semaphore
    2.13    2.00    5.00    0.23   90%  90% Post/Wait semaphore

    0.55    0.50    2.00    0.09   96%  96% Create counter
    0.67    0.00    1.50    0.26   62%   3% Get counter value
    0.50    0.00    1.00    0.06   87%   6% Set counter value
    1.03    0.50    2.00    0.12   84%   6% Tick counter
    0.64    0.00    1.50    0.23   68%   3% Delete counter

    0.42    0.00    0.50    0.13   84%  15% Init flag
    0.91    0.50    2.50    0.23   68%  28% Destroy flag
    0.81    0.50    2.00    0.27   53%  43% Mask bits in flag
    0.91    0.50    2.00    0.20   71%  25% Set bits in flag [no waiters]
    1.08    1.00    3.50    0.15   96%  96% Wait for flag [AND]
    0.98    0.50    2.00    0.09   87%   9% Wait for flag [OR]
    1.03    1.00    2.00    0.06   96%  96% Wait for flag [AND/CLR]
    0.98    0.50    2.00    0.09   87%   9% Wait for flag [OR/CLR]
    0.41    0.00    0.50    0.15   81%  18% Peek on flag

    0.58    0.50    3.00    0.15   96%  96% Create alarm
    1.67    1.00    5.50    0.36   87%   9% Initialize alarm
    1.00    0.50    2.00    0.06   90%   6% Disable alarm
    1.55    1.00    4.50    0.30   62%  21% Enable alarm
    1.06    1.00    2.50    0.12   93%  93% Delete alarm
    0.94    0.50    1.50    0.14   81%  15% Tick counter [1 alarm]
    3.67    3.50    4.00    0.23   65%  65% Tick counter [many alarms]
    1.39    1.00    2.50    0.22   68%  28% Tick & fire counter [1 alarm]
   17.56   17.50   18.00    0.11   87%  87% Tick & fire counters [>1 together]
```

```
    4.09    4.00    4.50    0.15   81%  81% Tick & fire counters [>1 separately]
    4.01    4.00    5.00    0.02   99%  99% Alarm latency [0 threads]
    4.47    4.00    5.50    0.21   63%  21% Alarm latency [2 threads]
    8.21    6.50   10.00    0.64   47%  11% Alarm latency [many threads]
    6.04    6.00   11.00    0.08   98%  98% Alarm -> thread resume latency

    1.01    0.50    4.00    0.00            Clock/interrupt latency

    1.85    1.00    5.50    0.00            Clock DSR latency

    5      0     272  (main stack:  1328)  Thread stack used (1360 total)
           All done, main stack : stack used  1328 size  3920
            All done :  Interrupt stack used   140 size  4096
            All done : Idlethread stack used   232 size  2048

Timing complete - 29920 ms total

PASS:<Basic timing OK>
EXIT:<done>
```

Other Issues

The platform HAL does not affect the implementation of other parts of the eCos HAL specification. The OMAP L1xx processor HAL, ARM9 variant HAL, and the ARM architectural HAL documentation should be consulted for further details.

Chapter 266. Freescale i.MXxx Processor Support

Name

Support for the Freescale i.MXxx Processor — Overview

Description

This document covers the configuration and usage of the Hardware Abstraction Layer (HAL) for the Freescale i.MXxx processor family. It is expected to be read in conjunction with platform HAL-specific documentation, as well as the eCos HAL specification. This processor HAL package complements the ARM architectural HAL, ARM9 variant HAL and the platform HAL. It provides functionality common to all i.MXxx-based board implementations.

This support is found in the eCos package located at `packages/hal/arm/arm9/imx` within the eCos source repository.

The i.MXxx processor HAL package is loaded automatically when eCos is configured for an i.MXxx-based platform. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Supported Hardware

Supported features of the i.MXxx processor within this processor HAL package include:

- i.MXxx-specific hardware definitions
- Interrupt controller
- Timers
- Serial UARTs
- Pin Configuration and GPIO Support
- Peripheral Clock Control

Support for the on-chip SPI device, SPI NOR flash, I²C, interrupt-driven serial, Ethernet and watchdog features of the i.MXxx are also present and can be found in separate packages, outside of this processor HAL.

Name

i.Mxx Hardware Definitions — Details on obtaining hardware definitions for i.MXxx

Register definitions

The file `<cyg/hal/imx.h>` can be included from application and eCos package sources to provide definitions related to IMX subsystems. These include register definitions for the interrupt controller, clock management controller, pin multiplexing, GPIO, UART, timers and watchdog subsystems. Register definitions for some devices are to be found in the appropriate driver packages and only base addresses and configuration definitions are provided here. This file will normally be included automatically if `<cyg/hal/hal_io.h>` is included, which is the preferred way of getting these definitions.

Initialization Helper Macros

The file `<cyg/hal/imx_init.inc>` contains definitions of helper macros which may be used by i.MXxx platform HALs in order to initialize common subsystems without excessive duplication between the platform HALs. Typically this file will be included by the `hal_platform_setup.h` header in the platform HAL, in turn included from the architectural HAL file `vectors.S`.

This file is solely intended to be used by platform HALs. At the same time, it is only present to assist initialization, and platform HALs are not obliged to use it if their startup requirements vary. NOTE: At present, the only extant i.MXxx port relies on either the on-chip boot loader, or the JTAG initialization script, to initialize the PLLs and memory controller, so these macros currently largely contain ARM9-generic setup only.

Name

i.MXxx Interrupt Controller — Advanced Interrupt Controller Definitions And Usage

Interrupt controller definitions

The file `<cyg/hal/var_ints.h>` (located at `hal/arm/arm9/imx/`*`VERSION`*`/include/var_ints.h` in the eCos source repository) contains interrupt vector number definitions for use with the eCos kernel and driver interrupt APIs.

The list of interrupt vectors may be augmented on a per-platform basis. Consult the platform HAL documentation for your platform for whether this is the case.

Interrupt Controller Functions

The source file `src/imx_misc.c` within this package provides most of the support functions to manipulate the interrupt controller. The `hal_IRQ_handler` queries the IRQ status register to determine the interrupt cause. Functions `hal_interrupt_mask` and `hal_interrupt_unmask` enable or disable interrupts within the interrupt controller.

Interrupts are configured in the `hal_interrupt_configure` function. Only GPIO interrupts are configurable, and at present we do not support full decoding of these, so this function is empty.

The `hal_interrupt_acknowledge` function acknowledges an interrupt. Since there is no action needed to acknowledge an interrupt, this function is empty.

The `hal_interrupt_set_level` is used to set the priority level of the supplied interrupt within the interrupt controller. The level value may range from 0 to 15, with 0 being the highest priority.

Note that in all the above, it is not recommended to call the described functions directly. Instead either the HAL macros (`HAL_INTERRUPT_MASK` et al) or preferably the kernel or driver APIs should be used to control interrupts.

Name

Timers — Use of on-chip timers

System Clock

The eCos kernel system clock is implemented using EPIT1 By default, the system clock interrupts once every 10ms, corresponding to a 100Hz clock. This can be changed by the configuration option `CYGNUM_HAL_RTC_DENOMINATOR` which corresponds to the clock frequency. Other clock-related settings are recalculated automatically if the denominator is changed. If the desired frequency cannot be expressed accurately solely with changes to `CYGNUM_HAL_RTC_DENOMINATOR`, then the configuration option `CYGNUM_HAL_RTC_NUMERATOR` may also be adjusted and the value of `CYGNUM_HAL_RTC_PERIOD` adjusted to match.

The same timer is used to implement the HAL microsecond delay function, `HAL_DELAY_US`. This is used by some device drivers, and in non-kernel configurations such as with RedBoot where this timer is needed for loading program images via X/Y-modem protocols and debugging via TCP/IP. Standalone applications which require RedBoot services, such as debugging, should avoid use of this timer.

Timer-based profiling support

Timer-based profiling support is implemented using EPIT2 If the gprof package, `CYGPKG_PROFILE_GPROF`, is included in the configuration, then EPIT2 reserved for use by the profiler.

Name

Serial UARTs — Configuration and Implementation Details of Serial UART Support

Overview

Support is included in this processor HAL package for up to five of the i.MXxx's on-chip serial UART devices. Interfaces `CYGINT_HAL_IMX_UART1` to `CYGINT_HAL_IMX_UART5` indicate for each UART whether it is connected to an external port and should be **implemented** as appropriate by the platform HAL CDL.

There are two forms of support: HAL diagnostic I/O; and a fully interrupt-driven serial driver. Unless otherwise specified in the platform HAL documentation, for all serial ports the default settings are 115200,8,N,1 with no flow control.

HAL Diagnostic I/O

This first form is polled mode HAL diagnostic output, intended primarily for use during debug and development. Operations are usually performed with global interrupts disabled, and thus this mode is not usually suitable for deployed systems. This can operate on any port, according to the configuration settings.

There are several configuration options usually found within a platform HAL which affect the use of this support in the IMX processor HAL. The CDL option `CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL` selects the serial port channel to use as the console at startup time. This will be the channel that receives output from, for example, `diag_printf()`. The CDL option `CYGNUM_HAL_VIRTUAL_VECTOR_DEBUG_CHANNEL` selects the serial port channel to use for GDB communication by default. Note that when using RedBoot, these options are usually inactive as it is RedBoot that decides which channels are used. Applications may override RedBoot's selections by enabling the `CYGSEM_HAL_VIRTUAL_VECTOR_CLAIM_COMMS` CDL option in the HAL. Baud rates for each channel are set with the `CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL_BAUD` and `CYGNUM_HAL_VIRTUAL_VECTOR_DEBUG_CHANNEL_BAUD` options.

Interrupt-driven Serial Driver

The second form of support is an interrupt-driven serial driver, which is integrated into the eCos standard serial I/O infrastructure (`CYGPKG_IO_SERIAL`). This support can be enabled on any port.

Note that it is not recommended to share this driver when using the HAL diagnostic I/O on the same port. If the driver is shared with the GDB debugging port, it will prevent ctrl-c operation when debugging.

The main part of this driver is contained in the generic `CYGPKG_IO_SERIAL_ARM_IMX` package. A platform specific package, for example `CYGPKG_IO_SERIAL_ARM_MCIMX25`, contains definitions that configure the generic driver for the platform. That driver package should also be consulted for documentation and configuration options. The driver is not active until the `CYGPKG_IO_SERIAL_DEVICES` configuration option within the generic serial driver support package `CYGPKG_IO_SERIAL` is enabled in the configuration.

Support for hardware flow control and modem control lines is present in the driver, but will only be enabled if these control signals are brought out to the physical serial port.

Name

Pin Configuration and GPIO Support — Use of pin configuration and GPIO

Synopsis

#include <cyg/hal/hal_io.h>

pin = **CYGHWR_HAL_IMX_PINMUX**(*padctl_off*, *padctl*, *muxctl_off*, *muxmode*, *selinput_off*, *selinput*);

CYGHWR_HAL_IMX_PINMUX_SET (*pin*);

pin = **CYGHWR_HAL_IMX_GPIO**(*bank*, *bit*, *mode*);

CYGHWR_HAL_IMX_GPIO_SET (*pin*);

CYGHWR_HAL_IMX_GPIO_OUT (*pin*, *val*);

CYGHWR_HAL_IMX_GPIO_IN (*pin*, *val*);

CYGHWR_HAL_IMX_GPIO_INTCFG (*pin*, *mode*);

CYGHWR_HAL_IMX_GPIO_INTSTAT (*pin*, *stat*);

CYGHWR_HAL_IMX_GPIO_INTMASK (*pin*, *enable*);

CYGHWR_HAL_IMX_GPIO_INTCLR (*pin*);

Description

The i.MXxx HAL provides a number of macros to support the encoding of pin multiplexing information and GPIO pin modes into descriptors. This is useful to drivers and other packages that need to configure and use different lines for different devices. Because there is not a simple correspondence between pin multiplexing information and GPIO bank and pin identities, these two things are treated separately.

Pin Multiplexing

There is no systematic relationship between the various registers that control the properties of a single io pin. So all the information needed to identify and configure a pin is encoded into a 64 bit descriptor. To define a new pin descriptor it is necessary to consult the appropriate processor reference manual for the register offsets and valid settings. A pin multiplexing descriptor is represented by the hal_imx_pin and is created with `CYGHWR_HAL_IMX_PINMUX()` which takes the following arguments:

padctl_off
: The offset of the pad control register in the IO multiplexor. If this is zero, no pad control configuration is performed.

padctl
: Pad control settings. This is just the last part of the name of a `CYGHWR_HAL_IMX_PINMUX_PADCTL_*` macro. Macros are defined to correspond to the fields of this register, and combination macros are defined to set several fields. The user can also define their own macros if the default set do not contain the required values.

muxctl_off
: The offset of the multiplexing control register in the IO multiplexor. If this is zero, no multiplexing is performed.

muxmode
: This sets the multiplexing mode for the pin. It may be either `MUX(x)` or `MUX_SION(x)` where *x* is the multiplexing mode to attach this pin to the selected device.

selinput_off Some device inputs can be attached to more than one pin. In this case this parameter contains the offset of the input select register that controls this. If this value is zero, no input selection is made.

selinput This is the input selection value. It may either be `NONE` or `SEL(x)`, where *x* is the selection value.

The following examples show how this macro may be used:

```
// UART1 RX line, with 100K Ohm pull down, mux 0
#define CYGHWR_HAL_IMX_UART1_RX \
        CYGHWR_HAL_IMX_PINMUX( 0x368, PUS_100KD, 0x170, MUX_SION(0),     0, NONE   )

// UART4 RX line, 100K Ohm pull down, mux 1, input selection 1
#define CYGHWR_HAL_IMX_UART4_RX \
        CYGHWR_HAL_IMX_PINMUX( 0x3B0, PUS_100KD, 0x1B8, MUX_SION(1), 0x570, SEL(1) )

// GPIO line, floating, mix 5
#define CYGHWR_HAL_IMX_FEC_RESET \
        CYGHWR_HAL_IMX_PINMUX( 0x238,     FLOAT, 0x01C, MUX(5),     0, NONE   )
```

The macro `CYGHWR_HAL_IMX_PINMUX_SET( pin )` sets the pin multiplexing setting according to the descriptor passed in.

GPIO Support

A GPIO descriptor is created with `CYGHWR_HAL_IMX_GPIO( bank, bit, mode)` which takes the following arguments:

bank This identifies the GPIO bank to which the pin is attached. This is a value between 1 and 4.

bit This gives the bit offset within the bank of the GPIO pin. This is a value between 0 and 31.

mode This defines whether this is an input or an output pin, and may take the values `INPUT` or `OUTPUT` respectively.

Additionally, the macro `CYGHWR_HAL_IMX_GPIO_NONE` may be used in place of a pin descriptor and has a value that no valid descriptor can take. It may therefore be used as a placeholder where no GPIO pin is present or to be used.

The following examples show how this macro may be used:

```
// PHY Reset pin on GPIO4, pin 8, output
#define CYGHWR_HAL_IMX_FEC_RESET_GPIO   CYGHWR_HAL_IMX_GPIO( 4, 8, OUTPUT )

// CSPI 1, chip select 0 on GPIO1, pin 16, output
#define CYGHWR_HAL_IMX_CSPI1_SS0_GPIO   CYGHWR_HAL_IMX_GPIO( 1, 16, OUTPUT )
```

The remaining macros all take a GPIO pin descriptor as an argument. `CYGHWR_HAL_IMX_GPIO_SET` configures the pin according to the descriptor and must be called before any other macros. `CYGHWR_HAL_IMX_GPIO_OUT` sets the output to the value of the least significant bit of the *`val`* argument. The *`val`* argument of `CYGHWR_HAL_IMX_GPIO_IN` should be a pointer to an int, which will be set to 0 if the pin input is zero, and 1 otherwise.

There is also support for GPIO interrupts. `CYGHWR_HAL_IMX_GPIO_INTCFG( pin, mode )` configures the interrupt mode of the pin. It may be either `LOW_LEVEL`, `HIGH_LEVEL`, `RISING_EDGE`, `FALLING_EDGE` or `EITHER_EDGE` to configure the pin, respectively, to interrupt on active-low, active-high, rising edge, falling edge or both rising and falling edges. For example:

```
// PHY interrupt on GPIO3 pin 19
#define CYGHWR_HAL_IMX_FEC_INTERRUPT_GPIO CYGHWR_HAL_IMX_GPIO( 3, 19, INPUT )

// Configure active-LOW interrupt
CYGHWR_HAL_IMX_GPIO_INTCFG( CYGHWR_HAL_IMX_FEC_INTERRUPT_GPIO, LOW_LEVEL );
```

The second argument to `CYGHWR_HAL_IMX_GPIO_INTSTAT( pin, stat )` must be a pointer to an int, which will be set to 1 if an interrupt has be received on the given pin, and 0 otherwise.

Note

GPIO interrupts are currently not decoded into per-pin interrupt vectors, only the shared per-bank vectors are available. If an application needs to get interrupts from more than one pin on a bank, it needs to install a shared ISR and decode the specific pins itself.

The second argument to `CYGHWR_HAL_IMX_GPIO_INTMASK( pin, enable )` when set to 1 will enable the relevant GPIO interrupt source for the configured pin, with 0 disabling the source. If required, `CYGHWR_HAL_IMX_GPIO_INTCLR( pin )` can be used to explicitly clear the interrupt status for a specific GPIO pin.

Name

Peripheral Clock Control — Description

Synopsis

#include <cyg/hal/hal_io.h>

`pin = `**`CYGHWR_HAL_IMX_CLOCK`**`(`*`cgcr`*`, `*`device`*`);`

`CYGHWR_HAL_IMX_CLOCK_ENABLE`` (`*`desc`*`);`

`CYGHWR_HAL_IMX_CLOCK_DISABLE`` (`*`desc`*`);`

Description

The i.MXxx HAL provides a number of macros to support the management of peripheral clocks. The macro `CYGHWR_HAL_IMX_CLOCK( cgcr, device )` encodes a clock control descriptor into a 32 bit value. The arguments are the clock group register name, and the name of the clock to be controlled.

The remaining functions all take a peripheral clock descriptor as an argument. `CYGHWR_HAL_IMX_CLOCK_ENABLE( desc )` enables the given clock. Likewise `CYGHWR_HAL_IMX_CLOCK_DISABLE( desc )` disables the clock.

The following examples show how a clock descriptor may be defined.

```
// EPIT input clock and EPIT1 device clock
#define CYGHWR_HAL_IMX_EPIT_PER_CLOCK   CYGHWR_HAL_IMX_CLOCK( CGCR0, PER_EPIT )
#define CYGHWR_HAL_IMX_EPIT1_CLOCK      CYGHWR_HAL_IMX_CLOCK( CGCR1, IPG_EPIT1 )

// UART1 clock
#define CYGHWR_HAL_IMX_UART1_CLOCK      CYGHWR_HAL_IMX_CLOCK( CGCR2, IPG_UART1 )
```

Chapter 267. Freescale MCIMX25WPDK Board Support

Name

eCos Support for the Freescale MCIMX25WPDK Board — Overview

Description

This document covers the configuration and usage of eCos and RedBoot on the Freescale MCIMX25WPDK board. The board consists of a base board, a CPU module and a debug board. The CPU module contains the i.MX25 processor, RAM, NAND flash and a connector carrying most on-chip peripherals to the Personality board. The Personality board contains the CPU module, 2Mbytes of serial NOR flash memory on CSPI1, a DP83640 PHY, external connections for Ethernet, and the various other peripherals supported by the CPUs. The Debug board connects to the Personality boards and carries power, RS232 serial, and JTAG connectors; it also contains a second Ethernet device which is not used. An LCD is connected to the Personality board, and is supported by a frame buffer driver. eCos support for the devices and peripherals on the board and the CPU is described below.

For typical eCos development, a RedBoot image is programmed into the SPI NOR flash memory, and the board will load this image from reset. RedBoot provides gdb stub functionality so it is then possible to download and debug stand-alone and eCos applications via the gdb debugger. This can happen over either the serial line or over Ethernet.

This documentation is expected to be read in conjunction with the i.MXxx processor HAL documentation and further device support and subsystems are described and documented there.

Supported Hardware

The SPI NOR flash consists of 32 blocks of 64Ki bytes each. In a typical setup, the first four blocks are reserved for the ROM RedBoot image. The topmost block is used to manage the flash and also holds RedBoot **fconfig** values. The remaining blocks can be used by application code.

Serial support is through the `CYGPKG_IO_SERIAL_ARM_IMX` driver package which is modified by the `CYGPKG_IO_SERIAL_ARM_MCIMX25` driver package for the MCIMX25WPDK. These packages can support all the serial devices on the i.MX25. However, this board only has UART1 connected to an external connector which this HAL indicates by implementing the `CYGINT_HAL_IMX_UART1` interface. This serial channel is used by RedBoot for communication with the host. If this device is needed by the application, either directly or via the serial driver, then it cannot also be used for RedBoot communication. Another communication channel such as Ethernet should be used instead. The serial driver package is loaded automatically when configuring for the `mcimx25x` target.

There is an Ethernet driver `CYGPKG_DEVS_ETH_FREESCALE_ENET` for the on-chip FEC Ethernet device. The platform HAL package is responsible for configuring this generic driver to the hardware. This driver is also loaded automatically when configuring for the mcimx25x board.

There is a watchdog driver `CYGPKG_DEVICES_WATCHDOG_ARM_IMX`. This driver is also loaded automatically when configuring for the board.

The platform HAL provides definitions to allow access to devices on the SPI bus. The HAL provides information to the more general CSPI driver (`CYGPKG_DEVS_SPI_ARM_CSPI`) which in turn provides the underlying implementation for the SPI API layer in the `CYGPKG_IO_SPI` package. All these packages are automatically loaded when configuring for the board.

Furthermore, the platform HAL package contains support for the SPI NOR flash. The HAL support integrates with the `CYGPKG_DEVS_FLASH_SPI_M25PXX` package as well as the above SPI packages. That package is automatically loaded when configuring for the target. This driver is capable of supporting the JFFS2 filesystem.

I²C support is provided by the `CYGPKG_DEVS_I2C_FREESCALE_I2C` package. Only I²C bus 1 is directly supported, to which are attached an EEPROM and power management chips. There is a test program to test access to the former and the latter are accessed to enable the Ethernet PHY during initialization.

ADC support is provided by the `CYGPKG_DEVS_ADC_ARM_TSC` package. Only ADC inputs `INAUX0`, `INAUX1` and `INAUX2` are supported at present.

LCD support is provided by the `CYGPKG_DEVS_FRAMEBUF_ARM_IMX` package. This supports a fixed 640x480 16 bits per pixel display mode.

In general, devices (Caches, GPIO, UARTs) are initialized only as far as is necessary for eCos to run. Other devices (SPI, Watchdog etc.) are not touched unless the appropriate driver is loaded, although in some cases, the HAL boot sequence will set up the appropriate power control and pin multiplexing configuration. Devices not used by eCos (MMC/SD, Audio, CAN etc.) are not touched at all.

Tools

The board support is intended to work with GNU tools configured for an arm-eabi target. The original port was undertaken using arm-eabi-gcc version 4.7.3, arm-eabi-gdb version 7.2, and binutils version 2.23.2.

Name

Setup — Preparing the board for eCos Development

Overview

In a typical development environment, the board boots from the SPI NOR and runs the RedBoot ROM monitor from SDRAM. eCos applications are configured for RAM startup and then downloaded and run on the board via the debugger **arm-eabi-gdb**. Preparing the board therefore usually involves programming a suitable RedBoot image into flash memory.

The following RedBoot configurations are supported:

Configuration	Description	Use	File
ROM	RedBoot loaded from SPI NOR flash to SDRAM	`redboot_ROM.ecm`	`redboot_ROM.bin`

For serial communications, all versions run with 8 bits, no parity, and 1 stop bit at 115200 baud. RedBoot also supports Ethernet communication and flash management.

Note that the use of the term ROM for the initial RedBoot configuration is a historical accident. RedBoot actually runs from SDRAM after being loaded there from NOR flash by the on-chip boot loader. The use of ROM for this configuration is intended to indicate that it initializes the microprocessor and board peripherals, as opposed to the RAM configuration which assumes that this has already been done.

Initial Installation

The board comes with Windows CE installed by default. The booting mechanism is that the on-chip boot loader examines a set of switches and based on their settings, reads a header, initialization data and an application from the selected external device. It then executes the application at a given address. Possible boot devices include NAND flash, SPI NOR flash and an MMC/SD card. For eCos we use the SPI NOR flash.

To write RedBoot to the SPI NOR flash we need to run RedBoot and then use that to download and program an image to SPI flash. There are two ways to do this: use a JTAG debugger to load and run RedBoot under GDB, or use an MMC/SD card to load it.

Note

The RedBoot image files referred to in the setup instructions below can be found in the `loaders/mcimx25wpdk` directory, and the PEEDI JTAG debugger configuration file and ECM files for rebuilding RedBoot can be found in the `packages/hal/arm/arm9/mcimx25x/<version>/misc` directory of the eCosPro installation. On a Linux host eCosPro is typically installed in the `/opt/ecospro/ecospro-<version>` sub-directory. On a Windows host eCosPro will typically be installed in the `C:\eCosPro\ecos-<version>` sub-directory.

The following section describes this process using the Ronetix PEEDI; other JTAG emulators will have similar steps. The next section describes how to do this using an MMC/SD card. A third common section then describes how to use that RedBoot to initialize the SPI NOR flash and program RedBoot into it.

Programming RedBoot into NOR flash using the PEEDI

The following gives the steps needed load RedBoot using the PEEDI.

1. Set up the PEEDI as described in the Ronetix documentation. The `peedi.mcimx25x.cfg` file should be used to setup and configure the hardware.
2. Connect a null-modem serial cable between the serial port of the Debug board and a serial port on a convenient host. Run a terminal emulator (TeraTerm or minicom) at 115200 baud.

3. Attach an Ethernet cable to the Personality board FEC socket, not the Debug board Ethernet socket. Make sure this is on the same network as the PEEDI and host.

4. Copy `redboot_ROM.img` to to a TFTP server on the same network as the PEEDI and MCIMX25WPDK board.

5. Apply power to the PEEDI.

6. Connect a telnet session to the PEEDI and power up the MCIMX25WPDK board. You should see something similar to the following output:

```
++ info: RESET and TRST asserted
++ info: TRST released
++ info: BYPASS check passed, 4 TAP controller(s) detected
++ info: TAP 0 : invalid IDCODE = 0x0
++ info: TAP 1 : invalid IDCODE = 0x0
++ info: TAP 2 : IDCODE = 0x07926041, ARM926E -> CORE0
++ info: TAP 3 : IDCODE = 0x1B900F0F, ARM ETB
++ info: RESET released
++ info: core 0: initialized

mx25>
```

7. Issue the following command, substituting your own TFTP server address:

```
mx25>> mem load tftp://10.0.1.1/redboot_ROM.img bin 0x80000000
++ info: Loading image file: tftp://10.0.1.1/redboot_ROM.img
++ info: At absolute address: 0x80000000
loading at 0x80000000
loading at 0x80008000
loading at 0x80010000
loading at 0x80018000
loading at 0x80020000

++ info: successfully loaded 149.00 KB in 0.59 sec
mx25>
```

8. Now set the CPSR to disable thumb mode (which the on-chip boot loader uses) and issue the go command:

```
       mx25> set cpsr 0xd3
       mx25> go 0x80008000
```

   You should see output similar to the following on the board serial line:

```
+**Warning** FLASH configuration checksum error or invalid key
Use 'fconfig -i' to [re]initialize database
Ethernet eth0: MAC address 0e:00:00:ea:18:f0
IP: 10.0.2.4/255.0.0.0, Gateway: 10.0.0.3
Default server: 10.0.1.1

RedBoot(tm) bootstrap and debug environment [ROM]
Non-certified release, version UNKNOWN - built 16:55:09, Apr 14 2014

Copyright (C) 2000-2009 Free Software Foundation, Inc.
Copyright (C) 2003-2014 eCosCentric Limited
RedBoot is free software, covered by the eCos license, derived from the
GNU General Public License. You are welcome to change it and/or distribute
copies of it under certain conditions. Under the license terms, RedBoot's
source code and full license terms must have been made available to you.
Redboot comes with ABSOLUTELY NO WARRANTY.

Platform: Freescale MCIMX25WPDK (ARM9)
RAM: 0x80000000-0x84000000 [0x800318f8-0x83fed000 available]
FLASH: 0x70000000-0x701fffff, 32 x 0x10000 blocks
RedBoot>
```

Now go to this section to complete the installation.

Programming RedBoot into NOR flash using an MMC/SD card

The following gives the steps needed to load RedBoot using an MMC/SD card.

1. Connect a null-modem serial cable between the serial port of the Debug board and a serial port on a convenient host. Run a terminal emulator (for example, TeraTerm or minicom) at 115200 baud.

2. Attach an Ethernet cable to the Personality board FEC socket, not the Debug board Ethernet socket. Make sure this is on the same network as your TFTP server.

3. Copy `redboot_ROM.img` to the TFTP server.

4. Locate an SD card whose existing content can be lost.

5. Using a suitable tool such as **dd** on Linux or an equivalent on Windows (e.g. dd for windows or win32 diskimager), write `redboot_ROM.img` into the first sectors of the SD card. For example, on Linux:

```
       $ sudo dd if=/path/to/redboot_ROM.img of=/dev/sdX
```

6. Set the Personality board switches as follows: SW21 all off except switches 1 and 2; SW22 all off. On the Debug board SW5..SW10 should all be off and SW4 all off except 1 and 8 (these are the default settings).

7. Insert the SD card into the SD card socket on the Personality board and apply power to the board. You should see output similar to the following on the board serial line:

```
+**Warning** FLASH configuration checksum error or invalid key
Use 'fconfig -i' to [re]initialize database
Ethernet eth0: MAC address 0e:00:00:ea:18:f0
IP: 10.0.2.4/255.0.0.0, Gateway: 10.0.0.3
Default server: 10.0.1.1

RedBoot(tm) bootstrap and debug environment [ROM]
Non-certified release, version UNKNOWN - built 16:55:09, Apr 14 2014

Copyright (C) 2000-2009 Free Software Foundation, Inc.
Copyright (C) 2003-2014 eCosCentric Limited
RedBoot is free software, covered by the eCos license, derived from the
GNU General Public License. You are welcome to change it and/or distribute
copies of it under certain conditions. Under the license terms, RedBoot's
source code and full license terms must have been made available to you.
Redboot comes with ABSOLUTELY NO WARRANTY.

Platform: Freescale MCIMX25WPDK (ARM9)
RAM: 0x80000000-0x84000000 [0x800318f8-0x83fed000 available]
FLASH: 0x70000000-0x701fffff, 32 x 0x10000 blocks
RedBoot>
```

Now follow the instructions in the following section to complete the RedBoot installation.

Initialize and Install RedBoot

With a RedBoot now running on the board, the following steps are common between the two methods.

1. Run the following command to initialize RedBoot's flash file system and flash configuration:

```
RedBoot> fis init
About to initialize [format] FLASH image system - continue (y/n)? y
*** Initialize FLASH Image System
... Erase from 0x701f0000-0x701fffff: .
... Program from 0x83ff0000-0x84000000 to 0x701f0000: .
RedBoot> fconfig -i
```

```
Initialize non-volatile configuration - continue (y/n)? y
Run script at boot: false
Use BOOTP for network configuration: true
Default server IP address: 10.0.1.1
Console baud rate: 115200
DNS domain name: example.com
DNS server IP address: 10.0.0.5
Network hardware address [MAC] for eth0: 0xec:0x05:0x00:0x00:0x02:0x24
GDB connection port: 9000
Force console for special debug messages: false
Network debug at boot time: false
Default network device: enet0_eth
Update RedBoot non-volatile configuration - continue (y/n)? y
... Erase from 0x701f0000-0x701fffff: .
... Program from 0x83ff0000-0x84000000 to 0x701f0000: .
RedBoot>
```

Substitute your own IP addresses and domain name in place of those above. If multiple boards are to be run on the same network, also ensure that they have unique MAC addresses.

2. We now need to download a copy of RedBoot and program it into the flash. Give the following command to RedBoot, substituting in your tftp server's IP address:

```
RedBoot> load -h 10.0.1.1 -r -b %{freememlo} redboot_ROM.img
Using default protocol (TFTP)
Raw file loaded 0x80031c00-0x80056fff, assumed entry at 0x80031c00
RedBoot>
```

3. Now program the RedBoot image to flash:

```
RedBoot> fis create RedBoot
An image named 'RedBoot' exists - continue (y/n)? y
... Erase from 0x70000000-0x7003ffff: ....
... Program from 0x80031c00-0x80057000 to 0x70000000: ...
... Erase from 0x701f0000-0x701fffff: .
... Program from 0x83ff0000-0x84000000 to 0x701f0000: .
RedBoot>
```

4. RedBoot installation is now complete. It is now necessary to set the board switches to select SPI NOR flash boot. Set the Personality board switches as follows: SW21 all off except switches 1, 2, 3, 4 and 7; SW22 all off. On the Debug board SW5..SW10 should all be off and SW4 all off except 1 and 8 (these are the default settings). If you have already booted via the MMC/SD card, you just need to move SW21 switches 3, 4 and 7 to on.

5. Disconnect the PEEDI or pop the SD card from the socket and power cycle the board. You should see output similar to the following on the serial line:

```
+Ethernet eth0: MAC address ec:05:00:00:02:24
IP: 10.0.2.9/255.0.0.0, Gateway: 10.0.0.3
Default server: 10.0.1.1

RedBoot(tm) bootstrap and debug environment [ROM]
Non-certified release, version UNKNOWN - built 16:55:09, Apr 14 2014

Copyright (C) 2000-2009 Free Software Foundation, Inc.
Copyright (C) 2003-2014 eCosCentric Limited
RedBoot is free software, covered by the eCos license, derived from the
GNU General Public License. You are welcome to change it and/or distribute
copies of it under certain conditions. Under the license terms, RedBoot's
source code and full license terms must have been made available to you.
Redboot comes with ABSOLUTELY NO WARRANTY.

Platform: Freescale MCIMX25WPDK (ARM9)
RAM: 0x80000000-0x84000000 [0x800318f8-0x83fed000 available]
FLASH: 0x70000000-0x701fffff, 32 x 0x10000 blocks
RedBoot>
```

If it proves necessary to re-install RedBoot, this may be achieved by repeating the **load** and **fis create** parts of the above process. It is not necessary to use a PEEDI, or an SD card, or to reinitialize the FIS and fconfig.

Rebuilding RedBoot

Should it prove necessary to rebuild a RedBoot binary, this is done most conveniently at the command line. The steps needed to rebuild the ROM version of RedBoot for the MCIMX25WPDK are:

```
$ mkdir redboot_mcimx25x_rom
$ cd redboot_mcimx25x_rom
$ ecosconfig new mcimx25x redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/arm/arm9/mcimx25x/VERSION/misc/redboot_ROM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

At the end of the build the `install/bin` subdirectory should contain the files `redboot.img`. `redboot.img` is a binary file that includes the headers needed by the on-chip boot loader to load RedBoot successfully.

Note

The `flashimg_imx` host executable provided within your eCosPro installation is required on the build host to wrap the RedBoot binary image into an image that can be loaded by the on-chip boot loader. This executable must be on your path when you build RedBoot and will normally be copied into the `ecospro/ecoshosttools/bin` sub-directory by the eCosPro installer from the `ecospro/ecos-<version>/host/bin-<hostos>` sub-directory of your eCosPro installation. The `ecospro/ecoshosttools/bin` sub-directory will be on your path if you use the eCos GUI configuration tool or the eCos CLI Shell environment provided by `ecosprofileenv`.

Name

Configuration — Platform-specific Configuration Options

Overview

The MCIMX25X platform HAL package is loaded automatically when eCos is configured for the `mcimx25x` target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The platform HAL package supports three separate startup types:

RAM
: This is the startup type which is normally used during application development. The board has RedBoot programmed into flash and boots into that initially. arm-eabi-gdb is then used to load a RAM startup application into memory and debug it. RAM applications can also be downloaded to the board, programmed into flash and then run from the RedBoot prompt. It is assumed that the hardware has already been initialized by RedBoot. By default the application will use the eCos virtual vectors mechanism to obtain certain services from RedBoot, including diagnostic output.

ROM
: This startup type can be used for finished applications which will be programmed into Flash. The application will be self-contained with no dependencies on services provided by other software. An initialization table in the image header and eCos startup code will perform all necessary hardware initialization.

JTAG
: This startup type can be used for finished applications which will be loaded via JTAG. The application will be self-contained with no dependencies on services provided by other software. The JTAG init file plus eCos startup code will perform all necessary hardware initialization.

RedBoot and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building RedBoot.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then the serial port will be claimed for HAL diagnostics.

Flash Driver

The board contains an 2Mbyte Macronix MX25L1605D SPI serial NOR flash device. The `CYGPKG_DEVS_FLASH_SPI_M25PXX` package contains all the code necessary to support this part and the platform HAL package contains definitions that customize the driver to the board. This driver is not active until the generic Flash support package, `CYGPKG_IO_FLASH`, is included in the configuration.

This driver is capable of supporting the JFFS2 filesystem. However, note that the SPI interface means that this file system has reduced bandwidth and increased latency compared with other implementations. All that is required to enable the support is to include the filesystem (`CYGPKG_FS_JFFS2`) and any of its package dependencies (including `CYGPKG_IO_FILEIO` and `CYGPKG_LINUX_COMPAT`) together with the flash infrastructure (`CYGPKG_IO_FLASH`).

Ethernet Driver

The board uses the internal FEC Ethernet device attached to an external Texas Instruments DP83640 PHY. The `CYGPKG_DEVS_ETH_FREESCALE_ENET` package contains all the code necessary to support this device and the platform HAL package contains definitions that customize the driver to the board. This driver is not active until the generic Ethernet support package, `CYGPKG_IO_ETH_DRIVERS`, is included in the configuration.

Watchdog Driver

The board uses the i.MXxx internal watchdog. The `CYGPKG_DEVICES_WATCHDOG_ARM_IMX` package contains all the code necessary to support this device. Within that package the `CYGNUM_DEVS_WATCHDOG_ARM_IMX_DESIRED_TIMEOUT_MS` configuration option controls the watchdog timeout, and by default will force a reset of the board upon timeout. This driver is not active until the generic watchdog device support package, `CYGPKG_IO_WATCHDOG`, is included in the configuration.

UART Serial Driver

The board uses the i.MXxx internal UART serial support as described in the i.MXxx processor HAL documentation. Only one serial connector is available on the board, which is connected to UART1. This connector has the RTS/CTS hardware flow control lines connected in addition to the data lines.

ADC Driver

ADC support is provided by the `CYGPKG_DEVS_ADC_ARM_TSC` package. Only ADC inputs `INAUX0`, `INAUX1` and `INAUX2` are supported at present. In addition to the TSC ADC device, this driver uses GPT1 to provide the sample rate clock. Application code should avoid using this timer if the ADC is to be used. This driver is only active if the generic ADC support package, `CYGPKG_IO_ADC`, is included in the configuration.

LCD Driver

LCD support is provided by the `CYGPKG_DEVS_FRAMEBUF_ARM_IMX` package. It supports a single frame buffer format, 640 by 480 pixels, 16 bits per pixel true colour. The frame buffer is placed at a fixed address in memory at 0x83F00000. This driver is only active if the generic framebuffer support package, `CYGPKG_IO_FRAMEBUF`, is included in the configuration.

Compiler Flags

The platform HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos. There are just three flags specific to this port:

`-mcpu=arm926ej-s`

The arm-eabi-gcc compiler supports many variants of the ARM architecture. A `-m` option should be used to select the specific variant in use, and with current tools `-mcpu=arm926ej-s` is the correct option for the ARM926EJ-S CPU in the i.MX25.

`-mthumb`

The arm-eabi-gcc compiler will compile C and C++ files into the Thumb instruction set when this option is used. The best way to build eCos in Thumb mode is to enable the configuration option `CYGHWR_THUMB`.

`-mthumb-interwork`

This option allows programs to be created that mix ARM and Thumb instruction sets. Without this option, some memory can be saved. This option should be used if -mthumb is used. The best way to build eCos with Thumb interworking is to enable the configuration option `CYGBLD_ARM_ENABLE_THUMB_INTERWORK`.

Name

JTAG support — Usage

Use of JTAG for debugging

JTAG can be used to single-step and debug loaded applications, including RedBoot. Only JTAG and ROM configuration applications should be debugged using JTAG, RAM applications assume the presence of RedBoot.

Ronetix PEEDI notes

On the Ronetix PEEDI, the `peedi.mcimx25x.cfg` file should be used to setup and configure the hardware to an appropriate state to load programs. This includes setting up the PLLs and SDRAM controller.

The `peedi.mcimx25x.cfg` file also contains an option to define whether hardware or software breakpoints are used by default, using the `CORE0_BREAKMODE` directive in the `[PLATFORM_ARM]` section. Edit this file if you wish to use hardware break points, and remember to restart the PEEDI to make the changes take effect.

On the PEEDI, debugging can be performed either via the telnet interface or using **arm-eabi-gdb** and the GDB interface. In the case of the latter, **arm-eabi-gdb** needs to connect to TCP port 2000 on the PEEDI's IP address. For example:

```
(gdb) target remote 111.222.333.444:2000
```

By default when the PEEDI is powered up, the target will always run the initialization section of the `peedi.mcimx25x.cfg` file, and halts the target. This behaviour is repeated with the **reset** command.

If the board is reset (either with the '**reset**', or by pressing the reset button) and the '**go**' command is then given, then the board will boot as normal.

Consult the PEEDI documentation for information on other features.

Running JTAG applications

Applications configured for either JTAG or ROM startup can be run directly under JTAG. Once loaded and running via JTAG, HAL diagnostic output will appear by default on the serial debug port. There is in fact minimal difference between ROM and JTAG applications and these can mostly be used interchangeably.

Installing user applications into Flash with JTAG

If you wish to install a ROM startup application into Flash to be automatically booted, you can follow a similar procedure to installing RedBoot into Flash. However before you can do so, you must first prepend a header to your application image in order for the on-chip boot loader to recognize it as a valid application.

You will need to locate the program **flashimg_imx** supplied with the eCosPro installation and generate a binary image of your program using the **arm-eabi-objcopy** command. The following gives an example simplified command sequence which can be run at a command shell prompt:

```
arm-eabi-objcopy -O binary myapp myapp.bin
flashimg_imx myapp.bin myapp.img
```

You will need to substitute your own paths and filenames where applicable.

Once you have the `.img` file, you can follow the same process as installing RedBoot via JTAG. Alternatively, you can write it to an SD card and boot it directly after changing the board switches.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the board hardware, and should be read in conjunction with that specification. The platform HAL package complements the ARM architectural HAL, the ARM9 variant HAL and the i.MXxx processor HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset, the HAL will initialize or reinitialize most of the on-chip peripherals. There is an exception for RAM startup applications which depend on a ROM monitor for certain services.

For ROM startup, the HAL will perform additional initialization. This is all done in the `PLATFORM_SETUP1` macro in the assembler header file `hal_platform_setup.h`.

Linker Scripts and Memory Maps

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

SDRAM

This is located at address 0x80000000 of the physical memory space and is 64MiB in size. The HAL configures the MMU to retain the SDRAM at virtual address 0x80000000, but in order to assign hardware exception vectors vectors at address 0x00000000, the HAL also uses the MMU to create an uncached clone of this memory at virtual address 0x00000000. The first 32 bytes are used for hardware exception vectors. The next 32 bytes are used for the VSR table and the next 256 bytes are normally used for the eCos virtual vectors, allowing RAM-based applications to use services provided by the ROM monitor. Memory is required for the MMU tables, and must be aligned on a 16Kbyte boundary. These therefore occupy memory from 0x4000 to 0x8000. 1MiB at 0x83f00000 is reserved for the LCD display frame buffer and is identity mapped uncached. For ROM startup, all remaining SDRAM is available. For RAM startup, available RAM starts at virtual location 0x80100000, with the bottom 1MB reserved for use by RedBoot. ROM applications are relocated starting at 0x80008000.

On-chip SRAM

There is 128KiB of SRAM which is identity mapped uncached at 0x78000000. eCos makes no current use of this memory, so it is available for application use.

SPI NOR Flash

SPI NOR flash media can only be accessed with the Flash API. For the purposes of this API a placeholder address range has been allocated as if the Flash is present at this address. The base of this address range is 0x70000000. This reserved range is not real memory and any attempt to access it directly by the processor other than via the Flash API will result in a memory address exception.

On-chip Peripheral Registers

These are located at various addresses in the physical memory space. When the MMU is enabled, it sets up a direct, uncached, unbuffered mapping so that these registers remain accessible at their physical locations.

Off-chip Peripherals

eCos uses the SDRAM, Ethernet PHY, and SPI flash and I^{2}C power controllers on the board. eCos does not currently make any use of any other off-chip peripherals present on this board.

Real-time characterization

The tm_basic kernel test gives statistics gathered about the real-time characterization and performance of the kernel. The sample output is shown here for information. The test was built in ARM mode, which provides better performance than Thumb mode.

Example 267.1. mcimx25x Real-time characterization

```
             Startup, main thrd : stack used   380 size  1792
             Startup :  Interrupt stack used  4096 size  4096
             Startup : Idlethread stack used    96 size  1280

eCos Kernel Timings
Notes: all times are in microseconds (.000001) unless otherwise stated

Reading the hardware clock takes 0 'ticks' overhead
... this value will be factored out of all other measurements
Clock interrupt took    3.09 microseconds (3 raw clock ticks)

Testing parameters:
   Clock samples:            32
   Threads:                  64
   Thread switches:         128
   Mutexes:                  32
   Mailboxes:                32
   Semaphores:               32
   Scheduler operations:    128
   Counters:                 32
   Flags:                    32
   Alarms:                   32
   Stack Size:             1088


                                 Confidence
     Ave     Min     Max     Var  Ave  Min  Function
  ======  ======  ======  ====== ========== ========
    4.05    3.00    8.00    0.42   65%  17% Create thread
    0.47    0.00    2.00    0.51   54%  54% Yield thread [all suspended]
    0.72    0.00    2.00    0.43   68%  29% Suspend [suspended] thread
    0.47    0.00    1.00    0.50   53%  53% Resume thread
    0.75    0.00    2.00    0.40   71%  26% Set priority
    0.23    0.00    1.00    0.36   76%  76% Get priority
    1.52    1.00    6.00    0.56   98%  54% Kill [suspended] thread
    0.47    0.00    1.00    0.50   53%  53% Yield [no other] thread
    0.83    0.00    2.00    0.34   76%  20% Resume [suspended low prio] thread
    0.53    0.00    2.00    0.51   50%  48% Resume [runnable low prio] thread
    0.58    0.00    1.00    0.49   57%  42% Suspend [runnable] thread
    0.47    0.00    2.00    0.51   54%  54% Yield [only low prio] thread
    0.45    0.00    1.00    0.50   54%  54% Suspend [runnable->not runnable]
    1.34    1.00    3.00    0.46   67%  67% Kill [runnable] thread
    1.22    1.00    4.00    0.36   81%  81% Destroy [dead] thread
    2.03    1.00    4.00    0.21   82%   7% Destroy [runnable] thread
    3.03    2.00    7.00    0.46   60%  20% Resume [high priority] thread
    0.75    0.00    4.00    0.41   71%  27% Thread switch

    0.16    0.00    1.00    0.26   84%  84% Scheduler lock
    0.39    0.00    1.00    0.48   60%  60% Scheduler unlock [0 threads]
    0.40    0.00    1.00    0.48   60%  60% Scheduler unlock [1 suspended]
    0.40    0.00    1.00    0.48   60%  60% Scheduler unlock [many suspended]
    0.39    0.00    1.00    0.48   60%  60% Scheduler unlock [many low prio]

    0.28    0.00    1.00    0.40   71%  71% Init mutex
    0.63    0.00    2.00    0.51   56%  40% Lock [unlocked] mutex
    0.69    0.00    4.00    0.56   56%  40% Unlock [locked] mutex
    0.50    0.00    1.00    0.50  100%  50% Trylock [unlocked] mutex
    0.41    0.00    1.00    0.48   59%  59% Trylock [locked] mutex
    0.22    0.00    1.00    0.34   78%  78% Destroy mutex
    2.94    2.00    6.00    0.29   81%  15% Unlock/Lock mutex

    0.47    0.00    1.00    0.50   53%  53% Create mbox
    0.47    0.00    1.00    0.50   53%  53% Peek [empty] mbox
    0.63    0.00    2.00    0.51   56%  40% Put [first] mbox
    0.13    0.00    1.00    0.22   87%  87% Peek [1 msg] mbox
    0.53    0.00    2.00    0.53   96%  50% Put [second] mbox
    0.19    0.00    1.00    0.30   81%  81% Peek [2 msgs] mbox
    0.63    0.00    2.00    0.51   56%  40% Get [first] mbox
    0.66    0.00    1.00    0.45   65%  34% Get [second] mbox
    0.50    0.00    2.00    0.53   96%  53% Tryput [first] mbox
    0.47    0.00    1.00    0.50   53%  53% Peek item [non-empty] mbox
    0.53    0.00    2.00    0.53   96%  50% Tryget [non-empty] mbox
    0.50    0.00    1.00    0.50  100%  50% Peek item [empty] mbox
    0.47    0.00    1.00    0.50   53%  53% Tryget [empty] mbox
    0.19    0.00    1.00    0.30   81%  81% Waiting to get mbox
    0.16    0.00    1.00    0.26   84%  84% Waiting to put mbox
    0.25    0.00    1.00    0.38   75%  75% Delete mbox
    1.19    1.00    5.00    0.34   90%  90% Put/Get mbox

    0.19    0.00    1.00    0.30   81%  81% Init semaphore
    0.47    0.00    1.00    0.50   53%  53% Post [0] semaphore
    0.47    0.00    1.00    0.50   53%  53% Wait [1] semaphore
    0.47    0.00    2.00    0.53   56%  56% Trywait [0] semaphore
    0.47    0.00    1.00    0.50   53%  53% Trywait [1] semaphore
    0.19    0.00    1.00    0.30   81%  81% Peek semaphore
    0.16    0.00    1.00    0.26   84%  84% Destroy semaphore
    1.06    1.00    3.00    0.12   96%  96% Post/Wait semaphore

    0.53    0.00    1.00    0.50   53%  46% Create counter
    0.44    0.00    1.00    0.49   56%  56% Get counter value
    0.25    0.00    1.00    0.38   75%  75% Set counter value
    0.56    0.00    1.00    0.49   56%  43% Tick counter
    0.22    0.00    1.00    0.34   78%  78% Delete counter

    0.22    0.00    1.00    0.34   78%  78% Init flag
    0.56    0.00    3.00    0.56   96%  50% Destroy flag
    0.44    0.00    1.00    0.49   56%  56% Mask bits in flag
    0.50    0.00    2.00    0.53   96%  53% Set bits in flag [no waiters]
    0.59    0.00    3.00    0.56   50%  46% Wait for flag [AND]
    0.63    0.00    1.00    0.47   62%  37% Wait for flag [OR]
    0.53    0.00    1.00    0.50   53%  46% Wait for flag [AND/CLR]
    0.56    0.00    1.00    0.49   56%  43% Wait for flag [OR/CLR]
    0.09    0.00    1.00    0.17   90%  90% Peek on flag

    0.66    0.00    2.00    0.49   59%  37% Create alarm
    0.88    0.00    3.00    0.33   78%  18% Initialize alarm
    0.50    0.00    2.00    0.53   96%  53% Disable alarm
    0.72    0.00    2.00    0.45   65%  31% Enable alarm
    0.50    0.00    1.00    0.50  100%  50% Delete alarm
    0.56    0.00    1.00    0.49   56%  43% Tick counter [1 alarm]
    2.25    2.00    3.00    0.38   75%  75% Tick counter [many alarms]
    0.88    0.00    2.00    0.27   81%  15% Tick & fire counter [1 alarm]
   11.78   11.00   12.00    0.34   78%  21% Tick & fire counters [>1 together]
    2.53    2.00    3.00    0.50   53%  46% Tick & fire counters [>1 separately]
    2.03    2.00    3.00    0.06   96%  96% Alarm latency [0 threads]
    2.74    2.00    3.00    0.38   74%  25% Alarm latency [2 threads]
    6.89    6.00    9.00    0.47   59%  26% Alarm latency [many threads]
    4.05    4.00    9.00    0.11   98%  98% Alarm -> thread resume latency

    1.30    1.00    4.00    0.00            Clock/interrupt latency

    1.24    1.00    5.00    0.00            Clock DSR latency

     242     172     272                    Worker thread stack used (stack size 1088)
            All done, main thrd : stack used   956 size  1792
            All done :  Interrupt stack used   156 size  4096
            All done : Idlethread stack used   408 size  1280

Timing complete - 29810 ms total

PASS:<Basic timing OK>
EXIT:<done>
```

Other Issues

The platform HAL does not affect the implementation of other parts of the eCos HAL specification. The i.MXxx processor HAL, ARM9 variant HAL, and the ARM architectural HAL documentation should be consulted for further details.

Bonjour Conformance Test

This section just provides information regarding the specific MCIMX25WPDK platform Bonjour test. For more detail regarding Bonjour testing please reference the Bonjour Conformance Test section of the mDNS package documentation.

For the mDNS Responder package testing the following equipment was used.

- Unit-Under-Test (UUT)

  The `mcimx25wpdk` target platform being tested, executing a suitable eCosPro mDNS configuration.

- Mac Mini (OS: 10.9.3)

  Executing the BonjourConformanceTest v1.3.0 in a Terminal shell window, and the Safari v7.0.4 web-browser.

- Billion BiPAC 7800N

  10base-T (wired) Ethernet connections to the UUT and the Mac Mini.

The result for an actual BCT run using the `mdns_example` application, executing on the MCIMX25WPDK platform, is available in the `doc/bct_mcimx25wpdk_result.txt` file. This file is the original, as produced by the BCT application. For reference, the `doc/bct_mcimx25wpdk_terminal.txt` contains the execution output captured from the MacOS Terminal window. Both of these files can be found in the mcimx25x HAL documentation directory: `packages/hal/arm/arm9/mcimx25x/<version>/doc`

The following is a listing of the `doc/bct_mcimx25wpdk_result.txt` file.

```
Bonjour Conformance Test Version 1.3.0
Started   Mon Jun  9 09:57:26 2014
Completed Mon Jun  9 13:19:23 2014


Link-Local Address Allocation
----------------------------
    PASSED: INITIAL PROBING
    PASSED: PROBING: RATE LIMITING
    PASSED: PROBING: CONFLICTING SIMULTANEOUS PROBES
    PASSED: PROBING: PROBE DENIALS
    PASSED: PROBING COMPLETION
    WARNING: SUBSEQUENT CONFLICTS: RE-PROBE AFTER FIRST CONFLICT
    WARNING: SUBSEQUENT CONFLICTS: RE-PROBE AFTER FIRST CONFLICT
    WARNING: SUBSEQUENT CONFLICTS: RE-PROBE AFTER FIRST CONFLICT
    WARNING: SUBSEQUENT CONFLICTS: RE-PROBE AFTER FIRST CONFLICT
    WARNING: SUBSEQUENT CONFLICTS: RE-PROBE AFTER FIRST CONFLICT
    WARNING: SUBSEQUENT CONFLICTS: RE-PROBE AFTER FIRST CONFLICT
    WARNING: SUBSEQUENT CONFLICTS: RE-PROBE AFTER FIRST CONFLICT
    WARNING: SUBSEQUENT CONFLICTS: RE-PROBE AFTER FIRST CONFLICT
    WARNING: SUBSEQUENT CONFLICTS: RE-PROBE AFTER FIRST CONFLICT
    PASSED: SUBSEQUENT CONFLICTS
    PASSED: HOT-PLUG: USE OF PREVIOUS ADDRESS AS FIRST PROBE CANDIDATE
    PASSED: CABLE CHANGE HANDLING
    PASSED: PREMATURE MDNS PROBING
PASSED with 9 warning(s).

Multicast DNS
-----------------------------
    PASSED: INITIAL PROBING
    PASSED: PROBING: SIMULTANEOUS PROBE CONFLICT
    PASSED: PROBING: RATE LIMITING
    PASSED: PROBING: PROBE DENIALS
    PASSED: WINNING SIMULTANEOUS PROBES - ANNOUNCEMENTS
    PASSED: WINNING SIMULTANEOUS PROBES: WINNING SIMULTANEOUS PROBES
    PASSED: SRV PROBING/ANNOUNCEMENTS
    PASSED: SUBSEQUENT CONFLICT - ANNOUNCEMENTS
    PASSED: SUBSEQUENT CONFLICT - A
    PASSED: SUBSEQUENT CONFLICT - ANNOUNCEMENTS
    PASSED: SUBSEQUENT CONFLICT - SRV
    PASSED: SIMPLE REPLY VERIFICATION
    PASSED: SHARED REPLY TIMING - UNIFORM RANDOM REPLY TIME DISTRIBUTION
    PASSED: SHARED REPLY TIMING
    PASSED: MULTIPLE QUESTIONS - SHARED REPLY TIMING - UNIFORM RANDOM REPLY TIME DISTRIBUTION
    PASSED: MULTIPLE QUESTIONS - SHARED REPLY TIMING
    PASSED: REPLY AGGREGATION
    PASSED: MANUAL NAME CHANGE - ANNOUNCEMENTS
    PASSED: HOT-PLUGGING: INITIAL PROBING
    PASSED: HOT-PLUGGING: PROBING: SIMULTANEOUS PROBE CONFLICT
    PASSED: HOT-PLUGGING: PROBING: RATE LIMITING
    PASSED: HOT-PLUGGING: PROBING: PROBE DENIALS
    PASSED: HOT-PLUGGING: WINNING SIMULTANEOUS PROBES - ANNOUNCEMENTS
    PASSED: HOT-PLUGGING: WINNING SIMULTANEOUS PROBES: WINNING SIMULTANEOUS PROBES
    PASSED: HOT-PLUGGING: SRV PROBING/ANNOUNCEMENTS
    PASSED: HOT-PLUGGING: SUBSEQUENT CONFLICT - ANNOUNCEMENTS
    PASSED: HOT-PLUGGING: SUBSEQUENT CONFLICT - A
    PASSED: HOT-PLUGGING: SUBSEQUENT CONFLICT - ANNOUNCEMENTS
    PASSED: HOT-PLUGGING: SUBSEQUENT CONFLICT - SRV
    PASSED: HOT-PLUGGING
    PASSED: NO DUPLICATE RECORDS IN PACKETS
    PASSED: REQUIRED ADDITIONAL RECORDS IN ANSWERS
    PASSED: LEGAL CHARACTERS IN ADDRESS RECORD NAMES
    PASSED: CACHE FLUSH BIT SET IN NON-SHARED RESPONSES
    PASSED: CACHE FLUSH BIT NOT SET IN PROPOSED ANSWER OF PROBES
PASSED with 0 warning(s).

Mixed-Network Interoperability
-----------------------------
    PASSED: LINK-LOCAL TO ROUTABLE COMMUNICATION
    PASSED: ROUTABLE TO LINK-LOCAL COMMUNICATION
    PASSED: CACHE FLUSH BIT NOT SET IN UNICAST RESPONSE
    PASSED: UNICAST INTEROPERABILITY
    PASSED: CHATTINESS
    PASSED: mDNS IP TTL CHECK
    PASSED: DUPLICATE RECORDS CHECK
    PASSED: ADDITIONAL RECORDS IN ANSWER CHECK
PASSED with 0 warning(s).


******************************************************************************
    CONGRATULATIONS: You successfully passed the Bonjour Conformance test
******************************************************************************
```

Chapter 268. Intel IQ80321 Board Support

Name

eCos Support for the Intel IQ80321 Board — Overview

Description

This document covers the Intel IQ80321 development board for the IOP321 XScale device. The IQ80321 contains the IOP321 processor, 128MB of SDRAM, 8MB of Intel Strataflash memory, an Intel i82544 Ethernet MAC and a TL16C550C UART.

For typical eCos development, a RedBoot image is programmed into the flash memory, and the board will boot this image from reset. RedBoot provides gdb stub functionality so it is then possible to download and debug stand-alone and eCos applications via the gdb debugger. This can happen over either a serial line or over ethernet.

Supported Hardware

The flash memory consists of 64 blocks of 128k bytes each. In a typical setup, the first two flash blocks are used for the ROM RedBoot image. The topmost block is used to manage the flash and hold RedBoot **fconfig** values. The remaining 62 blocks between 0xF0040000 and 0xF07DFFFF can be used by application code.

There is a serial driver `CYGPKG_IO_SERIAL_GENERIC_16X5X` which supports the 16C550 UART. The `CYGPKG_IO_SERIAL_ARM_IQ80321` package provides customization of this generic driver to the IQ80321 hardware. This device can be used by RedBoot for communication with the host. If this device is needed by the application, either directly or via the serial driver, then it cannot also be used for RedBoot communication. Another communication channel such as ethernet should be used instead. The serial driver packages are loaded automatically when configuring for the IQ80321 target.

There is an ethernet driver `CYGPKG_DEVS_ETH_INTEL_I82544` for the Intel i82544 ethernet device. A second package `CYGPKG_DEVS_ETH_ARM_IQ80321` is responsible for configuring this generic driver to the IQ80321 hardware. These drivers are also loaded automatically when configuring for the IQ80321 target.

eCos manages the on-chip interrupt controller. Timer 0 is used to implement the eCos system clock and the microsecond delay function. Other on-chip devices (Caches, MEMC, INTC, PCI bridge, ATU, MMU, I²C) are initialized only as far as is necessary for eCos to run. Other devices are not touched.

Tools

The IQ80321 port is intended to work with GNU tools configured for an arm-eabi target. The original port was undertaken using arm-elf-gcc version 3.2.1, arm-elf-gdb version 5.3, and binutils version 2.13.1.

Name

Setup — Preparing the IQ80321 board for eCos Development

Overview

In a typical development environment, the IQ80321 board boots from flash into the RedBoot ROM monitor. eCos applications are configured for RAM startup and then downloaded and run on the board via the debugger **arm-eabi-gdb**. Preparing the board therefore usually involves programming a suitable RedBoot image into flash memory.

The following RedBoot configurations are supported:

Configuration	Description	Use	File
ROM	RedBoot running from flash ROM boot sector	`redboot_ROM.ecm`	`redboot_ROM.bin`
RAM	RedBoot running from RAM with RedBoot in the flash boot sector	`redboot_RAM.ecm`	`redboot_RAM.bin`

For serial communications, all versions run with 8 bits, no parity, and 1 stop bit at 38400 baud. RedBoot also supports ethernet communication and flash management.

Hardware Setup

The 80321 board is highly configurable through a number of switches and jumpers. RedBoot and eCos make some assumptions about board configuration and attention must be paid to these assumptions for reliable operation:

- The onboard ethernet and the secondary slot may be placed in a private space so that they are not seen by a PC BIOS. If the board is to be used in a PC with BIOS, then the ethernet should be placed in this private space so that RedBoot/eCos and the BIOS do not conflict.
- RedBoot assumes that the board is plugged into a PC with BIOS. This requires RedBoot to detect when the BIOS has configured the PCI-X secondary bus. If the board is placed in a backplane, RedBoot will never see the BIOS configure the secondary bus. To prevent this wait, set switch S7E1-3 to ON when using the board in a backplane.
- For the remaining switch settings, the following is a known good configuration:

S1D1	All OFF
S7E1	7 is ON, all others OFF
S8E1	2,3,5,6 are ON, all others OFF
S8E2	2,3 are ON, all others OFF
S9E1	3 is ON, all others OFF
S4D1	1,3 are ON, all others OFF
J9E1	2,3 jumpered
J9F1	2,3 jumpered
J3F1	Nothing jumpered
J3G1	2,3 jumpered
J1G2	2,3 jumpered

Initial Installation

Flash Installation

The IQ80321 is supplied with a version of RedBoot in flash. It is recommended that this version of RedBoot be replaced with one that is built from the same set of sources as the eCos system to be run on it. There are several ways of doing this.

The board manufacturer provides a DOS application which is capable of programming the flash over the PCI bus, and this is required for initial installations of RedBoot. Please see the board manual for information on using this utility. In general, the process involves programming the ROM mode RedBoot image to flash. RedBoot should be programmed to flash address 0x00000000 using the DOS utility.

If a JTAG debugger is available (such as the Abatron BDI2000) that supports programming the flash, then this may be used to install the new RedBoot. See the documentation for the JTAG device to find out how to initialize the board and program the flash. RedBoot needs to be programmed to flash address 0x00000000.

Finally, RedBoot may be installed by using the resident RedBoot. Installing RedBoot is a matter of downloading a new binary image and overwriting the existing Boot monitor ROM image. This is a two stage process, you must first download a RAM-resident version of RedBoot and then use that to download the ROM image to be programmed into the flash memory.

Connect to RedBoot as described in the IQ80321 documentation using a terminal emulator (for example **HyperTerminal** on Windows, **minicom** on Linux). You should see the RedBoot startup banner, similar to the following:

```
RedBoot(tm) bootstrap and debug environment [ROM]

Platform: IQ80321 (XScale)
Copyright (C) 2000, 2001, 2002, Free Software Foundation, Inc.

RAM: 0x00000000-0x08000000, [0x0001b068-0x07fd1000] available
FLASH: 0xf0000000 - 0xf0800000, 64 blocks of 0x00020000 bytes each.
RedBoot>
```

The RAM image can be downloaded using the following RedBoot command:

```
RedBoot> load -m ymodem
```

Use the terminal emulator's Ymodem support to send the file `redboot_RAM.srec`. This should result in something like the following output:

```
Entry point: 0x00020040, address range: 0x00020000-0x000479f8
xyzModem - CRC mode, 2(SOH)/456(STX)/0(CAN) packets, 5 retries
RedBoot>
```

Now start the RAM version of RedBoot:

```
RedBoot> go
+No network interfaces found

RedBoot(tm) bootstrap and debug environment [RAM]
Non-certified release, version v2_0_24a1 - built 12:57:43, Sep 14 2004

Platform: IQ80321 (XScale)
Copyright (C) 2000, 2001, 2002, Free Software Foundation, Inc.
Copyright (C) 2003, 2004, eCosCentric Limited

RAM: 0x00000000-0x08000000, [0x00059668-0x07dd1000] available
FLASH: 0xf0000000 - 0xf0800000, 64 blocks of 0x00020000 bytes each.
RedBoot>
```

Now, the ROM image can be downloaded using the following RedBoot command:

```
RedBoot> load -r -b %{FREEMEMLO} -m ymodem
```

Use the terminal emulator's Ymodem support to send the file `redboot_ROM.bin`. This should result in something like the following output:

```
Raw file loaded 0x0005a000-0x000819f8, assumed entry at 0x0005a000
xyzModem - CRC mode, 911(SOH)/0(STX)/0(CAN) packets, 4 retries
RedBoot>
```

Once the file has been uploaded, you can check that it has been transferred correctly using the **cksum** command. On the host (Linux or Cygwin) run the **cksum** program on the binary file:

```
$ cksum redboot_ROM.bin
140216855 116332 redboot_ROM.bin
```

In RedBoot, run the **cksum** command on the data that has just been loaded:

```
RedBoot> cksum -b %{FREEMEMLO} -l 116332
POSIX cksum = 140216855 116332 (0x085b8a17 0x0001c66c)
```

The second number in the output of the host **cksum** program is the file size, which should be used as the argument to the `-l` option in the RedBoot **cksum** command. The first numbers in each instance are the checksums, which should be equal.

If the program has downloaded successfully, then it can be programmed into the flash using the following commands:

```
RedBoot> fis init
About to initialize [format] FLASH image system - continue (y/n)?y
*** Initialize FLASH Image System
... Unlock from 0xf07e0000-0xf0800000: .
... Erase from 0xf07e0000-0xf0800000: .
... Program from 0x07cc0000-0x07d00000 at 0xf07e0000: .
... Lock from 0xf07e0000-0xf0800000: .
RedBoot> fis create -b %{FREEMEMLO} RedBoot
An image named 'RedBoot' exists - continue (y/n)?y
... Unlock from 0xf0000000-0xf0040000: ..
... Erase from 0xf0000000-0xf0040000: ..
... Program from 0x0001c000-0x0005c000 at 0xf0000000: .
... Lock from 0xf0000000-0xf0040000: .
... Unlock from 0xf07e0000-0xf0800000: .
... Erase from 0xf07e0000-0xf0800000: .
... Program from 0x07cc0000-0x07d00000 at 0xf07e0000: .
... Lock from 0xf07e0000-0xf0800000: .
RedBoot>
```

The IQ80321 board may now be reset either by cycling the power, or with the **reset** command. It should then display the startup screen for the ROM version of RedBoot:

```
+No network interfaces found

RedBoot(tm) bootstrap and debug environment [ROM]
Non-certified release, version v2_0_24a1 - built 12:59:55, Sep 14 2004

Platform: IQ80321 (XScale)
Copyright (C) 2000, 2001, 2002, Free Software Foundation, Inc.
Copyright (C) 2003, 2004, eCosCentric Limited

RAM: 0x00000000-0x08000000, [0x0001b068-0x07fd1000] available
FLASH: 0xf0000000 - 0xf0800000, 64 blocks of 0x00020000 bytes each.
RedBoot>
```

LED Codes

RedBoot uses the two digit LED display to indicate status during board initialization. Possible codes are:

LED   Actions

Power-On/Reset

88
Set the CPSR
Enable coprocessor access
Drain write and fill buffer
Setup PBIU chip selects

A1
Enable the Icache

A2
Move FLASH chip select from 0x0 to 0xF0000000
Jump to new FLASH location

A3
Setup and enable the MMU

A4
I^{2}C interface initialization

90
Wait for I^{2}C initialization to complete

91
Send address (via I^{2}C) to the DIMM

92
Wait for transmit complete

93
Read SDRAM PD data from DIMM

94
Read remainder of EEPROM data.
An error will result in one of the following
error codes on the LEDs:
77 BAD EEPROM checksum
55 I^{2}C protocol error
FF bank size error

A5
Setup DDR memory interface

A6
Enable branch target buffer
Drain the write & fill buffers
Flush Icache, Dcache and BTB
Flush instruction and data TLBs
Drain the write & fill buffers

SL
ECC Scrub Loop

SE

A7
Clean, drain, flush the main Dcache

A8
Clean, drain, flush the mini Dcache
Flush Dcache
Drain the write & fill buffers

A9
Enable ECC

AA
Save SDRAM size
Move MMU tables into RAM

AB
Clean, drain, flush the main Dcache

Clean, drain, flush the mini Dcache

Drain the write & fill buffers

AC

Set the TTB register to DRAM mmu_table

AD

Set mode to IRQ mode

A7

Move SWI & Undefined "vectors" to RAM (at 0x0)

A6

Switch to supervisor mode

A5

Move remaining "vectors" to RAM (at 0x0)

A4

Copy DATA to RAM

Initialize interrupt exception environment

Initialize stack

Clear BSS section

A3

Call platform specific hardware initialization

A2

Run through static constructors

A1

Start up the eCos kernel or RedBoot

Special RedBoot Commands

A special RedBoot command, **diag**, is used to access a set of hardware diagnostics. To access the diagnostic menu, enter **diag** at the RedBoot prompt:

```
RedBoot> diag
Entering Hardware Diagnostics - Disabling Data Cache!

  IQ80321 Hardware Tests

 1 - Memory Tests
 2 - Repeating Memory Tests
 3 - Repeat-On-Fail Memory Tests
 4 - Rotary Switch S1 Test
 5 - 7 Segment LED Tests
 6 - i82544 Ethernet Configuration
 7 - Battery Status Test
 8 - Battery Backup SDRAM Memory Test
 9 - Timer Test
10 - PCI Bus test
11 - CPU Cache Loop (No Return)
 0 - quit
Enter the menu item number (0 to quit):
```

Tests for various hardware subsystems are provided, and some tests require special hardware in order to execute normally. The Ethernet Configuration item may be used to set the board ethernet address.

Memory Tests

This test is used to test installed DDR SDRAM memory. Five different tests are run over the given address ranges. If errors are encountered, the test is aborted and information about the failure is printed. When selected, the user will be prompted to enter the base address of the test range and its size. The numbers must be in hex with no leading “0x”

```
Enter the menu item number (0 to quit): 1
```

```
Base address of memory to test (in hex): 100000

Size of memory to test (in hex): 200000

Testing memory from 0x00100000 to 0x002fffff.

Walking 1's test:
0000000100000002000000040000000800000010000000200000004000000080
0000010000000200000004000000080000001000000020000000400000008000
0001000000020000000400000008000000100000002000000040000000800000
0100000002000000040000000800000010000000200000004000000080000000
passed
32-bit address test: passed
32-bit address bar test: passed
8-bit address test: passed
Byte address bar test: passed
Memory test done.
```

Repeating Memory Tests

The repeating memory tests are exactly the same as the above memory tests, except that the tests are automatically rerun after completion. The only way out of this test is to reset the board.

Repeat-On-Fail Memory Tests

This is similar to the repeating memory tests except that when an error is found, the failing test continuously retries on the failing address.

Rotary Switch S1 Test

This tests the operation of the sixteen position rotary switch. When run, this test will display the current position of the rotary switch on the LED display. Slowly dial through each position and confirm reading on LED.

7 Segment LED Tests

This tests the operation of the seven segment displays. When run, each LED cycles through 0 through F and a decimal point.

i82544 Ethernet Configuration

This test initializes the ethernet controller‚s serial EEPROM if the current contents are invalid. In any case, this test will also allow the user to enter a six byte ethernet MAC address into the serial EEPROM.

```
Enter the menu item number (0 to quit): 6


Current MAC address: 00:80:4d:46:00:02
Enter desired MAC address: 00:80:4d:46:00:01
Writing to the Serial EEPROM... Done

******** Reset The Board To Have Changes Take Effect ********
```

Battery Status Test

This tests the current status of the battery. First, the test checks to see if the battery is installed and reports that finding. If the battery is installed, the test further determines whether the battery status is one or more of the following:

- Battery is charging.
- Battery is fully discharged.

- Battery voltage measures within normal operating range.

Battery Backup SDRAM Memory Test

This tests the battery backup of SDRAM memory. This test is a three step process:

1. Select Battery backup test from main diag menu, then write data to SDRAM.
2. Turn off power for 60 seconds, then repower the board.
3. Select Battery backup test from main diag menu, then check data that was written in step 1.

Timer Test

This tests the internal timer by printing a number of dots at one second intervals.

PCI Bus Test

This tests the secondary PCI-X bus and socket. This test requires that an IQ80310 board be plugged into the secondary slot of the IOP80321 board. The test assumes at least 32MB of installed memory on the IQ80310. That memory is mapped into the IOP80321 address space and the memory tests are run on that memory.

CPU Cache Loop

This test puts the CPU into a tight loop run entirely from the ICache. This should prevent all external bus accesses.

Rebuilding RedBoot

Should it prove necessary to rebuild a RedBoot binary, this is done most conveniently at the command line. The steps needed to rebuild the the ROM version of RedBoot for the IQ80321 are:

```
$ mkdir redboot_iq80321_rom
$ cd redboot_iq80321_rom
$ ecosconfig new iq80321 redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/arm/arm9/iq80321/current/misc/redboot_ROM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

To rebuild the RAM RedBoot:

```
$ mkdir redboot_iq80321_ram
$ cd redboot_iq80321_ram
$ ecosconfig new iq80321 redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/arm/arm9/iq80321/current/misc/redboot_RAM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

At the end of the build the `install/bin` subdirectory should contain the file `redboot.bin`. This is the case for both the above builds, take care not to mix the two files up, since programming the RAM RedBoot into the ROM will render the board unbootable.

Name

Configuration — Platform-specific Configuration Options

Overview

The IQ80321 platform HAL package is loaded automatically when eCos is configured for a `iq80321` target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The IQ80321 platform HAL package supports two separate startup types:

RAM
: This is the startup type which is normally used during application development. The board has RedBoot programmed into flash and boots into that initially. arm-eabi-gdb is then used to load a RAM startup application into memory and debug it. It is assumed that the hardware has already been initialized by RedBoot. By default, the application will use the eCos virtual vectors mechanism to obtain certain services from RedBoot, including diagnostic output.

ROM
: This startup type can be used for finished applications which will be programmed into flash at physical address 0xF0000000. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization.

RedBoot and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building RedBoot.

If the application is required to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for RAM startup, disabled otherwise. It can be manually disabled for RAM startup, making the application self-contained, as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then the serial port will be claimed for HAL diagnostics.

Flash Driver

The IQ80321 board contains an 8MB Intel StrataFlash flash device. The `CYGPKG_DEVS_FLASH_STRATA` package contains all the code necessary to support these parts and the `CYGPKG_DEVS_FLASH_IQ80321` package contains definitions that customize the driver to the IQ80321 board.

Ethernet Driver

The IQ80321 board contains an Intel i82544 ethernet controller. The `CYGPKG_DEVS_ETH_INTEL_I82544` package contains all the code necessary to support this device and the `CYGPKG_DEVS_ETH_ARM_IQ80321` package contains definitions that customize the driver to the IQ80321 board.

System Clock

By default, the system clock interrupts once every 10ms, corresponding to a 100Hz clock. This can be changed by the configuration option `CYGNUM_HAL_RTC_DENOMINATOR` which corresponds to the clock frequency. Other clock-related settings are recalculated automatically if the denominator is changed.

Compiler Flags

The platform HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos. There is just one flag specific to this port:

`-mcpu=xscale`

The arm-eabi-gcc compiler supports many variants of the ARM architecture. A `-m` option should be used to select the specific variant in use, and with current tools `-mcpu=xscale` is the correct option for the XScale CPU on the IOP321.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the IQ80321 hardware, and should be read in conjunction with that specification. The IQ80321 platform HAL package complements the ARM architectural HAL, the XScale core HAL and the IOP321 (VERDE) variant HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset the HAL will initialize or reinitialize the on-chip peripherals that are used by eCos. There is an exception for RAM startup applications which depend on a ROM monitor for certain services.

For ROM startup, the HAL will perform additional initialization, setting up the external SDRAM and programming the various internal registers. This is all done in the `PLATFORM_SETUP1` macro in the assembler header file `hal_platform_setup.h`.

Memory Maps

The RAM based page table is located at RAM start + 0x4000.

NOTE

The virtual memory map in this section uses a C, B, and X column to indicate the caching policy for the region.

```
X C B  Description
- - -  ---------------------------------------------
0 0 0  Uncached/Unbuffered
0 0 1  Uncached/Buffered
0 1 0  Cached/Buffered    Write Through, Read Allocate
0 1 1  Cached/Buffered    Write Back, Read Allocate
1 0 0  Invalid -- not used
1 0 1  Uncached/Buffered  No write buffer coalescing
1 1 0  Mini DCache - Policy set by Aux Ctl Register
1 1 1  Cached/Buffered    Write Back, Read/Write Allocate

Physical Address Range     Description
-----------------------    ----------------------------------
0x00000000 - 0x7fffffff    ATU Outbound Direct Window
0x80000000 - 0x900fffff    ATU Outbound Translate Windows
0xa0000000 - 0xbfffffff    SDRAM
0xf0000000 - 0xf0800000    FLASH               (PBIU CS0)
0xfe800000 - 0xfe800fff    UART                (PBIU CS1)
0xfe840000 - 0xfe840fff    Left 7-segment LED  (PBIU CS3)
0xfe850000 - 0xfe850fff    Right 7-segment LED (PBIU CS2)
0xfe8d0000 - 0xfe8d0fff    Rotary Switch       (PBIU CS4)
0xfe8f0000 - 0xfe8f0fff    Battery Status      (PBIU CS5)
0xfff00000 - 0xffffffff    Verde Memory mapped Registers


Default Virtual Map      X C B  Description
-----------------------  - - -  ----------------------------------
0x00000000 - 0x1fffffff  1 1 1  SDRAM
0x20000000 - 0x9fffffff  0 0 0  ATU Outbound Direct Window
0xa0000000 - 0xb00fffff  0 0 0  ATU Outbound Translate Windows
0xc0000000 - 0xdfffffff  0 0 0  Uncached alias for SDRAM
0xe0000000 - 0xe00fffff  1 1 1  Cache flush region (no phys mem)
0xf0000000 - 0xf0800000  0 1 0  FLASH               (PBIU CS0)
0xfe800000 - 0xfe800fff  0 0 0  UART                (PBIU CS1)
```

```
0xfe840000 - 0xfe840fff  0 0 0  Left 7-segment LED  (PBIU CS3)
0xfe850000 - 0xfe850fff  0 0 0  Right 7-segment LED (PBIU CS2)
0xfe8d0000 - 0xfe8d0fff  0 0 0  Rotary Switch       (PBIU CS4)
0xfe8f0000 - 0xfe8f0fff  0 0 0  Battery Status      (PBIU CS5)
0xfff00000 - 0xffffffff  0 0 0  Verde Memory mapped Registers
```

Other Issues

The IQ80321 platform HAL does not affect the implementation of other parts of the eCos HAL specification. The XScale core HAL, the IOP321 (VERDE) variant HAL, and the ARM architectural HAL documentation should be consulted for further details.

Chapter 269. Intel XScale IXP4xx Network Processor Support

Name

Support for Intel XScale IXP4xx Network Processors — Overview

Description

This document covers the configuration and usage of the Hardware Abstraction Layer (HAL) for the Intel XScale IXP4xx Network Processor series, including the IXP425 and IXP465. It is expected to be read in conjunction with platform HAL-specific documentation, as well as the eCos HAL specification. This processor HAL package complements the ARM architectural HAL, XScale variant HAL and the platform HAL. It provides functionality common to IXP4xx-based board implementations.

This support is found in the eCos package located at `packages/hal/arm/xscale/ixp425` within the eCos source repository.

For historical reasons many of the definitions, filenames and configuration options in this package refer to the IXP425 specifically. In fact, unless otherwise noted, these definitions, filenames and configuration options apply equally to all members of the IXP4xx family.

The IXP4xx processor HAL package is loaded automatically when eCos is configured for an IXP4xx-based platform. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Supported Hardware

Supported features of the Intel XScale IXP4xx processors within this processor HAL package include:

- IXP4xx-specific hardware definitions
- Interrupt controller
- General-purpose timers
- Watchdog timer
- Serial UARTs
- PCI bus controller
- PCI bus IDE controllers
- CompactFlash cards in TrueIDE mode
- General Purpose I/O (GPIO)

For licensing-related reasons, support for the Network Processing Engines (NPEs) at this time is only available with an add-on EPK package from Intel. eCosCentric is unable to provide support for this add-on package.

Name

IXP4xx hardware definitions — Details on obtaining hardware definitions for IXP4xx

Register definitions

The file `<cyg/hal/hal_ixp425.h>` can be included from application and eCos package sources to provide definitions related to IXP4xx subsystems. These include register definitions for the PCI bus controller, SDRAM controller, DDR controller for IXP46x, expansion bus controller, I^{2}C® controller for IXP46x, General purpose I/O (GPIO), interrupt controller, general-purpose timers and watchdog timer.

I/O Definitions

The file `<cyg/hal/var_io.h>` contains definitions used by the PCI support, as well as memory mapping details and macros to read and write the I/O space (including the PCI I/O space) at a variety of widths. If PCI IDE support has been enabled, it also provides the relevant support macros.

Name

IXP4xx interrupt controller — Interrupt controller definitions and usage

Interrupt controller definitions

The file `<cyg/hal/hal_var_ints.h>` (located at `hal/arm/xscale/ixp425/VERSION/include/hal_var_ints.h` in the eCos source repository) contains interrupt vector number definitions for use with the eCos kernel and driver interrupt APIs:

```
#define CYGNUM_HAL_INTERRUPT_NONE        -1
#define CYGNUM_HAL_INTERRUPT_NPEA         0
#define CYGNUM_HAL_INTERRUPT_NPEB         1
#define CYGNUM_HAL_INTERRUPT_NPEC         2
#define CYGNUM_HAL_INTERRUPT_QM1          3
#define CYGNUM_HAL_INTERRUPT_QM2          4
#define CYGNUM_HAL_INTERRUPT_TIMER0       5
#define CYGNUM_HAL_INTERRUPT_GPIO0        6
#define CYGNUM_HAL_INTERRUPT_GPIO1        7
#define CYGNUM_HAL_INTERRUPT_PCI_INT      8
#define CYGNUM_HAL_INTERRUPT_PCI_DMA1     9
#define CYGNUM_HAL_INTERRUPT_PCI_DMA2     10
#define CYGNUM_HAL_INTERRUPT_TIMER1       11
#define CYGNUM_HAL_INTERRUPT_USB          12
#define CYGNUM_HAL_INTERRUPT_UART2        13
#define CYGNUM_HAL_INTERRUPT_TIMESTAMP    14
#define CYGNUM_HAL_INTERRUPT_UART1        15
#define CYGNUM_HAL_INTERRUPT_WDOG         16
#define CYGNUM_HAL_INTERRUPT_AHB_PMU      17
#define CYGNUM_HAL_INTERRUPT_XSCALE_PMU   18
#define CYGNUM_HAL_INTERRUPT_GPIO2        19
#define CYGNUM_HAL_INTERRUPT_GPIO3        20
#define CYGNUM_HAL_INTERRUPT_GPIO4        21
#define CYGNUM_HAL_INTERRUPT_GPIO5        22
#define CYGNUM_HAL_INTERRUPT_GPIO6        23
#define CYGNUM_HAL_INTERRUPT_GPIO7        24
#define CYGNUM_HAL_INTERRUPT_GPIO8        25
#define CYGNUM_HAL_INTERRUPT_GPIO9        26
#define CYGNUM_HAL_INTERRUPT_GPIO10       27
#define CYGNUM_HAL_INTERRUPT_GPIO11       28
#define CYGNUM_HAL_INTERRUPT_GPIO12       29
#define CYGNUM_HAL_INTERRUPT_SW_INT1      30
#define CYGNUM_HAL_INTERRUPT_SW_INT2      31

#ifdef CYGHWR_HAL_ARM_XSCALE_CPU_IXP46x
#define CYGNUM_HAL_INTERRUPT_USB_HOST     32
#define CYGNUM_HAL_INTERRUPT_I2C          33
#define CYGNUM_HAL_INTERRUPT_SPI          34
#define CYGNUM_HAL_INTERRUPT_TIMESYNC     35
#define CYGNUM_HAL_INTERRUPT_EAU_DONE     36
#define CYGNUM_HAL_INTERRUPT_SHA_DONE     37
#define CYGNUM_HAL_INTERRUPT_SWCP_PERR    58
#define CYGNUM_HAL_INTERRUPT_QMGR_PERR    60
#define CYGNUM_HAL_INTERRUPT_MCU_ERR      61
#define CYGNUM_HAL_INTERRUPT_EXP_PERR     62
#endif
```

The list of interrupt vectors may be augmented on a per-platform basis. Consult the platform HAL documentation for your platform for whether this is the case.

Interrupt controller functions

The source file src/ixp425_misc.c within this package provides most of the support functions to manipulate the interrupt controller. The hal_IRQ_handler queries the IRQ status register to determine the interrupt cause. Functions hal_interrupt_mask and hal_interrupt_unmask enable or disable interrupts within the interrupt controller.

GPIO interrupts are configured in the hal_interrupt_configure function, where the level and up arguments are interpreted as follows:

level	up	interrupt on
0	0	Falling Edge
0	1	Rising Edge
0	-1	Either Edge
1	0	Low Level
1	1	High Level

Some interrupts are acknowledged in the hal_interrupt_acknowledge function, although for many of the IXP4xx on-chip peripherals, the means to stop the interrupt from triggering again is to remove the cause of the interrupt in a device dependent way. This function does however clear GPIO interrupts.

Macros of the following forms may be defined by the platform HAL via the header file defined by its CYGBLD_HAL_PLATFORM_H macro definition in order to extend support to further vectors:

```
HAL_PLF_INTERRUPT_MASK(vector)
HAL_PLF_INTERRUPT_UNMASK(vector)
HAL_PLF_INTERRUPT_ACKNOWLEDGE(vector)
HAL_PLF_INTERRUPT_CONFIGURE(vector)
```

Note that in all the above, it is not recommended to call the described functions directly. Instead either the HAL macros (HAL_INTERRUPT_MASK et al) or preferably the kernel or driver APIs should be used to control interrupts.

Interrupt handling withing standalone applications

For non-eCos standalone applications running under RedBoot, it is possible to install an interrupt handler into the interrupt vector table manually. Memory mappings are platform-dependent and so the platform documentation should be consulted, but in general the address of the interrupt table can be determined by analyzing RedBoot's symbol table, and searching for the address of the symbol name hal_interrupt_handlers. Table slots correspond to the interrupt numbers above. Pointers inserted in this table should be pointers to a C/C++ function with the following prototype:

```
extern unsigned int isr( unsigned int vector, unsigned int data );
```

For non-eCos applications run from RedBoot, the return value can be ignored. The vector argument will also be the interrupt vector number. The data argument is extracted from a corresponding table named hal_interrupt_data which immediately follows the interrupt vector table. It is still the responsibility of the application to enable and configure the interrupt source appropriately if needed.

IXP46x

Support exists for the IXP46x in order to query and manipulate the extended interrupt controller registers handling interrupt numbers of 32 and above. Also on the IXP46x an interrupt handler is attached to handle ECC errors from the MCU.

Name

General-purpose timers — Use of IXP4xx general-purpose timers

General-purpose timer 0

The IXP4xx processor HAL provides general-purpose timer 0 (`OST_TIM0`) to the eCos kernel for use as a real-time clock. This timer is also used for implementing the `HAL_DELAY_US` functionality, which is used by some device drivers, and in non-kernel configurations such as with RedBoot where this timer is needed for loading program images via X/Y-modem protocols and debugging via TCP/IP. Standalone applications which require RedBoot services, such as debugging, should avoid use of this timer.

Available timers

General-purpose timer 1 and the timestamp timer are available for application use.

System clock configuration

By default for eCos applications, the system clock interrupts once every 10ms, corresponding to a 100Hz clock. This can be changed by the configuration option `CYGNUM_HAL_RTC_DENOMINATOR` which corresponds to the clock frequency. Other clock-related settings are recalculated automatically if the denominator is changed. If the desired frequency cannot be expressed accurately solely with changes to `CYGNUM_HAL_RTC_DENOMINATOR`, then the configuration option `CYGNUM_HAL_RTC_NUMERATOR` may also be adjusted, and again clock-related settings will automatically be recalculated.

Name

Watchdog — Describes use of the hardware watchdog

Description

This HAL package includes the hardware driver required to support the generic watchdog support defined in the `CYGPKG_IO_WATCHDOG` package. The functionality allows the watchdog either to cause the IXP4xx to reset, or to cause a callback function to be triggered as defined in the generic watchdog API.

The watchdog is also used to perform platform resets, such as with the **reset** command in the RedBoot CLI. Note that there is an IXP425 erratum affecting use of the watchdog on IXP425s with stepping level of A0. Stepping levels of B0 and above no longer have this issue.

Configuration

It is not possible to enable support for the IXP4xx on-chip watchdog unless the `CYGPKG_IO_WATCHDOG` package is included in the eCos configuration. Once included, the configuration options for this hardware support can then be found in the watchdog part of the eCos configuration tree under the CDL component name `CYGPKG_HAL_IXP4XX_WATCHDOG`, rather than in the HAL.

The option `CYGNUM_HAL_IXP4XX_WATCHDOG_DESIRED_TIMEOUT_US` allows the watchdog timeout interval to be set. An application must ensure that the watchdog is reset more frequently than this period. The units are microseconds, and depending on the selected period, values may be rounded up to the next clock tick..

If the option `CYGSEM_HAL_IXP4XX_WATCHDOG_RESET` is enabled, watchdog expiration will cause the IXP4xx to perform a soft reset. If this option is instead disabled, a callback function can be registered with the generic watchdog API, which will be called by an interrupt handler associated with the watchdog.

Name

Serial UARTs — Configuration and implementation details of serial UART support

Overview

There are two forms of support for the built-in high-speed and console serial UARTs. In all cases the default serial port settings are 115200,8,N,1 with no flow control.

HAL diagnostic I/O

One form is polled mode HAL diagnostic output, intended primarily for use during debug and development. Operations are usually performed with global interrupts disabled, and thus is not usually suitable for deployed systems. This can operate on either port, according to the configuration settings.

Several configuration options within the IXP4xx processor HAL affect HAL diagnostic operation. The CDL option CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL selects the serial port channel to use as the console at startup time. Channel 0 is the debug serial port, channel 1 is the high-speed serial port. This will be the channel that receives output from, for example, diag_printf(). The CDL option CYGNUM_HAL_VIRTUAL_VECTOR_DEBUG_CHANNEL selects the serial port channel to use for GDB communication by default. Note that when using RedBoot, these options are usually inactive as it is RedBoot that decides which channels are used. Applications may override RedBoot's selections by enabling the CYGSEM_HAL_VIRTUAL_VECTOR_CLAIM_COMMS CDL option in the HAL. Baud rates for each channel are set with the CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL_BAUD and CYGNUM_HAL_VIRTUAL_VECTOR_DEBUG_CHANNEL_BAUD options.

Interrupt-driven serial driver

The second form of support is an interrupt-driven serial driver, which is integrated into the eCos standard serial I/O infrastructure (CYGPKG_IO_SERIAL). This support can be enabled on either port. Note that if this form of serial driver is enabled on a port, it will prevent ctrl-c operation when debugging.

This driver uses the eCos generic 16x5x UART driver found in CYGPKG_IO_SERIAL_GENERIC_16X5X to provide most of the support. That driver package should also be consulted for documentation and configuration options.

Note that a standard 16550A compatible UART has receive FIFO trigger levels that can be set with the CYGPKG_IO_SERIAL_GENERIC_16X5X_FIFO_RX_THRESHOLD to 1, 4, 7, 8 or 14 bytes. However the IXP4xx family has a larger 64-byte FIFO, and so these values should be mapped to 1, 8, 16 or 32. In other words setting the CYGPKG_IO_SERIAL_GENERIC_16X5X_FIFO_RX_THRESHOLD option to 8 bytes would in fact cause the receive FIFO to trigger at 16 bytes and so forth.

The maximum baud rate supported by the generic eCos infrastructure is 230,400 and so at this time the higher baud rates of 460,800 and 921,600 baud are not supported.

Name

PCI bus controller — PCI bus controller support implementation details

Implementation details

The PCI bus controller is supported with the files <cyg/hal/var_io.h> and the source file src/ixp425_pci.c. These files provide almost all the required underlying support for use with the generic PCI package CYGPKG_IO_PCI.

Platform HALs are still required to provide any extra initialisation steps such as pin routing, involving GPIO, particularly for the interrupt pins. Accordingly any such interrupt pins will also require treatment with an implementation of the following function:

```
extern void cyg_hal_plf_pci_translate_interrupt(cyg_uint32 bus, cyg_uint32 devfn,
                                                CYG_ADDRWORD *vec, cyg_bool *valid);
```

Name

PCI bus IDE controllers — Configuring and using IDE controllers on the PCI bus

Overview

Support is included for IDE bus controllers connected via the PCI bus. This includes support for generic PCI IDE controllers as well as Promise 20275 or 20269 IDE controllers.

Configuration

This support can be enabled with the `CYGFUN_HAL_IXP4XX_PCI_IDE_SUPPORT` configuration option and is mutually exclusive with support for TrueIDE mode disks connected via a CompactFlash interface.

Use with RedBoot

Enabling this option allows RedBoot to be used to load program images off an IDE disk connected via a PCI IDE controller, for example a Linux kernel from an EXT2 partition of an IDE disk using the "disk" load method.

Files

The file `src/ixp_pci_ide.c` within this package is used to provide the necessary underlying support.

Name

CompactFlash cards in TrueIDE mode — Using CompactFlash cards in TrueIDE mode on the IXP4xx expansion bus

Overview

The IXP4xx processor HAL includes support for CompactFlash IDE devices accessed in True IDE mode directly on the IXP4xx expansion bus.

The hardware configuration for attaching the CF IDE devices must follow the specification described in Intel Application Note 30245603: “Intel IXP42X Product Line of Network Processors and IXC1100 Control Plane Processor: Using CompactFlash”. This describes use of the EXP_CS_N_1 (CS1) and EXP_CS_N_2 (CS2) signals to control the CF IDE device.

Configuration

Support for CF True IDE devices is contained within the IXP4xx variant HAL and is controlled with the `CYGFUN_HAL_IXP4XX_CF_TRUE_IDE_SUPPORT` configuration option. It is not possible to include PCI IDE support within the same configuration - eCos only allows one ID controller driver at the present time.

The driver requires the presence of the generic disk layer (`CYGPKG_IO_DISK`), as well as the IDE disk driver (`CYGPKG_DEVS_DISK_IDE`).

The block device name used to identify the disk is configured with the option `CYGDAT_HAL_IXP4XX_CF_TRUE_IDE_DISK_NAME` and defaults to `/dev/hd0/`. An MBR is expected to be present on the CF card, and individual partitions can be accessed as e.g. `/dev/hd0/1`, `/dev/hd0/2`, etc. or the whole device as `/dev/hd0/0`.

With this support it is possible to access filesystems on the CF IDE card with the further inclusion of the generic file I/O layer package (`CYGPKG_IO_FILEIO`) along with a standard eCos filesystem implementation such as FAT (`CYGPKG_FS_FAT`).

Use from RedBoot

Similarly, it is possible for RedBoot to load images from a filesystem using the "file" load method. For example:

```
RedBoot> fs mount -d /dev/hd0/1 -t fatfs
RedBoot> fs list
   2 -rwxrwxrwx  1 size 3961588 VMLINUX
1937 -rwxrwxrwx  1 size 1588984 ZIMAGE
RedBoot> load -m file -r -b %{freememlo} /ZIMAGE
Raw file loaded 0x00079800-0x001fd6f7, assumed entry at 0x00079800
RedBoot> exec
Using base address 0x00079800 and length 0x00183ef8
Uncompressing Linux...................................................................................................
Linux version 2.6.12 (root@andy) (gcc version 3.4.4) #43 Tue Nov 15 16:40:04 GMT 2005
CPU: XScale-IXP42x Family [690541c1] revision 1 (ARMv5TE)
CPU0: D VIVT undefined 5 cache
CPU0: I cache: 32768 bytes, associativity 32, 32 byte lines, 32 sets
CPU0: D cache: 32768 bytes, associativity 32, 32 byte lines, 32 sets
Machine: Intel IXDP425 Development Platform
[etc.]
```

Implementation details

The implementation assumes that the platform HAL will map memory accesses at 0x51000000 to expansion bus accesses with CS1 enabled, and similarly 0x52000000 to expansion bus accesses with CS2 enabled.

This driver operates in polled mode (PIO) only with no interrupt-driven operation nor DMA, and uses conservative bus configuration timings to allow for maximum compatibility with CF IDE cards. Note that some CF cards are not fully compliant with the CompactFlash standard and do not fully or correctly implement True IDE mode.

Name

GPIO — General purpose I/O

GPIO functions

As well as hardware definitions, the file `<cyg/hal/hal_ixp425.h>` provides a set of macros to assist GPIO manipulation:

```
HAL_GPIO_OUTPUT_ENABLE(line)
HAL_GPIO_OUTPUT_DISABLE(line)
HAL_GPIO_OUTPUT_SET(line)
HAL_GPIO_OUTPUT_CLEAR(line)
```

As described earlier, HAL support already exists for handling GPIO lines configured as interrupts.

Chapter 270. Intel XScale IXDP425 Network Processor Evaluation Board Support

Name

eCos and RedBoot Support for the Intel XScale IXDP425 Network Processor Evaluation Board — Overview

Description

This document covers the configuration and usage of eCos and RedBoot on the Intel XScale IXDP425 Network Processor Evaluation Board. The IXDP425 board contains the Intel XScale IXP425 processor, 256Mbytes of SDRAM, 16MByte of parallel NOR flash memory, an I²C EEPROM, hexadecimal debug display, LEDs, and external connections for two serial channels, two NPE ethernet daughterboards and an expansion bus based on the Utopia-2 interface standard. eCos and RedBoot support for the devices and peripherals on this board is described below.

In normal operation, a RedBoot image is programmed into the flash memory, and the board will boot into this image from reset. RedBoot provides gdb stub functionality so it is then possible to download and debug stand-alone applications via the gdb debugger. RedBoot can also load and execute Linux kernels. This can happen over either a serial line or over ethernet.

This document should be read in conjunction with the *Intel XScale IXP4xx Network Processor Support* processor HAL documentation in the eCos documentation set, as well as the generic HAL documentation.

Supported Hardware

The parallel NOR flash memory supplied by default with the IXDP425 - a single Intel StrataFlash 28F128J3 - consists of 128 blocks of 128Kbytes each. In a typical setup, the first four blocks, 512 Kbytes, are reserved for the use of the RedBoot ROM image. The topmost block is used to manage the flash and hold RedBoot **fconfig** values. The remaining blocks can be used by application code. There are 123 blocks available between 0x50080000 and 0x50FE0000.

RedBoot supports the built-in high-speed and console UARTs. The default serial port settings are 115200,8,N,1.

There is an ethernet driver `CYGPKG_DEVS_ETH_INTEL_I82559` intended for use with a PCI I82559-based ethernet device to allow communication and downloads. A separate package, `CYGPKG_DEVS_ETH_ARM_IXDP425_I82559`, is responsible for configuring this generic driver to the IXDP425 hardware. This driver is also loaded automatically when configuring for the IXDP425 target.

IDE support is available to support most PCI IDE controllers. Separate support is also available to support CompactFlash cards fitted to an optional daughterboard, in True IDE mode. These features are described in the IXP4xx processor HAL documentation.

Tools

The IXDP425 support is intended to work with GNU tools configured for an arm-eabi target. The original port was undertaken using arm-elf-gcc version 3.4.4, arm-elf-gdb version 6.3, and binutils version 2.15.

Name

Setup — Setting up the IXDP425 board

Overview

In a typical development environment, the IXDP425 board boots from the parallel NOR Flash and runs the RedBoot ROM monitor directly. Applications are then downloaded into RAM and run directly on the board using the command line interface, or for standalone applications, via the debugger **arm-eabi-gdb**. Alternatively applications can be stored in Flash to be subsequently loaded into RAM for execution. Preparing the board therefore usually involves programming a suitable RedBoot image into flash memory.

The following RedBoot configurations are supported:

Configuration	Description	Use	File
ROM	RedBoot running from ROM	`redboot_ROM.ecm`	`redboot_ROM.bin`
RAM	RedBoot running from RAM	`redboot_RAM.ecm`	`redboot_RAM.bin`
ROMRAM	RedBoot running from RAM, but contained in the board's flash boot sector	`redboot_ROMRAM.ecm`	`redboot_ROMRAM.bin`
ROM_CFIDE	Same as ROM, but with additional CompactFlash IDE and FAT FS support	`redboot_ROM_CFIDE.ecm`	`redboot_ROM_CFIDE.bin`
ROMRAM_CFIDE	Same as ROMRAM, but with additional CompactFlash IDE and FAT FS support	`redboot_ROMRAM_CFIDE.ecm`	`redboot_ROMRAM_CFIDE.bin`
ROMLE	RedBoot running from ROM configured for little-endian operation	`redboot_ROMLE.ecm`	`redboot_ROMLE.bin`
RAMLE	RedBoot running from RAM configured for little-endian operation	`redboot_RAMLE.ecm`	`redboot_RAMLE.bin`
ROMRAMLE	RedBoot running from RAM and configured for little-endian operation, but contained in the board's flash boot sector	`redboot_ROMRAMLE.ecm`	`redboot_ROMRAMLE.bin`

For serial communications, all versions run with 8 bits, no parity, and 1 stop bit at 115200 baud. RedBoot also supports ethernet communication and flash management.

If the ROM version is to be chosen, then the RAM version is provided to allow for updating the resident RedBoot image in Flash. The ample provision of RAM memory on the board allows the ROMRAM version of RedBoot to be used instead of the standard ROM version which executes directly from Flash.

Initial Installation

Installation with a Flash programmer

This approach to initial installation may be used if a Flash device programmer is available. The IXDP425 Flash is socketed at U22. Although the supplied Flash part is an Intel StrataFlash 28F128J3, any 28FxxxJ3 part may be substituted - RedBoot will use the Common Flash Interface (CFI) to determine the Flash device geometry.

In this mode, the ROM mode RedBoot is programmed into the boot flash at offset 0x00000000.

Installation via JTAG

A JTAG device may also be used to perform initial programming. Some JTAG devices have in-built capabilities that permit programming of the Flash part. Other JTAG devices are able to load a RAM RedBoot image into SDRAM, from where a ROM RedBoot image can then in turn be loaded and then programmed.

If using a JTAG device that supports direct programming of the Flash part, the Flash is usually located at 0x50000000 in the CPU memory map, assuming the JTAG device initialisation has not remapped it elsewhere. It will usually need to first unlocked and then erased before programming.

For other JTAG devices, to load a RAM RedBoot image into SDRAM it will first be required to initialise the memory interface, and in particular configure the SDRAM controller. Consult your JTAG device documentation on how to access IXP425 registers.

One example of a JTAG device capable of direct programming of Flash is the Abatron BDI2000, and in the following sections are the steps required to program RedBoot using it.

Preparing the Abatron BDI2000 JTAG debugger

The BDI2000 must first be configured to allow communication with your local network, and configured with the parameters for interfacing with the target board. The following steps should be followed:

1. Prepare a PC to act as a host PC and start a TFTP server on it.

2. Connect the Abatron BDI2000 JTAG debugger via both serial and ethernet to the host PC and power it on. Use the serial cable supplied with the BDI2000.

3. Install the Abatron BDI2000 bdiGDB support software on the host PC.

4. Confirm that the XScale firmware is resident in the BDI2000. For example, using the **bdisetup** utility:

   ```
   $ bdisetup -v -s
   BDI Type : BDI2000 Rev.C (SN: xxxxxxxx)
   Loader   : V1.05
   Firmware : V1.08 bdiGDB for XScale
   Logic    : V1.03 XScale
   MAC      : 00-0c-01-96-64-92
   IP Addr  : 192.168.7.220
   Subnet   : 255.255.255.0
   Gateway  : 192.168.7.1
   Host IP  : 192.168.7.9
   Config   : /ixdp425.cfg
   ```

   To upload firmware if needed, follow the procedures in the BDI2000 manual, for example using the **bdisetup** utility:

   ```
   $ bdisetup -u -p/dev/ttyS0 -b57 -aGDB -tXSCALE
   Connecting to BDI loader
   Erasing CPLD
   Programming firmware with ./b20xscgd.108
   Erasing firmware flash ....
   Erasing firmware flash passed
   Programming firmware flash ....
   Programming firmware flash passed
   Programming CPLD with ./xscjed21.103
   ```

5. Locate the file `ixdp425.cfg` within the BDI2000 software installation.

6. Locate the file `regIXP425.def` within the installation of the BDI2000 bdiGDB support software.

7. Place the `ixdp425.cfg` file in a location on the PC accessible to the TFTP server. Later you will configure the BDI2000 to load this file via TFTP as its configuration file.

8. Similarly place the file `regIXP425.def` in a location accessible to the TFTP server.

9. Open `ixdp425.cfg` in an editor such as emacs or notepad and if necessary adjust the path of the `regIXP425.def` file in the `[REGS]` section to match its location relative to the TFTP server root. Also comment out with a ';' the IP line in the `[HOST]` section, or update it to refer to the development PC.

10. Install and configure the Abatron BDI2000 in line with the bdiGDB instruction manual. Configure the BDI2000 to use the `ixdp425.cfg` configuration file at the appropriate point of this process. For example, using the **bdisetup** utility:

```
$ bdisetup -c -p/dev/ttyS0 -b57 -i192.168.7.220 -h192.168.7.9 -m255.255.255.0 -g192.168.7.1 -f/ixdp425.cfg
Connecting to BDI loader
Writing network configuration
Configuration passed
```

    The above command uses the first serial port at 57,600 baud to set the BDI2000's IP address to 192.168.7.220, its default TFTP host to 192.168.7.9, the network mask to 255.255.255.0, the default gateway to 192.168.7.1, and the configuration file to load to `/ixdp425.cfg`.

Preparing the IXDP425 board for programming

Follow the steps in this section in order to allow communication between the board and the host PC, and between the board and the JTAG device.

1. First you must connect a straight through DB9 serial cable between the high speed serial port labelled UART0 on the board and a serial port on the host computer.

2. Start a suitable terminal emulator on the host computer such as **minicom** or HyperTerminal. Set the communication parameters to 115200 baud, 8 data bits, no parity bit and 1 stop bit with no flow control.

3. If using a PCI ethernet card, connect the board to your host PC's LAN with an Ethernet cable.

4. If using a PCI ethernet card, you may need to designate the ethernet interface with a new Ethernet MAC address. The RedBoot binary image contains a default address, but each board requires its own unique address. It is advisable to mark each board with its programmed MAC address for future identification.

5. Connect the board to the BDI2000 using a 20-pin ARM/Xscale cable from the JTAG/ICE interface connector (J12) to the Target A port on the BDI2000.

6. Power up the IXDP425 board. You should see the hex display and various LEDs illuminate.

7. Connect to the BDI2000's CLI interface via TCP/IP on the standard telnet port 23. The **telnet** application is suitable for this. You should see usage information followed by the prompt:

```
Core#0>
```

8. Confirm correct connection with the BDI2000 with the **reset halt** command as follows:

```
Core#0> reset halt
- TARGET: processing reset request
- TARGET: BDI asserts RESET and TRST
- TARGET: BDI removed TRST
- TARGET: Bypass check: 0x000000001 => 0x00000001
- TARGET: JTAG exists check passed
- Core#0: ID code is 0x19274013
- Core#0: BDI sets hold_rst and halt mode
- TARGET: BDI removes RESET
- Core#0: BDI sets hold_rst and halt mode again
- Core#0: BDI loads debug handler to mini IC
- Core#0: BDI clears hold_rst
- TARGET: resetting target passed
- TARGET: processing target startup ....
- TARGET: processing target startup passed
Core#0>
```

9. Locate the `redboot_ROM.bin` image within the `loaders` subdirectory of the base of the eCos installation.

10. Copy the `redboot_ROM.bin` file into a location on the host computer accessible to its TFTP server.

Using the BDI2000 to directly program RedBoot into Flash

As previously mentioned, there are two methods of programming a RedBoot image into the parallel NOR Flash via JTAG, depending on the capabilities of the JTAG device. The BDI2000 supports direct programming of the Flash device and so that is the approach described here.

This is a three stage process. The relevant Flash blocks must first be unlocked, then erased, and finally programmed. This can be accomplished with the following steps:

1. Connect to the BDI2000 telnet port as before.

2. Use the following commands in the BDI2000 telnet session to unlock and then erase the relevant Flash blocks that will contain RedBoot.

```
Core#0>unlock 0x50000000 0x20000 4
Unlocking flash at 0x50000000
Unlocking flash at 0x50020000
Unlocking flash at 0x50040000
Unlocking flash at 0x50060000
Unlocking flash passed
Core#0>erase 0x50000000 0x20000 4
Erasing flash at 0x50000000
Erasing flash at 0x50020000
Erasing flash at 0x50040000
Erasing flash at 0x50060000
Erasing flash passed
```

3. Program the RedBoot image into Flash with the following command, replacing *`/RBPATH`* with the location of the redboot_ROM.bin file relative to the TFTP server root directory:

```
Core#0>prog 0x50000000 /RBPATH/redboot_ROM.bin bin
Programming /RBPATH/redboot_ROM.bin , please wait ....
Programming flash passed
Core#0>
```

   This operation can take some time.

The RedBoot installation is now complete. This can be tested by powering off the board, disconnecting the JTAG, and then powering on the board again. The RedBoot banner should be visible on the serial port.

```
flash configuration checksum error or invalid key
```

This message does not indicate a problem at this stage of installation. It just means that RedBoot Flash configuration has yet to be performed, as described below.

If it proves necessary to re-install RedBoot, this may be achieved by repeating the above process. Alternatively, a new image may be downloaded and programmed into flash more directly using RedBoot's own commands. See the RedBoot documentation for details.

RedBoot Flash configuration

The following steps describe how to initialize RedBoot's Flash configuration. This must be performed when using a RAM RedBoot to program Flash, but is also applicable to initial configuration of a ROM or ROMRAM RedBoot loaded using JTAG or with a Flash device programmer.

1. Use the following command to initialize RedBoot's Flash Information System (FIS):

   ```
   RedBoot> fis init -f
   About to initialize [format] FLASH image system - continue (y/n)? y
   *** Initialize FLASH Image System
   ... Erase from 0x50080000-0x50fdffff: .............................................................................
   ... Unlocking from 0x50fe0000-0x50ffffff: .
   ... Erase from 0x50fe0000-0x50ffffff: .
   ... Program from 0x0ffd0000-0x0fff0000 to 0x50fe0000: .
   ... Locking from 0x50fe0000-0x50ffffff: .
   RedBoot>
   ```

2. Now configure RedBoot's Flash configuration with the command:

   ```
   RedBoot> fconfig -i
   ```

   If a BOOTP/DHCP server is not available, then IP configuration may be set manually. The default server IP address can be set to a PC that will act as a TFTP host for future RedBoot load operations, or may be left unset. The following gives an example configuration:

   ```
   RedBoot> fconfig -i
   Initialize non-volatile configuration - continue (y/n)? y
   Run script at boot: false
   Use BOOTP for network configuration: false
   Gateway IP address: 192.168.7.11
   Local IP address: 192.168.7.222
   Local IP address mask: 255.255.255.0
   Default server IP address: 192.168.7.9
   Console baud rate: 115200
   DNS server IP address: 192.168.7.11
   GDB connection port: 9000
   Force console for special debug messages: false
   Network debug at boot time: false
   Update RedBoot non-volatile configuration - continue (y/n)? y
   ... Unlocking from 0x50fe0000-0x50ffffff: .
   ... Erase from 0x50fe0000-0x50ffffff: .
   ... Program from 0x0ffd0000-0x0fff0000 to 0x50fe0000: .
   ... Locking from 0x50fe0000-0x50ffffff: .
   RedBoot>
   ```

Rebuilding RedBoot

Should it prove necessary to rebuild a RedBoot binary, this is done most conveniently at the command line. The steps needed to rebuild the the ROM version of RedBoot for the IXDP425 are:

```
$ mkdir redboot_ixdp425_romram
$ cd redboot_ixdp425_romram
$ ecosconfig new ixdp425 redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/arm/xscale/ixdp425/VERSION/misc/redboot_ROM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

At the end of the build the `install/bin` subdirectory should contain the file `redboot.bin`.

The other versions of RedBoot - RAM, ROMRAM or the little-endian variants - may be similarly built by choosing the appropriate alternative .ecm file.

Name

Configuration — Platform-specific Configuration Options

Overview

The IXDP425 platform HAL package is loaded automatically when eCos is configured for the `ixdp425` target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Configuring for the `ixdp425` target also causes the IXP4xx processor HAL to be included. Many configuration options in relation to the peripheral IXP425 devices including serial UARTs, clocks, etc. are described in the IXP4xx processor HAL documentation, which should be referred to.

Startup

The platform HAL package supports three separate startup types:

RAM
: This is the startup type which is normally used during application development. The board has RedBoot programmed into flash and boots into that initially. arm-eabi-gdb is then used to load a RAM startup application into memory and debug it. It is assumed that the hardware has already been initialized by RedBoot. By default a standalone application will use the eCos virtual vectors mechanism to obtain certain services from RedBoot, including diagnostic output.

ROM
: This startup type can be used for finished applications which will be programmed into flash at physical address 0x50000000. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization.

ROMRAM
: This startup type can be used for finished applications which will be programmed into flash at physical location 0x50000000. However, when it starts up, the application will first copy itself to RAM at virtual address 0x00000000 and then run from there. RAM is generally faster than flash memory, so the program will run more quickly than a ROM-startup application. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization.

RedBoot and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building RedBoot.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then the serial port will be claimed for HAL diagnostics.

Flash Driver

The IXDP425 is supplied by default with a 16Mbyte Intel StrataFlash 28F128J3 parallel Flash device.

The `CYGPKG_DEVS_FLASH_STRATA_V2` package contains all the code necessary to support this part and the platform HAL package contains definitions that customize the driver to the IXDP425 board. This driver is not active until the generic Flash support package, `CYGPKG_IO_FLASH`, is included in the configuration.

The device is located in a socket on a board labelled both U22 and "BOOT ROM", and can be replaced with compatible Intel StrataFlash 28FxxxJ3 parts. RedBoot will use the Common Flash Interface (CFI) to determine the Flash device geometry.

Ethernet Driver

The IXDP425 development kit includes an Intel i82559-based PCI Ethernet NIC. The `CYGPKG_DEVS_ETH_INTEL_I82559` package contains all the code necessary to support this device and the `CYGPKG_DEVS_ETH_ARM_IXDP425_I82559` package contains definitions that customize the driver to the IXDP425 board. This driver is not active until the generic Ethernet support package, `CYGPKG_IO_ETH_DRIVERS`, is included in the configuration.

The IXDP425 is also supplied with Network Processing Engine (NPE) modules, however these are not supported in this release.

CompactFlash True IDE Driver

The IXP425 variant HAL includes support for CompactFlash IDE devices accessed in True IDE mode directly on the IXP425 expansion bus. This is described further in the IXP4xx processor HAL documentation.

However note that the use of CS2 conflicts with use of the hex display, which also operates from the CS2 chip select, and so the use of the hex display by eCos/RedBoot will be disabled if CF IDE support is enabled. In addition it is expected that the hex display will show random unpredictable values during CF IDE accesses.

Other IXP425 peripherals

Details in relation to on-chip IXP425 peripherals such as PCI IDE driver, watchdog support, serial support, clocks, interrupts and so forth are described further in the IXP4xx processor HAL documentation.

Compiler Flags

The platform HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos. There are just two flags specific to this port:

`-mcpu=xscale`
: The arm-eabi-gcc compiler supports many variants of the ARM architecture. A `-m` option should be used to select the specific variant in use, and with current tools `-mcpu=xscale` is the correct option for the XScale IXP425 CPU.

`-mbig-endian`
: The arm-eabi-gcc compiler will compile all code into big endian (most significant byte first) format. This is the default endianness for this port. Without this flag, arm-eabi-gcc generates little endian code. You must ensure your application is built for the same endianness as RedBoot.

  Although RedBoot endianness can be controlled by enabling or disabling the configuration option `CYGHWR_HAL_ARM_BIGENDIAN`, it is more convenient to use the minimal configuration files (.ecm files) as described earlier.

`-mthumb`
: The arm-eabi-gcc compiler will compile C and C++ files into the Thumb instruction set when this option is used. The best way to build eCos in Thumb mode is to enable the configuration option `CYGHWR_THUMB`.

`-mthumb-interwork`
: This option allows programs to be created that mix ARM and Thumb instruction sets. Without this option, some memory can be saved. This option should be used if -mthumb is used. The best way to build eCos with Thumb interworking is to enable the configuration option `CYGBLD_ARM_ENABLE_THUMB_INTERWORK`.

Name

JTAG support — Usage

Use of JTAG for debugging

JTAG can be used to single-step and debug loaded RAM applications, or even applications resident in ROM, including RedBoot.

Debugging of ROM applications is only possible if using hardware breakpoints. The XScale only supports four such hardware breakpoints - two for instruction breakpoints and two for data breakpoints, and so they should be used sparingly. If using a GDB front-end such as Eclipse, check it has not set unnecessary extra breakpoints. Some JTAG devices give the option of whether to set hardware or software breakpoints by default. Be sure to configure your device appropriately.

Abatron BDI2000 notes

On the Abatron BDI2000, the `ixdp425.cfg` file that is included with the BDI2000 software should be used to setup and configure the hardware to an appropriate state to load programs. This includes setting up the SDRAM controller.

The `ixdp425.cfg` file also contains an option to define whether hardware or software breakpoints are used by default, using the `BREAKMODE` directive in the `[TARGET]` section. Edit this file if you wish to use software break points, and remember to use the **boot** command on the BDI2000 command line interface to make the changes take effect.

On the BDI2000, debugging can be performed either via the telnet interface or using **arm-eabi-gdb** and the bdiGDB interface. In the case of the latter, **arm-eabi-gdb** needs to connect to TCP port 2001 on the BDI2000's IP address. For example:

```
(gdb) target remote 111.222.333.444:2001
```

By default when the BDI2000 is powered up, the target will always run the initialization section of the `ixdp425.cfg` file (which configures the SDRAM among other things), and halts the target. This behavior is repeated with the **reset halt** command.

If the board is reset when in '**reset halt**' mode (either with the '**reset halt**' or '**reset**' commands, or by pressing the reset button) and the '**go**' command is then given, then the board will boot as normal. If a ROMRAM RedBoot is resident in Flash, it will be run.

It is also possible for the target to always run, without initialization, after the reset button has been pressed. This mode is selected with the **reset run** command. This conveniently allows the target to be connected to the JTAG debugger, and be able to reset it with the reset button, without being required to always type '**go**' every time. Thereafter, invoking the **reset** command will repeat the previous reset style. Also in this mode, exceptions will be handled by board software, rather than causing the JTAG debugger to halt the CPU.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the IXDP425 hardware, and should be read in conjunction with that specification. The IXDP425 platform HAL package complements the ARM architectural HAL, the XScale variant HAL and the IXP425 processor HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset, the HAL will initialize or reinitialize most of the on-chip peripherals. There is an exception for RAM startup applications which depend on a ROM monitor for certain services.

For ROM or ROMRAM startup, the HAL will perform additional initialization, setting up the external RAM and programming the various internal registers. This is all done in the `PLATFORM_SETUP1` macro in the assembler header file `hal_platform_setup.h`.

LED Codes

Unless the Compact Flash IDE configuration option is selected, RedBoot uses the 4 digit LED display to indicate status during board initialization. Possible codes are:

LED   Actions

    Power-On/Reset
        Set the CPSR
        Enable coprocessor access
        Drain write and fill buffer
        Setup expansion bus chip selects
1001
        Enable Icache
1002
        Initialize SDRAM controller
1003
        Switch flash (CS0) from 0x00000000 to 0x50000000
1004
        Copy MMU table to RAM
1005
        Setup TTB and domain permissions
1006
        Enable MMU
1007
        Enable DCache
1008
        Enable branch target buffer
1009
        Drain write and fill buffer
        Flush caches
100A
        Start up the eCos kernel or RedBoot
0001

Linker Scripts and Memory Maps

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

Flash

This is located at address 0x50000000 of the physical memory space. At initialization, the HAL uses the MMU to retain it at virtual address 0x50000000, while also providing an uncached mapping at 0xB0000000 and a data coherent mapping at 0xA0000000.

SDRAM

This is located at address 0x00000000 of the physical memory space. The HAL uses the MMU to retain this at virtual address 0x00000000, along with an alias at 0x10000000. The same memory is also accessible uncached at virtual location 0x20000000 for use by devices, and at 0x30000000 for data coherent access. The first 32 bytes are used for hardware exception vectors. The next 32 bytes are used for the VSR table and the next 256 bytes are normally used for the eCos virtual vectors, allowing RAM-based applications to use services provided by the ROM monitor. Memory is required for the MMU tables, and must be aligned on a 16Kbyte boundary. These therefore occupy memory from 0x4000 to 0x8000. For ROM/ROMRAM startup, all remaining SDRAM is available. For RAM startup, available RAM starts at virtual location 0x00080000, with the bottom 512 kilobytes reserved for use by RedBoot.

On-chip Peripheral Registers

There are several regions in the memory map devoted to on-chip peripherals or on-chip device controllers. When the MMU is enabled, all these regions are set up with a direct, uncached and unbuffered mapping so that these registers remain accessible at their physical locations.

As such, the address space for the AHB Queue Manager (AQM) resides at 0x60000000; the PCI controller resides at 0xC0000000; the expansion bus controller configuration registers reside at 0xC4000000; the SDRAM controller configuration registers resides at 0xCC000000; and all remaining IXP425 on-chip peripherals reside in the block at 0xC8000000. This latter block includes peripheral control for on-chip high-speed and console UARTs, internal bus performance monitoring unit, interrupt controller, GPIO controller, timers, WAN/Voice and Ethernet NPEs, Ethernet MACs, and the USB controller.

Off-chip Peripherals

RedBoot and eCos access the SDRAM, parallel NOR flash, and hex display on CS2 (mapped to 0x52000000). In addition a CompactFlash True IDE mode disk may be accessed via the expansion bus on CS1/CS2 (0x51000000/0x52000000), although the hex display is not usable in that case.

In addition a 64MiB PCI window is mapped to 0x48000000, for communication with devices on the PCI bus.

RedBoot and eCos do not currently make any use of any other off-chip peripherals present on the IXDP425 board.

Memory map summary

The virtual memory maps in this section use a C, B, and X column to indicate the caching policy for the region.

```
X C B  Description
- - -  ---------------------------------------------
0 0 0  Uncached/Unbuffered
0 0 1  Uncached/Buffered
0 1 0  Cached/Buffered    Write Through, Read Allocate
```

```
0 1 1  Cached/Buffered    Write Back, Read Allocate
1 0 0  Invalid - not used
1 0 1  Uncached/Buffered  No write buffer coalescing
1 1 0  Mini DCache - Policy set by Aux Ctl Register
1 1 1  Cached/Buffered    Write Back, Read/Write Allocate

Virtual Address   Physical Address  XCB  Size (MB)  Description
   0x00000000       0x00000000      010     256     SDRAM (cached)
   0x10000000       0x00000000      010     256     SDRAM (alias)
   0x20000000       0x00000000      000     256     SDRAM (uncached)
   0x30000000       0x00000000      010     256     SDRAM (cached, DC)
   0x48000000       0x48000000      000      64     PCI Data
   0x50000000       0x50000000      010      16     Flash (CS0, cached)
   0x51000000       0x51000000      000      16     CF True IDE mode chip select #0 (CS1)
   0x52000000       0x52000000      000      16     Hex display/CF True IDE mode chip select #1 (CS2)
   0x53000000       0x53000000      000      80     CS3 - CS7
   0x60000000       0x60000000      000      64     Queue Manager
   0xA0000000       0x50000000      010      16     Flash (CS0, cached, DC)
   0xB0000000       0x50000000      000      16     Flash (CS0, uncached)
   0xC0000000       0xC0000000      000       1     PCI Controller
   0xC4000000       0xC4000000      000       1     Exp. Bus Config
   0xC8000000       0xC8000000      000       1     Misc IXP425 IO
   0xCC000000       0xCC000000      000       1     SDRAM Config
```

Other Issues

The IXDP425 platform HAL does not affect the implementation of other parts of the eCos HAL specification. The XScale variant HAL, the IXP4xx processor HAL documentation and the ARM architectural HAL documentation should be consulted for further details.

Chapter 271. Altera Hard Processor System Support

Name

Support for the Altera HPS — Overview

Description

This document covers the configuration and usage of the Hardware Abstraction Layer (HAL) for the Hard Processor System (HPS) present on the Cyclone V and Arria 10 FPGAs. It is expected to be read in conjunction with platform HAL-specific documentation, as well as the eCos HAL specification. This processor HAL package complements the ARM architectural HAL, Cortex-A variant HAL and the platform HAL. It provides functionality common to all HPS-based implementations.

This support is found in the eCos package located at `packages/hal/arm/cortexa/altera_hps` within the eCos source repository.

The Altera HPS HAL package is loaded automatically when eCos is configured for an HPS-based platform. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Supported Hardware

Supported features of the HPS within this processor HAL package include:

- HPS-specific hardware definitions
- Interrupt controller selection
- Timers
- Serial UART configuration
- I²C two wire interface configuration
- Pin Configuration and GPIO Support

Support for the on-chip QSPI device, SPI NOR flash, interrupt-driven serial, watchdog and wallclock (RTC) features of the HPS are also present and can be found in separate packages, outside of this processor HAL.

Support for SMP operation of the two Cortex-A9 CPUs in the HPS is available, although debugging support is restricted to use of an external JTAG debugger. The HAL does not contain support for the Cortex-A's NEON SIMD engine.

Name

HPS Hardware Definitions — Details on obtaining hardware definitions for HPS

Register definitions

The file `<cyg/hal/altera_hps.h>` can be included from application and eCos package sources to provide definitions related to HPS subsystems. These include base addresses for various devices and register definitions for HPS-specific devices such as the system manager, reset manager, pin multiplexing and GPIO. The registers for various devices, such as QSPI, I²C, Ethernet, and the UARTs are defined in the drivers for those devices. The interrupt controller and system timer are implemented in the Cortex-A HAL. This file will normally be included automatically if `<cyg/hal/hal_io.h>` is included, which is the preferred way of getting these definitions.

Initialization Helper Macros

The file `<cyg/hal/altera_hps_init.inc>` contains definitions of helper macros which may be used by HPS platform HALs in order to initialize common subsystems without excessive duplication between these platform HALs. Typically this file will be included by the `hal_platform_setup.h` header in the platform HAL, in turn included from the architectural HAL file `vectors.S`.

This file is solely intended to be used by platform HALs. At the same time, it is only present to assist initialization, and platform HALs are not obliged to use it if their startup requirements vary. NOTE: At present, the only extant HPS port relies on the Altera preloader to initialize the PLLs and memory controller, so these macros currently largely contain Cortex-A-generic setup only.

Name

GIC Interrupt Controller — Advanced Interrupt Controller Definitions And usage

Interrupt controller definitions

The file `<cyg/hal/var_ints.h>` (located at `hal/arm/cortexa/altera_hps/*VERSION*/include/var_ints.h` in the eCos source repository) contains interrupt vector number definitions for use with the eCos kernel and driver interrupt APIs.

The list of interrupt vectors may be augmented on a per-platform basis. Consult the platform HAL documentation for your platform for whether this is the case.

Interrupt Controller Functions

The HPS uses a standard ARM Generic Interrupt Controller (GIC), which is implemented in the Cortex-A HAL. The Cortex-A HAL exports the standard interrupt vector management functions. The `hal_IRQ_handler` queries the IRQ status register to determine the interrupt cause. Functions `hal_interrupt_mask` and `hal_interrupt_unmask` enable or disable interrupts within the interrupt controller.

Interrupts are configured in the `hal_interrupt_configure` function. Refer to the HPS documentation for any limitations as to what types of signal can be detected.

The `hal_interrupt_eoi` function performs End-Of-Interrupt processing and is called automatically by the architecture HAL. The `hal_interrupt_acknowledge` function is intended to acknowledge an interrupt, although for the GIC it is a NUL function since the necessary work it handled by the EOI function.

The `hal_interrupt_set_level` is used to set the priority level of the supplied interrupt within the GIC. Priorities may range between 0 and 255, with lower values mapping to higher priority. The GIC in the HPS only implements the top 5 bits of the priority value, so there are actually 32 distinct priority levels available.

Note that in all the above, it is not recommended to call the described functions directly. Instead either the HAL macros (`HAL_INTERRUPT_MASK` et al) or preferably the kernel or driver APIs should be used to control interrupts.

Name

Timers — Use of on-chip timers

System Clock

The eCos kernel system clock is implemented using the private timer of CPU0. By default, the system clock interrupts once every 10ms, corresponding to a 100Hz clock. This can be changed by the configuration option `CYGNUM_HAL_RTC_DENOMINATOR` which corresponds to the clock frequency. Other clock-related settings are recalculated automatically if the denominator is changed. If the desired frequency cannot be expressed accurately solely with changes to `CYGNUM_HAL_RTC_DENOMINATOR`, then the configuration option `CYGNUM_HAL_RTC_NUMERATOR` may also be adjusted. However, if this is done then `CYGNUM_HAL_RTC_PERIOD` must be changed to provide the described clock frequency.

The same Timer is also used to implement the HAL microsecond delay function, `HAL_DELAY_US`. This is used by some device drivers, and in non-kernel configurations, such as RedBoot, where this timer is needed for loading program images via X/Y-modem protocols and debugging via TCP/IP. Standalone applications which require RedBoot services, such as debugging, should avoid use of this timer.

If SMP is enabled, then the HAL switches to using the global timer. This is because it is necessary to read the timer from any CPU, not just from CPU0.

Timer-based profiling support

If the gprof package, `CYGPKG_PROFILE_GPROF`, is included in the configuration, then the MPCore global timer is reserved for use by the profiler.

Profiling is only supported in single-core systems. If SMP is enabled then the profiling timer is disabled. This is because the SMP scheduler needs to use the global timer rather than the private timers. Additionally, the `CYGPKG_PROFILE_GPROF` package is not SMP-aware and the results would, in any case, be invalid.

If profiling is wanted, then it is recommended that a hardware tool like the Lauterbach Trace32 debugger be used.

Name

Serial UARTs — Configuration and Implementation Details of Serial UART Support

Overview

Support is included in this processor HAL package for the on-chip serial UART devices.

There are two forms of support: HAL diagnostic I/O; and a fully interrupt-driven serial driver. Unless otherwise specified in the platform HAL documentation, for all serial ports the default settings are 57600,8,N,1 with no flow control.

HAL Diagnostic I/O

This first form is polled mode HAL diagnostic output, intended primarily for use during debug and development. Operations are usually performed with global interrupts disabled, and thus this mode is not usually suitable for deployed systems. This can operate on any port, according to the configuration settings.

There are several configuration options usually found within a platform HAL which affect the use of this support in the HPS processor HAL. The CDL option `CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL` selects the serial port channel to use as the console at startup time. This will be the channel that receives output from, for example, `diag_printf()`. The CDL option `CYGNUM_HAL_VIRTUAL_VECTOR_DEBUG_CHANNEL` selects the serial port channel to use for GDB communication by default. Note that when using RedBoot, these options are usually inactive as it is RedBoot that decides which channels are used. Applications may override RedBoot's selections by enabling the `CYGSEM_HAL_VIRTUAL_VECTOR_CLAIM_COMMS` CDL option in the HAL. Baud rates for each channel are set with the `CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL_BAUD` and `CYGNUM_HAL_VIRTUAL_VECTOR_DEBUG_CHANNEL_BAUD` options.

Interrupt-driven Serial Driver

The second form of support is an interrupt-driven serial driver, which is integrated into the eCos standard serial I/O infrastructure (`CYGPKG_IO_SERIAL`). This support can be enabled on any port.

Note that it is not recommended to share this driver when using the HAL diagnostic I/O on the same port. If the driver is shared with the GDB debugging port, it will prevent Ctrl-C operation when debugging.

The main part of this driver is contained in the generic `CYGPKG_IO_SERIAL_GENERIC_16X5X` package. The package `CYGPKG_IO_SERIAL_ARM_ALTERA_HPS` contains definitions that configure the generic driver for the HPS. That driver package should also be consulted for documentation and configuration options. The driver is not active until the `CYGPKG_IO_SERIAL_DEVICES` configuration option within the generic serial driver support package `CYGPKG_IO_SERIAL` is enabled in the configuration.

Support for hardware flow control and modem control lines is present in the driver, but should only be enabled if these control signals are brought out to the physical serial port.

Name

Multimedia Card Interface (MMC/SD) Driver — Using MMC/SD cards with block drivers and filesystems

Overview

The MultiMedia Card Interface (MMC/SD) driver in this processor HAL package allows use of MultiMedia Cards (MMC cards) and Secure Digital (SD) flash storage cards within eCos, exported as block devices. This makes them suitable for use as the underlying devices for filesystems such as FAT.

The HPS architecture only has support for one SDMMC controller.

Configuration

This driver provides the necessary support for the generic MMC bus layer within the `CYGPKG_DEVS_DISK_MMC` package to export a disk block device. The disk block device is only available if the generic disk I/O layer found in the package `CYGPKG_IO_DISK` is included in the configuration.

The block device may then be used as the device layer for a filesystem such as FAT. Example devices are "`/dev/mmcsd0/1`" to refer to the first partition on the card, or "`/dev/mmcsd0/0`" to address the whole device including potentially the partition table at the start.

The driver may be forcibly disabled within this processor HAL package with the configuration option `CYGPKG_HAL_ARM_CORTEXA_ALTERA_HPS_MMC`.

If the driver is enabled, the following options are available to control it:

`CYGPKG_HAL_ARM_CORTEXA_ALTERA_HPS_MMC_DMA_DESCRIPTORS`

> This option specifies the number of descriptors in the DMA ringbuffer. This value can be tuned if needed to optimise performance vs the memory footprint.

`CYGIMP_HAL_ARM_CORTEXA_ALTERA_HPS_MMC_INTMODE`

> This indicates that the driver should operate in interrupt-driven mode if possible. This is enabled by default if the eCos kernel is enabled. Note though that if the driver finds that global interrupts are off when running, then it will fall back to polled mode even if this option is enabled. This allows for use of the MMC/SD driver in an initialisation context.

`CYGNUM_HAL_ARM_CORTEXA_ALTERA_HPS_MMC_INT_PRI`

> This is the MMC/SD bus interrupt priority. It may range from 0 to 255.

`CYGPKG_HAL_SDMMC_DWC_INSTRUMENTATION`

> When the `CYGPKG_KERNEL_INSTRUMENTATION` support is configured, this option allows control over whether SDMMC device driver instrumentation is generated. Sub-options are provided to control which event code instrumentation records are generated. Normally this option is only useful when developing/debugging the driver; but it can be used to monitor performance of the driver in a running system.

Usage Notes

The driver will detect the appropriate card sizes.

The MMC/SD bus layer will parse partition tables, although it can be configured to allow handling of cards with no partition table.

This driver implements multi-sector I/O operations. If you are using the FAT filesystem, see the generic MMC/SD driver documentation which describes how to exploit this feature when using FAT.

Note

Any cards removed from a socket before an explicit unmounting or a `sync()` call to flush filesystem buffers will likely result in a corrupted filesystem on the removed card.

Name

I2C Interface — Using I²C devices

Overview

The I²C driver in the `CYGPKG_DEVS_I2C_DWI2C` supports the use of I²C devices within eCos. Access to the driver will be via the standard I²C interface subsystem.

This driver provides support for all four I²C busses available on the HPS. The number of interfaces supported is defined by the platform HAL.

Configuration

The HAL contains the following configuration options for each the I²C busses:

`CYGINT_HAL_ARM_CORTEXA_ALTERA_HPS_I2C_BUSX`

> This interface controls the inclusion of support for I²C bus *X*. This will normally be implemented by the platform HAL to indicate that there are I²C devices attached to the given bus, or that the SCL and SDA lines are routed to an external connector.

`CYGNUM_HAL_ARM_CORTEXA_ALTERA_HPS_I2C_BUSX_CLOCK`

> This is the I²C bus *X* clock speed in Hz. Frequencies of either 100kHz or 400kHz can be chosen, the latter sometimes known as fast mode.

`CYGNUM_HAL_ARM_CORTEXA_ALTERA_HPS_I2C_BUSX_INTR_PRI`

> This is the I²C bus *X* interrupt priority. It may range from 1 to 255; the default of 128 places it in the centre of the priority range.

Usage Notes

The design of the I²C device does not make it possible to start a new bus transfer without also sending a START condition on the bus. This means that divided transactions are not possible. A divided transaction would look like this:

```
    cyg_i2c_transaction_begin(&cyg_aardvark_at24c02);
    cyg_i2c_transaction_tx(&cyg_aardvark_at24c02, 1, tx_buf1, 1, 0);
    cyg_i2c_transaction_tx(&cyg_aardvark_at24c02, 0, tx_buf2, 2, 0);
    cyg_i2c_transaction_tx(&cyg_aardvark_at24c02, 0, tx_buf3, 6, 1);
    cyg_i2c_transaction_end(&cyg_aardvark_at24c02);
```

In this transaction a START and one byte are sent from tx_buf1, then 2 bytes of data from tx_buf2, finishing with 6 bytes from tx_buf3 followed by a STOP. The device will not allow the tx_buf2 and tx_buf3 transfers to happen without also sending a START. The only solution to this is to combine the data into a single buffer and perform a single transfer:

```
    memcpy( tx_buf, tx_buf1, 1);
    memcpy( tx_buf+1, tx_buf2, 2);
    memcpy( tx_buf+3, tx_buf3, 6);
    cyg_i2c_tx(&cyg_aardvark_at24c02, tx_buf, 9);
```

Name

Pin Configuration and GPIO Support — Use of pin configuration and GPIO

Synopsis

#include <cyg/hal/hal_io.h>

`pin = **CYGHWR_HAL_HPS_PINMUX**(*reg*, *func*);`

CYGHWR_HAL_HPS_PINMUX_SET (*pin*);

pin = **CYGHWR_HAL_HPS_GPIO**(*line*, *mode*);

CYGHWR_HAL_HPS_GPIO_SET (*pin*);

CYGHWR_HAL_HPS_GPIO_OUT (*pin*, *val*);

CYGHWR_HAL_HPS_GPIO_IN (*pin*, *val*);

Description

The HPS HAL provides a number of macros to support the encoding of pin multiplexing information and GPIO pin modes into 32 bit descriptors. This is useful to drivers and other packages that need to configure and use different lines for different devices. Because there is not a simple correspondence between pin multiplexing information and GPIO line numbers, these two things are treated separately.

Pin Multiplexing

A pin multiplexing descriptor is created with `CYGHWR_HAL_HPS_PINMUX( reg, func )` which takes the following arguments:

reg This identifies the PINMUX register which controls this pin. This consists of the name of a pin group parameterized by the pin within that group.

func This defines the function code to program into the PINMUX register. There is no systematic consistency between functions and function codes, and the same function for a pin may be represented by different codes in different PINMUX registers. You should refer to the HPS documentation for the correct value to be used here.

The following examples show how this macro may be used:

```
// UART0 RX line is controlled by register GENERALIO(17), function 2 = UART0.RX
#define CYGHWR_HAL_HPS_UART0_RX                 CYGHWR_HAL_HPS_PINMUX(GENERALIO(17),2)

// GPIO 41 is controlled by register FLASHIO(5), function 0 = GPIO 41
#define CYGHWR_HAL_HPS_LED0                     CYGHWR_HAL_HPS_PINMUX(FLASHIO(5),0)
```

The macro `CYGHWR_HAL_HPS_PINMUX_SET( pin )` sets the pin multiplexing setting according to the descriptor passed in.

GPIO Support

A GPIO descriptor is created with `CYGHWR_HAL_HPS_GPIO( line, mode)` which takes the following arguments:

line This gives the GPIO line number, between 0 and 70.

mode This defines whether this is an input or an output pin, and whether it is de-bounced. It may take the values `IN` or `OUT`, or `INDB` for de-bounced input. GPIO lines 48 to 70 can take their input from one of two pins; modes `IN_ALT` and `INDB_ALT` cause the GPIO line to be connected to the alternate input pin.

Additionally, the macro `CYGHWR_HAL_HPS_GPIO_NONE` may be used in place of a pin descriptor and has a value that no valid descriptor can take. It may therefore be used as a placeholder where no GPIO pin is present or to be used.

The following examples show how this macro may be used:

```
// LED 0 is at GPIO41, bank 1 bit 13, output mode
#define CYGHWR_HAL_HPS_LED0_GPIO                CYGHWR_HAL_HPS_GPIO( 41, OUT )
```

The remaining macros all take a GPIO pin descriptor as an argument. `CYGHWR_HAL_HPS_GPIO_SET` configures the pin according to the descriptor and must be called before any other macros. `CYGHWR_HAL_HPS_GPIO_OUT` sets the output to the value of the least significant bit of the *val* argument. The *val* argument of `CYGHWR_HAL_HPS_GPIO_IN` should be a pointer to an int, which will be set to 0 if the pin input is zero, and 1 otherwise.

Chapter 272. Broadcom IProc Support

Name

Support for the Broadcom IProc — Overview

Description

This document covers the configuration and usage of the Hardware Abstraction Layer (HAL) for the IProc subsystem present on some Broadcom devices. It is expected to be read in conjunction with platform HAL-specific documentation, as well as the eCos HAL specification. This processor HAL package complements the ARM architectural HAL, Cortex-A variant HAL and the platform HAL. It provides functionality common to all IProc-based implementations.

This support is found in the eCos package located at `packages/hal/arm/cortexa/broadcom_iproc` within the eCos source repository.

The Broadcom IProc HAL package is loaded automatically when eCos is configured for an IProc-based platform. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Supported Hardware

Supported features of the IProc within this processor HAL package include:

- IProc-specific hardware definitions
- Interrupt controller selection
- Timers
- Serial UART configuration

Support for the on-chip QSPI device, SPI NOR flash and interrupt-driven serial are also present and can be found in separate packages, outside of this processor HAL.

Note that the HAL does not currently contain support for the Cortex-A's NEON SIMD engine.

Name

IProc Hardware Definitions — Details on obtaining hardware definitions for IProc

Register definitions

The file `<cyg/hal/broadcom_iproc.h>` can be included from application and eCos package sources to provide definitions related to IProc subsystems. These include base addresses for various devices and register definitions for IProc-specific devices such as the Clock Reset Manager and Device Management Unit. The registers for various devices, such as QSPI and the UARTs, are defined in the drivers for those devices. The interrupt controller and system timer are implemented in the Cortex-A HAL. This file will normally be included automatically if `<cyg/hal/hal_io.h>` is included, which is the preferred way of getting these definitions.

Initialization Helper Macros

The file `<cyg/hal/broadcom_iproc_init.inc>` contains definitions of helper macros which may be used by IProc platform HALs in order to initialize common subsystems without excessive duplication between these platform HALs. Typically this file will be included by the `hal_platform_setup.h` header in the platform HAL, in turn included from the architectural HAL file `vectors.S`.

This file is solely intended to be used by platform HALs. At the same time, it is only present to assist initialization, and platform HALs are not obliged to use it if their startup requirements vary.

Name

GIC Interrupt Controller — Advanced Interrupt Controller Definitions and Usage

Interrupt Controller Definitions

The file `<cyg/hal/var_ints.h>` (located at `hal/arm/cortexa/broadcom_iproc/`*`VERSION`*`/include/var_ints.h` in the eCos source repository) contains interrupt vector number definitions for use with the eCos kernel and driver interrupt APIs.

The list of interrupt vectors may be augmented on a per-platform basis. Consult the platform HAL documentation for your platform for whether this is the case.

Interrupt Controller Functions

The IProc uses a standard ARM Generic Interrupt Controller (GIC), which is implemented in the Cortex-A HAL. The Cortex-A HAL exports the standard interrupt vector management functions. The `hal_IRQ_handler` queries the IRQ status register to determine the interrupt cause. Functions `hal_interrupt_mask` and `hal_interrupt_unmask` enable or disable interrupts within the interrupt controller.

Interrupts are configured in the `hal_interrupt_configure` function. Refer to the IProc documentation for any limitations as to what types of signal can be detected.

The `hal_interrupt_eoi` function performs End-Of-Interrupt processing and is called automatically by the architecture HAL. The `hal_interrupt_acknowledge` function is intended to acknowledge an interrupt, although for the GIC it is a NUL function since the necessary work it handled by the EOI function.

The `hal_interrupt_set_level` is used to set the priority level of the supplied interrupt within the GIC. Priorities may range between 0 and 255, with lower values mapping to higher priority. The GIC in the IProc only implements the top 5 bits of the priority value, so there are actually 32 distinct priority levels available.

Note that in all the above, it is not recommended to call the described functions directly. Instead either the HAL macros (`HAL_INTERRUPT_MASK` et al) or preferably the kernel or driver APIs should be used to control interrupts.

Name

Timers — Use of on-chip timers

System Clock

The eCos kernel system clock is implemented using the private timer of CPU0. By default, the system clock interrupts once every 10ms, corresponding to a 100Hz clock. This can be changed by the configuration option `CYGNUM_HAL_RTC_DENOMINATOR` which corresponds to the clock frequency. Other clock-related settings are recalculated automatically if the denominator is changed. If the desired frequency cannot be expressed accurately solely with changes to `CYGNUM_HAL_RTC_DENOMINATOR`, then the configuration option `CYGNUM_HAL_RTC_NUMERATOR` may also be adjusted. However, if this is done then `CYGNUM_HAL_RTC_PERIOD` must be changed to provide the described clock frequency.

The same Timer is also used to implement the HAL microsecond delay function, `HAL_DELAY_US`. This is used by some device drivers, and in non-kernel configurations, such as RedBoot, where this timer is needed for loading program images via X/Y-modem protocols and debugging. Standalone applications which require RedBoot services, such as debugging, should avoid use of this timer.

Timer-based profiling support

If the gprof package, `CYGPKG_PROFILE_GPROF`, is included in the configuration, then the MPCore global timer is reserved for use by the profiler.

Name

Serial UARTs — Configuration and Implementation Details of Serial UART Support

Overview

Support is included in this processor HAL package for the on-chip serial UART devices.

There are two forms of support: HAL diagnostic I/O; and a fully interrupt-driven serial driver. Unless otherwise specified in the platform HAL documentation, for all serial ports the default settings are 115200,8,N,1 with no flow control.

HAL Diagnostic I/O

This first form is polled mode HAL diagnostic output, intended primarily for use during debug and development. Operations are usually performed with global interrupts disabled, and thus this mode is not usually suitable for deployed systems. This can operate on any port, according to the configuration settings.

There are several configuration options usually found within a platform HAL which affect the use of this support in the IProc processor HAL. The CDL option CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL selects the serial port channel to use as the console at startup time. This will be the channel that receives output from, for example, diag_printf(). The CDL option CYGNUM_HAL_VIRTUAL_VECTOR_DEBUG_CHANNEL selects the serial port channel to use for GDB communication by default. Note that when using RedBoot, these options are usually inactive as it is RedBoot that decides which channels are used. Applications may override RedBoot's selections by enabling the CYGSEM_HAL_VIRTUAL_VECTOR_CLAIM_COMMS CDL option in the HAL. Baud rates for each channel are set with the CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL_BAUD and CYGNUM_HAL_VIRTUAL_VECTOR_DEBUG_CHANNEL_BAUD options.

Interrupt-driven Serial Driver

The second form of support is an interrupt-driven serial driver, which is integrated into the eCos standard serial I/O infrastructure (CYGPKG_IO_SERIAL). This support can be enabled on any port.

Note that it is not recommended to share this driver when using the HAL diagnostic I/O on the same port. If the driver is shared with the GDB debugging port, it will prevent Ctrl-C operation when debugging.

The main part of this driver is contained in the generic CYGPKG_IO_SERIAL_GENERIC_16X5X package. The package CYGPKG_IO_SERIAL_ARM_BROADCOM_IPROC contains definitions that configure the generic driver for the IProc. That driver package should also be consulted for documentation and configuration options. The driver is not active until the CYGPKG_IO_SERIAL_DEVICES configuration option within the generic serial driver support package CYGPKG_IO_SERIAL is enabled in the configuration.

Support for hardware flow control and modem control lines is present in the driver, but should only be enabled if these control signals are brought out to the physical serial port.

Chapter 273. Broadcom BCM283X Processor Support

Name

Support for the Broadcom BCM283X Processor Family — Overview

Description

This document covers the configuration and usage of the Hardware Abstraction Layer (HAL) for the Broadcom BCM283X devices present on the Raspberry Pi boards. It is expected to be read in conjunction with platform HAL-specific documentation, as well as the eCos HAL specification. This processor HAL package complements the ARM architectural HAL, Cortex-A variant HAL and the Raspberry Pi platform HAL. It provides functionality common to all BCM283X-based implementations.

This support is found in the eCos package located at `packages/hal/arm/cortexa/bcm283x` within the eCos source repository.

The BCM283X HAL package is loaded automatically when eCos is configured for any Raspberry Pi board. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Supported Hardware

Supported features of the BCM283X within this processor HAL package include:

- BCM283X-specific hardware definitions
- Interrupt Controller Support
- Timers
- Serial UART configuration
- I²C two wire interface configuration
- GPIO Support
- DMA Support

Support for I²C, SPI, USB, Ethernet, MMC/SD and interrupt-driven serial features of the BCM283X are also present and can be found in separate packages, outside of this processor HAL.

Support for SMP operation of the four Cortex-A CPUs in the BCM2836 and BCM2837 is available, although debugging support is restricted to use of an external JTAG debugger.

Name

Hardware Definitions — Details on obtaining hardware definitions for BCM283X

Register definitions

The file `<cyg/hal/bcm283x.h>` can be included from application and eCos package sources to provide definitions related to BCM283X subsystems. These include base addresses for various devices and register definitions for BCM283X-specific devices such as the interrupt controller, timers, mailboxes, DMA and GPIO. The registers for various devices, such as I²C, SPI, MMC/SD, and the UARTs are defined in the drivers for those devices. This file will normally be included automatically if `<cyg/hal/hal_io.h>` is included, which is the preferred way of getting these definitions.

Initialization Helper Macros

The file `<cyg/hal/bcm283x_init.inc>` contains definitions of helper macros which may be used by platform HALs in order to initialize common subsystems without excessive duplication between these platform HALs. Typically this file will be included by the `hal_platform_setup.h` header in the platform HAL, in turn included from the architectural HAL file `vectors.S`.

This file is solely intended to be used by platform HALs. At the same time, it is only present to assist initialization, and platform HALs are not obliged to use it if their startup requirements vary. NOTE: At present, the only extant BCM283X port relies on the GPU to initialize the PLLs and memory controller, so these macros currently largely contain Cortex-A-generic setup only.

Name

Interrupt Controller — Interrupt Controller Definitions and Usage

Interrupt Vector Assignments

The file `<cyg/hal/var_ints.h>` (located at `hal/arm/cortexa/bcm283x/`*`VERSION`*`/include/var_ints.h` in the eCos source repository) contains interrupt vector number definitions for use with the eCos kernel and driver interrupt APIs.

The BCM283X interrupt controller is somewhat unsystematic and in order to derive some efficiency from it, the eCos interrupt vector table is large and in some places sparse. Some secondary decoding is also performed to demultiplex some shared vectors. The following table shows the overall interrupt mapping scheme.

Range	**Description**
0..31	IRQ pending register 1
32..63	IRQ pending register 2
64..84	IRQ basic pending register
85..95	Unused, reserved for expansion
96..111	CPU0, per-CPU vectors
112..127	CPU1, per-CPU vectors
128..143	CPU2, per-CPU vectors
144..159	CPU3, per-CPU vectors
160..162	Auxiliary device vectors -- demultiplexed
163..216	GPIO pin vectors -- demultiplexed

Interrupt Controller Functions

The BCM283X HAL exports the standard interrupt vector management functions which are in turn called by the generic interrupt management macros.

The `hal_IRQ_handler()` function queries the IRQ pending registers to determine the interrupt cause. Functions `hal_interrupt_mask()` and `hal_interrupt_unmask()` enable or disable interrupts within the interrupt controller.

The BCM283X does not have any provision for prioritizing interrupts, setting their edge/level, per-vector CPU affinity, or even acknowledge individual vectors. So the functions `hal_interrupt_set_level`, `hal_interrupt_configure`, `hal_interrupt_set_cpu`, `hal_interrupt_get_cpu` and `hal_interrupt_acknowledge` are all empty functions.

The only available mechanism in the BCM283X for redirecting interrupts to different CPUs is to move all device interrupt to a given CPU. This does not match the model to which eCos has been implemented. So, all device interrupts are directed to CPU0, which cannot be changed. CPUs 1 to 3 only handle explicitly multi-core interrupts.

Note that in all the above, it is not recommended to call the described functions directly. Instead either the HAL macros (`HAL_INTERRUPT_MASK` et al) or preferably the kernel or driver APIs should be used to control interrupts.

Name

Timers — Use of on-chip timers

System Clock

The eCos kernel system clock is implemented using the BCM283X System Timer device. By default, the system timer is programmed to interrupt once every 10ms, corresponding to a 100Hz clock. This can be changed with the configuration option `CYGNUM_HAL_RTC_DENOMINATOR` which corresponds to the clock frequency. Other clock-related settings are recalculated automatically if the denominator is changed. The system timer is clocked at 1MHz, so there are limits on the accuracy of any frequency that is not a factor of 1000000.

The same timer is also used to implement the HAL microsecond delay function, `HAL_DELAY_US`. This is used by some device drivers, and in non-kernel configurations, such as RedBoot, where this timer is needed for loading program images via X/Y-modem protocols and debugging via TCP/IP. Standalone applications which require RedBoot services, such as debugging, should avoid use of this timer.

Timer-based Profiling Support

At present timer based profiling is not supported. This is mainly because the current profiling support is not SMP-aware.

Name

Serial UARTs — Configuration and Implementation Details of Serial UART Support

Overview

Support is included in this processor HAL package for the auxiliary mini UART device.

There are two forms of support: HAL diagnostic I/O; and a fully interrupt-driven serial driver. Unless otherwise specified in the platform HAL documentation, for all serial ports the default settings are 115200,8,N,1 with no flow control.

HAL Diagnostic I/O

This first form is polled mode HAL diagnostic output, intended primarily for use during debug and development. Operations are usually performed with global interrupts disabled, and thus this mode is not usually suitable for deployed systems.

There are several configuration options usually found within a platform HAL which affect the use of this support in the BCM283X processor HAL. The CDL option CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL selects the serial port channel to use as the console at startup time. This will be the channel that receives output from, for example, diag_printf(). The CDL option CYGNUM_HAL_VIRTUAL_VECTOR_DEBUG_CHANNEL selects the serial port channel to use for GDB communication by default. Note that when using RedBoot, these options are usually inactive as it is RedBoot that decides which channels are used. Applications may override RedBoot's selections by enabling the CYGSEM_HAL_VIRTUAL_VECTOR_CLAIM_COMMS CDL option in the HAL. Baud rates for each channel are set with the CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL_BAUD and CYGNUM_HAL_VIRTUAL_VECTOR_DEBUG_CHANNEL_BAUD options.

Interrupt-driven Serial Driver

The second form of support is an interrupt-driven serial driver, which is integrated into the eCos standard serial I/O infrastructure (CYGPKG_IO_SERIAL).

Note that it is not recommended to share this driver when using the HAL diagnostic I/O on the same port. If the driver is shared with the GDB debugging port, it will prevent Ctrl-C operation when debugging.

The main part of this driver is contained in the generic CYGPKG_IO_SERIAL_GENERIC_16X5X package. The package CYGPKG_IO_SERIAL_ARM_BCM283X contains definitions that configure the generic driver for the BCM283X. That driver package should also be consulted for documentation and configuration options. The driver is not active until the CYGPKG_IO_SERIAL_DEVICES configuration option within the generic serial driver support package CYGPKG_IO_SERIAL is enabled in the configuration.

Support for hardware flow control is present in the driver, but should only be enabled if the RTS and CTS control signals are connected to accessible GPIO lines. The mini UART is also lacking some 16550 functionality, it is only capable of 7 or 8 bit characters, and only supports one stop bit and no parity; attempts to select other settings will be accepted, but will have no effect.

Name

I²C Interface — Using I²C devices

Overview

The I²C driver in the CYGPKG_DEVS_I2C_BSC package supports the use of I²C devices within eCos. Access to the driver will be via the standard I²C interface subsystem.

This driver can provide support for all three I²C busses available on the BCM283X. However, I2C0 is reserved for use by the GPU and I2C2 is dedicated to the HDMI interface, so in practice only I2C1 is available.

Configuration

The HAL contains the following configuration options for each the I²C busses:

CYGINT_HAL_ARM_CORTEXA_BCM283X_I2C_BUSX

> This interface controls the inclusion of support for I²C bus *X*. This will normally be implemented by the platform HAL to indicate that there are I²C devices attached to the given bus, or that the SCL and SDA lines are routed to an external connector.

CYGNUM_HAL_ARM_CORTEXA_BCM283X_I2C_BUSX_CLOCK

> This is the I²C bus *X* clock speed in Hz. Frequencies of either 100kHz or 400kHz can be chosen, the latter sometimes known as fast mode.

Usage Notes

The design of the I²C device does not make it possible to start a new bus transfer without also sending a START condition on the bus. This means that divided transactions are not possible. A divided transaction would look like this:

```
    cyg_i2c_transaction_begin(&cyg_aardvark_at24c02);
    cyg_i2c_transaction_tx(&cyg_aardvark_at24c02, 1, tx_buf1, 1, 0);
    cyg_i2c_transaction_tx(&cyg_aardvark_at24c02, 0, tx_buf2, 2, 0);
    cyg_i2c_transaction_tx(&cyg_aardvark_at24c02, 0, tx_buf3, 6, 1);
    cyg_i2c_transaction_end(&cyg_aardvark_at24c02);
```

In this transaction a START and one byte are sent from tx_buf1, then 2 bytes of data from tx_buf2, finishing with 6 bytes from tx_buf3 followed by a STOP. The device will not allow the tx_buf2 and tx_buf3 transfers to happen without also sending a START. The only solution to this is to combine the data into a single buffer and perform a single transfer:

```
    memcpy( tx_buf, tx_buf1, 1);
    memcpy( tx_buf+1, tx_buf2, 2);
    memcpy( tx_buf+3, tx_buf3, 6);
    cyg_i2c_tx(&cyg_aardvark_at24c02, tx_buf, 9);
```

Name

GPIO Support — Use of GPIO

Synopsis

#include <cyg/hal/hal_io.h>

`desc =` **`CYGHWR_HAL_BCM283X_GPIO`**`(`*`pin`*`, `*`alt`*`, `*`mode`*`);`

`desc =` **`CYGHWR_HAL_BCM283X_GPIO_VAR`**`(`*`pin`*`, `*`alt`*`, `*`mode`*`);`

`CYGHWR_HAL_BCM283X_GPIO_SET` `(`*`desc`*`);`

`desc =` **`CYGHWR_HAL_BCM283X_GPIO_GET`** `(`*`pin`*`);`

`CYGHWR_HAL_BCM283X_GPIO_OUT` `(`*`desc`*`, `*`val`*`);`

`CYGHWR_HAL_BCM283X_GPIO_IN` `(`*`desc`*`, `*`val`*`);`

Description

The BCM283X HAL provides a number of macros to support the encoding of pin multiplexing and GPIO pin modes into a 32 bit descriptor. This is useful to drivers and other packages that need to configure and use different lines for different devices. Pin multiplexing is handled in the GPIO controller, so it is handled by the same interface.

A GPIO descriptor is created with `CYGHWR_HAL_BCM283X_GPIO( pin, alt, mode)` which takes the following arguments:

`pin`
: This gives the GPIO pin number, between 0 and 53.

`alt`
: This gives the pin's function. It may be one of `GPIO_IN`, `GPIO_OUT`, `ALT0`, `ALT1`, `ALT2`, `ALT3`, `ALT4` or `ALT5`. The first two set the pin to be a GPIO input or output respectively, the remainder select one of six alternate functions for the pin, usually assigning it to a particular device signal. The alternate function mappings can be found in the BCM2835 documentation, and may also be seen using the RedBoot **gpio table** command.

`mode`
: This defines any additional properties that this pin should have. These either define the signal input condition on which an interrupt is raised, or the nature of any pull-up or -down to be applied to the pin. The values `RISING` and `FALLING` program the line to interrupt on a rising or falling edge; similarly `HIGH` and `LOW` interrupt on high or low levels. The values `PULL_DOWN` and `PULL_UP` apply a pull resistor in the given direction.

  The value in this argument is the last element of a macro of the form, `CYGHWR_HAL_BCM283X_GPIO_MODE_XXXXX`. It is possible to define additional macros than enable combinations of modes. The HAL defines some such macros: `EDGE` causes and interrupt on any signal edge; `PULL_UP_FALLING` applies a pull up on the line and interrupts on a falling edge, and `PULL_DOWN_RISING` does the inverse.

Additionally, the macro `CYGHWR_HAL_BCM283X_GPIO_NONE` may be used in place of a pin descriptor and has a value that no valid descriptor can take. It may therefore be used as a placeholder where no GPIO pin is present or to be used. It may be passed to any GPIO macro which takes a descriptor and will be treated as a no-op.

The following examples show how this macro may be used:

```
// ACT LED is on GPIO16 on some boards
#define CYGHWR_HAL_PI_LED                       CYGHWR_HAL_BCM283X_GPIO( 16, GPIO_OUT, NONE )

// UART TX and TX pins are on GPIO14 and GPIO15
#define CYGHWR_HAL_BCM283X_UART0_TX             CYGHWR_HAL_BCM283X_GPIO( 14, ALT0, NONE )
```

```
#define CYGHWR_HAL_BCM283X_UART0_RX             CYGHWR_HAL_BCM283X_GPIO( 15, ALT0, NONE )
```

Most of the remaining macros all take a GPIO descriptor as an argument. `CYGHWR_HAL_BCM283X_GPIO_SET` configures the pin according to the descriptor and must be called before any the following macros. `CYGHWR_HAL_BCM283X_GPIO_OUT` sets the pin output to the value of the least significant bit of the *val* argument. The *val* argument of `CYGHWR_HAL_BCM283X_GPIO_IN` should be a pointer to an int, which will be set to 0 if the pin input is zero, and 1 otherwise.

The macro `CYGHWR_HAL_BCM283X_GPIO_GET` takes a pin number as an argument and returns a descriptor that encodes the current state of that pin. The `CYGHWR_HAL_BCM283X_GPIO_VAR` performs the same descriptor encoding as `CYGHWR_HAL_BCM283X_GPIO` except that the arguments are not interpreted as macro name fragments. This is useful for generating a descriptor at runtime from variable field values.

Name

DMA Support — Description

Synopsis

#include <cyg/hal/bcm283x_dma.h>

ok = **hal_dma_channel_init**(*hal_dma_channel *chan*, *cyg_uint8 permap*, *cyg_bool fast*, *hal_dma_callback *callback*, *CYG_ADDRWORD data*);

hal_dma_channel_delete(*hal_dma_channel *chan*);

hal_dma_channel_set_polled(*hal_dma_channel *chan*, *cyg_bool polled*);

hal_dma_poll(*void*);

hal_dma_cb_init(*hal_dma_cb *cb*, *cyg_uint32 ti*, *void *source*, *void *dest*, *cyg_uint32 size*);

hal_dma_add_cb(*hal_dma_channel *chan*, *hal_dma_cb *cb*);

hal_dma_channel_start(*hal_dma_channel *chan*);

hal_dma_channel_stop(*hal_dma_channel *chan*);

Description

The HAL provides support for access to the DMA channels. This support is not intended to expose the full functionality of these devices and is mainly limited to supporting peripheral DMA. The API is therefore mainly oriented for use by device drivers rather than applications. The user is referred to the BCM2835 documentation for full details of the DMA channels, and to the SDHOST driver for an example of this API in use.

The DMA hardware consist of sixteen independent channels. DMA is initiated by attaching a chain of control blocks to a channel and starting it running. Each control block contains source and destination addresses, size, transfer direction and a number of other parameters. Of the sixteen channels available, some are reserved for use by the GPU. Also, the channels are divided into full function channels and *lite* channels that lack some functionality and have lower bandwidth. Full details are available in the BCM2835 documentation.

A DMA channel is represented by a hal_dma_channel object that the client must allocate. Control blocks are similarly represented by a hal_dma_cb object, which again must be allocated by the client. DMA control blocks must be aligned on a 32 byte boundary. The type definition in `bcm283x_dma.h` has an alignment attribute so that static allocations should be correctly aligned by default; however care should be taken to align dynamic allocations.

A DMA channel is initialized by calling `hal_dma_channel_init`, the parameters are as follows:

chan
: A pointer to the channel object to be initialized.

permap
: Peripheral map value. This is one of the `CYGHWR_HAL_BCM283X_DMA_DREQ_XXXX` values defined in `bcm238x.h`. It specifies the peripheral to or from which the transfer will be made.

fast
: This specifies whether the DMA channel should be a full featured fast channel or a reduced bandwidth lite channel. If a lite channel is specified and none are available a fast channel will be allocated. However, if a fast channel is specified and none are available then this routine will return 0 to indicate an error.

callback
: A pointer to a function that will be called when the DMA transfer has been completed. This will be called with a pointer to the channel, an event code, nd a copy of the *data* parameter. The event code will be ei-

ther CYGHWR_REG_BCM283X_DMA_CS_END to indicate a successful completion of the transfer or CYGHWR_REG_BCM283X_DMA_CS_ERROR to indicate an error.

data An uninterpreted data value that will be passed to the callback. This would typically be a pointer to a client data structure.

If the initialization is successful the routine will return 1. The current implementation of the DMA API permanently allocates a physical channel when this routine is called. In the future the allocation of physical channels may be more dynamic, so the client should not assume that the channel in use is constant.

The hal_dma_channel_delete function deletes the given channel, releasing any resources and making them available for reuse.

The hal_dma_channel_set_polled function marks a channel for polled operation only. Otherwise the channel will enable interrupts and wait for an interrupt to complete. If a channel is marked polled then it will only be completed and its callback called during calls to hal_dma_poll. Note that channels not marked polled may also be completed during this call if their interrupt has not yet fired.

A transfer control block is initialized by calling hal_dma_cb_init. The parameters are as follows:

cb A pointer to the control block to be initialized.

ti This is an initial value for the TI register field of the control block. This may contain any of the bits and fields specified for this register except the PERMAP field, which will be set from the value set in the channel. For simplicity the standard settings for common operations are defined by the DMA API; HAL_DMA_INFO_DEV2MEM initializes the control block for a single buffer transfer from a device to memory, and HAL_DMA_INFO_MEM2DEV for a transfer in the reverse direction. If a client needs to perform scatter/gather transfers, then it needs to set this argument more explicitly. In particular, the INTEN bit should normally only be set on the last control block of a chain.

source The source address for the transfer, either the start of a memory buffer or the data register of the appropriate device.

dest The destination address for the transfer, either the start of a memory buffer or the data register of the appropriate device.

size Transfer size in bytes.

Once initialized a control block may be added to a channel by calling hal_dma_cb_add. Control blocks will be chained together on the channel in the order in which they are added. The DMA engines operate on addresses in the GPU address space, not the physical address space visible the the ARM CPUs or the virtual address space set up by the MMU. During initialization the source and destination addresses will be translated into GPU addresses, and after it is added, the dma_next field of the control block will be translated to a GPU address. So, care should be taken when inspecting an active control block and it should not be changed.

Once a channel had been initialized and any control blocks have been added the transfers may be started by calling hal_dma_channel_start. For channels not marked polled, interrupts will fire and the callback will be called from a DSR when the control block chain has been completed. For polled channel, it will be necessary to call hal_dma_poll until all channels have completed.

An ongoing transfer may be halted by calling hal_dma_channel_stop. This function should also be called as a matter of course when a transfer has completed normally.

Name

GPU Communication Support — Use of GPU mailbox

Synopsis

#include <cyg/hal/hal_io.h>

```
void hal_bcm283x_board_model(cyg_uint32 *model);
```

```
void hal_bcm283x_board_revision(cyg_uint32 *revision);
```

```
void hal_bcm283x_board_serial(cyg_uint64 *serial);
```

```
void hal_bcm283x_mac_address(cyg_uint8 *macaddr);
```

```
void hal_bcm283x_arm_memory(cyg_uint32 *base, cyg_uint32 *size);
```

```
void hal_bcm283x_clock_rate(cyg_uint32 clock, cyg_uint32 *rate);
```

```
void hal_bcm283x_clock_rate_max(cyg_uint32 clock, cyg_uint32 *rate);
```

```
void hal_bcm283x_clock_rate_min(cyg_uint32 clock, cyg_uint32 *rate);
```

```
void hal_bcm283x_clock_rate_set(cyg_uint32 clock, cyg_uint32 rate);
```

```
void hal_bcm283x_dma_channels(cyg_uint32 *channels);
```

```
void hal_bcm283x_set_gpu_gpio(cyg_uint32 pin, cyg_uint32 val);
```

```
void hal_bcm283x_temperature(cyg_uint32 id, cyg_uint32 *temperature);
```

```
void hal_bcm283x_temperature_max(cyg_uint32 id, cyg_uint32 *temperature);
```

```
void hal_bcm283x_property_exchange(cyg_uint32 channel, void *prop, int size);
```

Description

The BCM283X GPU provides information about and control over various aspects of the system. A hardware mailbox mechanism is used to exchange messages with the GPU to query or control these aspects. The BCM283X HAL provides some functions to access this interface. The default functions are limited to just those exchanges that are of direct relevance to eCos, although there is scope for expanding these, or for performing a property exchange directly.

`hal_bcm283x_board_model()` and `hal_bcm283x_board_revision()` fetch the board model and revision numbers. The model number is usually zero, but the revision number encodes the board properties as described Raspberry Pi Revision codes page. Normally the eCos HAL reads the revision code at startup and decodes it into a set of variables.

`hal_bcm283x_board_serial()` returns the board's serial number; this is a 64 bit identifier unique to this board. The function `hal_bcm283x_mac_address()` returns a MAC address for this board; this consists of the lower 24 bits of the serial number combined with a vendor ID assigned to the Raspberry Pi Foundation.

`hal_bcm283x_arm_memory()` returns the base and size of the RAM available to the ARM processor. The base will be zero and the size will indicate the top of memory after the GPU memory specified in the `config.txt` file has been allocated.

`hal_bcm283x_clock_rate()` returns the current rate of the given clock in Hz. Clock identifiers can be found in `bcm238x.h`. `hal_bcm283x_clock_rate_max()` and `hal_bcm283x_clock_rate_min()` return the maximum and minimum rates for the clock. `hal_bcm283x_clock_rate_set()` sets the given clock to the supplied rate.

`hal_bcm283x_dma_channels()` returns a mask of the available DMA channels.

`hal_bcm283x_set_gpu_gpio()` sets the value (0 or 1) of a GPIO pin controlled by the GPU through the external GPIO extender. This is mainly used to control the ACT LED on Pi3B and Pi3A boards.

`hal_bcm283x_temperature()` reads the temperature of the SoC in thousandths of a degree C. `hal_bcm283x_temperature_max()` returns the maximum SoC temperature, above which overclocking will be disabled. In both cases the id should be zero.

`hal_bcm283x_property_exchange()` performs a property exchange with the GPU and may be used to implement requests that are not covered above. The protocol used is described on the Raspberry Pi Mailbox properties GitHub page. This function copies the array of property tags pointed to by *`prop`* into a properly aligned buffer after the standard buffer header and terminates it an end tag. It then sends a mailbox message to the GPU and waits for a response. Finally, the updated property buffer is copied back out to *`prop`*. In SMP systems a spinlock is used to control concurrent access from different CPUs.

Name

Frequency Control — control ARM and CORE frequencies

Synopsis

#include <cyg/hal/hal_io.h>

```
void hal_bcm283x_set_freqencies(cyg_uint32 arm, cyg_uint32 core);
```

Description

The BCM283X GPU has control over the frequencies of the various clocks that feed the hardware. Of these the most important are the ARM clock, which feeds the CPUs and the CORE clock which feeds most of the peripherals. On booting, the HAL reads the current, minimum and maximum values of these clocks and stores them in global variables. The UART and EMMC clocks are also read and recorded. The ARM and CORE clocks are then set to either the maximum values, or values set in the HAL configuration (`CYGHWR_HAL_ARM_CORTEXA_BCM283X_ARM_FREQ` and `CYGHWR_HAL_ARM_CORTEXA_BCM283X_CORE_FREQ`).

eCos does not contain any support for dynamic frequency management in the same way that Linux does. The CPUs run at a single frequency throughout. Normally this is the maximum frequency permitted without overclocking. A different frequency may be selected in the configuration, or it may be set at runtime by calling `hal_bcm283x_set_freqencies()`.

The function `hal_bcm283x_set_freqencies()` updates either or both the ARM or CORE clock frequencies. The frequencies are given in Hz, and if zero are not changed. Values that lie below the minimum frequency for the clock are increased to the minimum and those above the maximum are reduced to the maximum. Other clock-related values, such as the mini-UART baud rate divider (which divides the CORE frequency), may also be changed. Finally, the curent values of the main clocks in the system are read back from the GPU to ensure that the correct values are used.

Chapter 274. Broadcom BCM56150 Reference Board Support

Name

eCos Support for the Broadcom BCM56150 Reference Board — Overview

Description

This document covers the configuration and usage of eCos and RedBoot on the Broadcom BCM956150 SVK reference board. This board is fitted with a BCM56150 CPU and it is referred to in this document and the configuration system as a BCM56150 Reference, or bcm56150_ref, to differentiate it from other Broadcom boards.

In addition to the BCM56150, the board contains 512MiB SDRAM main memory, a 256Mib (32MiB) SPI NOR Flash, a connector for CCA UART1, Ethernet sockets for both the IProc and switch Ethernet interfaces, plus a variety of connectors for other interfaces. The extent of eCos support for the devices and peripherals on the board and the CPU is described below.

For typical eCos development, a RedBoot image is programmed into the SPI NOR flash memory, and the board will load this image from reset. RedBoot provides gdb stub functionality so it is then possible to download and debug eCos applications via the gdb debugger using the serial line.

This documentation is expected to be read in conjunction with the Broadcom IProc processor HAL documentation and further device support and subsystems are described and documented there.

Supported Hardware

The SPI NOR flash consists of 512 blocks of 64KiB each. In a typical setup, the first 13 blocks are reserved for the use of the ROMRAM RedBoot image. One block is reserved to contain DRAM memory parameters. The topmost block is used to manage the flash and also holds RedBoot **fconfig** values. The remaining blocks can be used by application code.

Serial support is through the `CYGPKG_IO_SERIAL_GENERIC_16X5X` generic driver package which is modified by the `CYGPKG_IO_SERIAL_ARM_BROADCOM_IPROC` driver package for the IProc. These packages support both the serial devices on the IProc ChipCommonA device. However, this board only has UART1 connected to an external connector which this HAL indicates by implementing the `CYGINT_HAL_ARM_CORTEXA_BROADCOM_IPROC_UART1` interface. This serial channel is used by RedBoot for communication with the host. If this device is needed by the application, either directly or via the serial driver, then it cannot also be used for RedBoot communication. The serial driver package is loaded automatically when configuring for the `bcm56150-ref` target.

The platform HAL provides definitions to enable access to flash devices on the SPI bus. The HAL enables the QSPI driver (`CYGPKG_DEVS_FLASH_QSPI_IPROC`) which in turn provides the underlying implementation for access to the Micron N25Q256 SPI NOR flash. The QSPI support integrates with the `CYGPKG_DEVS_FLASH_SPI_M25PXX` package. These packages are automatically loaded when configuring for the target. This driver is capable of supporting the JFFS2 filesystem, although at greatly reduced performance compared with a parallel flash device.

In general, devices (Caches, GPIO, UARTs) are initialized only as far as is necessary for eCos to run. Other devices (RTC, QSPI, Ethernet etc.) are not touched unless the appropriate driver is loaded, although in some cases, the HAL boot sequence will set up the appropriate power control and pin multiplexing configuration.

Tools

The board support is intended to work with GNU tools configured for an arm-eabi target. The original port was undertaken using arm-eabi-gcc version 4.7.3, arm-eabi-gdb version 7.2, and binutils version 2.23.2.

Name

Setup — Preparing the BCM56150 Reference board for eCos Development

Overview

In a typical development environment, the board boots from the SPI NOR Flash and runs the RedBoot bootloader and debug agent from SDRAM. eCos applications are configured for RAM startup and then downloaded and run on the board via the debugger **arm-eabi-gdb**. Preparing the board therefore usually involves programming a suitable RedBoot image into flash memory.

The following RedBoot configurations are supported:

Configuration	Description	Use	File
ROMRAM	RedBoot loaded from SPI NOR flash to SDRAM	`redboot_ROMRAM.ecm`	`redboot_ROMRAM.bin`
ROM	RedBoot executed directly from SPI NOR flash	`redboot_ROM.ecm`	`redboot_ROM.bin`

For serial communications, all versions run with 8 bits, no parity, and 1 stop bit at 115200 baud. RedBoot also supports flash management.

By default the ROMRAM startup should be used since it relocates itself to RAM after initializing the SDRAM controller. The ROM startup RedBoot is mainly used for debugging board initialization.

Initial Installation

The BCM56150 Reference board comes with U-Boot and Linux installed by default on the SPI NOR flash. For eCos development we need to install RedBoot in the flash. The booting mechanism is that the on-chip firmware interrogates the boot mode pins and jumps to the start of the memory region corresponding to the device selected. For our purposes, this is the start of the memory-mapped SPI NOR flash.

RedBoot must be written to the first few blocks of the SPI NOR flash. Thirteen sectors or 832KiB are reserved for this. There are several ways that this flash may be written. The N25Q256 is removable and may be programmed externally. The SPI CS/SCLK/MOSI/MISO lines are available on a header and may be used to program the flash from an external device. It may also be programmed using software running on the board. The following explains how to program RedBoot using a Lauterbach Power Debug via Trace32.

Programming RedBoot into NOR flash using Trace32

The following gives the steps needed to program RedBoot into the SPI NOR Flash using a Lauterbach Power Debug Interface and Trace32.

1. Install Trace32 onto your host system. Attach the Power Debug module to the JTAG header (JP1801) and to a USB port on the host. Attach the serial adaptor cable from J1801 to a host serial port and run a suitable terminal communications program (e.g. TeraTerm or Putty on Windows, minicom on Linux). The default serial settings are 115200,8,N,1 with no flow control.

2. Locate the boot selection jumper (JP900) and move the jumper from the 5-6 position to 1-2. Power up the board and the Power Debug.

3. Copy the following files from `hal/arm/cortexa/bcm56150_ref/`*`VERSION`*`/misc` in the repository to a working directory: `ecospro.cmm`, `layout.cmm`, `hr2_prog_sflash_trace32`

4. Copy `redboot_ROMRAM.bin` to the same directory, renaming it `app.bin`.

5. With the work directory as your current directory, start Trace32.

6. From the `File` menu, select `Run Script` and run the `ecospro.cmm` script.

7. From the `eCosPro` menu select the `Program APP.BIN to Flash` entry. This should proceed to load `hr2_prog_sflash_trace32` into on-chip SRAM, erase the flash and program the contents of `app.bin` into flash. This will take a while to complete.

8. Once the programming is finished, shut down Trace32. Power off the board. Detach the Power Debug from the board.

8. Move the jumper on JP900 back to the 5-6 position.

9. Power up the board. The DRAM initialization code may spend some time optimizing the memory parameters. This may take several minutes and result in many lines of diagnostic output. This will only need to be done once, in the future these parameters will be loaded from flash.

10. When the DRAM initialization is finished, output similar to the following will be seen on the serial output:

```
+**Warning** FLASH configuration checksum error or invalid key
RedBoot(tm) bootstrap and debug environment [ROMRAM]
Non-certified release, version UNKNOWN - built 18:05:47, Nov 11 2015

Copyright (C) 2000-2009 Free Software Foundation, Inc.
Copyright (C) 2003-2014 eCosCentric Limited
RedBoot is free software, covered by the eCos license, derived from the
GNU General Public License. You are welcome to change it and/or distribute
copies of it under certain conditions. Under the license terms, RedBoot's
source code and full license terms must have been made available to you.
Redboot comes with ABSOLUTELY NO WARRANTY.

Platform: Broadcom BCM56150 Reference Platform (Cortex-A9)
RAM: 0x60000000-0x80000000 [0x6012c680-0x7ffed000 available]
FLASH: 0x1c000000-0x1dffffff, 512 x 0x10000 blocks
RedBoot>
```

12. Run the following commands to initialize RedBoot's flash file system and flash configuration:

```
RedBoot> fis init
About to initialize [format] FLASH image system - continue (y/n)? y
*** Initialize FLASH Image System
... Erase from 0x1dff0000-0x1dffffff: .
... Program from 0x7fff0000-0x80000000 to 0x1dff0000: .
RedBoot>
RedBoot> fconfig -i
Initialize non-volatile configuration - continue (y/n)? y
Run script at boot: false
Console baud rate: 115200
Update RedBoot non-volatile configuration - continue (y/n)? y
... Erase from 0x1dff0000-0x1dffffff: .
... Program from 0x7fff0000-0x80000000 to 0x1dff0000: .
RedBoot>
```

The RedBoot installation is now complete. This can be tested by power cycling the board again. Output similar to the following should be seen on the serial port.

```
Clocks (MHz): CPU 1000 Periph 500 AXI 400 APB 250
DEV ID = 0xdb56
SKU ID = 0xb150
DDR type: DDR3
MEMC 0 DDR speed = 667MHz
PHY revision version: 0x00024006
ddr_init2: Calling soc_ddr40_set_shmoo_dram_config
ddr_init2: Calling soc_ddr40_phy_calibrate
C01. Check Power Up Reset_Bar
```

```
C02. Config and Release PLL from reset
C03. Poll PLL Lock
C04. Calibrate ZQ (ddr40_phy_calib_zq)
C05. DDR PHY VTT On (Virtual VTT setup) DISABLE all Virtual VTT
C06. DDR40_PHY_DDR3_MISC
C07. VDL Calibration
C07.1
C07.2
C07.4
C07.4.1
C07.4.4
VDL calibration result: 0x30000003 (cal_steps = 0)
C07.4.5
C07.4.6
C07.5
C08. DDR40_PHY_DDR3_MISC : Start DDR40_PHY_RDLY_ODT....
C09. Start ddr40_phy_autoidle_on (MEM_SYS_PARAM_PHY_AUTO_IDLE) ....
C10. Wait for Phy Ready...Done.
DDR phy calibration passed
Programming controller register
ddr_init2: Wait for MemC ready
ddr_init2: MemC initialization complete
Validate Shmoo parameters stored in flash ..... OK
Restoring Shmoo parameters from flash ..... done
Running simple memory test ..... OK
DeepSleep wakeup: ddr init bypassed 3
DDR Interface Ready
+
RedBoot(tm) bootstrap and debug environment [ROMRAM]
Non-certified release, version UNKNOWN - built 18:05:47, Nov 11 2015

Copyright (C) 2000-2009 Free Software Foundation, Inc.
Copyright (C) 2003-2014 eCosCentric Limited
RedBoot is free software, covered by the eCos license, derived from the
GNU General Public License. You are welcome to change it and/or distribute
copies of it under certain conditions. Under the license terms, RedBoot's
source code and full license terms must have been made available to you.
Redboot comes with ABSOLUTELY NO WARRANTY.

Platform: Broadcom BCM56150 Reference Platform (Cortex-A9)
RAM: 0x60000000-0x80000000 [0x6012c680-0x7ffed000 available]
FLASH: 0x1c000000-0x1dffffff, 512 x 0x10000 blocks
RedBoot> fis list
Name              FLASH addr  Mem addr    Length      Entry point
RedBoot           0x1C000000  0x6012C800  0x000E0000  0x6012C800
Memory Params     0x1C0E0000  0x1C0E0000  0x00010000  0x00000000
FIS directory     0x1DFF0000  0x1DFF0000  0x0000F000  0x00000000
RedBoot config    0x1DFFF000  0x1DFFF000  0x00001000  0x00000000
RedBoot>
```

If it proves necessary to install a new version of RedBoot, this may be done from RedBoot itself over the serial line. From RedBoot run the following commands:

```
RedBoot> load -r -m y -b %{freememlo}
C
```

From the terminal program, transmit the new RedBoot binary using Y-Modem protocol. When it is finished, you should see something similar to the following:

```
CRaw file loaded 0x6012c800-0x60154a07, assumed entry at 0x6012c800
xyzModem - CRC mode, 1287(SOH)/0(STX)/0(CAN) packets, 4 retries
```

Now write the loaded binary over the current RedBoot in flash:

```
RedBoot> fis cre RedBoot
An image named 'RedBoot' exists - continue (y/n)? y
... Erase from 0x1c000000-0x1c0dffff: .............
... Program from 0x6012c800-0x60154a08 to 0x1c000000: ...
... Erase from 0x1dff0000-0x1dffffff: .
... Program from 0x7fff0000-0x80000000 to 0x1dff0000: .
RedBoot>
```

The RedBoot image has now been replaced. Power cycle the board to execute it.

Rebuilding RedBoot

Should it prove necessary to rebuild a RedBoot binary, this is done most conveniently at the command line. The steps needed to rebuild the ROMRAM version of RedBoot for the BCM56150 Reference are:

```
$ mkdir redboot_bcm56150_ref_romram
$ cd redboot_bcm56150_ref_romram
$ ecosconfig new bcm56150_ref redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/arm/cortexa/bcm56150_ref/VERSION/misc/redboot_ROMRAM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

At the end of the build the `install/bin` subdirectory should contain the file `redboot.bin`. This is a binary file that can be programmed directly to the flash.

Name

Configuration — Platform-specific Configuration Options

Overview

The BCM56150 Reference platform HAL package is loaded automatically when eCos is configured for the `bcm56150-ref` target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The platform HAL package supports four separate startup types:

RAM

> This is the startup type which is normally used during application development. The board has RedBoot programmed into flash and boots into that initially. arm-eabi-gdb is then used to load a RAM startup application into memory via the serial line and debug it. It is assumed that the hardware has already been initialized by RedBoot. By default the application will use the eCos virtual vectors mechanism to obtain services from RedBoot, including diagnostic output.

ROMRAM

> This startup type can be used for finished applications which will be programmed into Flash. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization. The application starts execution from flash, but relocates itself to RAM during initialization.

ROM

> This startup type can be used for finished applications which will be programmed into Flash. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization. The application executes from flash. However, since the SPI flash is very slow, it is not recommended for production applications. It is mainly useful for debugging application startup code via a JTAG debugger.

JTAG

> This startup type can be used for finished applications that are to be loaded into RAM via a JTAG debugger. Since DRAM needs to be initialized before loading a JTAG application, it is necessary to allow RedBoot to start and run from flash. After this the JTAG probe may be connected and the application loaded. Once running the application will be self-contained with no dependencies on services provided by other software.

RedBoot and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building RedBoot.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then the serial port will be claimed for HAL diagnostics.

Flash Driver

The BCM56150 Reference board contains an 32Mbyte Micron N25Q256 SPI serial NOR flash device attached to the QSPI controller. The `CYGPKG_DEVS_FLASH_SPI_M25PXX` and `CYGPKG_DEVS_FLASH_QSPI_IPROC` packages contains all the code necessary to support this part and the platform HAL package contains definitions that customize the driver to the BCM56150 Reference board. This driver is not active until the generic Flash support package, `CYGPKG_IO_FLASH`, is included in the configuration.

This driver is capable of supporting the JFFS2 filesystem. However, note that the SPI interface means that this file system has reduced bandwidth and increased latency compared with other implementations. All that is required to enable the support is to include the filesystem (`CYGPKG_FS_JFFS2`) and any of its package dependencies (including `CYGPKG_IO_FILEIO` and `CYGPKG_LINUX_COMPAT`) together with the flash infrastructure (`CYGPKG_IO_FLASH`).

UART Serial Driver

The board uses the IProc's internal UART serial support as described in the IProc processor HAL documentation. Only one serial connector is available on the board, which is connected to UART1 via the J1801 connector. An adaptor cable is needed to connect to a standard 9-pin serial connector.

Compiler Flags

The platform HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos. The following flags are specific to this port:

`-mcpu=cortex-a9`
: The arm-eabi-gcc compiler supports many variants of the ARM architecture. A `-m` option should be used to select the specific variant in use, and with current tools `-mcpu=cortex-a9` is the correct option for the CPU in the IProc.

`-mthumb`
: The arm-eabi-gcc compiler will compile C and C++ files into the Thumb2 instruction set when this option is used. The best way to build eCos in Thumb mode is to enable the configuration option `CYGHWR_THUMB`.

`-mno-unaligned-access`
: The Cortex-A CPU allows unaligned memory accesses and the default for arm-eabi-gcc is to generate instructions that make unaligned accesses. However, for this port, alignment exceptions are enabled, so unaligned accesses should not be made. This option disables unaligned accesses. Note that there is a performance and code size cost in doing this, since all accesses to unaligned data must now be made using individual byte accesses.

`-fno-jump-tables`
: For ROMRAM startup, some code is executed from flash before code is relocated to RAM. As a result, any code or constant data references will use the wrong address. To prevent **switch** statements using tables at the wrong location, this option is enabled for ROMRAM startup builds.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the BCM56150 Reference hardware, and should be read in conjunction with that specification. The platform HAL package complements the ARM architectural HAL, the Cortex-A variant HAL and the Broadcom IProc processor HAL. It provides functionality which is specific to the target board.

Startup

Following a reset, the HAL will initialize or reinitialize most of the on-chip peripherals. There is an exception for RAM startup applications which depend on a ROM monitor for certain services.

For ROMRAM startup, the HAL will perform additional initialization. This is all done in the `PLATFORM_SETUP1` macro in the assembler header file `hal_platform_setup.h`.

Linker Scripts and Memory Maps

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

SDRAM

This is located at address 0x60000000 of the physical memory space. The HAL configures the MMU to retain the SDRAM at virtual address 0x60000000 with caching enabled. The same memory is also accessible uncached at virtual location 0xA0000000 for use by device drivers. The first 128MiB of SDRAM is also mapped to physical address 0x00000000, which is identity mapped uncached. The exception vectors and VSR table occupy the bottom 64 bytes of this region. Memory is required for the MMU tables, and must be aligned on a 16Kbyte boundary. These therefore occupy memory from 0x60010000 to 0x60014000. For ROMRAM startup, the application relocates to 0x60100000 and all remaining SDRAM is available. The virtual vector table is allocated as part of the RedBoot image and occupies 256 bytes from 0x60100050. RAM applications load from 0x60200000, reserving 1MiB for RedBoot. JTAG applications load at 0x60100000, overwriting any ROMRAM application already present.

On-chip SRAM

On-chip SRAM is located at 0x1B000000 and is 16KiB in size. While this memory is used during DRAM initialization, it is not used by eCos for any other purpose and is available for application use.

SPI NOR Flash

SPI NOR flash is supported by a QSPI device that translates read accesses to memory starting at 0x1C000000 into SPI read transactions. Writes to the SPI flash go via a different device which disables this mapping. Because of this, and its poor performance, code should not be executed from SPI flash except during initialization. The Serial flash area at 0x1C000000 is identity mapped uncached, although it should normally be accessed only using the flash API.

Peripheral Registers

These are located at various addresses in the physical memory space. When the MMU is enabled, it contain direct, uncached, identity mappings so that these registers remain accessible at their physical locations.

SPI NOR Flash

eCos supports QSPI access to the NOR flash on the board. The device is typically used to contain RedBoot and flash configuration data.

Accesses to SPI flash are performed via the Flash API, using 0x1C000000 as its base address. Access from RedBoot should be made using **fis** command operations.

Real-time characterization

The tm_basic kernel test gives statistics gathered about the real-time characterization and performance of the kernel. The sample output is shown here for information. The test was built in ARM32 mode.

Example 274.1. bcm56150_ref Real-time characterization

```
             Startup, main thrd : stack used   388 size  1792
             Startup :  Interrupt stack used  4096 size  4096
             Startup : Idlethread stack used    88 size  1280

eCos Kernel Timings
Notes: all times are in microseconds (.000001) unless otherwise stated

Reading the hardware clock takes 0 'ticks' overhead
... this value will be factored out of all other measurements
Clock interrupt took    1.03 microseconds (1 raw clock ticks)

Testing parameters:
   Clock samples:            32
   Threads:                  64
   Thread switches:         128
   Mutexes:                  32
   Mailboxes:                32
   Semaphores:               32
   Scheduler operations:    128
   Counters:                 32
   Flags:                    32
   Alarms:                   32
   Stack Size:             1088


                                 Confidence
     Ave     Min     Max     Var  Ave  Min  Function
  ======  ======  ======  ====== ========== ========
INFO:<Ctrl-C disabled until test completion>
    1.42    1.00    2.00    0.49   57%  57% Create thread
    0.33    0.00    1.00    0.44   67%  67% Yield thread [all suspended]
    0.33    0.00    1.00    0.44   67%  67% Suspend [suspended] thread
    0.30    0.00    1.00    0.42   70%  70% Resume thread
    0.44    0.00    1.00    0.49   56%  56% Set priority
    0.16    0.00    1.00    0.26   84%  84% Get priority
    0.73    0.00    2.00    0.41   70%  28% Kill [suspended] thread
    0.33    0.00    1.00    0.44   67%  67% Yield [no other] thread
    0.41    0.00    1.00    0.48   59%  59% Resume [suspended low prio] thread
    0.31    0.00    1.00    0.43   68%  68% Resume [runnable low prio] thread
    0.36    0.00    1.00    0.46   64%  64% Suspend [runnable] thread
    0.33    0.00    1.00    0.44   67%  67% Yield [only low prio] thread
    0.31    0.00    1.00    0.43   68%  68% Suspend [runnable->not runnable]
    0.67    0.00    1.00    0.44   67%  32% Kill [runnable] thread
    0.64    0.00    1.00    0.46   64%  35% Destroy [dead] thread
    1.05    1.00    2.00    0.09   95%  95% Destroy [runnable] thread
    1.05    1.00    2.00    0.09   95%  95% Resume [high priority] thread
    0.42    0.00    1.00    0.49   57%  57% Thread switch

    0.14    0.00    1.00    0.24   85%  85% Scheduler lock
    0.30    0.00    1.00    0.42   70%  70% Scheduler unlock [0 threads]
    0.31    0.00    1.00    0.43   68%  68% Scheduler unlock [1 suspended]
    0.32    0.00    1.00    0.44   67%  67% Scheduler unlock [many suspended]
    0.30    0.00    1.00    0.42   70%  70% Scheduler unlock [many low prio]

    0.19    0.00    1.00    0.30   81%  81% Init mutex
    0.25    0.00    1.00    0.38   75%  75% Lock [unlocked] mutex
    0.41    0.00    1.00    0.48   59%  59% Unlock [locked] mutex
    0.25    0.00    1.00    0.38   75%  75% Trylock [unlocked] mutex
    0.28    0.00    1.00    0.40   71%  71% Trylock [locked] mutex
    0.09    0.00    1.00    0.17   90%  90% Destroy mutex
    1.03    1.00    2.00    0.06   96%  96% Unlock/Lock mutex

    0.25    0.00    1.00    0.38   75%  75% Create mbox
    0.16    0.00    1.00    0.26   84%  84% Peek [empty] mbox
    0.38    0.00    1.00    0.47   62%  62% Put [first] mbox
    0.19    0.00    1.00    0.30   81%  81% Peek [1 msg] mbox
    0.41    0.00    1.00    0.48   59%  59% Put [second] mbox
    0.16    0.00    1.00    0.26   84%  84% Peek [2 msgs] mbox
    0.34    0.00    1.00    0.45   65%  65% Get [first] mbox
    0.38    0.00    1.00    0.47   62%  62% Get [second] mbox
    0.31    0.00    1.00    0.43   68%  68% Tryput [first] mbox
    0.28    0.00    1.00    0.40   71%  71% Peek item [non-empty] mbox
    0.28    0.00    1.00    0.40   71%  71% Tryget [non-empty] mbox
    0.28    0.00    1.00    0.40   71%  71% Peek item [empty] mbox
    0.34    0.00    1.00    0.45   65%  65% Tryget [empty] mbox
    0.13    0.00    1.00    0.22   87%  87% Waiting to get mbox
    0.22    0.00    1.00    0.34   78%  78% Waiting to put mbox
    0.19    0.00    1.00    0.30   81%  81% Delete mbox
    0.94    0.00    1.00    0.12   93%   6% Put/Get mbox

    0.22    0.00    1.00    0.34   78%  78% Init semaphore
    0.31    0.00    1.00    0.43   68%  68% Post [0] semaphore
    0.34    0.00    1.00    0.45   65%  65% Wait [1] semaphore
    0.38    0.00    1.00    0.47   62%  62% Trywait [0] semaphore
    0.31    0.00    1.00    0.43   68%  68% Trywait [1] semaphore
    0.31    0.00    1.00    0.43   68%  68% Peek semaphore
    0.19    0.00    1.00    0.30   81%  81% Destroy semaphore
    0.97    0.00    1.00    0.06   96%   3% Post/Wait semaphore

    0.25    0.00    1.00    0.38   75%  75% Create counter
    0.16    0.00    1.00    0.26   84%  84% Get counter value
    0.13    0.00    1.00    0.22   87%  87% Set counter value
    0.31    0.00    1.00    0.43   68%  68% Tick counter
    0.03    0.00    1.00    0.06   96%  96% Delete counter

    0.25    0.00    1.00    0.38   75%  75% Init flag
    0.34    0.00    1.00    0.45   65%  65% Destroy flag
    0.34    0.00    1.00    0.45   65%  65% Mask bits in flag
    0.31    0.00    1.00    0.43   68%  68% Set bits in flag [no waiters]
    0.28    0.00    1.00    0.40   71%  71% Wait for flag [AND]
    0.31    0.00    1.00    0.43   68%  68% Wait for flag [OR]
    0.44    0.00    1.00    0.49   56%  56% Wait for flag [AND/CLR]
    0.34    0.00    1.00    0.45   65%  65% Wait for flag [OR/CLR]
    0.03    0.00    1.00    0.06   96%  96% Peek on flag

    0.28    0.00    1.00    0.40   71%  71% Create alarm
    0.44    0.00    1.00    0.49   56%  56% Initialize alarm
    0.31    0.00    1.00    0.43   68%  68% Disable alarm
    0.41    0.00    1.00    0.48   59%  59% Enable alarm
    0.06    0.00    1.00    0.12   93%  93% Delete alarm
    0.25    0.00    1.00    0.38   75%  75% Tick counter [1 alarm]
    1.28    1.00    2.00    0.40   71%  71% Tick counter [many alarms]
    0.47    0.00    1.00    0.50   53%  53% Tick & fire counter [1 alarm]
    6.50    6.00    7.00    0.50  100%  50% Tick & fire counters [>1 together]
    1.50    1.00    2.00    0.50  100%  50% Tick & fire counters [>1 separately]
    1.00    1.00    1.00    0.00  100% 100% Alarm latency [0 threads]
    1.00    1.00    1.00    0.00  100% 100% Alarm latency [2 threads]
    1.04    1.00    2.00    0.08   96%  96% Alarm latency [many threads]
    2.00    2.00    2.00    0.00  100% 100% Alarm -> thread resume latency

    0.00    0.00    0.00    0.00            Clock/interrupt latency
    1.00    1.00    1.00    0.00            Clock DSR latency

     222     164     272                    Worker thread stack used (stack size 1088)
            All done, main thrd : stack used   812 size  1792
            All done :  Interrupt stack used   156 size  4096
            All done : Idlethread stack used   232 size  1280

Timing complete - 30830 ms total

PASS:<Basic timing OK>
EXIT:<done>
```

Other Issues

The platform HAL does not affect the implementation of other parts of the eCos HAL specification. The IProc processor HAL and the ARM architectural HAL documentation should be consulted for further details.

Chapter 275. Altera Cyclone V SX Board Support

Name

eCos Support for the Altera Cyclone V SX Board — Overview

Description

This document covers the configuration and usage of eCos and RedBoot on the Altera Cyclone V SoC Development kit. This board is fitted with an SX variant of the Cyclone V family of FPGAs and it therefore referred to in this document and the configuration system as a Cyclone V SX, or cyclone5_sx, to differentiate it from other Cyclone V SoC development boards.

In addition to the Cyclone V FPGA, the board contains 1GiB SDRAM main memory, a 1Gib (128GiB) SPI NOR flash, a micro-SD card socket, a USB bridge connected to UART0, Ethernet sockets for both the HPS and FPGA Ethernet interfaces, plus a variety of connectors for other interfaces plus resources devoted to the FPGA. The extent of eCos support for the devices and peripherals on the board and the CPU is described below.

For typical eCos development, a RedBoot image is programmed into the SPI NOR flash memory, and the board will load this image from reset. RedBoot provides gdb stub functionality so it is then possible to download and debug stand-alone and eCos applications via the gdb debugger. This can happen over either a serial line or over Ethernet.

Support for SMP operation of the two Cortex-A9 CPUs on the Cyclone V SoC is available, although debugging support is restricted to use of an external JTAG debugger. There is no SMP support in RedBoot.

This documentation is expected to be read in conjunction with the Altera HPS processor HAL documentation and further device support and subsystems are described and documented there.

Supported Hardware

The SPI NOR flash consists of 2048 blocks of 64Ki bytes each. In a typical setup, the first 4 blocks are reserved for the second-level bootstrap, the preloader. The 10 blocks from block 6 are reserved for the use of the ROM RedBoot image. The topmost block is used to manage the flash and also holds RedBoot **fconfig** values. The remaining blocks can be used by application code.

Serial support is through the `CYGPKG_IO_SERIAL_GENERIC_16X5X` generic driver package which is modified by the `CYGPKG_IO_SERIAL_ARM_ALTERA_HPS` driver package for the HPS. These packages can support all the serial devices on the HPS. However, this board only has UART0 connected to an external connector which this HAL indicates by implementing the `CYGINT_HAL_ARM_CORTEXA_ALTERA_HPS_UART0` interface. This serial channel is used by RedBoot for communication with the host. If this device is needed by the application, either directly or via the serial driver, then it cannot also be used for RedBoot communication. Another communication channel such as Ethernet should be used instead. The serial driver package is loaded automatically when configuring for the `cyclone5-sx` target.

There is an Ethernet driver `CYGPKG_DEVS_ETH_DWC_GMAC` for the on-chip Ethernet device. A separate package, `CYGPKG_DEVS_ETH_CYCLONE5_SX` configures this generic driver to the hardware. This driver is also loaded automatically when configuring for the `cyclone5-sx` board.

There is a watchdog driver `CYGPKG_DEVICES_WATCHDOG_DWWDT`. This driver is also loaded automatically when configuring for the board.

There is a driver for the DS1339C real-time clock (RTC) at `CYGPKG_DEVICES_WALLCLOCK_DALLAS_DS1307`, with which it is compatible. This driver is also loaded automatically when configuring for the target and when I²C support is included.

The HPS processor HAL contains a driver for the MultiMedia Card Interface (MMC/SD). This driver is loaded automatically when configuring `CYGPKG_DEVS_DISK_MMCSD_BUS` for this target and allows use of MultiMediaCard (MMC) and Secure Digital (SD) flash storage cards within eCos, exported as block devices. Further documentation on the driver may be found in the Altera Hard Processor System Support HAL documentation.

Note

There is no working card-detect (media change) signal available on the Altera Cyclone V SoC Development board for the J3 MicroSD slot.

The platform HAL provides definitions to enable access to flash devices on the SPI bus. The HAL enables the QSPI driver (`CYGPKG_DEVS_FLASH_QSPI`) which in turn provides the underlying implementation for access to the Micron N25Q00AA SPI NOR flash. The QSPI support integrates with the `CYGPKG_DEVS_FLASH_SPI_M25PXX` package. These packages are automatically loaded when configuring for the target. This driver is capable of supporting the JFFS2 filesystem, although at reduced performance compared with a parallel flash device.

In general, devices (Caches, GPIO, UARTs) are initialized only as far as is necessary for eCos to run. Other devices (RTC, SPI, MMC/SD etc.) are not touched unless the appropriate driver is loaded, although in some cases, the HAL boot sequence will set up the appropriate power control and pin multiplexing configuration.

Tools

The board support is intended to work with GNU tools configured for an arm-eabi target. The original port was undertaken using arm-eabi-gcc version 4.4.5, arm-eabi-gdb version 7.2, and binutils version 2.20.

Name

Setup — Preparing the Cyclone V SX board for eCos Development

Overview

In a typical development environment, the board boots from the SPI NOR and runs the RedBoot ROM monitor from SDRAM. eCos applications are configured for RAM startup and then downloaded and run on the board via the debugger **arm-eabi-gdb**. Preparing the board therefore usually involves programming a suitable RedBoot image into flash memory.

The following RedBoot configurations are supported:

Configuration	Description	Use	File
ROM	RedBoot loaded from SPI NOR flash to SDRAM	`redboot_ROM.ecm`	`redboot_ROM.bin`

For serial communications, all versions run with 8 bits, no parity, and 1 stop bit at 57600 baud. RedBoot also supports Ethernet communication and flash management.

Note that the use of the term ROM for the initial RedBoot configuration is a historical accident. RedBoot actually runs from SDRAM after being loaded there from NOR flash by the preloader. The use of ROM for this configuration is intended to indicate that it initializes the microprocessor and board peripherals, as opposed to the RAM configuration which assumes that this has already been done.

Initial Installation

The Cyclone V SX board comes with U-Boot and Linux installed by default on a micro-SD card. For eCos development we want to install RedBoot in the SPI NOR flash. The booting mechanism is that the on-chip firmware loads a small preloader from the start of NOR flash which then loads RedBoot from later in the flash.

To write RedBoot to the SPI NOR flash, you must use the command line tools from the Altera QuartusII SoC Embedded Development System.

Programming RedBoot into NOR flash using Altera Tools

The following gives the steps needed to program RedBoot into the SPI NOR Flash using the Altera EDS tools. This uses the **quartus_hps** flash programmer command line utility to do the work.

Note

The examples below are for a Linux host where eCosPro is typically installed in the `/opt/ecospro/ecospro-<version>` sub-directory. On a Windows host eCosPro will typically be installed in the `C:\eCosPro\ecos-<version>` sub-directory.

1. Install QuartusII and the SoC EDS as described by Altera onto your host system. Start an Embedded Command Shell by running either `<SoC EDS Installation Folder>\embedded\Embedded_Command_Shell.bat` in Windows or `<SoC EDS Installation Folder>/embedded/embedded_command_shell.sh` in Linux.
2. Ensure that the BOOTSEL jumpers are set as follows: BOOTSEL0(J28) 2-3, BOOTSEL1(J29) 1-2, BOOTSEL2(J30) 1-2.
3. Connect USB cables between the mini USB socket at J37 (used by the USB-Blaster), and the mini USB socket at J8 (used to provide serial device access) and your host. Power up the board and run a terminal emulator on your host, attaching it to the serial USB channel which should appear on your host, setting the baud rate to 57600.
4. Run the following command on the host to detect the USB-Blaster:

```
# jtagconfig
1) USB-BlasterII [USB 1-1]
  4BA00477   SOCVHPS
  02D020DD   5CS(EBA6ES|XFC6C6ES)/..

#
```

It may be necessary to run this a couple of times before the above result is obtained. If the program failure persists then you should check that your EDS installation is correct.

5. Run the following command to configure the JTAG interface:

   ```
   # jtagconfig --setparam 1 JtagClock 16M
   #
   ```

   This program terminates silently if it is successful.

6. Run the following command to install the preloader, passing it a path to the preloader image from the installed eCos distribution. This preloader was built with the Altera bsp-editor according to the instructions in the "Altera SoC Embedded Design Suite User Guide", chapter "HPS Preloader User Guide". The `BOOT_FROM_QSPI` boot option was used.

   ```
   # quartus_hps -c 1 -o PV /opt/ecospro/ecos-<version>/loaders/cyclone5_sx/preloader-mkpimage.bin
   Info: *******************************************************************
   Info: Running Quartus II 32-bit Programmer
       Info: Version 13.0.0 Build 156 04/24/2013 SJ Full Version
       Info: Copyright (C) 1991-2013 Altera Corporation. All rights reserved.
       Info: Your use of Altera Corporation's design tools, logic functions
       Info: and other software and tools, and its AMPP partner logic
       Info: functions, and any output files from any of the foregoing
       Info: (including device programming or simulation files), and any
       Info: associated documentation or information are expressly subject
       Info: to the terms and conditions of the Altera Program License
       Info: Subscription Agreement, Altera MegaCore Function License
       Info: Agreement, or other applicable license agreement, including,
       Info: without limitation, that your use is for the sole purpose of
       Info: programming logic devices manufactured by Altera and sold by
       Info: Altera or its authorized distributors.  Please refer to the
       Info: applicable agreement for further details.
       Info: Processing started: Wed Jun 26 12:51:47 2013
   Info: Command: quartus_hps -c 1 -o PV /path/to/preloader-mkpimage.bin
   Current hardware is: USB-BlasterII [USB 1-1]
   Found HPS device at index 0
   HPS Device IDCODE: 0x4BA00477
   AHB Port is located at port 0
   APB Port is located at port 1
   Boot Info: 1.8V QSPI Flash
   Start HPS Quad SPI flash programming ...
   Initialize QSPI peripheral and flash controller ...
   Read Silicon ID of Quad SPI flash ...
      Quad SPI Flash silicon ID is 0x1021BA20
         Flash device matched
            Manufacturer: Micron
            Device: QSPI_1024
   Enable Four Byte Addressing ...
   Sector Erase Quad SPI flash ...
      Sector Erase Info: Start Addr at 0x00000000 for 4 sector(s)
         Sector Erase Quad SPI flash at 0x00000000
         Sector Erase Quad SPI flash at 0x00010000
         Sector Erase Quad SPI flash at 0x00020000
         Sector Erase Quad SPI flash at 0x00030000
   Program Quad SPI flash ...
   Verify Quad SPI flash ...
   Info: Quartus II 32-bit Programmer was successful. 0 errors, 0 warnings
       Info: Peak virtual memory: 46 megabytes
       Info: Processing ended: Wed Jun 26 12:53:55 2013
   ```

```
    Info: Elapsed time: 00:02:08
    Info: Total CPU time (on all processors): 00:00:04
#
```

7. Now run the following commands to install RedBoot, passing a path to the RedBoot image from the installed eCos distribution. Note that the **quartus_hps** command can only accept files with a `.bin` extension, so it is necessary to copy the generated RedBoot image to a file with the correct extension before running the command. The RedBoot image file will be located alongside `preloader-mkpimage.bin` in the `loaders/cyclone5_sx` sub-directory of the eCosPro installation as illustrated above. You may ignore the `.bin`, `.elf` and `.srec` files also located in the same sub-directory. Also note that the commands provided are for a Linux host so the `PATH` and copy command will vary.

```
# cp /opt/ecospro/ecos-<version>/loaders/cyclone5_sx/redboot_ROM.img /tmp/redboot.bin
# quartus_hps -c 1 -a 0x60000 -o PV /tmp/redboot.bin
Info: *******************************************************************
Info: Running Quartus II 32-bit Programmer
    Info: Version 13.0.0 Build 156 04/24/2013 SJ Full Version
    Info: Copyright (C) 1991-2013 Altera Corporation. All rights reserved.
    Info: Your use of Altera Corporation's design tools, logic functions
    Info: and other software and tools, and its AMPP partner logic
    Info: functions, and any output files from any of the foregoing
    Info: (including device programming or simulation files), and any
    Info: associated documentation or information are expressly subject
    Info: to the terms and conditions of the Altera Program License
    Info: Subscription Agreement, Altera MegaCore Function License
    Info: Agreement, or other applicable license agreement, including,
    Info: without limitation, that your use is for the sole purpose of
    Info: programming logic devices manufactured by Altera and sold by
    Info: Altera or its authorized distributors.  Please refer to the
    Info: applicable agreement for further details.
    Info: Processing started: Wed Jun 26 13:02:07 2013
Info: Command: quartus_hps -c 1 -a 0x60000 -o PV /tmp/redboot.bin
Current hardware is: USB-BlasterII [USB 1-1]
Found HPS device at index 0
HPS Device IDCODE: 0x4BA00477
AHB Port is located at port 0
APB Port is located at port 1
Boot Info: 1.8V QSPI Flash
Start HPS Quad SPI flash programming ...
Initialize QSPI peripheral and flash controller ...
Read Silicon ID of Quad SPI flash ...
   Quad SPI Flash silicon ID is 0x1021BA20
      Flash device matched
         Manufacturer: Micron
         Device: QSPI_1024
Enable Four Byte Addressing ...
Sector Erase Quad SPI flash ...
   Sector Erase Info: Start Addr at 0x00060000 for 2 sector(s)
      Sector Erase Quad SPI flash at 0x00060000
      Sector Erase Quad SPI flash at 0x00070000
Program Quad SPI flash ...
Verify Quad SPI flash ...
Info: Quartus II 32-bit Programmer was successful. 0 errors, 0 warnings
    Info: Peak virtual memory: 46 megabytes
    Info: Processing ended: Wed Jun 26 13:03:10 2013
    Info: Elapsed time: 00:01:03
    Info: Total CPU time (on all processors): 00:00:02
#
```

8. Detach the USB cable for the USB Blaster from the mini USB socket at J37 and connect an ethernet cable to J2 on the board (located on the opposite side from the ENET1 and ENET2 dual ethernet interfaces) and an ethernet hub that is connected to your host's network.

9. Power cycle the board. You should see the following output on the serial line:

```
+**Warning** FLASH configuration checksum error or invalid key
```

```
Ethernet eth0: MAC address 12:34:aa:bb:cc:ee
IP: 10.0.2.4/255.0.0.0, Gateway: 10.0.0.3
Default server: 10.0.1.1

RedBoot(tm) bootstrap and debug environment [ROM]
Non-certified release, version UNKNOWN - built 11:45:41, Jun 26 2013

Copyright (C) 2000-2009 Free Software Foundation, Inc.
Copyright (C) 2003-2012 eCosCentric Limited
RedBoot is free software, covered by the eCos license, derived from the
GNU General Public License. You are welcome to change it and/or distribute
copies of it under certain conditions. Under the license terms, RedBoot's
source code and full license terms must have been made available to you.
Redboot comes with ABSOLUTELY NO WARRANTY.

Platform: Altera Cyclone V SX Development Kit (Cortex-A9)
RAM: 0x00000000-0x40000000 [0x00259208-0x3ffed000 available]
FLASH: 0x80000000-0x87ffffff, 2048 x 0x10000 blocks
RedBoot>
```

Since the serial USB bridge is also power cycled, you may lose the serial device under Windows or one or two lines from the beginning of this output. If this happens, simply reconnect if necessary and type `version` to see the full output again.

10. Run the following commands to initialize RedBoot's flash file system and flash configuration:

```
RedBoot> fis init
About to initialize [format] FLASH image system - continue (y/n)? y
*** Initialize FLASH Image System
... Erase from 0x87ff0000-0x87ffffff: .
... Program from 0x3fff0000-0x40000000 to 0x87ff0000: .
RedBoot>
RedBoot> fconfig -i
Initialize non-volatile configuration - continue (y/n)? y
Run script at boot: false
Use BOOTP for network configuration: true
Default server IP address: 10.0.1.1
Console baud rate: 57600
Network hardware address [MAC] for eth0: 0x12:0x34:0xAA:0xBB:0xCC:0x08
GDB connection port: 9000
Force console for special debug messages: false
Network debug at boot time: false
Default network device: dwc_gmac
Update RedBoot non-volatile configuration - continue (y/n)? y
... Erase from 0x87ff0000-0x87ffffff: .
... Program from 0x3fff0000-0x40000000 to 0x87ff0000: .
RedBoot>
```

You should substitute your own server IP address for the one shown above. You may also want to change the MAC address if more than one board is present on the network, or use one of the MAC addresses assigned by Altera to this board. If you want to use a static IP address, then choose false for the "Use BOOTP" option and enter the gateway, IP address and netmask that you have assigned.

The RedBoot installation is now complete. This can be tested by power cycling the board again. Output similar to the following should be seen on the serial port.

```
+Ethernet eth0: MAC address 12:34:aa:bb:cc:08
IP: 10.0.2.6/255.0.0.0, Gateway: 10.0.0.3
Default server: 10.0.1.1

RedBoot(tm) bootstrap and debug environment [ROM]
Non-certified release, version UNKNOWN - built 11:45:41, Jun 26 2013

Copyright (C) 2000-2009 Free Software Foundation, Inc.
Copyright (C) 2003-2013 eCosCentric Limited
RedBoot is free software, covered by the eCos license, derived from the
```

```
GNU General Public License. You are welcome to change it and/or distribute
copies of it under certain conditions. Under the license terms, RedBoot's
source code and full license terms must have been made available to you.
Redboot comes with ABSOLUTELY NO WARRANTY.

Platform: Altera Cyclone V SX Development Kit (Cortex-A9)
RAM: 0x00000000-0x40000000 [0x00259208-0x3ffed000 available]
FLASH: 0x80000000-0x87ffffff, 2048 x 0x10000 blocks
RedBoot> fis list
Name              FLASH addr  Mem addr    Length      Entry point
(reserved)        0x80000000  0x80000000  0x00060000  0x00000000
RedBoot           0x80060000  0x80060000  0x000A0000  0x00000000
FIS directory     0x87FF0000  0x87FF0000  0x0000F000  0x00000000
RedBoot config    0x87FFF000  0x87FFF000  0x00001000  0x00000000
RedBoot>
```

If it proves necessary to install a new version of RedBoot, this may be done from RedBoot itself. Place the new image on a TFTP server on the configured server. From RedBoot run the following commands:

```
RedBoot> load -r -b %{freememlo} redboot.img
Using default protocol (TFTP)
Raw file loaded 0x00259400-0x00277c0b, assumed entry at 0x00259400
RedBoot> fis cre RedBoot
An image named 'RedBoot' exists - continue (y/n)? y
... Erase from 0x80060000-0x800fffff: ..........
... Program from 0x00259400-0x00277c0c to 0x80060000: ..
... Erase from 0x87ff0000-0x87ffffff: .
... Program from 0x3fff0000-0x40000000 to 0x87ff0000: .
RedBoot>
```

Rebuilding RedBoot

Should it prove necessary to rebuild a RedBoot binary, this is done most conveniently at the command line. The steps needed to rebuild the ROM version of RedBoot for the Cyclone V SX are:

```
$ mkdir redboot_cyclone5_sx_rom
$ cd redboot_cyclone5_sx_rom
$ ecosconfig new cyclone5_sx redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/arm/cortexa/cyclone5_sx/VERSION/misc/redboot_ROM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

At the end of the build the `install/bin` subdirectory should contain the file `redboot.img`. This is a binary file that includes a header needed by the preloader to load and run RedBoot successfully.

Note

The `flashimg_cv` host executable provided within your eCosPro installation is required on the build host to wrap the RedBoot binary image into an image that can be programmed by the tools from the Altera QuartusII SoC Embedded Development System. This executable must be on your path when you build RedBoot and will normally located in the `/opt/ecospro/ecos-<version>/host/bin-<hostos>` sub-directory of your eCosPro installation. Shells created by future revisions of the eCos GUI configuration tool or `ecosprofileenv` will add this directory to the `PATH` environment variable.

Installing user applications into Flash

If you wish to install a ROM startup application into Flash to be automatically booted instead of RedBoot, you can follow a similar procedure to installing RedBoot into Flash. However before you can do so, you must first prepend a header to your application image in order for the preloader to recognise it as a valid application.

You will need the **flashimg_cv** command, which should be in the host tools binary directory as described above. You will also need to generate a binary image of your program using the **arm-eabi-objcopy** command. The following gives an example simplified command sequence which can be run at a command shell prompt:

```
$ arm-eabi-objcopy -O binary myapp myapp.bin
$ flashimg_cv myapp.bin myapp.img
```

You will need to subsitute your own paths and filenames where applicable.

Once you have the `.img` file, you can follow the same process as above for installing RedBoot via the USB-BlasterII. Once the initial setup has been done once, it is only then necessary to re-install the ROM executable. It is not necessary to reinstall the preloader each time. A typical command sequence might be:

```
$ arm-eabi-objcopy -O binary myapp myapp.bin0
$ flashimg_cv myapp.bin0 myapp.bin
$ quartus_hps -c 1 -a 0x60000 -o PV myapp.bin
```

Name

Configuration — Platform-specific Configuration Options

Overview

The Cyclone V SX platform HAL package is loaded automatically when eCos is configured for the `cyclone5-sx` target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The platform HAL package supports four separate startup types:

RAM
: This is the startup type which is normally used during application development. The board has RedBoot programmed into flash and boots into that initially. arm-eabi-gdb is then used to load a RAM startup application into memory and debug it. It is assumed that the hardware has already been initialized by RedBoot. By default the application will use the eCos virtual vectors mechanism to obtain services from RedBoot, including diagnostic output.

ROM
: This startup type can be used for finished applications which will be programmed into Flash. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization. This startup type can also be used for applications loaded via JTAG.

SRAM
: This startup type can be used for finished applications that are to be loaded into the on-chip SRAM. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization. This startup type can also be used for applications loaded via JTAG.

SMP
: This startup type can be used for finished applications that can be loaded into RAM via RedBoot. The load address is set to the same as for RAM applications, however, the application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization. Once started, this application takes full control of the system and RedBoot will not be called again. This means that debugging via RedBoot will not be possible, only JTAG-based hardware debugging is supported. The intent of this startup type is to allow SMP test programs to be run from RedBoot, most SMP applications should use the ROM startup type. This startup type can also be used for applications loaded directly via JTAG.

RedBoot and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building RedBoot.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then the serial port will be claimed for HAL diagnostics.

Flash Driver

The Cyclone V SX board contains an 128Mbyte Micron N25Q00AA SPI serial NOR flash device attached to the QSPI controller. The `CYGPKG_DEVS_FLASH_SPI_M25PXX` package contains all the code necessary to support this part and the platform HAL package contains definitions that customize the driver to the Cyclone V SX. This driver is not active until the generic Flash support package, `CYGPKG_IO_FLASH`, is included in the configuration.

This driver is capable of supporting the JFFS2 filesystem. However, note that the SPI interface means that this file system has reduced bandwidth and increased latency compared with other implementations. All that is required to enable the support is to include the filesystem (CYGPKG_FS_JFFS2) and any of its package dependencies (including CYGPKG_IO_FILEIO and CYGPKG_LINUX_COMPAT) together with the flash infrastructure (CYGPKG_IO_FLASH).

Ethernet Driver

The board uses the HPS's internal GMAC Ethernet device attached to an external Micrel KSZ9021 Gigabit PHY. The CYGPKG_DEVS_ETH_DWC_GMAC package contains all the code necessary to support this device and the CYGPKG_DEVS_ETH_CYCLONE5_SX package contains definitions that customize the driver to the board. This driver is not active until the generic Ethernet support package, CYGPKG_IO_ETH_DRIVERS, is included in the configuration.

Support for PHY events is made available by default when the Kernel (CYGPKG_KERNEL) is available in a configuration.

RTC Driver

The Cyclone V SX board has a Maxim DS1339C I²C based RTC chip installed. For the functionality used by eCos, this device is compatible with the DS1307. Therefore, the CYGPKG_DEVICES_WALLCLOCK_DALLAS_DS1307 package contains all the code necessary to support this device. This driver is not active until the generic wallclock device support package, CYGPKG_IO_WALLCLOCK, is included in the configuration and I²C support is enabled.

Watchdog Driver

The board uses the HPS's internal watchdog support. The CYGPKG_DEVICES_WATCHDOG_DWWDT package contains all the code necessary to support this device. Within that package the CYGNUM_DEVS_WATCHDOG_DWWDT_DESIRED_TIMEOUT_MS configuration option controls the watchdog timeout, and by default will force a reset of the board upon timeout. This driver is not active until the generic watchdog device support package, CYGPKG_IO_WATCHDOG, is included in the configuration.

UART Serial Driver

The board uses the HPS's internal UART serial support as described in the HPS processor HAL documentation. Only one serial connector is available on the board, which is connected to UART0 via a USB bridge. Only the UART data lines are connected to the bridge, so hardware flow control is not supported.

MMC/SD Driver

As the Cyclone V SX MMC/SD driver is part of the HPS processor HAL, nothing is required to load it. Similarly the MMC/SD bus driver layer (CYGPKG_DEVS_DISK_MMC) is automatically included as part of the hardware-specific configuration for this target. All that is required to enable the support is to include the generic disk I/O infrastructure package (CYGPKG_IO_DISK), along with the intended filesystem, typically, the FAT filesystem (CYGPKG_FS_FAT) and any of its package dependencies (e.g. for FAT also including CYGPKG_LIBC_STRING and CYGPKG_LINUX_COMPAT packages).

Various options can be used to control specifics of the MMC/SD driver. Consult the HPS processor HAL documentation for information on its configuration.

This board does not have a working MMC/SD card detect for MicroSD socket (J3), thus the disk I/O layer's removeable media support is not available.

Compiler Flags

The platform HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos. The following flags are specific to this port:

`-mcpu=cortex-a9`
: The arm-eabi-gcc compiler supports many variants of the ARM architecture. A `-m` option should be used to select the specific variant in use, and with current tools `-mcpu=cortex-a9` is the correct option for the CPU in the HPS.

`-mthumb`
: The arm-eabi-gcc compiler will compile C and C++ files into the Thumb2 instruction set when this option is used. The best way to build eCos in Thumb mode is to enable the configuration option `CYGHWR_THUMB`.

`-mno-unaligned-access`
: The Cortex-A CPU allows unaligned memory accesses and the default for arm-eabi-gcc is to generate instructions that make unaligned accesses. However, for this port, alignment exceptions are enabled, so unaligned accesses should not be made. This option disables unaligned accesses. Note that there is a performance and code size cost in doing this, since all accesses to unaligned data must now be made using individual byte accesses.

Name

SMP Support — Usage

Overview

Support is available for SMP operation of the two CPUs on this platform. However, debugging support is restricted to using an external SMP-aware JTAG debugger like ARM's DS-5 or a Lauterbach Power Debug probe. RedBoot does not have support for multi-core debugging.

A board intended to be used for SMP operation should be initialized in the same way as a single core board by installing the preloader and RedBoot. If RedBoot is not needed then the preloader must still be installed and a ROM startup application, with headers added by **flashimg_cv**, can be written to flash in place of RedBoot.

SMP support is enabled by setting `CYGPKG_KERNEL_SMP_SUPPORT` to true. SMP applications should only be built using either ROM or SMP startup types. ROM applications can be loaded by the pre-loader in place of RedBoot. The SMP startup is identical to a ROM startup except that the load address is set to allow the application to be loaded into a higher location in RAM from RedBoot. Both application types may also be loaded via a JTAG debugger.

Loading an SMP startup application via RedBoot can be done from the RedBoot command line via serial or Ethernet. It may also be loaded via a GDB connection on serial or Ethernet. However, once started running the SMP application will take full control of the system, including redirecting all interrupt sources, exception vectors and virtual vector table entries. This means that RedBoot will no-longer be active. Any breakpoints planted by GDB will result in an exception to the application, Ctrl-C will not work, any Ethernet connections will be lost and serial output will come from the application in plain ASCII. Any GDB connection will be lost and GDB may start reporting packet errors.

JTAG support has been tested using the ARM DS-5 debugger and the Lauterbach Trace32 debugger. Most of our SMP development has been done using Trace32.

ARM DS-5 Support

Support for this board is included in the Altera QuartusII SoC EDS. Select the Cyclone V dual core option in the debug configuration to debug SMP applications. Otherwise follow the EDS, Eclipse and DS-5 documentation to set up and operate the debugger.

A suitable license file will also be needed.

Trace32 Support

Support for SMP debugging using the Lauterbach **Trace32** debugger using a Power Debug probe is available. You need suitable licenses for the Cortex-A/R families and multicore debugging in order to do this.

The **Trace32** debugger needs a startup script to initialize it for the Cyclone V. Some example scripts are present in the Cyclone V SX platform HAL (i.e. `packages/hal/arm/cortexa/cyclone5_sx/`*`VERSION`*`/misc`). The `ecosprompt.cmm` file provides setup for the standard **Trace32** application, and `ecosprompt-qt.cmm` is for the QT based variant avaliable on some host operating systems. These files are identical except for the layout file they load to define the window arrangement (`layoutmp.cmm` or `layoutmp-qt.cmm`). The expectation in these scripts is that all files are present in the same directory, along with the application being debugged. It is recommended that these files be copied out of the source repository into a working directory to which the application can also be copied and that Trace32 be started from the command line as follows:

```
$ cd /path/to/work/directory
$ t32marm-qt -s ecosprompt-qt.cmm
```

The lack of a clean hardware reset on this board means that the safest approach for debugging any application is to power cycle the board, start Trace32 and then load and run the application. To re-run the application, exit Trace32 before power cycling the board.

In addition to attaching to the target, these startup files define an additional `eCosPro` menu in the **Trace32** GUI. It contains the following entries:

MMU Table List

This entry causes a window showing the curent state of the MMU tables for the current CPU to be displayed. In eCos, all CPUs should be using the same shared table.

Load eCos.t32

This loads the **Trace32** eCos RTOS specialization extenstion. This file should be copied out of the Trace32 installation into the working directory. Note that the eCos RTOS support is not SMP-aware, so some information it displays in SMP application may be a little misleading. See the Lauterbach documentation on the RTOS debugger for eCos for more details of the functionality available.

Display Threads

Displays a list of current threads. Note that in SMP systems only the thread running on CPU 0 will be marked running. Those on other CPUs will be marked ready.

Display Scheduler

Displays state of scheduler. Only those parts of the scheduler state common between single and multi-core systems will be displayed.

Jump to CPU entrypoint

This entry disables the Caches and MMU and sets PC to zero. This may cause the system to restart once it is started running, however, it may also cause a crash since the hardware will not be in its initial state. In particular, if the second CPU is running, it is likely to cause a problem. Under most circumstances the board should be power cycled.

Load APP.ELF

This entry disables the caches and MMU, loads the file `app.elf` from the current directory, and loads any breakpoints from `breakpoints.cmm`. This is the menu entry that should be used to load and run SMP applications for development and testing. The `breakpoints.cmm` file allows a set of current breakpoints to be saved using the Store button on the breakpoint list window and have them reloaded automatically each time the application is reloaded.

Load APP.ELF Syms+Bkpts

This entry loads just the symbol tables and debug information from the application and also loads breakpoints from `breakpoints.cmm`. This is useful if the application is already running when Trace32 is attached. For example if it is a ROM startup application that has been loaded from flash, or an SMP startup application that was loaded by RedBoot.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the Cyclone V SX hardware, and should be read in conjunction with that specification. The platform HAL package complements the ARM architectural HAL, the Cortex-A variant HAL and the Altera HPS processor HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset, the HAL will initialize or reinitialize most of the on-chip peripherals. There is an exception for RAM startup applications which depend on a ROM monitor for certain services.

For ROM startup, the HAL will perform additional initialization. This is all done in the `PLATFORM_SETUP1` macro in the assembler header file `hal_platform_setup.h`.

Linker Scripts and Memory Maps

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

SDRAM
: This is located at address 0x00000000 of the physical memory space. The HAL configures the MMU to retain the SDRAM at virtual address 0x00000000 with caching enabled. The same memory is also accessible uncached and unbuffered at virtual location 0x40000000 for use by device drivers. The first 1MiB of RAM is left unmapped, allowing NULL pointer accesses to be trapped. Memory is required for the MMU tables, and must be aligned on a 16Kbyte boundary. These therefore occupy memory from 0x00100000 to 0x00004000. For ROM startup, all remaining SDRAM is available, although ROM applications actually load from 0x00200000. The virtual vector table is allocated as part of the RedBoot image and occupied 256 bytes from 0x00200050. RAM startup applications are loaded from location 0x00300000, reserving 1MiB for RedBoot.

On-chip SRAM
: On-chip SRAM is located at 0xFFFF0000 and occupied all of the remaining 64KiB to the top of the address space. It is identity mapped uncached. This port locates the exception vectors to high memory, at 0xFFFF0000. So, the first 32 bytes of SRAM are used for hardware exception vectors and the next 32 bytes are used for the VSR table. SRAM from 0xFFFF0040 is available for application use.

SPI NOR Flash
: SPI NOR flash media can only be accessed with the Flash API. For the purposes of this API a placeholder address range has been allocated as if the Flash is present at this address. The base of this address range is 0x80000000. This reserved range is not real memory and any attempt to access it directly by the processor other than via the Flash API will result in a memory address exception.

Peripheral Registers
: These are located at various addresses in the physical memory space above 0xC0000000. When the MMU is enabled, it sets up a direct, uncached, unbuffered mapping so that these registers remain accessible at their physical locations.

SPI NOR Flash

eCos supports QSPI access to the NOR flash on the board. The device is typically used to contain RedBoot and flash configuration data.

Accesses to SPI flash are performed via the Flash API, using 0x80000000 or as the nominal address of the device, although it does not truly exist in the processor address space.

Since SPI flash is not directly addressable, access from RedBoot is only possible using **fis** command operations.

Real-time characterization

The tm_basic kernel test gives statistics gathered about the real-time characterization and performance of the kernel. The sample output is shown here for information. The test was built in ARM32 mode.

Example 275.1. cyclone5_sx Real-time characterization

```
             Startup, main thrd : stack used   380 size  1792
             Startup :  Interrupt stack used  4096 size  4096
             Startup : Idlethread stack used    88 size  1280

eCos Kernel Timings
Notes: all times are in microseconds (.000001) unless otherwise stated

Reading the hardware clock takes 1 'ticks' overhead
... this value will be factored out of all other measurements
Clock interrupt took    1.12 microseconds (22 raw clock ticks)

Testing parameters:
   Clock samples:            32
   Threads:                  64
   Thread switches:         128
   Mutexes:                  32
   Mailboxes:                32
   Semaphores:               32
   Scheduler operations:    128
   Counters:                 32
   Flags:                    32
   Alarms:                   32
   Stack Size:             1088


                                 Confidence
     Ave     Min     Max     Var  Ave  Min  Function
  ======  ======  ======  ====== ========== ========
    0.75    0.55    1.25    0.13   57%  42% Create thread
    0.07    0.05    0.25    0.03   98%  57% Yield thread [all suspended]
    0.08    0.05    0.30    0.04   78%  67% Suspend [suspended] thread
    0.07    0.05    0.15    0.03   62%  62% Resume thread
    0.13    0.05    0.70    0.06   75%  71% Set priority
    0.01    0.00    0.10    0.02   78%  78% Get priority
    0.30    0.20    1.50    0.08   71%  64% Kill [suspended] thread
    0.08    0.05    0.15    0.02   65%  32% Yield [no other] thread
    0.19    0.15    0.40    0.04   87%  50% Resume [suspended low prio] thread
    0.08    0.05    0.30    0.03   93%  60% Resume [runnable low prio] thread
    0.11    0.10    0.25    0.02   82%  82% Suspend [runnable] thread
    0.07    0.05    0.15    0.03   57%  57% Yield [only low prio] thread
    0.06    0.05    0.20    0.02   84%  84% Suspend [runnable->not runnable]
    0.27    0.20    0.70    0.05   62%  82% Kill [runnable] thread
    0.20    0.15    0.65    0.03   53%  32% Destroy [dead] thread
    0.44    0.35    1.10    0.05   81%  68% Destroy [runnable] thread
    0.78    0.55    1.90    0.14   59%  26% Resume [high priority] thread
    0.26    0.20    0.40    0.04   46%  27% Thread switch

    0.00    0.00    0.15    0.00   99%  99% Scheduler lock
    0.05    0.00    0.10    0.01   89%   9% Scheduler unlock [0 threads]
    0.07    0.05    0.15    0.02   67%  67% Scheduler unlock [1 suspended]
    0.08    0.05    0.20    0.02   60%  36% Scheduler unlock [many suspended]
    0.08    0.05    0.15    0.02   64%  34% Scheduler unlock [many low prio]

    0.03    0.00    0.75    0.05   96%  90% Init mutex
    0.09    0.05    0.30    0.04   46%  43% Lock [unlocked] mutex
    0.11    0.05    0.60    0.04   68%  25% Unlock [locked] mutex
    0.07    0.05    0.40    0.04   93%  75% Trylock [unlocked] mutex
    0.05    0.00    0.15    0.01   87%   3% Trylock [locked] mutex
    0.02    0.00    0.15    0.03   84%  84% Destroy mutex
    0.74    0.65    1.05    0.06   71%  65% Unlock/Lock mutex

    0.02    0.00    0.35    0.03   96%  78% Create mbox
    0.01    0.00    0.10    0.01   93%  93% Peek [empty] mbox
    0.14    0.10    0.85    0.05   93%  93% Put [first] mbox
    0.00    0.00    0.00    0.00  100% 100% Peek [1 msg] mbox
    0.10    0.05    0.25    0.01   84%   9% Put [second] mbox
    0.00    0.00    0.10    0.01   93%  93% Peek [2 msgs] mbox
    0.12    0.05    0.75    0.05   84%   9% Get [first] mbox
    0.10    0.05    0.25    0.02   87%   6% Get [second] mbox
    0.08    0.05    0.30    0.03   50%  46% Tryput [first] mbox
    0.09    0.05    0.40    0.05   84%  59% Peek item [non-empty] mbox
    0.12    0.05    0.45    0.04   75%  15% Tryget [non-empty] mbox
    0.07    0.05    0.20    0.03   62%  62% Peek item [empty] mbox
    0.10    0.05    0.30    0.02   68%  25% Tryget [empty] mbox
    0.01    0.00    0.15    0.02   84%  84% Waiting to get mbox
    0.01    0.00    0.15    0.01   90%  90% Waiting to put mbox
    0.03    0.00    0.25    0.04   90%  71% Delete mbox
    0.46    0.40    0.90    0.04   68%  25% Put/Get mbox

    0.00    0.00    0.15    0.01   96%  96% Init semaphore
    0.06    0.05    0.15    0.01   87%  87% Post [0] semaphore
    0.10    0.05    0.55    0.04   56%  34% Wait [1] semaphore
    0.07    0.05    0.30    0.03   84%  84% Trywait [0] semaphore
    0.06    0.05    0.15    0.01   90%  90% Trywait [1] semaphore
    0.00    0.00    0.05    0.01   93%  93% Peek semaphore
    0.02    0.00    0.20    0.03   87%  87% Destroy semaphore
    0.40    0.35    0.85    0.05   34%  46% Post/Wait semaphore

    0.02    0.00    0.25    0.03   78%  78% Create counter
    0.02    0.00    0.10    0.02   75%  75% Get counter value
    0.00    0.00    0.05    0.00   96%  96% Set counter value
    0.08    0.05    0.15    0.02   59%  37% Tick counter
    0.02    0.00    0.15    0.03   84%  84% Delete counter

    0.01    0.00    0.20    0.01   96%  96% Init flag
    0.06    0.05    0.40    0.03   90%  90% Destroy flag
    0.09    0.05    0.25    0.03   62%  31% Mask bits in flag
    0.11    0.05    0.45    0.04   62%  25% Set bits in flag [no waiters]
    0.13    0.10    0.65    0.05   93%  84% Wait for flag [AND]
    0.12    0.10    0.35    0.03   71%  71% Wait for flag [OR]
    0.12    0.10    0.45    0.03   90%  90% Wait for flag [AND/CLR]
    0.10    0.05    0.40    0.02   71%  21% Wait for flag [OR/CLR]
    0.00    0.00    0.00    0.00  100% 100% Peek on flag

    0.05    0.00    0.50    0.03   71%  25% Create alarm
    0.12    0.05    0.85    0.06   68%  81% Initialize alarm
    0.07    0.00    0.35    0.04   78%  12% Disable alarm
    0.11    0.05    0.70    0.04   78%  18% Enable alarm
    0.07    0.05    0.20    0.03   71%  71% Delete alarm
    0.10    0.05    0.20    0.01   84%   9% Tick counter [1 alarm]
    0.46    0.40    0.60    0.03   68%  18% Tick counter [many alarms]
    0.16    0.10    0.50    0.03   71%  18% Tick & fire counter [1 alarm]
    2.64    2.60    3.15    0.06   90%  90% Tick & fire counters [>1 together]
    0.56    0.50    1.10    0.05   50%  90% Tick & fire counters [>1 separately]
    0.92    0.90    1.25    0.03   99%  58% Alarm latency [0 threads]
    0.98    0.85    1.35    0.06   58%  26% Alarm latency [2 threads]
    1.20    1.00    1.65    0.09   51%  14% Alarm latency [many threads]
    1.35    1.30    2.15    0.02   79%  18% Alarm -> thread resume latency

    0.35    0.35    0.90    0.00            Clock/interrupt latency

    0.39    0.30    1.10    0.00            Clock DSR latency
```

```
     224     172     460                    Worker thread stack used (stack size 1088)
            All done, main thrd : stack used  1012 size  1792
            All done :  Interrupt stack used   156 size  4096
            All done : Idlethread stack used   232 size  1280

Timing complete - 29810 ms total

PASS:<Basic timing OK>
EXIT:<done>
```

Other Issues

The platform HAL does not affect the implementation of other parts of the eCos HAL specification. The HPS processor HAL and the ARM architectural HAL documentation should be consulted for further details.

Chapter 276. Dream Chip A10 Board Support

Name

eCos Support for the Dream Chip A10 Board — Overview

Description

This document covers the configuration and usage of eCos and RedBoot on the Dream Chip Arria 10 SoM and baseboard Development kit. This board is fitted with avariant of the Arria 10 family of FPGAs and it therefore referred to in this document and the configuration system as the Dream Chip A10, to differentiate it from other Arria 10 development boards.

In addition to the Arria 10 FPGA, the board contains 2GiB SDRAM main memory, a 500Mib (64GiB) SPI NOR flash, a micro-SD card socket, a USB bridge connected to UART1, an Ethernet socket for an HPS Ethernet interface, plus a variety of connectors for other interfaces plus resources devoted to the FPGA. The extent of eCos support for the devices and peripherals on the board and the CPU is described below.

For typical eCos development, a RedBoot image is programmed into the SPI NOR flash memory, and the board will load this image from reset. RedBoot provides gdb stub functionality so it is then possible to download and debug stand-alone and eCos applications via the gdb debugger. This can happen over either a serial line or over Ethernet. Alternatively, RedBoot may be loaded from an SD card.

Support for SMP operation of the two Cortex-A9 CPUs on the SoC is available, although debugging support is restricted to use of an external JTAG debugger. There is no SMP support in RedBoot.

This documentation is expected to be read in conjunction with the Altera HPS processor HAL documentation and further device support and subsystems are described and documented there.

Supported Hardware

The SPI NOR flash consists of 1024 blocks of 64Ki bytes each. In a typical setup, the first 16Mib are reserved for the second-level bootstrap, in this case a port of U-Boot, plus the FPGA bitstreams. The 10 blocks from 1MiB are reserved for the ROM RedBoot image. The topmost block is used to manage the flash and also holds RedBoot **fconfig** values. The remaining blocks can be used by application code.

Serial support is through the `CYGPKG_IO_SERIAL_GENERIC_16X5X` generic driver package which is modified by the `CYGPKG_IO_SERIAL_ARM_ALTERA_HPS` driver package for the HPS. These packages can support all the serial devices on the HPS. However, this board only has UART1 connected to an external connector which this HAL indicates by implementing the `CYGINT_HAL_ARM_CORTEXA_ALTERA_HPS_UART1` interface. This serial channel is used by RedBoot for communication with the host. If this device is needed by the application, either directly or via the serial driver, then it cannot also be used for RedBoot communication. Another communication channel such as Ethernet should be used instead. The serial driver package is loaded automatically when configuring for the `dreamchip-a10` target.

There is an Ethernet driver `CYGPKG_DEVS_ETH_DWC_GMAC` for the on-chip Ethernet device. A separate package, `CYGPKG_DEVS_ETH_DREAMCHIP_A10` configures this generic driver to the hardware. This driver is also loaded automatically when configuring for the `dreamchip-a10` board.

There is a watchdog driver `CYGPKG_DEVICES_WATCHDOG_DWWDT`. This driver is also loaded automatically when configuring for the board.

The HPS processor HAL contains a driver for the MultiMedia Card Interface (MMC/SD). This driver is loaded automatically when configuring `CYGPKG_DEVS_DISK_MMCSD_BUS` for this target and allows use of MultiMediaCard (MMC) and Secure Digital (SD) flash storage cards within eCos, exported as block devices. Further documentation on the driver may be found in the Altera Hard Processor System Support HAL documentation.

Note

There is no working card-detect (media change) signal available on the board for the J3 MicroSD slot.

The platform HAL provides definitions to enable access to flash devices on the SPI bus. The HAL enables the QSPI driver (CYGPKG_DEVS_FLASH_QSPI) which in turn provides the underlying implementation for access to the Micron N25Q512A SPI NOR flash. The QSPI support integrates with the CYGPKG_DEVS_FLASH_SPI_M25PXX package. These packages are automatically loaded when configuring for the target. This driver is capable of supporting the JFFS2 filesystem, although at reduced performance compared with a parallel flash device.

In general, devices (Caches, GPIO, UARTs) are initialized only as far as is necessary for eCos to run. Other devices (RTC, SPI, MMC/SD etc.) are not touched unless the appropriate driver is loaded, although in some cases, the HAL boot sequence will set up the appropriate power control and pin multiplexing configuration.

All development and testing was undertaken using the DCT10A22L20G2T4C3ES variant of the Dream Chip Arria 10 SoM and DCT10ABASE Evaluation Baseboard.

Tools

The board support is intended to work with GNU tools configured for an arm-eabi target. The original port was undertaken using arm-eabi-gcc version 7.3.0, arm-eabi-gdb version 8.1, and binutils version 2.30.

Name

Setup — Preparing the Dream Chip A10 board for eCos Development

Overview

In a typical development environment, the board boots from the SPI NOR and runs the RedBoot ROM monitor from SDRAM. eCos applications are configured for RAM startup and then downloaded and run on the board via the debugger **arm-eabi-gdb**. Preparing the board therefore usually involves programming a suitable RedBoot image into flash memory.

The following RedBoot configurations are supported:

Configuration	Description	Use	File
ROM	RedBoot loaded from SPI NOR flash to SDRAM	`redboot_ROM.ecm`	`redboot_ROM.bin`

For serial communications, all versions run with 8 bits, no parity, and 1 stop bit at 115200 baud. RedBoot also supports Ethernet communication and flash management.

Note that the use of the term ROM for the initial RedBoot configuration is a historical accident. RedBoot actually runs from SDRAM after being loaded there from NOR flash by U-Boot. The use of ROM for this configuration is intended to indicate that it initializes the microprocessor and board peripherals, as opposed to the RAM configuration which assumes that this has already been done.

Initial Installation

For eCos development we want to install RedBoot in the SPI NOR flash. The booting mechanism is that the on-chip firmware loads U-Boot from the start of NOR flash which then loads RedBoot from later in the flash.

To write RedBoot to the SPI NOR flash, you must use the command line tools from the Intel Quartus FPGA Development Tools and the SoC FPGA Embedded Development Suite. Any of the Prime, Standard or Lite versions of these tools may be used.

Programming RedBoot into NOR flash using Altera Tools

The following gives the steps needed to program RedBoot into the SPI NOR Flash using the Altera tools. This uses the **quartus_hps** flash programmer command line utility to do the work.

Note

The examples below are for a Linux host where eCosPro is typically installed in the `/opt/ecospro/ecospro-<version>` sub-directory. On a Windows host eCosPro will typically be installed in the `C:\eCosPro\ecos-<version>` sub-directory.

1. Install Quartus and the SoC EDS as described by Intel onto your host system. Quartus and EDS should be installed into the same directory if the following script is to work. Start an Embedded Command Shell by running either `<SoC EDS Installation Folder>\embedded\Embedded_Command_Shell.bat` in Windows or `<SoC EDS Installation Folder>/embedded/embedded_command_shell.sh` in Linux.

2. Ensure that there is no SD card installed.

3. Connect USB cables between the mini USB socket at J14 (used by the USB-Blaster), and the mini USB socket at J13 (used to provide serial device access) and your host. Power up the board and run a terminal emulator on your host, attaching it to the serial USB channel which should appear on your host, setting the baud rate to 115200. Attach an Ethernet cable to the RJ45 connector.

4. Run the following command on the host to detect the USB-Blaster:

```
   # jtagconfig
   1) DCT10ABASE [1-5]
     02E020DD   10AS022C(3|4)/10AS022E(3|4)
     4BA00477   SOCVHPS

   #
   ```

   It may be necessary to run this a couple of times before the above result is obtained. If the program failure persists then you should check that your EDS installation is correct.

5. If installing the prebuilt RedBoot image provided with the eCosPro release:
   change directory to `loaders/dreamchip_a10/etc/qspi_boot` within the `ecos-x.y.z` installation sub-directory.

   If installing the RedBoot image you built:
   change directory to `etc/qspi_boot` in the `install` directory.

6. Run the script `qspi_prog`. If you are running on a Windows host, you will need to run the script using the **bash** command. This will use the **quartus_hps** to program U-Boot, the FPGA bit streams and RedBoot into the QSPU flash. This script will produce a lot of output, which is not reproduced here. Monitor the output to ensure each component is written to the flash correctly.

7. Power cycle the board. Output similar to the following should be seen on the serial line:

   ```
   U-Boot 2014.10-dirty (Feb 20 2023 - 12:40:11)

   CPU   : Altera SOCFPGA Arria 10 Platform
   BOARD : Dream Chip Arria 10 SoM base
   I2C:   ready
   DRAM:  WARNING: Caches not enabled
   SF: Read data capture delay calibrated to 3 (0 - 6)
   SF: Detected N25Q512A with page size 256 Bytes, erase size 4 KiB, total 64 MiB
   FPGA: Early Release Succeeded.
   SF: Detected N25Q512A with page size 256 Bytes, erase size 4 KiB, total 64 MiBDDRCAL: Success
   INFO  : Skip relocation as SDRAM is non secure memory
   Reserving 2048 Bytes for IRQ stack at: ffe386e8
   DRAM  : 2 GiB
   WARNING: Caches not enabled
   MMC:   SOCFPGA DWMMC: 0
   SF: Read data capture delay calibrated to 8 (0 - 15)
   SF: Detected N25Q512A with page size 256 Bytes, erase size 4 KiB, total 64 MiB
   In:    serial
   Out:   serial
   Err:   serial
   Model: Dreamchip Arria10 SoM
   Net:   dwmac.ff802000
   Hit any key to stop autoboot:  0
   SF: Read data capture delay calibrated to 3 (0 - 6)
   SF: Detected N25Q512A with page size 256 Bytes, erase size 4 KiB, total 64 MiB
   Full Configuration Succeeded.
   SF: Detected N25Q512A with page size 256 Bytes, erase size 4 KiB, total 64 MiB
   SF: 655360 bytes @ 0x1000000 Read: OK
   ## Starting application at 0x00100000 ...
   +**Warning** FLASH configuration checksum error or invalid key
   Use 'fconfig -i' to [re]initialize database
   Ethernet eth0: MAC address 12:34:aa:bb:cc:ee
   IP: 10.0.2.2/255.0.0.0, Gateway: 10.0.0.3
   Default server: 10.0.1.1
   DNS server IP: 10.0.0.5, DNS domain name: <null>

   RedBoot(tm) bootstrap and debug environment [ROM]
   eCosCentric certified release, version 4.7.7 - built 14:58:38, Apr 12 2023

   Copyright (C) 2000-2009 Free Software Foundation, Inc.
   Copyright (C) 2003-2023 eCosCentric Limited
   The RedBoot bootloader is a component of the eCos real-time operating system.
   ```

```
Want to know more? Visit www.ecoscentric.com for everything eCos & RedBoot related.
This is free software, covered by the eCosPro Non-Commercial Public License
and eCos Public License. You are welcome to change it and/or distribute copies
of it under certain conditions. Under the license terms, RedBoot's source code
and full license terms must have been made available to you.
Redboot comes with ABSOLUTELY NO WARRANTY.

Platform: Dreamchip Arria 10 SoM Development Kit (Cortex-A9)
RAM: 0x00000000-0x40000000 [0x0017be78-0x3fd2d000 available]
  Arena: base 0x3fe00000, size 0x200000, 99% free, maxfree 0x1fffec
FLASH: 0x80000000-0x83ffffff, 1024 x 0x10000 blocks
RedBoot>
```

8. Run the following commands to initialize RedBoot's flash file system and flash configuration:

```
RedBoot> fis init
About to initialize [format] FLASH image system - continue (y/n)? y
*** Initialize FLASH Image System
... Erase from 0x83ff0000-0x83ffffff: .
... Program from 0x3fdf0000-0x3fe00000 to 0x83ff0000: .
RedBoot>
RedBoot> fconfig -i
Initialize non-volatile configuration - continue (y/n)? y
Run script at boot: false
Use BOOTP for network configuration: true
Default server IP address: 10.0.1.1
Console baud rate: 115200
DNS domain name: calivar.com
DNS server IP address: 10.0.0.5
Network hardware address [MAC] for eth0: 0x12:0x34:0xAA:0xBB:0xCC:0xEE
GDB connection port: 9000
Force console for special debug messages: false
Network debug at boot time: false
Default network device: dwc_gmac
Update RedBoot non-volatile configuration - continue (y/n)? y
... Erase from 0x83ff0000-0x83ffffff: .
... Program from 0x3fdf0000-0x3fe00000 to 0x83ff0000: .
RedBoot>
```

   You should substitute your own server IP address for the one shown above. You may also want to change the MAC address if more than one board is present on the network. If you want to use a static IP address, then choose false for the "Use BOOTP" option and enter the gateway, IP address and netmask that you have assigned.

The RedBoot installation is now complete. Power cycling the board should show

```
U-Boot 2014.10-dirty (Feb 20 2023 - 12:40:11)

CPU   : Altera SOCFPGA Arria 10 Platform
BOARD : Dream Chip Arria 10 SoM base
I2C:   ready
DRAM:  WARNING: Caches not enabled
SF: Read data capture delay calibrated to 3 (0 - 6)
SF: Detected N25Q512A with page size 256 Bytes, erase size 4 KiB, total 64 MiB
FPGA: Early Release Succeeded.
SF: Detected N25Q512A with page size 256 Bytes, erase size 4 KiB, total 64 MiBDDRCAL: Success
INFO  : Skip relocation as SDRAM is non secure memory
Reserving 2048 Bytes for IRQ stack at: ffe386e8
DRAM  : 2 GiB
WARNING: Caches not enabled
MMC:   SOCFPGA DWMMC: 0
SF: Read data capture delay calibrated to 8 (0 - 15)
SF: Detected N25Q512A with page size 256 Bytes, erase size 4 KiB, total 64 MiB
In:    serial
Out:   serial
Err:   serial
Model: Dreamchip Arria10 SoM
```

```
Net:   dwmac.ff802000
Hit any key to stop autoboot:  0
SF: Read data capture delay calibrated to 3 (0 - 6)
SF: Detected N25Q512A with page size 256 Bytes, erase size 4 KiB, total 64 MiB
Full Configuration Succeeded.
SF: Detected N25Q512A with page size 256 Bytes, erase size 4 KiB, total 64 MiB
SF: 655360 bytes @ 0x1000000 Read: OK
## Starting application at 0x00100000 ...
+Ethernet eth0: MAC address 12:34:aa:bb:cc:ee
IP: 10.0.2.2/255.0.0.0, Gateway: 10.0.0.3
Default server: 10.0.1.1
DNS server IP: 10.0.0.5, DNS domain name: <null>

RedBoot(tm) bootstrap and debug environment [ROM]
eCosCentric certified release, version 4.7.7 - built 14:58:38, Apr 12 2023

Copyright (C) 2000-2009 Free Software Foundation, Inc.
Copyright (C) 2003-2023 eCosCentric Limited
The RedBoot bootloader is a component of the eCos real-time operating system.
Want to know more? Visit www.ecoscentric.com for everything eCos & RedBoot related.
This is free software, covered by the eCosPro Non-Commercial Public License
and eCos Public License. You are welcome to change it and/or distribute copies
of it under certain conditions. Under the license terms, RedBoot's source code
and full license terms must have been made available to you.
Redboot comes with ABSOLUTELY NO WARRANTY.

Platform: Dreamchip Arria 10 SoM Development Kit (Cortex-A9)
RAM: 0x00000000-0x40000000 [0x0017be78-0x3fd2d000 available]
  Arena: base 0x3fe00000, size 0x200000, 99% free, maxfree 0x1fffec
FLASH: 0x80000000-0x83ffffff, 1024 x 0x10000 blocks
RedBoot>
RedBoot> fis list
Name              FLASH addr  Mem addr    Length      Entry point
(reserved)        0x80000000  0x80000000  0x01000000  0x00000000
RedBoot           0x81000000  0x81000000  0x000A0000  0x00000000
FIS directory     0x83FF0000  0x83FF0000  0x0000F000  0x00000000
RedBoot config    0x83FFF000  0x83FFF000  0x00001000  0x00000000
RedBoot>
```

If it proves necessary to install a new version of RedBoot, this may be done from RedBoot itself. Place the new image on a TFTP server on the configured server. From RedBoot run the following commands:

```
RedBoot> load -r -b %{freememlo} redboot.bin
Using default protocol (TFTP)
Raw file loaded 0x001a6800-0x0020e953, assumed entry at 0x001a6800
RedBoot> fis cre RedBoot
An image named 'RedBoot' exists - continue (y/n)? y
... Erase from 0x81000000-0x8109ffff: ..........
... Program from 0x001a6800-0x0020e954 to 0x81000000: .......
... Erase from 0x83ff0000-0x83ffffff: .
... Program from 0x3fdf0000-0x3fe00000 to 0x83ff0000: .
RedBoot>
```

Booting RedBoot from an SD card

The following gives the steps needed to run RedBoot from an SD card. This does not need any tools from Quartus.

Notes:

1. These instructions are only applicable to developers running on a Linux host.
2. eCosPro is typically installed in the `/opt/ecospro/ecospro-<version>` sub-directory.

1. Insert a blank SD card into an SD card drive on your host machine and identify the device name by either monitoring the system messages or using **lsblk**.

2. Connect USB cables between the mini USB socket at J14 (used by the USB-Blaster), and the mini USB socket at J13 (used to provide serial device access) and your host. Run a terminal emulator on your host, attaching it to the serial USB channel which should appear on your host, setting the baud rate to 115200. Attach an Ethernet cable to the RJ45 connector.

3. If installing the prebuilt RedBoot image provided with the eCosPro release:
   change directory to `loaders/dreamchip_a10/etc/sd_boot` within the `ecos-x.y.z` installation sub-directory.

   If installing the RedBoot image you built:
   change directory to `etc/sd_boot` in the `install` directory.

4. Run the script `sd_prog` as root giving it the name of the SD device as an argument. For example **sudo ./sdcard_build /dev/sdx**. This will format the SD card and copy U-Boot, the FPGA bit streams and RedBoot on to it. flash. The script should produce output similar to the following:

```
$ sudo ./sdcard_build /dev/sdh
1024+0 records in
1024+0 records out
1048576 bytes (1.0 MB, 1.0 MiB) copied, 0.261784 s, 4.0 MB/s
Checking that no-one is using this disk right now ... OK

Disk /dev/sdh: 14.84 GiB, 15931539456 bytes, 31116288 sectors
Disk model: MassStorageClass
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes

>>> Script header accepted.
>>> Script header accepted.
>>> Script header accepted.
>>> Script header accepted.
>>> Script header accepted.
>>> Created a new DOS disklabel with disk identifier 0x6d0fd677.
/dev/sdh1: Created a new partition 1 of type 'W95 FAT32 (LBA)' and of size 100 MiB.
Partition #1 contains a vfat signature.
/dev/sdh2: Created a new partition 2 of type 'W95 FAT32 (LBA)' and of size 100 MiB.
Partition #2 contains a vfat signature.
/dev/sdh3: Created a new partition 3 of type 'Unknown' and of size 10 MiB.
/dev/sdh4: Done.

New situation:
Disklabel type: dos
Disk identifier: 0x6d0fd677

Device     Boot  Start    End Sectors  Size Id Type
/dev/sdh1        24576 229375  204800  100M  c W95 FAT32 (LBA)
/dev/sdh2       229376 434175  204800  100M  c W95 FAT32 (LBA)
/dev/sdh3         2048  22528   20481   10M a2 unknown

Partition table entries are not in disk order.

The partition table has been altered.
Calling ioctl() to re-read partition table.
Syncing disks.
2048+0 records in
2048+0 records out
1048576 bytes (1.0 MB, 1.0 MiB) copied, 0.0472776 s, 22.2 MB/s
8+0 records in
8+0 records out
4096 bytes (4.1 kB, 4.0 KiB) copied, 0.0525689 s, 77.9 kB/s
mkfs.fat 4.1 (2017-01-24)
$
```

5. Remove the SD card from the host and insert into the SD socket on the board.

6. Power up the board. Output similar to the following should appear on the serial line:

```
U-Boot 2014.10-dirty (Feb 20 2023 - 12:40:11)

CPU   : Altera SOCFPGA Arria 10 Platform
BOARD : Dream Chip Arria 10 SoM base
I2C:   ready
DRAM:  WARNING: Caches not enabled
SF: Read data capture delay calibrated to 3 (0 - 6)
SF: Detected N25Q512A with page size 256 Bytes, erase size 4 KiB, total 64 MiB


U-Boot 2014.10-dirty (Feb 15 2023 - 11:20:29)

CPU   : Altera SOCFPGA Arria 10 Platform
BOARD : Dream Chip Arria 10 SoM base
I2C:   ready
DRAM:  WARNING: Caches not enabled
SOCFPGA DWMMC: 0
FPGA: writing ghrd_10AS048E4.periph.rbf ...
FPGA: Early Release Succeeded.
DDRCAL: Success
INFO  : Skip relocation as SDRAM is non secure memory
Reserving 2048 Bytes for IRQ stack at: ffe386e8
DRAM  : 2 GiB
WARNING: Caches not enabled
MMC:   In:    serial
Out:   serial
Err:   serial
Model: Dreamchip Arria10 SoM
Net:   dwmac.ff802000
Hit any key to stop autoboot:  0
FPGA: writing ghrd_10AS048E4.core.rbf ...
Full Configuration Succeeded.
293108 bytes read in 16 ms (17.5 MiB/s)
## Starting application at 0x00100000 ...
+Ethernet eth0: MAC address 12:34:aa:bb:cc:ee
IP: 10.0.2.2/255.0.0.0, Gateway: 10.0.0.3
Default server: 10.0.1.1
DNS server IP: 10.0.0.5, DNS domain name: <null>

RedBoot(tm) bootstrap and debug environment [ROM]
eCosCentric certified release, version 4.7.7 - built 14:58:38, Apr 12 2023

Copyright (C) 2000-2009 Free Software Foundation, Inc.
Copyright (C) 2003-2023 eCosCentric Limited
The RedBoot bootloader is a component of the eCos real-time operating system.
Want to know more? Visit www.ecoscentric.com for everything eCos & RedBoot related.
This is free software, covered by the eCosPro Non-Commercial Public License
and eCos Public License. You are welcome to change it and/or distribute copies
of it under certain conditions. Under the license terms, RedBoot's source code
and full license terms must have been made available to you.
Redboot comes with ABSOLUTELY NO WARRANTY.

Platform: Dreamchip Arria 10 SoM Development Kit (Cortex-A9)
RAM: 0x00000000-0x40000000 [0x001865e0-0x3fe00000 available]
  Arena: base 0x3fe00000, size 0x200000, 99% free, maxfree 0x1fffec
RedBoot>
```

7. Due to the different FPGA bitstreams, The SD card version of RedBoot cannot access the QSPI flash. Consequently any configuration such as Ethernet MAC address or IP address must be configured in to the RedBoot executable.

If it proves necessary te reinstall RedBoot then rerunning the above script is the simplest approach. Alterntively the first partition on the SD card can be mounted on the host and the new RedBoot executable copied over.

Rebuilding RedBoot

Should it prove necessary to rebuild a RedBoot binary, this is done most conveniently at the command line. The steps needed to rebuild the ROM version of RedBoot for the Dream Chip A10 are:

```
$ mkdir redboot_dreamchip_a10_rom
$ cd redboot_dreamchip_a10_rom
$ ecosconfig new dreamchip_a10 redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/arm/cortexa/dreamchip_a10/VERSION/misc/redboot_ROM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

At the end of the build the `install/bin` subdirectory should contain the file `redboot.bin`. This is a binary file that includes a header needed by the preloader to load and run RedBoot successfully.

These instructions build a RedBoot that can be booted from the QSPI flash. To build a RedBoot to boot from SD, use `redboot_ROM_SD.ecm` in the above import command.

Name

Configuration — Platform-specific Configuration Options

Overview

The Dream Chip A10 platform HAL package is loaded automatically when eCos is configured for the `dreamchip-a10` target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The platform HAL package supports three startup types:

RAM
: This is the startup type which is normally used during application development. The board has RedBoot programmed into flash and boots into that initially. arm-eabi-gdb is then used to load a RAM startup application into memory and debug it. It is assumed that the hardware has already been initialized by RedBoot. By default the application will use the eCos virtual vectors mechanism to obtain services from RedBoot, including diagnostic output.

ROM
: This startup type can be used for finished applications which will be programmed into Flash. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization. This startup type can also be used for applications loaded via JTAG.

SMP
: This startup type can be used for finished applications that can be loaded into RAM via RedBoot. The load address is set to the same as for RAM applications, however, the application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization. Once started, this application takes full control of the system and RedBoot will not be called again. This means that debugging via RedBoot will not be possible, only JTAG-based hardware debugging is supported. The intent of this startup type is to allow SMP test programs to be run from RedBoot, most SMP applications should use the ROM startup type. This startup type can also be used for applications loaded directly via JTAG.

RedBoot and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building RedBoot.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then the serial port will be claimed for HAL diagnostics.

Flash Driver

The Dream Chip A10 board contains a 64Mbyte Micron SPI serial NOR flash device attached to the QSPI controller. The `CYGPKG_DEVS_FLASH_SPI_M25PXX` package contains all the code necessary to support this part and the platform HAL package contains definitions that customize the driver to the Dream Chip A10. This driver is not active until the generic Flash support package, `CYGPKG_IO_FLASH`, is included in the configuration.

This driver is capable of supporting the JFFS2 filesystem. However, note that the SPI interface means that this file system has reduced bandwidth and increased latency compared with other implementations. All that is required to enable the support is to include the filesystem (`CYGPKG_FS_JFFS2`) and any of its package dependencies (including `CYGPKG_IO_FILEIO` and `CYGPKG_LINUX_COMPAT`) together with the flash infrastructure (`CYGPKG_IO_FLASH`).

Ethernet Driver

The board uses the HPS's internal GMAC Ethernet device attached to an external Micrel KSZ9021 Gigabit PHY. The `CYGPKG_DEVS_ETH_DWC_GMAC` package contains all the code necessary to support this device and the `CYGPKG_DEVS_ETH_DREAMCHIP_A10` package contains definitions that customize the driver to the board. This driver is not active until the generic Ethernet support package, `CYGPKG_IO_ETH_DRIVERS`, is included in the configuration.

Support for PHY events is made available by default when the Kernel (`CYGPKG_KERNEL`) is available in a configuration.

Watchdog Driver

The board uses the HPS's internal watchdog support. The `CYGPKG_DEVICES_WATCHDOG_DWWDT` package contains all the code necessary to support this device. Within that package the `CYGNUM_DEVS_WATCHDOG_DWWDT_DESIRED_TIMEOUT_MS` configuration option controls the watchdog timeout, and by default will force a reset of the board upon timeout. This driver is not active until the generic watchdog device support package, `CYGPKG_IO_WATCHDOG`, is included in the configuration.

UART Serial Driver

The board uses the HPS's internal UART serial support as described in the HPS processor HAL documentation. Only one serial connector is available on the board, which is connected to UART1 via a USB bridge. Only the UART data lines are connected to the bridge, so hardware flow control is not supported.

MMC/SD Driver

As the Dream Chip A10 MMC/SD driver is part of the HPS processor HAL, nothing is required to load it. Similarly the MMC/SD bus driver layer (`CYGPKG_DEVS_DISK_MMC`) is automatically included as part of the hardware-specific configuration for this target. All that is required to enable the support is to include the generic disk I/O infrastructure package (`CYGPKG_IO_DISK`), along with the intended filesystem, typically, the FAT filesystem (`CYGPKG_FS_FAT`) and any of its package dependencies (e.g. for FAT also including `CYGPKG_LIBC_STRING` and `CYGPKG_LINUX_COMPAT` packages).

Various options can be used to control specifics of the MMC/SD driver. Consult the HPS processor HAL documentation for information on its configuration.

This board does not have a working MMC/SD card detect for MicroSD socket (J3), thus the disk I/O layer's removeable media support is not available.

Compiler Flags

The platform HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos. The following flags are specific to this port:

`-mcpu=cortex-a9`
The arm-eabi-gcc compiler supports many variants of the ARM architecture. A `-m` option should be used to select the specific variant in use, and with current tools `-mcpu=cortex-a9` is the correct option for the CPU in the HPS.

`-mthumb`
The arm-eabi-gcc compiler will compile C and C++ files into the Thumb2 instruction set when this option is used. The best way to build eCos in Thumb mode is to enable the configuration option `CYGHWR_THUMB`.

`-mno-unaligned-access`
The Cortex-A CPU allows unaligned memory accesses and the default for arm-eabi-gcc is to generate instructions that make unaligned accesses. However, for this port, alignment exceptions are enabled, so unaligned accesses should not be made. This option disables un-

aligned accesses. Note that there is a performance and code size cost in doing this, since all accesses to unaligned data must now be made using individual byte accesses.

Name

JTAG support — Usage

Use of JTAG for debugging

JTAG can be used to single-step and debug loaded applications, including RedBoot. Only ROM configuration applications should be debugged using JTAG, RAM applications assume the presence of RedBoot.

OpenOCD notes

OpenOCD support is available for loading and running ROM or SMP startup applications. Single CPU applications can be debugged via OpenOCD but SMP support is not currently available.

An OpenOCD configuration file, `openocd.dreamchip_a10.cfg`, is available in the `misc` subdirectory of the platform hal subdirectory. This will be installed as `etc/openocd.cfg` in the `install` subdirectory when building the eCos libarary for use by the eCos GUI Configuration Tool or Eclipse when running tests or debugging your application respectively. This openocd configuration file should be used with a recent release of OpenOCD.

Name

SMP Support — Usage

Overview

Support is available for SMP operation of the two CPUs on this platform. However, debugging support is restricted to using an external SMP-aware JTAG debugger like ARM's DS-5 or a Lauterbach Power Debug probe. RedBoot does not have support for multi-core debugging.

A board intended to be used for SMP operation should be initialized in the same way as a single core board by installing the preloader and RedBoot. If RedBoot is not needed then the preloader must still be installed and a ROM startup application, with headers added by **flashimg**, can be written to flash in place of RedBoot.

SMP support is enabled by setting `CYGPKG_KERNEL_SMP_SUPPORT` to true. SMP applications should only be built using either ROM or SMP startup types. ROM applications can be loaded by U-Boot in place of RedBoot. The SMP startup is identical to a ROM startup except that the load address is set to allow the application to be loaded into a higher location in RAM from RedBoot. Both application types may also be loaded via a JTAG debugger.

Loading an SMP startup application via RedBoot can be done from the RedBoot command line via serial or Ethernet. It may also be loaded via a GDB connection on serial or Ethernet. However, once started running the SMP application will take full control of the system, including redirecting all interrupt sources, exception vectors and virtual vector table entries. This means that RedBoot will no-longer be active. Any breakpoints planted by GDB will result in an exception to the application, Ctrl-C will not work, any Ethernet connections will be lost and serial output will come from the application in plain ASCII. Any GDB connection will be lost and GDB may start reporting packet errors.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the Dream Chip A10 hardware, and should be read in conjunction with that specification. The platform HAL package complements the ARM architectural HAL, the Cortex-A variant HAL and the Altera HPS processor HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset, the HAL will initialize or reinitialize most of the on-chip peripherals. There is an exception for RAM startup applications which depend on a ROM monitor for certain services.

For ROM startup, the HAL will perform additional initialization. This is all done in the `PLATFORM_SETUP1` macro in the assembler header file `hal_platform_setup.h`.

Linker Scripts and Memory Maps

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

SDRAM
: This is located at address 0x00000000 of the physical memory space. The SDRAM is 2GiB in size, although only the first 1GiB is used by eCos. The HAL configures the MMU to retain the SDRAM at virtual address 0x00000000 with caching enabled. The same memory is also accessible uncached and unbuffered at virtual location 0x40000000 for use by device drivers. Memory is required for the MMU tables, and must be aligned on a 16Kbyte boundary. These therefore occupy memory from 0x00010000 to 0x00014000. For ROM startup, all remaining SDRAM is available, although ROM applications actually load from 0x00100000. The virtual vector table is allocated as part of the RedBoot image and occupied 256 bytes from 0x00000050. RAM startup applications are loaded from location 0x00200000, reserving 1MiB for RedBoot.

SPI NOR Flash
: SPI NOR flash media can only be accessed with the Flash API. For the purposes of this API a placeholder address range has been allocated as if the Flash is present at this address. The base of this address range is 0x80000000. This reserved range is not real memory and any attempt to access it directly by the processor other than via the Flash API will result in a memory address exception.

Peripheral Registers
: These are located at various addresses in the physical memory space above 0xC0000000. When the MMU is enabled, it sets up a direct, uncached, unbuffered mapping so that these registers remain accessible at their physical locations.

SPI NOR Flash

eCos supports QSPI access to the NOR flash on the board. The device is typically used to contain RedBoot and flash configuration data.

Accesses to SPI flash are performed via the Flash API, using 0x80000000 or as the nominal address of the device, although it does not truly exist in the processor address space.

Since SPI flash is not directly addressable, access from RedBoot is only possible using **fis** command operations.

Real-time characterization

The tm_basic kernel test gives statistics gathered about the real-time characterization and performance of the kernel. The sample output is shown here for information. The test was built in ARM32 mode.

Example 276.1. dreamchip_a10 Real-time characterization

```
             Startup, main thrd : stack used   404 size  2304
             Startup :  Interrupt stack used  4096 size  4096
             Startup : Idlethread stack used    96 size  1280

eCos Kernel Timings
Notes: all times are in microseconds (.000001) unless otherwise stated

Reading the hardware clock takes 0 'ticks' overhead
... this value will be factored out of all other measurements
Clock interrupt took    1.00 microseconds (1 raw clock ticks)

Testing parameters:
   Clock samples:            32
   Threads:                  64
   Thread switches:         128
   Mutexes:                  32
   Mailboxes:                32
   Semaphores:               32
   Scheduler operations:    128
   Counters:                 32
   Flags:                    32
   Alarms:                   32
   Stack Size:             1088


                                 Confidence
     Ave     Min     Max     Var  Ave  Min  Function
  ======  ======  ======  ====== ========== ========
INFO:<Ctrl-C disabled until test completion>
    0.92    0.00    2.00    0.23   82%  12% Create thread
    0.17    0.00    1.00    0.28   82%  82% Yield thread [all suspended]
    0.20    0.00    1.00    0.32   79%  79% Suspend [suspended] thread
    0.16    0.00    1.00    0.26   84%  84% Resume thread
    0.23    0.00    1.00    0.36   76%  76% Set priority
    0.13    0.00    1.00    0.22   87%  87% Get priority
    0.30    0.00    1.00    0.42   70%  70% Kill [suspended] thread
    0.25    0.00    1.00    0.38   75%  75% Yield [no other] thread
    0.23    0.00    1.00    0.36   76%  76% Resume [suspended low prio] thread
    0.16    0.00    1.00    0.26   84%  84% Resume [runnable low prio] thread
    0.22    0.00    1.00    0.34   78%  78% Suspend [runnable] thread
    0.19    0.00    1.00    0.30   81%  81% Yield [only low prio] thread
    0.17    0.00    1.00    0.28   82%  82% Suspend [runnable->not runnable]
    0.33    0.00    1.00    0.44   67%  67% Kill [runnable] thread
    0.34    0.00    1.00    0.45   65%  65% Destroy [dead] thread
    0.50    0.00    1.00    0.50  100%  50% Destroy [runnable] thread
    0.63    0.00    1.00    0.47   62%  37% Resume [high priority] thread
    0.34    0.00    1.00    0.45   66%  66% Thread switch

    0.09    0.00    1.00    0.16   91%  91% Scheduler lock
    0.17    0.00    1.00    0.28   82%  82% Scheduler unlock [0 threads]
    0.17    0.00    1.00    0.28   82%  82% Scheduler unlock [1 suspended]
    0.17    0.00    1.00    0.28   82%  82% Scheduler unlock [many suspended]
    0.15    0.00    1.00    0.25   85%  85% Scheduler unlock [many low prio]

    0.06    0.00    1.00    0.12   93%  93% Init mutex
    0.16    0.00    1.00    0.26   84%  84% Lock [unlocked] mutex
    0.19    0.00    1.00    0.30   81%  81% Unlock [locked] mutex
    0.16    0.00    1.00    0.26   84%  84% Trylock [unlocked] mutex
    0.00    0.00    0.00    0.00  100% 100% Trylock [locked] mutex
    0.06    0.00    1.00    0.12   93%  93% Destroy mutex
    0.03    0.00    1.00    0.06   96%  96% Unlock/Lock mutex

    0.19    0.00    1.00    0.30   81%  81% Create mbox
    0.13    0.00    1.00    0.22   87%  87% Peek [empty] mbox
    0.19    0.00    1.00    0.30   81%  81% Put [first] mbox
    0.06    0.00    1.00    0.12   93%  93% Peek [1 msg] mbox
    0.22    0.00    1.00    0.34   78%  78% Put [second] mbox
    0.06    0.00    1.00    0.12   93%  93% Peek [2 msgs] mbox
    0.16    0.00    1.00    0.26   84%  84% Get [first] mbox
    0.06    0.00    1.00    0.12   93%  93% Get [second] mbox
    0.16    0.00    1.00    0.26   84%  84% Tryput [first] mbox
    0.19    0.00    1.00    0.30   81%  81% Peek item [non-empty] mbox
    0.16    0.00    1.00    0.26   84%  84% Tryget [non-empty] mbox
    0.22    0.00    1.00    0.34   78%  78% Peek item [empty] mbox
    0.16    0.00    1.00    0.26   84%  84% Tryget [empty] mbox
    0.09    0.00    1.00    0.17   90%  90% Waiting to get mbox
    0.09    0.00    1.00    0.17   90%  90% Waiting to put mbox
    0.06    0.00    1.00    0.12   93%  93% Delete mbox
    0.00    0.00    0.00    0.00  100% 100% Put/Get mbox

    0.09    0.00    1.00    0.17   90%  90% Init semaphore
    0.19    0.00    1.00    0.30   81%  81% Post [0] semaphore
    0.16    0.00    1.00    0.26   84%  84% Wait [1] semaphore
    0.06    0.00    1.00    0.12   93%  93% Trywait [0] semaphore
    0.16    0.00    1.00    0.26   84%  84% Trywait [1] semaphore
    0.13    0.00    1.00    0.22   87%  87% Peek semaphore
    0.13    0.00    1.00    0.22   87%  87% Destroy semaphore
    0.00    0.00    0.00    0.00  100% 100% Post/Wait semaphore

    0.13    0.00    1.00    0.22   87%  87% Create counter
    0.09    0.00    1.00    0.17   90%  90% Get counter value
    0.06    0.00    1.00    0.12   93%  93% Set counter value
    0.19    0.00    1.00    0.30   81%  81% Tick counter
    0.06    0.00    1.00    0.12   93%  93% Delete counter

    0.09    0.00    1.00    0.17   90%  90% Init flag
    0.19    0.00    1.00    0.30   81%  81% Destroy flag
    0.13    0.00    1.00    0.22   87%  87% Mask bits in flag
    0.19    0.00    1.00    0.30   81%  81% Set bits in flag [no waiters]
    0.22    0.00    1.00    0.34   78%  78% Wait for flag [AND]
    0.19    0.00    1.00    0.30   81%  81% Wait for flag [OR]
    0.16    0.00    1.00    0.26   84%  84% Wait for flag [AND/CLR]
    0.19    0.00    1.00    0.30   81%  81% Wait for flag [OR/CLR]
    0.06    0.00    1.00    0.12   93%  93% Peek on flag

    0.13    0.00    1.00    0.22   87%  87% Create alarm
    0.22    0.00    1.00    0.34   78%  78% Initialize alarm
    0.16    0.00    1.00    0.26   84%  84% Disable alarm
    0.19    0.00    1.00    0.30   81%  81% Enable alarm
    0.09    0.00    1.00    0.17   90%  90% Delete alarm
    0.16    0.00    1.00    0.26   84%  84% Tick counter [1 alarm]
    0.66    0.00    1.00    0.45   65%  34% Tick counter [many alarms]
    0.31    0.00    1.00    0.43   68%  68% Tick & fire counter [1 alarm]
    3.00    3.00    3.00    0.00  100% 100% Tick & fire counters [>1 together]
    0.78    0.00    1.00    0.34   78%  21% Tick & fire counters [>1 separately]
    0.00    0.00    0.00    0.00  100% 100% Alarm latency [0 threads]
    0.43    0.00    1.00    0.49   57%  57% Alarm latency [2 threads]
    0.63    0.00    1.00    0.47   62%  37% Alarm latency [many threads]
    1.00    1.00    1.00    0.00  100% 100% Alarm -> thread resume latency

    0.00    0.00    0.00    0.00            Clock/interrupt latency

    1.00    1.00    1.00    0.00            Clock DSR latency

     233     172     288                    Worker thread stack used (stack size 1088)
```

```
            All done, main thrd : stack used  1204 size  2304
            All done :  Interrupt stack used   420 size  4096
            All done : Idlethread stack used   248 size  1280

Timing complete - 29810 ms total

PASS:<Basic timing OK>
EXIT:<done>
```

Other Issues

The platform HAL does not affect the implementation of other parts of the eCos HAL specification. The HPS processor HAL and the ARM architectural HAL documentation should be consulted for further details.

Chapter 277. Atmel ATSAMA5D3 Variant HAL

Name

`CYGPKG_HAL_ARM_CORTEXA_SAMA5D3` — eCos Support for the Atmel SAMA5D3 Microprocessor Family

Description

This document covers the configuration and usage of the Hardware Abstraction Layer (HAL) for the Atmel SAMA5D3 series of Cortex-A5 microcontrollers. It is expected to be read in conjunction with platform HAL-specific documentation, as well as the eCos HAL specification. This processor package complements the ARM architectural HAL, Cortex-A variant HAL and the platform HAL. It provides functionality common to all SAMA5D3-based implementations.

This support is found in the eCos package located at `packages/hal/arm/cortexa/sama5d3/var` within the eCos source repository.

The SAMA5D3 HAL package is loaded automatically when eCos is configured for a SAMA5D3-based platform. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

The Cortex-A5 CPU uses the common ARM Cortex-A package which provides run-time support for cache, MMU, etc. The Cortex-A5 has full application compatability with Cortex-A8, Cortex-A9, and Cortex-A15 processors, but in a smaller, lower power, package.

In addition to the core CPU support, the SAMA5D3 processor variant package provides common functionality and definitions that SAMA5D3 based platforms may require, as well as definitions useful to application developers.

The platform HAL documentation Setup section gives an overview of hardware debugging support, e.g. PEEDI, J-Link, etc.

Name

SAMA5D3 hardware definitions — Details on obtaining hardware definitions for SAMA5D3

Register definitions

The file `<cyg/hal/sama5d3.h>` can be included from application and eCos package sources to provide definitions related to SAMA5D3 subsystems. These include register definitions for the interrupt controller, power management controller, clock generator, memory controller, external bus interface, GPIO, USART, and other subsystems.

Initialization helper macros

The file `<cyg/hal/sama5d3_init.inc>` contains definitions of helper macros which may be used by SAMA5D3 platform HALs in order to initialise common subsystems without excessive duplication between the platform HALs. Typically this file will be included by the `hal_platform_setup.h` header in the platform HAL, in turn included from the architectural HAL file `vectors.S`.

This file is solely intended to be used by platform HALs. At the same time, it is only present to assist initialization, and platform HALs are not obliged to use it if their startup requirements vary.

Name

Bootstrap — System startup

Overview

The BMS signal to the CPU controls the initial system bootstrap selection. This is used in conjunction with the OTP (FUSE) on-chip configuration, and reset “input” pin state, to select the boot source for the CPU.

Table 277.1. BMS signal

BMS signal	Description
BMS_BIT=0	The embedded (on-chip) `RomBOOT` first-level bootloader is used. The actual boot source used will depend on the presence of suitable binaries on the sources scanned by the `RomBOOT` code. The first acceptable binary is loaded into SRAM and executed. If no valid non-volatile-memory (NVM) binary is found then the `RomBOOT` will enter the Atmel SAM-BA monitor.
BMS_BIT=1	The application installed on the EBI_CS0 device is executed in-situ. The on-chip `RomBOOT` configures the chip-select for 12MHz RC access prior to executing the code mapped from address 0x00000000.

The “Standard Boot Strategies” section of the SAMA5D3 Series Datasheet provides details about the boot sequence configuration, and should be read in conjunction with this documentation.

The hardware platform documentation should be read in conjunction with this generic SAMA5D3 CPU documentation with regards to specific jumper settings that may affect the bootstrap code executed.

For some designs directly booting from an `EBI_CS0` memory-mapped device is the simplest option. However, making use of the SAMA5D3 `RomBOOT` world to load a small second-level boot loader allows the easier possibility of providing support for in-field upgrades and selection of multiple application images.

Note

When `BMS_BIT=1` and code is started from the `EBI_CS0` device then the Atmel Secure Boot functionality is not available.

Name

On-chip Subsystems and Peripherals — Hardware Support

Hardware support

On-chip Memory

The Atmel SAMA5D3 parts include 128K of on-chip SRAM for general application use. Other SRAM memory exists as part of specific on-chip I/O controllers and may be available for general purpose use if the corresponding I/O is not being used (e.g. NFC SRAM), though eCos by default does not provide explicit support for such use.

The SAMA5D3 parts also provide I/O controllers which allow access to various external memory types, which eCos may use where supported by the relevant platform HAL. Application execution is normally based on such external memory due to the limited size of the on-chip SRAM available.

Interrupts

The SAMA5D3 HAL provides interrupt support via the on-chip Advanced Interrupt Controller.

Interrupt controller definitions

The file `<cyg/hal/sama5d3_ints.h>` (located at `hal/arm/cortexa/sama5d3/var/`*`VERSION`*`/include/sama5d3_ints.h` in the eCos source repository) contains interrupt vector number definitions for use with the eCos kernel and driver interrupt APIs.

It should be noted that further decoding is performed on the multiplexed system `CYGNUM_HAL_INTERRUPT_SYS` interrupt to identify the cause more specifically. Note that as a result, placing an interrupt handler on the `CYGNUM_HAL_INTERRUPT_SYS` interrupt will not work as expected. Conversely, masking a decoded derivative of the `CYGNUM_HAL_INTERRUPT_SYS` interrupt will not work as this would mask other `CYGNUM_HAL_INTERRUPT_SYS` interrupts, but masking the `CYGNUM_HAL_INTERRUPT_SYS` interrupt itself will work. On the other hand, unmasking a decoded `CYGNUM_HAL_INTERRUPT_SYS` interrupt *will* unmask the `CYGNUM_HAL_INTERRUPT_SYS` interrupt as a whole, thus unmasking interrupts for the other units on this shared interrupt.

If the CDL option `CYGHWR_HAL_ARM_CORTEXA_SAMA5D3_PIO_DEMUX` is configured then the variant HAL also provides support for de-multiplexing PIO interrupt sources, allowing the standard eCos interrupt system to be used to control the use of individual PIO pin interrupt handlers. This support avoids the need for the developer to manually handle multiple active interrupt PIO pins on a single controller vector ISR function, and as such the feature is normally recommended. However for very small memory footprint systems the RAM overhead of maintaining a significantly increased set of ISR descriptor vectors may be deemed inappropriate, and so the developer is free to disable the extension and manually support the base per-controller `CYGNUM_HAL_INTERRUPT_PIO#` shared interrupt source as required for their PIO interrupt requirements.

Note

If the variant demultiplexing support is disabled then certain standard drivers may have restricted functionality on specific platforms if they depend on using the per-pin interrupt support for certain features.

The list of interrupt vectors may be augmented on a per-platform basis. Consult the platform HAL documentation for your platform for whether this is the case.

Interrupt controller Functions

The source file `src/sama5d3_misc.c` within this package provides most of the support functions to manipulate the interrupt controller. The `hal_IRQ_handler` queries the IRQ status register to determine the interrupt cause. Functions `hal_interrupt_mask` and `hal_interrupt_unmask` enable or disable interrupts within the interrupt controller.

Interrupts are configured in the hal_interrupt_configure function, where the level and up arguments are interpreted as follows:

level	up	interrupt on
0	0	Falling Edge
0	1	Rising Edge
1	0	Low Level
1	1	High Level

To fit into the eCos interrupt model, interrupts essentially must be acknowledged immediately once decoded. The hal_interrupt_acknowledge function explicitly acknowledges the PIO controller interrupt sources.

The hal_interrupt_set_level is used to set the priority level of the supplied interrupt within the Advanced Interrupt Controller.

Note that in all the above, it is not recommended to call the described functions directly. Instead either the HAL macros (HAL_INTERRUPT_MASK et al) or preferably the kernel or driver APIs should be used to control interrupts.

Using the Advanced Interrupt Controller for VSRs

The SAMA5D3 HAL has been designed to exploit benefits of the on-chip Advanced Interrupt Controller (AIC) on the SAMA5D3. Support has been included for exploiting its ability to provide hardware vectoring for VSR interrupt handlers.

The interrupt decoding path has been optimised by allowing the AIC to be interrogated for the interrupt handler VSR to use. These vectored interrupts are by default still configured to point to the default ARM architecture HAL IRQ and FIQ VSRs. However applications may set their own VSRs to override this default behaviour to allow optimised interrupt handling.

The VSR vector numbers to use when overriding are also defined in the <cyg/hal/sama5d3_ints.h> header. Consult the kernel and generic HAL documentation for more information on VSRs and how to set them.

Interrupt handling withing standalone applications

For non-eCos standalone applications running under RedBoot, it is possible to install an interrupt handler into the interrupt vector table manually. Memory mappings are platform-dependent and so the platform documentation should be consulted, but in general the address of the interrupt table can be determined by analyzing RedBoot's symbol table, and searching for the address of the symbol name hal_interrupt_handlers. Table slots correspond to the interrupt numbers as detailed above. Pointers inserted in this table should be pointers to a C/C++ function with the following prototype:

```
extern unsigned int isr( unsigned int vector, unsigned int data );
```

For non-eCos applications run from RedBoot, the return value can be ignored. The vector argument will also be the interrupt vector number. The data argument is extracted from a corresponding table named hal_interrupt_data which immediately follows the interrupt vector table. It is still the responsibility of the application to enable and configure the interrupt source appropriately if needed.

Periodic Interval Timer

The eCos kernel system clock is implemented using the Periodic Interval Timer (PIT) controller. By default, the system clock interrupts once every 10ms, corresponding to a 100Hz clock. This can be changed by the configuration option CYGNUM_HAL_RTC_DENOMINATOR which corresponds to the clock frequency. Other clock-related settings are recalculated automatically if the denominator is changed. If the desired frequency cannot be expressed accurately solely with changes to CYGNUM_HAL_RTC_DENOMINATOR, then the configuration option CYGNUM_HAL_RTC_NUMERATOR may also be adjusted, and again clock-related settings will automatically be recalculated.

The PIT is also used to implement the HAL microsecond delay function, HAL_DELAY_US. This is used by some device drivers, and in non-kernel configurations such as with RedBoot where this timer is needed for loading program images via X/Y-modem protocols and debugging via TCP/IP. Standalone applications which require RedBoot services, such as debugging, should avoid use of this timer.

GPIO

The variant HAL provides support for packaging the configuration of a GPIO line into a single 32-bit descriptor that can then be used with macros to configure the pin and set and read its value. Details are supplied later.

RTC/Wallclock

eCos includes RTC (known in eCos as a wallclock) device drivers for the on-chip RTC in the SAMA5D3 family. This support is located in the package `CYGPKG_DEVICES_WALLCLOCK_ARM_AT91` ("AT91 wallclock driver"). Normally this package is included automatically by the relevant platform HAL.

Profiling Support

The SAMA5D3 HAL contains support for **gprof**-based profiling using a sampling timer. The default timer used is channel 0 of Timer 0 (`CYGHWR_HAL_SAMA5D3_TC0`). This timer is only enabled when the gprof profiling package (`CYGPKG_PROFILE_GPROF`) is included and enabled in the eCos configuration, otherwise it remains available for application use.

Not all SAMA5D3 variants have multiple timer blocks. For example, when targetting the SAMA5D31 variant, only `TC0` is available and so profiling support may not be possible if the application requires the use of that timer block.

Serial I/O

The SAMA5D3 variant HAL supports basic polled HAL diagnostic I/O over any of the on-chip serial devices. There is also a fully interrupt-driven serial device driver suitable for eCos applications for all on-chip serial devices. The serial driver consists of an eCos package: `CYGPKG_IO_SERIAL_ARM_AT91` which provides all support for the SAMA5D3 on-chip serial devices. Using the HAL diagnostic I/O support, any of these devices can be used by the ROM monitor or RedBoot for communication with GDB. If a device is needed by the application, either directly or via the serial driver, then it cannot also be used for GDB communication using the HAL I/O support. An alternative serial port should be used instead.

The HAL defines CDL interfaces for each of the available UARTs, `CYGINT_HAL_ARM_CORTEXA_SAMA5D3_DBGU`. `CYGINT_HAL_ARM_CORTEXA_SAMA5D3_USART0` to `CYGINT_HAL_ARM_CORTEXA_SAMA5D3_USART3`, and (if available for the target CPU) `CYGINT_HAL_ARM_CORTEXA_SAMA5D3_UART0` and `CYGINT_HAL_ARM_CORTEXA_SAMA5D3_UART1`. The platform HAL CDL should contain an **implements** directive for each such UART that is available for use on the board. This will enable use of the UART for diagnostic use.

The SAMA5D3 UARTs provide TX and RX data lines plus hardware flow control using RTS/CTS for those UARTs that have them available and connected.

SPI

The platform HAL provides definitions to allow access to devices on the SPI buses. The HAL provides information to the general AT91 SPI driver (`CYGPKG_DEVS_SPI_ARM_AT91`) which in turn provides the underlying implementation for the SPI API layer in the `CYGPKG_IO_SPI` package. All these packages are automatically loaded when configuring for the board.

The SAMA5D3 variant implements the SPI buses as configured for the platform:

- `cyg_spi_at91_bus0`
- `cyg_spi_at91_bus1`

The platform specific HAL may implement specific SPI device instances as relevant for the underlying hardware. e.g. SPI Dataflash devices.

Two-Wire Interface (TWI)

The CYGPKG_DEVS_I2C_ATMEL_TWI package implements a Two-Wire Interface (TWI) driver for the TWI controllers present on the SAMA5D3 family of devices. This type of bus is also known as I²C®. The generic API for this driver may be found within the CYGPKG_IO_I2C package, along with the API documentation. The CYGPKG_DEVS_I2C_ATMEL_TWI hardware package is normally included by default when configuring a SAMA5D3 platfornm, and so does not need to be manually added to a configuration.

Support for TWI functionality is controlled within the SAMA5D3 variant HAL via the CYGPKG_HAL_ARM_CORTEXA_SAMA5D3_I2C CDL component. The component controls, via sub-options, whether TWI driver support is enabled for the different on-chip TWI buses available.

For each individual bus that is available the driver package itself provides the CYGNUM_DEVS_I2C_ATMEL_TWI*x*_CLOCK option to configure the TWI bus clock speed, where *x* is replaced by the relevant bus number.

The driver package implements the necessary I²C bus instances as appropriate:

- hal_atmel_i2c_bus0
- hal_atmel_i2c_bus1
- hal_atmel_i2c_bus2

The platform or application code can then register devices attached tospecific buses as needed.

MCI (MMC/SD card controller)

If a suitable socket or sockets exist, the platform HAL may provide definitions to allow use of MMC or SD cards accessed using the on-chip Multimedia Card Interface (MCI) peripheral block. The SAMA5D3 processor has support for up to three independent high speed MCI (HSMCI) controllers; although at present in eCos, only one may be configured for use, fixed at configuration time.

The device driver itself is found in the separate package CYGPKG_DEVS_MMCSD_ATMEL_SAM_MCI, which is loaded automatically when selecting any platform supporting it, and further documentation can be found in that package. Typically, disk support (CYGPKG_IO_DISK), generic filesystem support (CYGPKG_IO_FILEIO) and a filesystem such as FAT (CYGPKG_FS_FAT along with its prerequisite CYGPKG_LINUX_COMPAT) are then added to allow application use.

Features such as card insertion/removal detection, SDIO or write protection detection are hardware and platform specific, and platform documentation should be consulted for more information on those features.

USB

The platform HAL provides definitions to the general USB controller driver (CYGPKG_DEVS_USB_AT91) which in turn provides the underlying implementation for the USB API layer (CYGPKG_IO_USB).

The driver layer supports both OHCI host functionality via the CYGPKG_DEVS_USB_OHCI package, allowing peripheral devices to be attached to an eCos “host”, and for device functionality via the CYGPKG_DEVS_USB_PCD_UDPHS package, where the eCos application implements the peripheral device support for attaching to an external host.

All the necessary hardware packages are automatically loaded when configuring for the board. However, the top-level CYGPKG_IO_USB package needs to be included and configured when USB functionality is required.

OHCI

The CYGPKG_DEVS_USB_OHCI package implements the generic parts of the OHCI (host controller) support, and in conjunction with the platform specific driver and the CYGPKG_IO_USB package allows eCos to act as a “host” for attached USB devices.

The eCos USB host stack includes a number of class drivers and the ability for users to write additional ones using the eCos USB host API. See the USB chapter for further details, including a complete listing of supported classes.

Device

The CYGPKG_DEVS_USB_PCD_UDPHS package implements the generic parts of the Atmel UDPHS (USB High Speed Device Port) peripheral controller support, and in conjunction with the CYGPKG_IO_USB package provides support for implementing USB device/peripheral class drivers.

The USB chapter provides target mode stack details including a list of supported device/peripheral class drivers and information related to adding new class drivers.

Clock Control

Depending on how an eCos SAMA5D3 application is started will influence the CPU and peripheral clock frequencies used for application execution. For ROM startup, or SRAM when started by the on-chip RomBOOT, the eCos configuration supplied CYGHWR_HAL_ARM_CORTEXA_SAMA5D3_CLOCK_PLL_DIVA and CYGHWR_HAL_ARM_CORTEXA_SAMA5D3_CLOCK_PLL_MULA options, in conjunction with the platform supplied CYGHWR_HAL_ARM_CORTEXA_SAMA5D3_OSC_MAIN value, will be used to set the CPU and I/O clock frequencies. For the ROMRAM and RAM startup types the clock frequenies in effect when the application is loaded (either via another application, debug monitor of hardware debugger) are used. As such, the SAMA5D3 variant HAL provides access to variables that hold the currently configured clock frequencies:

```
cyg_uint32 hal_sama5d3_slck;   // SLCK
cyg_uint32 hal_sama5d3_mainck; // MAINCK
cyg_uint32 hal_sama5d3_pllack; // PLLA frequency
cyg_uint32 hal_sama5d3_upllck; // UPLL frequency
cyg_uint32 hal_sama5d3_pclk;   // Processor clock
cyg_uint32 hal_sama5d3_mclk;   // Main peripheral clock
```

It is not expected that applications will need to interpret or use the values, but the HAL makes use of the values to ensure valid clock configurations are used.

Name

GPIO Support on SAMA5D3 processors — Details

Synopsis

#include <cyg/hal/hal_io.h>

cyg_uint32 pin = **CYGHWR_HAL_SAMA5D3_PIN**(*port*, *bit*, *mode*, *md*, *pupd*, *if*, *int*, *conf*);

cyg_uint32 pin = **CYGHWR_HAL_SAMA5D3_PIN_OUT**(*port*, *bit*, *md*, *pupd*);

cyg_uint32 pin = **CYGHWR_HAL_SAMA5D3_PIN_IN**(*port*, *bit*, *md*, *pupd*, *if*, *int*);

CYGHWR_HAL_SAMA5D3_PIN_SET (*pin*);

CYGHWR_HAL_SAMA5D3_GPIO_OUT (*pin*, *val*);

CYGHWR_HAL_SAMA5D3_GPIO_IN (*pin*, *val*);

Description

The SAMA5D3 HAL provides a number of macros to support the encoding of the GPIO pin identity and I/O configuration into a single 32-bit descriptor. This is useful to drivers and other packages that need to configure and use different lines for different devices.

A descriptor is created with one of the 3 variants depending on how the pin is to be used. The support is implemented by the CYGHWR_HAL_SAMA5D3_PIN macro, with CYGHWR_HAL_SAMA5D3_PIN_IN and CYGHWR_HAL_SAMA5D3_PIN_OUT being shorthand helpers when direct control of a pin is required: CYGHWR_HAL_SAMA5D3_PIN_IN defines the pin as an input whose value can be accessed by the user using the macro CYGHWR_HAL_SAMA5D3_GPIO_IN (see later), CYGHWR_HAL_SAMA5D3_PIN_OUT defines the pin as an output where the user can set the pin output value with the macro CYGHWR_HAL_SAMA5D3_GPIO_OUT (see later).

The CYGHWR_HAL_SAMA5D3_PIN macro can be used when defining a pin that will be controlled by an on-chip peripheral.

Note

The HAL supplied header file sama5d3.h provides existing configuration definitions for the majority of the on-chip peripherals supported by eCos, thus obviating the need for the developer to provide their own pin definitions.

The macro variants take a subset of arguments from the following list:

port
: This identifies the PIO controller to which the pin is attached. Ports are identified by letters from A to E.

bit
: This gives the bit, or pin number, within the controller port. These are numbered from 0 to 31.

mode
: This parameter indicates whether the pin is controlled by an on-chip peripheral, or is to be used as a GPIO pin under application control.

Table 277.2. Pin Mode

mode	Details
GPIOIN	The pin is to be configured as an INPUT, and after configuration the CYGHWR_HAL_SAMA5D3_GPIO_IN macro can be used to ascertain the pin state.

mode	Details
GPIOOUT	The pin is to be configured as an OUTPUT, and after configuration the CYGHWR_HAL_SAMA5D3_GPIO_OUT macro can be used to drive the pin level.
PER_A, PER_B, PER_C, PER_D	The required peripheral mapping when the pin is to be assigned to an on-chip peripheral. The multiplexing of peripheral signals is defined by the CPU variant being targetted, and is beyond the scope of this documentation. When creating pin configurations for on-chip peripherals the relevant Atmel datasheet or technical reference manual should be consulted.

md
: This setting indicates whether the pin should be driven in open-drain mode (OPENDRAIN). If the pin is not to be configured as OPENDRAIN this value is unused, but for clarity can be given the setting NA.

pupd
: If this is an input pin, or an output pin configured in open-drain mode (whether controlled by GPIO or a peripheral), this setting can be used to indicate whether a weak pull-up resistor (PU) is used, or a weak pull-down resistor (PD) is used. If neither are to be used, then a value of NONE can be given.

if
: For input pins (GPIO or peripheral) a glitch (GLITCH) or debouncing (DEBOUNCE) filter can be configured for the pin. When no input filtering is required, or when the field is not relevant due to the other pin configuration fields, the value NONE can be specified.

int
: This parameter indicates whether the pin should have an interrupt configuration defined.

Table 277.3. Interrupt Type

int	Details
EDGE_ANY	When an interrupt event should be raised on a pin edge event (rising or falling).
EDGE_RISE	An interrupt should only be raised on rising (LOW->HIGH) edge transitions.
EDGE_FALL	An interrupt should only be raised on falling (HIGH->LOW) edge transitions.
LEVEL_HIGH	Interrupts should be asserted when the pin is at a HIGH level.
LEVEL_LOW	Interrupts should be asserted when the pin is at a LOW level.
NA	This value can be used when an interrupt configuration is not required, or not applicable due to the other pin configuration parameters.

conf
: This parameter provides a simple “extension” mechanism; and is treated as a 32-bit binary value that is OR-ed into the pin descriptor. Care must be taken to ensure that existing bitfields within the binary descriptor are not corrupted.

The following examples show how these macros may be used:

```
// Define port B pin 28 as being controlled by peripheral multiplex A,
// which for this pin on SAMA5D3 devices is USART1, without any
// pull-ups/pull-downs:
#define CYGHWR_HAL_SAMA5D3_USART1_RXD CYGHWR_HAL_SAMA5D3_PIN(B,28,PER_A,OPENDRAIN,NONE,NONE,NA,(0))

// Define port E pin 24 as a GPIO output with a pull-down, for an
// active-low LED:
#define CYGHWR_HAL_SAMA5D3_LED_RED CYGHWR_HAL_SAMA5D3_PIN_OUT(E,24,NA,PD)
```

Additionally, the manifest CYGHWR_HAL_SAMA5D3_PIN_NONE may be used in place of a pin descriptor and has a value that no valid descriptor can take. It may therefore be used as a placeholder where no GPIO pin is present or to be used. This can be useful when defining pin configurations for a series of instances of a peripheral (e.g. USART ports), but where not all instances support all the same pins (e.g. hardware flow control lines).

The remaining macros all take a suitably constructed GPIO pin descriptor as an argument. The CYGHWR_HAL_SAMA5D3_PIN_SET macro configures the pin according to the descriptor and must be called before any other macros. CYGHWR_HAL_SAMA5D3_GPIO_OUT sets the output to the value of the least significant bit of the *val* argument. The *val* argument of CYGHWR_HAL_SAMA5D3_GPIO_IN should be a pointer to an int, which will be set to 0 if the pin input is zero, and 1 otherwise.

Further helper macros are available, and it is recommended to consult the header file <cyg/hal/sama5d3.h> (also present in the include subdirectory of the SAMA5D3 variant HAL package within the eCos source repository), for the complete list if needed.

The Interrupt controller definitions section provides an overview of variant support for demultiplexing PIO interrupts.

Name

Peripheral clock control — Details

Synopsis

#include <cyg/hal/hal_io.h>

`cyg_uint32` **`CYGHWR_HAL_ATMEL_CLOCK_ENABLE`**`(`*`pid`*`);`

`CYGHWR_HAL_ATMEL_CLOCK_DISABLE` `(`*`pid`*`);`

Description

The HAL provides macros which may be used to enable or disable peripheral clocks. Effectively this indicates whether the peripheral is powered on (enabled) or powered down (disabled), and so may be used to ensure unused peripherals are turned off to save power. The `CYGHWR_HAL_ATMEL_CLOCK_ENABLE` macro will enforce the maximum frequency limitations for particular peripheral blocks, and will return the frequency of the clock used for the enabled periperhal. Such frequency information may be useful to device drivers if clock divider configuration is required.

It is important to remember that before a peripheral can be used, it must be enabled. It is safe to re-enable a peripheral that is already enabled, although usually a device driver will only do so once in its initialisation. eCos will automatically initialise some peripheral blocks where it needs to use the associated peripherals (such as memory controllers and some (but usually not all) PIO banks), and in eCos-supplied device drivers which are included in the eCos configuration. However this should not be relied on - it is always safest to enable the peripheral clocks anyway just in case. Finally, remember that each PIO bank must be enabled separately.

Each peripheral has a unique ID defined by the HAL, and these values are used as the *`pid`* parameter to the enable and disable macros.

Name

DMA Support — Details

Description

The package `CYGPKG_DEVS_DMA_ATMEL_DMAC` implements the DMA support for the eCos run-time, and the documentation for that package should be referenced as appropriate.

The DMAC package is automatically loaded when eCos is configured for a SAMA5D3-based platform. It should never be necessary to load this package explicitly.

Name

Configuration — Common SAMA5D3 configuration options

Overview

Startup

The SAMA5D3 variant HAL package supports four separate startup types for application configuration:

ROMRAM

> This startup type is for *stand-alone* applications that are loaded into external DDR2-SDRAM via a second-level boot loader (e.g. the default SAMA5D3x-CM based AT91BootLoader, or the eCosPro `BootUp` loader), or via a JTAG/SWD hardware debugger. The data and BSS areas will be put into suitable memory locations as defined by the relevant platform HAL linker script. The application will be self-contained with no dependencies on services provided by other software. The eCos startup code will perform all necessary hardware initialization.
>
> ### Note
>
> The application is copied from its persistent location by the second-level application bootloader, or is directly loaded to RAM via a JTAG/SWD hardware debugger. The `ROMRAM` startup code does not perform the copy to RAM itself.

SRAM

> This startup type is used for *stand-alone* applications. The application will be loaded from a suitable supported “bootable” device by the on-chip first-level (`RomBOOT`) bootloader, or directly via a hardware (JTAG/SWD) debugger. The application will be self-contained with no dependencies on services provided by other software. It is not envisaged that the `SRAM` startup type will normally be used for full applications, primarily being used for developing `RomBOOT` loaded second-level bootloaders. However, it can be useful for small platform verification tests loaded via JTAG.

ROM

> This startup type is used for *stand-alone* applications which will be programmed into bootable memory located in the EBI_CS0 space from 0x10000000. The data and BSS areas will be put into suitable memory as defined by the relevant platform HAL linker script. The application will be self-contained with no dependencies on services provided by other software. The program expects to boot from reset with the EBI_CS0 space mapped from location zero. It will then transfer control to the 0x10000000 region. The eCos startup code will perform all necessary hardware initialization.

RAM

> This startup type is used for RAM based applications that are dependent on a “debug” monitor (e.g. RedBoot or GDB stubs) for some run-time services. By default the application will use the eCos virtual vectors mechanism to obtain services from the monitor, including diagnostic output. Normally arm-eabi-gdb is used to load a `RAM` startup application into memory, and then to debug it. It is assumed that the hardware has already been initialized by the relevant debug monitor.
>
> This startup type can be useful during development, since it allows debugging applications where a hardware debug solution is not available.

RedBoot and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when Building RedBoot.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option CYGSEM_HAL_USE_ROM_MONITOR should be set. By default this option is enabled when building for a RAM startup, and disabled otherwise.

If the application does not rely on a ROM monitor for diagnostic services then, by default, the on-chip DBGU serial port will be used for HAL diagnostic output.

RedBoot Location

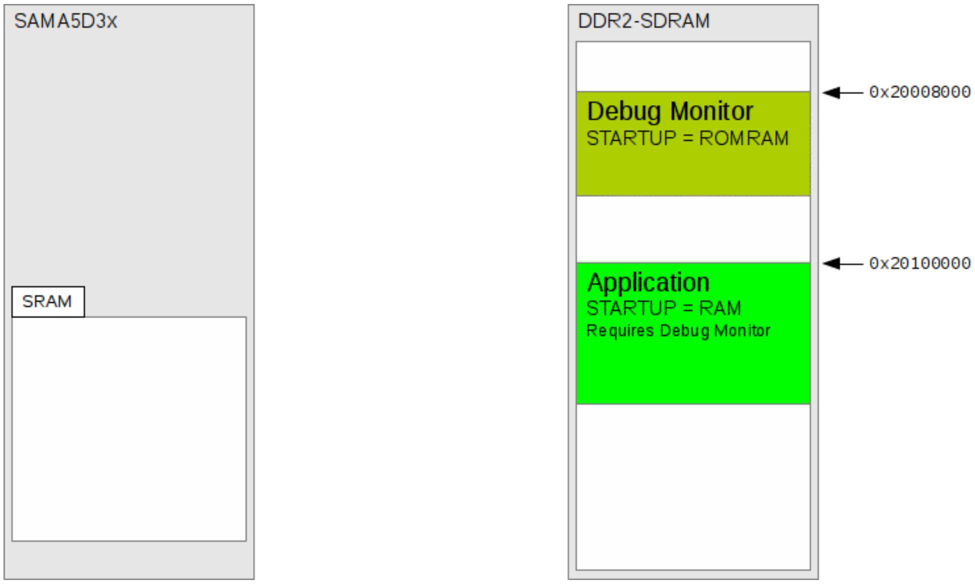
RedBoot can be used in conjunction with the host debugger arm-eabi-gdb to load and debug applications configured for the RAM startup type. Depending on the product requirements the RedBoot binary can either be executed in-situ from the NOR flash, or loaded dynamically into the DDR2-SDRAM for execution.

The ROMRAM RedBoot diagram shows the memory layout for a RAM resident RedBoot. The RedBoot binary is loaded into its final DDR2-SDRAM location either via a second-level boot loader (e.g. BootUp) or directly using a hardware debugger.

Note

The addresses shown in the following diagrams are based on the default linker script values.

Figure 277.1. ROMRAM RedBoot

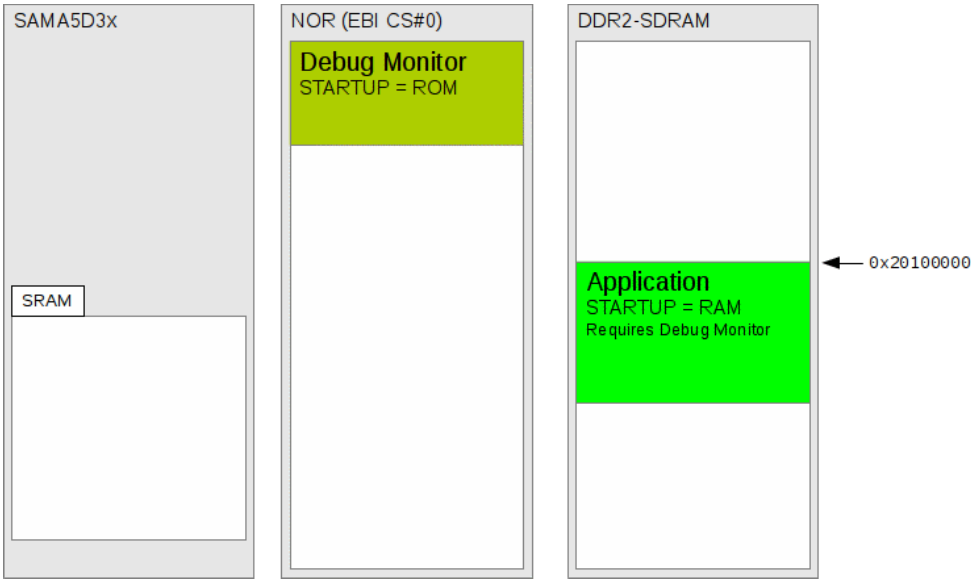


An illustration of a ROMRAM RedBoot executing from DDR2-SDRAM

Any application can be configured for the ROM startup type, and when stored from offset 0 in the NOR flash (mapped to address 0x10000000) the CPU can be configured (via BMS_BIT=1) to execute the code upon reset. The application does not need to be a debug monitor, but can be the final application if required.

The ROM RedBoot diagram shows the memory layout for a NOR flash resident RedBoot.

Figure 277.2. ROM RedBoot



An illustration of a ROM RedBoot executing from NOR flash

Name

Test Programs — Details

Test Programs

The SAMA5D3 variant HAL contains a test program which allows various aspects of the microcontroller to be tested.

varinfo

The simple `varinfo` test provides some output identifying the CPU, and the system configuration. It is designed as a very basic “Hello World” test to validate an eCos run-time build.

Chapter 278. Atmel SAMA5D3x-MB (MotherBoard) Platform HAL

Name

CYGPKG_HAL_ARM_CORTEXA_SAMA5D3X_MB — eCos Support for the SAMA5D3x-MB Board

Description

This document covers the configuration and usage of eCos and RedBoot on the Atmel SAMA53x-MB based evaluation kits. This motherboard is fitted with a SAMA5D3x-CM CPU module containing a variant of the SAMA5D3 family of microcontrollers. The board is referred to in this document using SAMA5D3x-MB. The eCos configuration system uses explicit target names to identify the available CPU evaluation kits based on this board, e.g. atsama5d31_ek, atsama5d33_ek, etc.

The CPU Module contains the available memories, as well as some I/O functionality to extend the features available on the base SAMA5D3x-MB motherboard. The CPU Module is connected to the J12 SODIMM 200 connector.

For typical eCos development it is expected that programs will be downloaded and debugged via the on-board J-Link USB interface (J14), or via a hardware debugger (JTAG/SWD) attached to the standard ARM 20-pin JTAG (J9) connector. Use of a hardware debugging interface avoids the requirement for a debug monitor application to be present on the platform.

Note

Hardware modification of the board may be required to add support to allow use of the 20-pin J9 connector.

See the CPU Module documentation regarding the use of flash for holding a RedBoot or GDB Stubs image if a debug monitor is required for development.

Supported Hardware

The SAMA5D3x-MB motherboard is common to different "EK" systems, but depending on the actual CPU installed some differences exist as to motherboard peripherals/connections that may be useable.

The motherboard provides the 10/100 Ethernet (MII/RMII) KSZ8051RNL PHY providing support via the J24 (labelled "ETH1") connector. For CPU Module systems where the Gigabit GMAC interface is available the PHY and support logic is provided by the CPU Module. The base motherboard provides the J17 GETH connector.

The Multimedia Card Interface (MCI) on the CPU provides support for the two card sockets on the motherboard. The MCI0 peripheral is connected to the full size SD/MMCplus card socket at J7. While the socket supports 8-bit mode, at present in eCos, only 1- or 4-bit modes are supported. MCI1 is connected to the MicroSD card socket at J6. Both sockets support card insertion/removal detection, but neither support write protect tab detection. Support for MCI use of these sockets is provided by definitions in the platform HAL along with the main SAM MCI device driver in CYGPKG_DEVS_MMCSD_ATMEL_SAM_MCI.

The motherboard supports dual CAN bus interfaces CAN0/CAN1, via the RJ12 connectors at J18/J27. Note that the pin assignments on these RJ12 sockets differ from some previous Atmel boards, so care needs to be taken to ensure that CAN cables are wired correctly. Support for the CAN interfaces is provided by the Atmel SAM CAN Driver.

The BMS signal to the CPU controls how the system is booted. The CPU variant bootstrap overview should be read in conjunction with this documentation. For example, when used with the SAMA5D3x-CM daughterboards the JP9 jumper controls the CPU bootstrap source:

Table 278.1. JP9 BMS

BMS signal (JP9)	Description
JP9 OPEN	Embedded (on-chip) RomBOOT first-level bootloader is used. Actual boot source used will depend on a combination of the CPU Module I/O configuration and the presence of suitable binaries.

BMS signal (JP9)	Description
JP9 CLOSED	Application installed on the EBI_CS0 parallel NOR flash on the CPU Module daughterboard is executed in-situ.

Note

On the SAMA5D3x-MB the BMS signal is connected to GND through JP9 (shipped “open” by default).

Tools

The board port is intended to work with GNU tools configured for an **arm-eabi** target. The original port was done using **arm-eabi-gcc** version 4.7.3e, **arm-eabi-gdb** version 7.6.1, and **binutils** version 2.23.2. All development work and testing was undertaken using SAMA5D3x-MB REV.C hardware.

Name

Setup — Preparing the SAMA5D3x-MB Board for eCos Development

Overview

In a typical development environment the SAMA5D3x-MB board is programmed via a JTAG/SWD interface. This will either be by loading smaller applications into the on-chip SRAM, or into suitably initialised DDR2-SDRAM memory. Alternatively applications may be loaded into bootable memory-mapped devices, e.g. EBI_CS0 NOR flash, or loaded via the on-chip `RomBOOT` code via a second-level SRAM boot scheme. The following sections initially deal with JTAG/SWD hardware based debugging approaches.

For debugging applications are loaded and then executed on the board via the debugger **arm-eabi-gdb**, or via the Eclipse IDE.

The SAMA5D3x-MB motherboard provides a built-in J-Link hardware debug solution, as well as optionally providing the J9 JTAG connector for attaching 3rd-party hardware debuggers (e.g. Ronetix PEEDI).

J-Link

By default the motherboard provides access to both the SAMA5's DBGU (debug UART) and the J-Link adapter through the J14 USB connector. Output on the DBGU channel is accessible via USB on a connected host as a USB CDC-ACM serial class device. Additionally, debugging support is accessible through the proprietary J-Link adaptor via the Segger J-Link GDB server.

The motherboard can be modified (by adding and removing resistors) to route DBGU to the J8 USART1 connector. However, if diagnostics via that connector are desired, it is simpler to re-configure eCos to directly send diagnostics to the J8 port. This can be achieved by setting the HAL diagnostic console channel (and if using RedBoot, GDB channel) to port 1 (the default setting of 0 indicates routing to the DBGU). However, if it is necessary after all to modify the hardware and route DBGU to J8, then support for the SAMA5's USART1 device must be manually disabled by the user as it will no longer be physically connected (removing the "`implements CYGINT_HAL_ARM_CORTEXA_SAMA5D3_USART1`" line from `hal_arm_cortexa_sama5d3x_mb.cdl`).

JLinkGDBServer

Segger's JLinkGDBServer application provides a network-based GDB server connection to J-Link hardware debuggers, including on-board J-Link hardware as used on the SAMA5D3x-MB. The Segger webpage http://www.segger.com/jlink-software.html provides a J-Link software package download that incorporates the J-link GDB Server.

Note

When downloading the software you may need to click on a link “I do not have a serial number because I own an eval board with J-Link on-board. How can I download J-Link software for it?”. This will allow you to download the J-Link software without prior knowledge of a serial number.

Experience has also shown that a reboot of the PC may be required after installation of the jlink software.

To ensure compatibility between the on-board J-Link firmware and software, agree to any prompt you may encounter to update the firmware during the first run of the J-Link software.

A helper script `sama5d3xek-ddram.jlink` is provided in the `misc` directory of the `sama5d3x_mb` package. This should be specified when executing the `JLinkGDBServer` application, and implements a set of commands that are executed when a GDB connection is established to the server. For example, under Linux:

```
$ JLinkGDBServer -device ATSAMA5D31 -xc $ECOS_REPOSITORY/packages/hal/arm/cortexa/sama5d3/sama5d3x_mb/current/misc/sam
```

Under Windows:

```
C:\eCosPro> "C:\Program Files (x86)\SEGGER\JLink_V612j\JLinkGDBServerCL.exe" -device ATSAMA5D31 -xc $ECOS_REPOSITORY\p
```

Note that in the above examples the -device argument should reference the relevant processor installed on the SAMA5D3x-MB motherboard. The eCos <vsn> and JLink_Vxxxx version names will change over time; you will need to use the path specific to the actual versions installed.

Other JLinkGDBServer options may be used as required, for example the `-silent` option reduces the output generated by the server.

A normal GDB debug session can then be started by connecting to the relevant JLinkGDBServer port (default 2331), for example:

```
(gdb) target remote localhost:2331
(gdb) load
```

A simpler `sama5d3xek.jlink` script is provided that does *NOT* initialise the clocks or DDR2-SDRAM, and can be used if explictly debugging `ROM` or `SRAM` startup types where execution of the relevant run-time initialization is desired.

From the GDB command-line when connected to JLinkGDBServer the “monitor regs” command can be used to see all of the non-VFP register state.

Note

Currently, as-of JLinkGDBServer version `V484`, there is *no* support for accessing the Cortex-A5 VFPv4-D16 (VFP) registers when using JLinkGDBServer. e.g. from the GDB “info all-reg” or “info float” commands.

If low-level debugging of floating point operations is required then n alternative debugger should be used. The PEEDI hardware JTAG debugger provides support for displaying the VFP register state, as does the GDB stubs present in the software based RedBoot monitor.

OpenOCD

Using the OpenOCD debug server via the J-Link (J14) interface, is not yet possible. For the moment it is recommended to use the JLinkGDBServer on the host if hardware debug via the J-Link interface is required.

PEEDI

For the Ronetix PEEDI, the `sama5d31ek.peedi.cfg` file should be used to setup and configure the hardware to an appropriate state to load programs. This includes setting up the PLLs and SDRAM controller. You can also check the Ronetix website http://download.ronetix.info/peedi/cfg_examples/cortex-a for updated versions.

Note

Use of a PEEDI debugger requires hardware modification of the standard SAMA5D3x-MB board, which will provide a 20-pin ARM JTAG connector at J9. The required modifications are detailed in the board's user manual.

The `sama5d31ek.peedi.cfg` file also contains an option to define whether hardware or software breakpoints are used by default, using the `CORE0_BREAKMODE` directive in the `[PLATFORM_CortexA8]` section (NOTE: The PEEDI firmware identifies not just A8 CPUs with the `CortexA8` tag). Edit this file if you wish to use hardware break points, and remember to restart the PEEDI to make the changes take effect.

On the PEEDI, debugging can be performed either via the telnet interface or using **arm-eabi-gdb** and the GDB interface. In the case of the latter, **arm-eabi-gdb** needs to connect to TCP port 2000 on the PEEDI's IP address. For example:

```
(gdb) target remote 111.222.333.444:2000
```

By default when the PEEDI is powered up, the target will always run the initialization section of the `sama5d31ek.peedi.cfg` file, and halts the target. This behaviour is repeated with the PEEDI **reset** command.

If the board is reset (either with the '**reset**', or by pressing the reset button) and the '**go**' command is then given, then the board will boot as normal. Depending on how the platform jumper state is configured this could run either a second-level bootstrap, or a `ROM` RedBoot resident in flash.

By default the arm-eabi-gdb connection to the PEEDI will default to displaying the obsolete `FPA` registers. To enable access to the `VFP` registers a suitable target description file should be configured prior to connecting to the target system. This can either be done manually every time a GDB session is started, or more sensibly embedded in the users `.gdbinit` used to configure GDB.

The `sama5d3x-tdesc.xml` file can be used to define the target description using the GDB **set tdesc filename *<file>*** command. For example the `.gdbinit` could contain something similar to:

```
set tdesc filename $ECOS_REPOSITORY/packages/hal/arm/cortexa/sama5d3/sama5d3x_mb/current/misc/sama5d3x-tdesc.xml
```

So that it has the relevent target description available prior to the remote debug connection being established. This will allow access to the VFP registers via the PEEDI, for example via the GDB **info all-reg** command.

Consult the PEEDI documentation for information on other features.

Name

Configuration — Platform-specific Configuration Options

Overview

The SAMA5D3x-MB motherboard platform HAL package is loaded automatically when eCos is configured for a suitable target, e.g. `atsama5d31_ek`. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Name

HAL Port — Implementation Details

Overview

The platform consists of the pairing of the MotherBoard and a CPU Module. The only motherboard specific support is the EMAC PHY. The majority of the port is covered by the CPU Module package.

The CPU Module startup documentation provides an overview of the system startup.

Name

BootUp Integration — Detail

BootUp

BootUp is a lightweight BootROM package that can support features such as in-field upgrades and in the case of the SAMA5 in particular, a secure boot capability.

The BootUp support for the SAMA5D3 targets is primarily implemented in the `sama5d3_mb_misc.c` file. The functions are only included when the `CYGPKG_BOOTUP` package is being used to construct the actual BootUp loader binary.

The BootUp code is designed to be very simple, and it is envisaged that once an implementation has been defined the binary will only need to be installed onto a device once. Its only purpose is to allow the startup of the main `ROMRAM` application.

This platform specific documentation should be read in conjunction with the generic BootUp package documentation.

The BootUp package provides a basic but fully functional implementation for the platform. It is envisaged that the developer will customize and further extend the platform side support to meet their specific application identification and update requirements.

The BootUp binary can be installed on *any* SAMA5D3x bootable media, and is not restricted to being placed into SPI Dataflash.

Note

The NOR flash mapped to EBI_CS#0 is *NOT* an acceptable boot source for the on-chip first-level `RomBOOT` code.

On execution BootUp will copy the `ROMRAM` configured final application from its Non-Volatile-Memory (NVM) location, which currently can either be the SPI Dataflash (default) or the memory-mapped NOR flash. The configuration option `CYGIMP_BOOTUP_SAMA5D3_SOURCE` selects whether the second-level BootUp code will look for the final application image in `SPI` or `NOR`.

Warning

When selecting `SPI` then the relevant CPU Module hardware configuration to allow the on-chip `RomBOOT` to boot from SPI should be used. This requires that jumper `JP1` on the Embest/Flextronics “SAMA5D3x-CM_rev.E” be fitted, and for the Ronetix “SAMA5D3x-CM v2.0” daughterboard jumper `J2` should be fitted.

The following diagrams give an overview of the first-level (on-chip `RomBOOT`) and second-level (`SRAM`) boot sequence:

Figure 278.1. On-chip `RomBOOT` executes

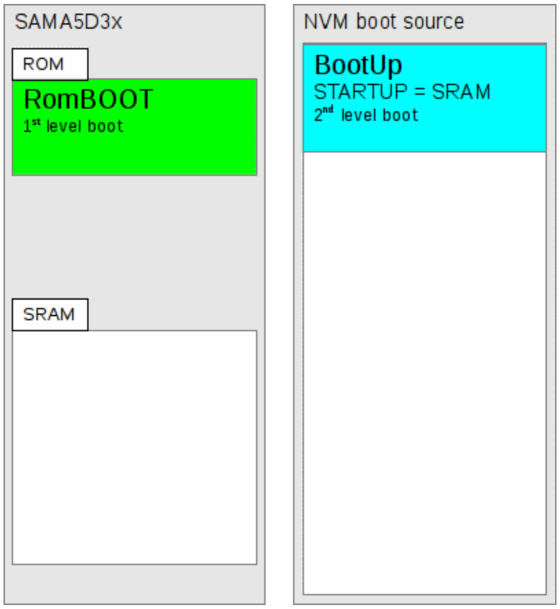


Figure 278.2. On-chip `RomBOOT` copies second-level boot code from NVM to on-chip SRAM

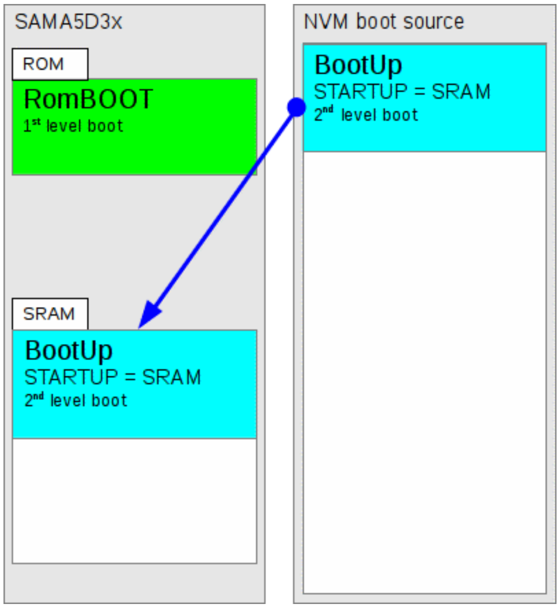


Figure 278.3. SRAM loaded second-level boot code is executed

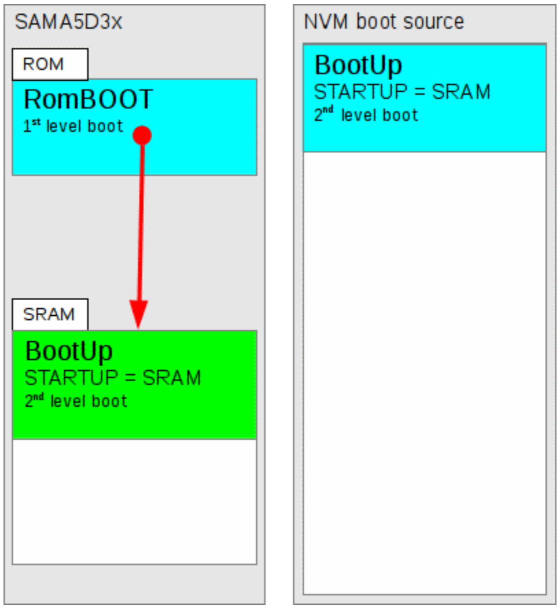


Figure 278.4. Final application `ROMRAM` is located in SPI or NOR NVM

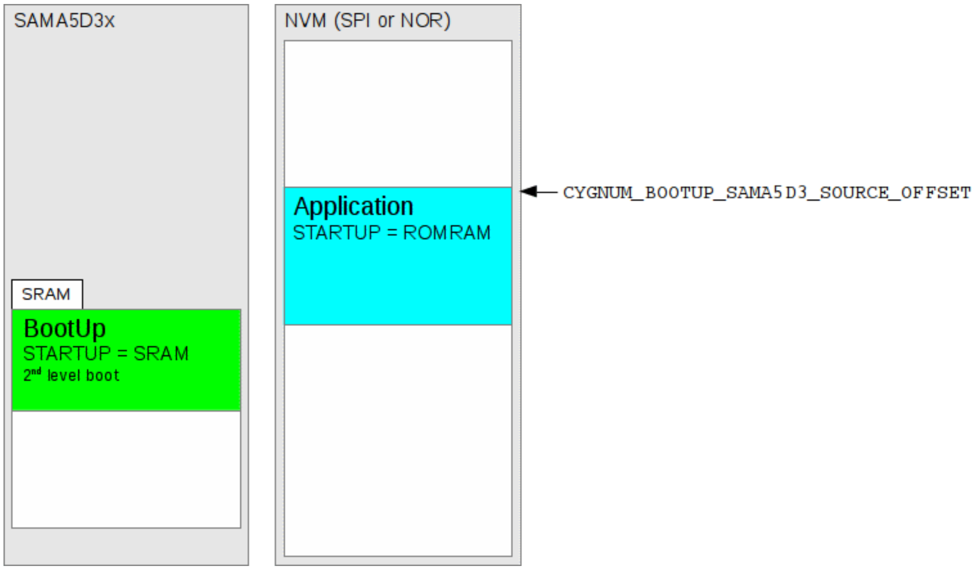


Figure 278.5. Second-level boot copies `ROMRAM` from NVM to DDR2-SDRAM

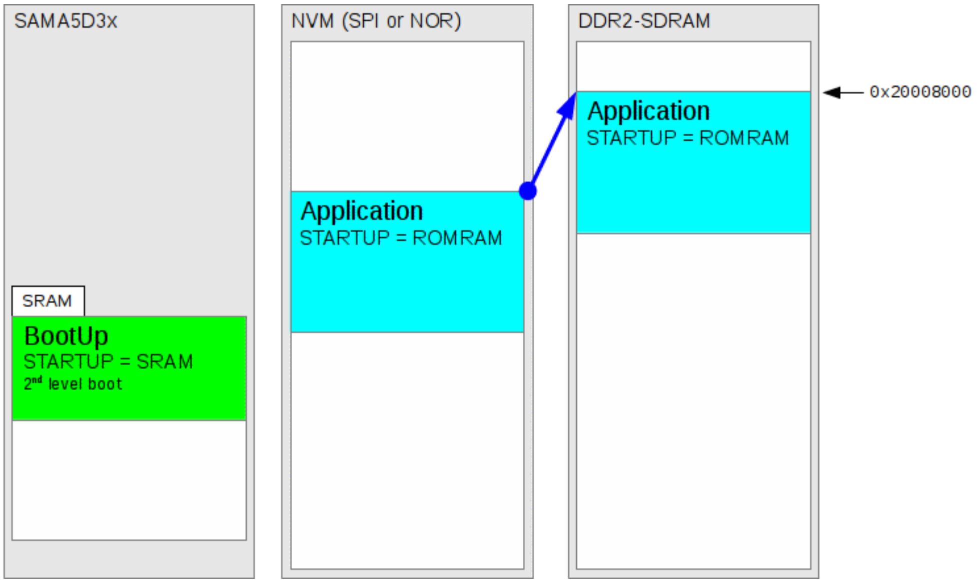
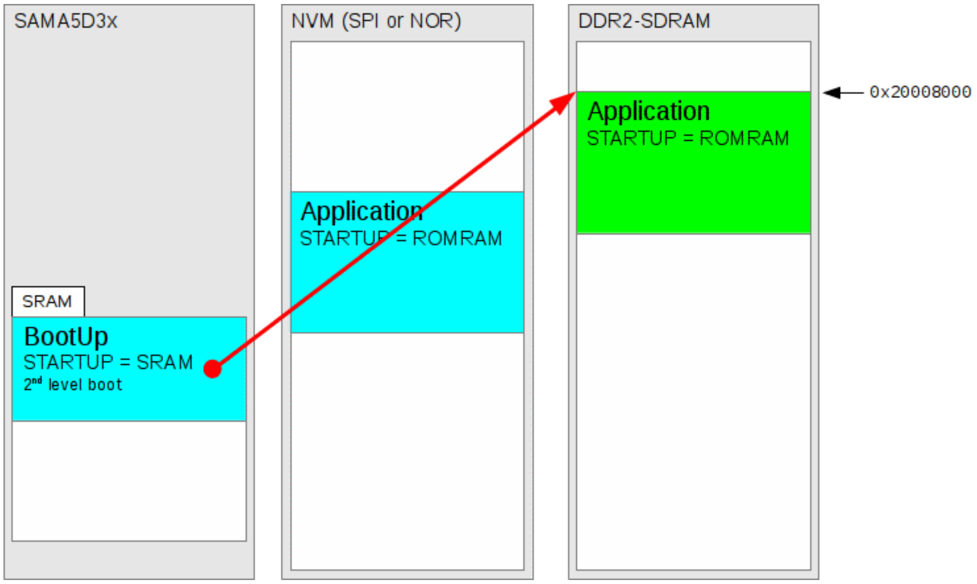


Figure 278.6. Application is started



Encrypted Application

The BootUp SRAM application can be configured with a built-in AES-256 key which can be used to decrypt the final `ROMRAM` application image (as stored on the NVM) prior to execution from its final RAM location. For ease of use during development, the BootUp binary can load both encrypted and plaintext (unencrypted) application images. This allows testing of the boot sequence prior to finalising the application key, or the final application build encryption process.

Note

This feature is really only relevant (and useful) as part of a secure boot process. When secure boot is *NOT* being used then by itself encrypting the final `ROMRAM` application provides no security since the SRAM second-level boot loader can be examined to extract the AES-256 key used to decrypt the application. Using the BootUp decryption support only makes sense when it is part of a complete reset->application security implementation.

As with the normal SRAM based second-level bootstrap world, the on-chip `RomBOOT` code loads and executes the second-level boot code from any suitable boot source. When BootUp is used as the second-level bootstrap it is configured to look at either the `SPI` or `NOR` memories for the encrypted final `ROMRAM` application image.

Figure 278.7. Second-level boot code built with AES-256 key

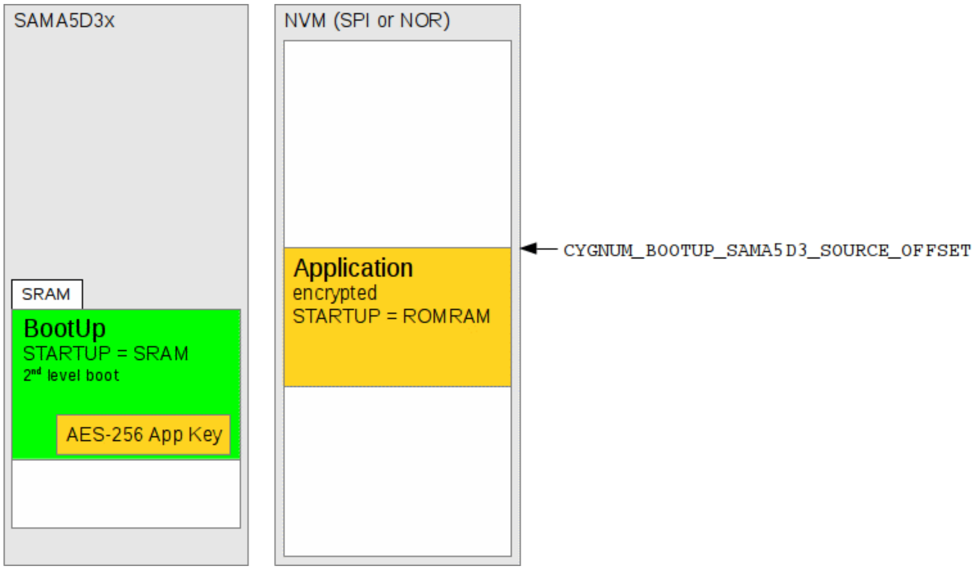


Figure 278.8. Stored key is used to decrypt NVM application into RAM

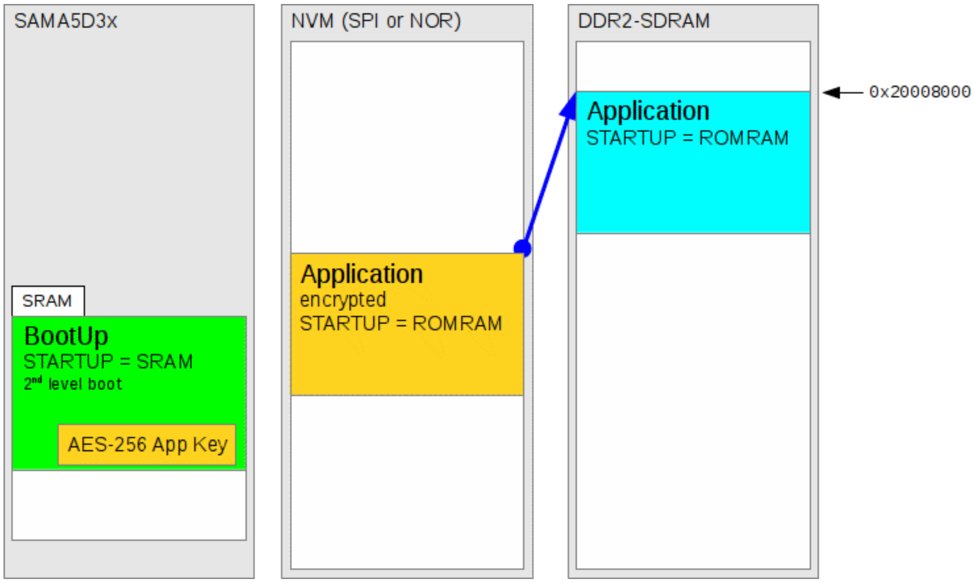
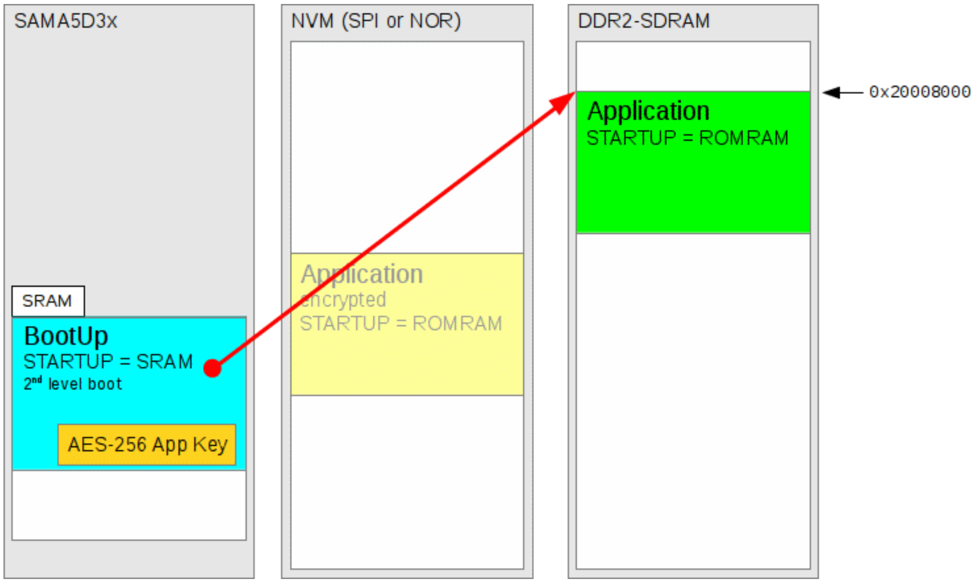


Figure 278.9. Decrypted application is started



Encrypting The “Final” Application

An example host tool to encrypt the final `ROMRAM` application using a AES-256 key is provided. A pre-built executable is provided, and available from the installed tools path as for the other build related tools. The source for the tool is available in the `$ECOS_REPOSITORY/packages/hal/arm/cortexa/sama5d3/sama5d3x_mb/current/host/claes.c` file. If required, a `Makefile` (for GNU make) is provided in the `host` directory to allow the command to be built locally.

The AES-256 key used to encrypt the application binary *MUST* be the same key that is *built-into* the second-level boot loader used to copy the application from its NVM location into RAM for execution.

An example key file is provided in `$ECOS_REPOSITORY/packages/hal/arm/cortexa/sama5d3/sama5d3x_mb/current/misc/example256.key`, but as always for a production environment the developer is responsible for managing the actual key values used, and for ensuring that embedded keys are not distributed publicly.

Assuming that the application binary is available in the file `finalapp.bin`, the following example shows how the `claes` tool can be used to create the encrypted image.

```
$ claes -k example256.key finalapp.bin -o encrypted.bin
```

The encrypted image `encrypted.bin` would then be stored at the relevant NVM offset using whatever production methodology is in use. If required, for example, the default SAMA5D3 RedBoot application provides the necessary network and flash operations to allow it to be used to initialise the NVM contents from supplied binary images.

The claes command provides a brief description of its options, which can be viewed by requesting `--help` (or the shorthand `-h`) on the command-line:

```
$ claes --help
```

Atmel Secure Boot

Details of the Atmel Secure Boot process are beyond the scope of this document, since a customer specific Non-Disclosure-Agreement (NDA) with Atmel needs to be in place. The reader is referred to the Atmel website www.atmel.com with regards to the “Secure Boot on SAMAD3 Series” (Atmel literature number `11165A`) documentation.

Building BootUp

Building a BootUp loader image is most conveniently done at the command line. The steps needed to rebuild the SRAM version of BootUp are:

```
$ mkdir bootup_SRAM
$ cd bootup_SRAM
$ ecosconfig new sama5d31_ek minimal
[ … ecosconfig output elided … ]
$ ecosconfig import $ECOS_REPOSITORY/packages/hal/arm/cortexa/sama5d3/sama5d3_mb/current/misc/bootup_SRAM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

The resulting install/bin/bootup.bin binary can then be programmed into a suitable non-volatile memory as supported by the SAMA5D3 on-chip RomBOOT.

The example bootup_SRAM.ecm is configured to expect to find the ROMRAM application stored in the SPI Dataflash at offset CYGNUM_BOOTUP_SAMA5D3_SOURCE_OFFSET. It also is configured with application encryption support (CYGFUN_BOOTUP_SAMA5D3_SOURCE_SECURE option) to allow decryption of the SPI Dataflash stored application to its final RAM destination.

Note

The application image to be loaded does *not* need to be encrypted. The BootUp code checks the embedded application binary identity markers to check for a plaintext (unencrypted) image to be started, prior to attempting to decrypt the data and check for a valid encrypted image at offset CYGNUM_BOOTUP_SAMA5D3_SOURCE_OFFSET in the source NVM.

Chapter 279. Atmel SAMA5D3x-CM (CPU Module) Platform HAL

Name

CYGPKG_HAL_ARM_CORTEXA_SAMA5D3X_CM — eCos Support for the SAMA5D3x-CM CPU Module

Description

There are currently two supported variants of CPU Module (CM). The Embest/Flextronics “SAMA5D3x-CM_rev.E” and the Ronetix “SAMA5D3x-CM v2.0” daughterboards. The boards can be identified by the silk screened names on the respective PCBs.

The currently supported CM variants are identical in functionality, with the exception of the type of NOR flash installed on EBI_CS0. However, the eCos port is configured to allow a single binary to execute irrespective of the installed CPU Module.

The common CM features include 512MiB of DDR2-SDRAM, 16MiB NOR flash, routing for the 10/100 EMAC and 10/100/1000 GMAC, and blue and red LEDs. Some common CM features are not yet supported by the eCos port, e.g. NAND.

For typical eCos development it is expected that hardware debugging will be used, as detailed in the SAMA5D3x-MB MotherBoard documentation.

However, RedBoot can be used to provide development support. RedBoot provides gdb stub functionality. so it is then possible to download and debug stand-alone and eCos applications via the gdb debugger. This can happen over either a serial line or over Ethernet. The RedBoot image can be a ROM application programmed into the parallel NOR flash, or a ROMRAM application loaded via the second-level bootloader. See RedBoot Location for an overview.

The bootstrap options for the SAMA5D3x parts are documented in the CPU variant documentation. How the CPU boots depends on the BMS signal (supplied by the motherboard via the CPU Module main connector pin 184). The CM has a Pull-Up on the CPU BMS pin so will default to BMS_BIT=0.

Note

The Embest/Flextronics CM is pre-installed with example application software. The software execute depends on the setting of CM jumper JP1. When JP1 is open then the NAND based AT91Bootstrap second-level boot loader is loaded into SRAM and used to start a NAND stored application world. When JP1 is closed then a system “test” application is loaded into SRAM from the SPI Dataflash and executed.

Warning

When JP1 is closed, if the example Atmel code is present and the NOR test (single test 05) is executed then it will erase the first 384K of the NOR flash (0x10000000..0x1005FFFF). This could erase any ROM application (e.g. RedBoot) that the user may have installed into the parallel NOR flash.

Name

HAL Port — Implementation Details

Overview

The SAMA5D3x-CM daughterboard platform HAL package is loaded automatically when eCos is configured for a suitable target, e.g. `atsama5d31_ek`. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

In the release view of the world (depending on the state of the `BMS` signal) the SAMA5D3x-CM daughterboard either boots into the BootUp “application loader” from a suitable on-chip `RomBOOT` supported memory, or alternatively from the EBI-CS0 parallel NOR flash based `ROM` startup type application.

When using the second-stage BootUp loader the main (`ROMRAM` startup type) application is then loaded from the configured non-volatile storage (e.g. parallel NOR flash) into the DDR2-SDRAM for execution.

The CPU variant bootstrap overview should be read in conjunction with this documentation.

Following a hard or soft reset the HAL will initialize or reinitialize many of the on-chip peripherals. There is an exception for `RAM` startup applications which depend on a ROM monitor for certain services, and so will not attempt to re-initialize the underlying peripheral.

For `ROM`, `ROMRAM` and `SRAM` startups the HAL will perform additional initialization, programming the various internal registers including the PLL, peripheral clocks and GPIO pins as required. The details of the early platform hardware startup may be found in the `plf_hardware_init()` function within the source file `src/sama5d3x_mb_misc.c`.

Memory Map

The SAMA5D3X_CM HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

External RAM

This is located at address 0x20000000 of the memory space, and is 512MiB long. For `ROM`, `ROMRAM` and `SRAM` applications the initial 32KiB is set aside, primarily for the first level MMU table and (depending on the startup type) the eCos VSR table. The rest of the RAM is then available for application use. For `RAM` startup applications the first 1MiB of RAM is reserved for the “debug” monitor (e.g. RedBoot), and the top `CYGNUM_HAL_COMMON_INTERRUPTS_STACK_SIZE` bytes are reserved for the interrupt stack. The remainder is then available for the `RAM` application.

Internal RAM

This is located at address 0x00300000 of the memory space, and is 128KiB in size.

NOR FLASH

This is located at address 0x10000000 of the memory space and *MAY* be mapped to 0x00000000 at reset if the SAMA5D3 is so configured. This region is 16MiB in size. `ROM` applications are by default configured to run from this memory. When RedBoot is being used then this memory is managed by RedBoot's FIS system, otherwise it is the applications responsibility to manage the NOR flash space.

On-chip Peripherals

The I/O is primarily accessible from location 0xF0000000 upwards, though some I/O is mapped into the initial 10MiB of the address space. Descriptions of the contents can be found in the Atmel SAMA5D3 Series Datasheet.

Linker Scripts

The platform linker script defines the following symbol:

hal_mmu_page_directory_base — This symbol defines where the initialization code will place the level-1 table when initialising the MMU.

Diagnostic LEDs

Two LEDs are fitted onto the CPU Module for diagnostic purposes, one red and one blue.

The platform HAL header file at `<cyg/hal/sama5d3x_cm_io.h>` defines the following convenience function to allow the LEDs to be controlled:

```
extern void hal_sama5d3x_cm_led(cyg_uint32 bitmask);
```

The low-order 2-bits of the argument *`bitmask`* correspond to each of the 2 LEDs. The red LED is logically mapped to bit 0, with the blue LED mapped to bit 1.

Note

The blue LED on GPIO line PE25 is also used as the 1-wire bus, so accesses to 1-wire devices will cause that LED to flicker.

SPI Dataflash

The variant HAL SPI support provides the necessary underlying SPI bus definition. The CM platform layer defines the `spi_dataflash_dev0` device instance describing the SPI Dataflash hardware.

The SPI Dataflash media can only be accessed with the Flash API using the flash device `m25pxx_flash_device`. For the purposes of this API a placeholder address range has been allocated as if the Flash is present at this address. The base of this address range is `0xE0000000`. This reserved range is not real memory and any attempt to access it directly by the processor other than via the Flash API will result in a memory address exception.

A test application at25df321 is provided within the `tests` subdirectory of the `CYGPKG_HAL_ARM_CORTEXA_SAMA5D3X_CM` package. This test communicates with the SPI Dataflash on the CM to perform read and write operations using the flash API.

Ethernet Driver

Depending on the processor specific SAMA5D3x-CM module used, either or both EMAC and GMAC Ethernet devices are available to use. The `CYGPKG_DEVS_ETH_ARM_AT91` package supports both of these devices. This driver is not active until the generic Ethernet support package, `CYGPKG_IO_ETH_DRIVERS`, is included in the configuration.

Both the standard (BSD and lwIP compatible) and direct (lwIP only) device drivers are supported. The standard driver is enabled by default; the direct driver can be enabled by setting `CYGOPT_IO_ETH_DRIVERS_LWIP_DRIVER_DIRECT` option. At the time of writing, the direct driver only supports the EMAC (ETH1), and not the GMAC (ETH0).

RedBoot Installation

Note

Unless you explicitly need network based debugging, or are interested in other aspects of the RedBoot functionality, it is generally the case that development and debugging using a direct hardware JTAG/SWD approach is generally superior and obviates the need to install RedBoot on the target.

RedBoot, by default, is configured to use the EBI_CS0 NOR flash as storage for its FIS and config information, regardless of whether it is executing from RAM or in place as a `ROM` application.

Building RedBoot

RedBoot will normally be a ROMRAM startup, since it will be loaded via the second-level bootloader, or loaded directly using a hardware JTAG debugger, into the DDR2-SDRAM memory for execution.

Note

Pre-built RedBoot binary images are supplied with the eCos release in the loaders sub-directory.

The following example illustrates the command-line steps needed to configure and build a ROMRAM RedBoot:

```
$ mkdir redboot_ROMRAM
$ cd redboot_ROMRAM
$ ecosconfig new atsama5d31_ek redboot
$ ecosconfig import $ECOS_REPOSITORY/packages/hal/arm/cortexa/sama5d3/sama5d3x_mb/current/misc/redboot_ROMRAM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

However, if required, a ROM based RedBoot can be executed directly from the EBI_CS0 NOR flash when JP9 on the SAMA5D3x-MB motherboard is closed.

```
$ mkdir redboot_ROM
$ cd redboot_ROM
$ ecosconfig new atsama5d31_ek redboot
[ … ecosconfig output elided … ]
$ ecosconfig import $ECOS_REPOSITORY/packages/hal/arm/cortexa/sama5d3/sama5d3x_mb/current/misc/redboot_ROM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

The RedBoot Location section of the generic SAMA5D3 variant documentation provides a graphical representation of both these ROMRAM and ROM models.

Programming RedBoot

Some hardware debuggers will allow direct programming of the EBI_CS0 NOR flash, but for those that do not provide such support a JTAG loaded ROMRAM executable can be used to load and write the ROM image to its NOR flash destination. For example, if the ROMRAM RedBoot as built above is loaded using a hardware debugger and executed, then the following shows that RAM based RedBoot being used to load a 0x16020 byte long redboot.bin image from a TFTP server and written to the RedBoot FIS section in the NOR.

```
RedBoot> load -r -h 192.168.7.39 -b 0x30000000 redboot.bin
Using default protocol (TFTP)
Raw file loaded 0x30000000-0x30016020, assumed entry at 0x30000000
RedBoot> fis unlock RedBoot
... Unlocking from 0x10000000-0x100fffff: ........
RedBoot> fis write -f 0x10000000 -b 0x30000000 -l 0x16020
* CAUTION * about to program FLASH
            at 0x10000000..0x1001ffff from 0x30000000 - continue (y/n)? y
... Erase from 0x10000000-0x1001ffff: .
... Program from 0x30000000-0x30020000 to 0x10000000: .
RedBoot> mcmp -s 0x30000000 -d 0x10000000 -l 0x16020
```

With the binary image written to the start of the NOR flash, and the motherboard BMS signal suitably configured, then after a power-on reset the board will boot using the ROM RedBoot application. Such a RedBoot world allows for GDB debugging of RAM startup applications loaded via the J8 USART1 serial connection, the 10/100 (EMAC) Ethernet connection, or the 10/10/1000 (GMAC) Ethernet connection. Selection of which Ethernet interface to use is set with the RedBoot **fconfig** command and its Default network device: entry. Entering at91_eth0 selects the Gigabit Ethernet (J17/GETH) interface, whereas at91_eth1 selects the 10/100 Ethernet (J24/ETH1) interface.

This method of using a RAM based RedBoot, and loading a binary into RAM (or SRAM) before writing to the NOR flash, can be used for any `ROM` application that needs to be executed from startup with a BMS closed configured platform. Alternatively, Atmel provides tools to work in conjunction with the on-chip SAM-BA monitor that can also be used to program the various CPU Module memories.

Real-time characterization

The tm_basic kernel test gives statistics gathered about the real-time characterization and performance of the kernel. The sample output is shown here for information. The test was built for ROMRAM startup, where both code and data are using the external DDR2-SDRAM.

Example 279.1. sama5d3x_cm Real-time characterization

```
            Startup, main thrd : stack used   444 size  1792
            Startup :  Interrupt stack used  4096 size  4096
            Startup : Idlethread stack used    88 size  1280

eCos Kernel Timings
Notes: all times are in microseconds (.000001) unless otherwise stated

Reading the hardware clock takes 1 'ticks' overhead
... this value will be factored out of all other measurements
Clock interrupt took    2.80 microseconds (23 raw clock ticks)

Testing parameters:
  Clock samples:            32
  Threads:                  64
  Thread switches:         128
  Mutexes:                  32
  Mailboxes:                32
  Semaphores:               32
  Scheduler operations:    128
  Counters:                 32
  Flags:                    32
  Alarms:                   32
  Stack Size:             1088

                                Confidence
    Ave     Min     Max     Var  Ave  Min  Function
 ======  ======  ======  ====== ========== ========
   1.67    1.09    2.67    0.31   53%  28% Create thread
   0.17    0.12    0.85    0.07   65%  65% Yield thread [all suspended]
   0.20    0.12    1.33    0.10   85%  59% Suspend [suspended] thread
   0.16    0.12    0.61    0.06   71%  71% Resume thread
   0.26    0.12    0.97    0.05   82%   9% Set priority
   0.01    0.00    0.24    0.02   92%  92% Get priority
   0.62    0.48    3.88    0.15   92%  84% Kill [suspended] thread
   0.17    0.12    0.85    0.07   65%  65% Yield [no other] thread
   0.31    0.24    1.09    0.08   90%  64% Resume [suspended low prio] thread
   0.15    0.12    0.36    0.04   78%  78% Resume [runnable low prio] thread
   0.25    0.12    1.09    0.03   85%   9% Suspend [runnable] thread
   0.18    0.12    1.09    0.07   98%  65% Yield [only low prio] thread
   0.16    0.12    0.61    0.05   78%  78% Suspend [runnable->not runnable]
   0.55    0.48    1.94    0.09   89%  73% Kill [runnable] thread
   0.48    0.36    2.30    0.08   59%  31% Destroy [dead] thread
   0.90    0.73    1.94    0.09   84%   7% Destroy [runnable] thread
   1.30    1.09    3.03    0.19   51%  62% Resume [high priority] thread
   0.42    0.36    1.21    0.06   61%  61% Thread switch

   0.02    0.00    0.24    0.03   86%  86% Scheduler lock
   0.12    0.00    0.24    0.01   92%   5% Scheduler unlock [0 threads]
   0.12    0.00    0.24    0.01   92%   5% Scheduler unlock [1 suspended]
   0.12    0.00    0.36    0.02   88%   6% Scheduler unlock [many suspended]
   0.12    0.00    0.61    0.02   89%   5% Scheduler unlock [many low prio]
```

```
    0.08    0.00    0.73    0.09   87%  59% Init mutex
    0.22    0.12    1.21    0.09   46%  46% Lock [unlocked] mutex
    0.27    0.12    1.94    0.14   50%  90% Unlock [locked] mutex
    0.20    0.12    1.09    0.10   90%  62% Trylock [unlocked] mutex
    0.16    0.12    0.61    0.06   78%  78% Trylock [locked] mutex
    0.03    0.00    0.48    0.05   84%  84% Destroy mutex
    1.23    1.21    1.82    0.04   96%  96% Unlock/Lock mutex

    0.19    0.00    0.85    0.09   87%   6% Create mbox
    0.01    0.00    0.36    0.02   96%  96% Peek [empty] mbox
    0.30    0.24    1.45    0.11   90%  90% Put [first] mbox
    0.00    0.00    0.00    0.00  100% 100% Peek [1 msg] mbox
    0.28    0.24    0.61    0.06   81%  81% Put [second] mbox
    0.00    0.00    0.12    0.01   96%  96% Peek [2 msgs] mbox
    0.25    0.12    1.33    0.07   78%  18% Get [first] mbox
    0.25    0.12    0.73    0.04   81%  12% Get [second] mbox
    0.28    0.12    1.09    0.09   78%  12% Tryput [first] mbox
    0.21    0.12    0.61    0.07   59%  37% Peek item [non-empty] mbox
    0.23    0.12    1.09    0.07   62%  34% Tryget [non-empty] mbox
    0.20    0.12    0.48    0.07   46%  46% Peek item [empty] mbox
    0.19    0.12    0.61    0.07   96%  50% Tryget [empty] mbox
    0.02    0.00    0.12    0.03   87%  87% Waiting to get mbox
    0.01    0.00    0.12    0.01   93%  93% Waiting to put mbox
    0.05    0.00    0.48    0.07   68%  68% Delete mbox
    0.89    0.73    1.94    0.09   90%   6% Put/Get mbox

    0.03    0.00    0.24    0.05   75%  75% Init semaphore
    0.15    0.12    0.48    0.05   81%  81% Post [0] semaphore
    0.19    0.12    0.36    0.06   50%  46% Wait [1] semaphore
    0.15    0.12    0.61    0.05   90%  90% Trywait [0] semaphore
    0.13    0.12    0.24    0.01   93%  93% Trywait [1] semaphore
    0.03    0.00    0.36    0.05   81%  81% Peek semaphore
    0.01    0.00    0.36    0.02   96%  96% Destroy semaphore
    0.81    0.73    1.94    0.10   93%  62% Post/Wait semaphore

    0.15    0.00    0.61    0.11   62%  28% Create counter
    0.05    0.00    0.36    0.07   68%  68% Get counter value
    0.01    0.00    0.12    0.01   93%  93% Set counter value
    0.20    0.12    0.48    0.06   56%  40% Tick counter
    0.02    0.00    0.24    0.03   87%  87% Delete counter

    0.02    0.00    0.36    0.03   90%  90% Init flag
    0.17    0.12    0.97    0.08   96%  75% Destroy flag
    0.14    0.12    0.73    0.04   93%  93% Mask bits in flag
    0.20    0.12    1.09    0.09   93%  62% Set bits in flag [no waiters]
    0.32    0.24    2.06    0.13   93%  93% Wait for flag [AND]
    0.30    0.24    1.21    0.09   96%  75% Wait for flag [OR]
    0.26    0.12    1.09    0.06   81%  12% Wait for flag [AND/CLR]
    0.25    0.12    0.97    0.04   84%  12% Wait for flag [OR/CLR]
    0.00    0.00    0.12    0.01   96%  96% Peek on flag

    0.20    0.00    0.85    0.11   78%   6% Create alarm
    0.26    0.12    1.09    0.06   78%  12% Initialize alarm
    0.14    0.12    0.36    0.03   90%  90% Disable alarm
    0.27    0.12    1.21    0.07   84%   6% Enable alarm
    0.16    0.12    0.36    0.05   71%  71% Delete alarm
    0.24    0.12    0.61    0.03   81%  12% Tick counter [1 alarm]
    1.00    0.97    1.21    0.04   81%  81% Tick counter [many alarms]
    0.37    0.24    1.09    0.04   84%  12% Tick & fire counter [1 alarm]
    5.82    5.70    6.55    0.05   75%  18% Tick & fire counters [>1 together]
    1.17    1.09    1.82    0.09   93%  56% Tick & fire counters [>1 separately]
    2.26    2.18    4.73    0.08   49%  49% Alarm latency [0 threads]
    2.57    2.30    4.24    0.11   67%  10% Alarm latency [2 threads]
    3.08    2.67    4.12    0.21   65%  20% Alarm latency [many threads]
    2.98    2.91    4.97    0.08   97%  60% Alarm -> thread resume latency
    1.07    0.97    2.91    0.00            Clock/interrupt latency

    0.67    0.48    1.70    0.00            Clock DSR latency

     239     172     272                    Worker thread stack used (stack size 1088)
            All done, main thrd : stack used   988 size  1792
            All done :  Interrupt stack used   156 size  4096
            All done : Idlethread stack used   232 size  1280

Timing complete - 29820 ms total

PASS:<Basic timing OK>
EXIT:<done>
```

Chapter 280. Atmel SAMA5D3 Xplained Platform HAL

Name

`CYGPKG_HAL_ARM_CORTEXA_SAMA5D3XPLD` — eCos Support for the SAMA5D3 Xplained platform

Description

The SAMA5D3-XPLD board consists of an ATSAMA5D36 Cortex-A5 microcontroller, 128kiB of SRAM, 256MiB of RAM and 256MiB of NAND flash. It has connectors for Ethernet, LCD, USB and other expansion boards as well as an SD card slot.

For typical eCos development it is expected that `ROMRAM` startup type applications will be downloaded and debugged via a hardware debugger (JTAG) attached to connector `J24`. Use of a hardware debugging interface avoids the requirement for a debug monitor application to be present on the platform. However, if required, a RedBoot image can be programmed into the onboard NAND flash to provide an interactive bootloader environment. When using a GDB stubs monitor it is then possible to download and debug eCos `RAM` startup applications via the gdb debugger over USB (using the TTL debug channel) or Ethernet.

The bootstrap options for the SAMA5D3x parts are documented in the CPU variant documentation. How the CPU boots depends on the BMS signal. The SAMA5D3 Xplained has a Pull-Up on the CPU BMS pin so will default to `BMS_BIT=0`, resulting in the RomBOOT, on-chip, first-level boot loader being executed.

Note

The Atmel SAMA5D3 Xplained is pre-installed with example application software. The board is supplied with jumper `JP5` ("NAND CS") closed, which causes the board to boot using its factory-programmed NAND-based boot loader and then to the sample application image. It is necessary to overwrite these images in order to place RedBoot or any other application on the NAND flash.

Tools

The board port is intended to work with GNU tools configured for an **arm-eabi** target. The original port was done using **arm-eabi-gcc** version 4.7.3e, **arm-eabi-gdb** version 7.6.1, and **binutils** version 2.23.2.

Name

Setup — Preparing the SAMA5D3-XPLD Board for eCos Development

Overview

In a typical development environment, a direct hardware JTAG connection is used to load and execute the eCos application. For hardware debugging, eCos applications would normally be configured for `ROMRAM` startup. The same application binary can be executed via a hardware debugger environment, or programmed into the onboard NAND flash for loading via a second-level boot loader.

Alternatively, the SAMA5D3-XPLD board can boot from NAND flash into a `ROMRAM` RedBoot. eCos applications are then configured for `RAM` startup and then downloaded and run on the board via the debugger **arm-eabi-gdb**.

Note

Unless you explicitly need network based debugging, are interested in other aspects of the RedBoot functionality, or you lack the requisite hardware support, it is generally the case that development and debugging using a direct hardware JTAG/SWD approach is generally superior and obviates the need to install RedBoot on the target.

Atmel Secure Boot code and the normal `RomBOOT` code will disable JTAG over certain startups.

For JTAG debugging the `RomBOOT` code may need to be started with *NO* bootable source (e.g. JP5 `NAND CS` open and no bootable SD card) to enable JTAG access. After reset, a CDC-ACM tty connection (to the on-chip ROM provided SAM-BA terminal) via the J6 USB-A interface can be used to enter a # key to enable JTAG.

Bootstrap process

The typical bootstrap process for this board has several steps:

1. RomBOOT (on-chip, cannot be modified).
2. eCosPro BootUp or AT91Bootstrap
3. Your choice of `ROMRAM` eCos application (such as RedBoot).

For more information about the bootstrap process, refer to the CPU variant bootstrap overview.

RedBoot Installation

The following RedBoot configurations are supported:

Configuration	Description	Use	File
ROMRAM	RedBoot running from external RAM. Loaded from NAND by second-level boot loader, or directly via JTAG hardware debugger.	`redboot_ROMRAM.ecm`	`redboot_ROMRAM.bin`

Note

The DBGU diagnostic (serial) debug channel is only exposed as TTL on J23 pins. A RS232 TTL to USB adapter can be used to allow serial diagnostic output. The serial terminal connection is not required for normal Ethernet based

GDB debugging, but for initial board bring-up where no valid flash configuration is available it can be easier to see what network address has been supplied to the board.

If the standard serial diagnostic/terminal is not available, then it is worth considering configuring the local DHCPD to present a known (fixed) IP address. The current fixed RedBoot MAC address is `12:34:56:78:9a:bc`. RedBoot defaults to using port `9000`.

Programming RedBoot

There are several ways to program the target flash:

1. Some hardware debuggers will allow direct programming of the target NAND flash.
2. If a hardware debugger is available but does not support NAND flash, it is possible to load and run a `ROMRAM` executable that does know how to write the image to its flash destination.
3. Atmel provides tools to work in conjunction with the on-chip SAM-BA monitor that can also be used to program the various memories. Detailed instructions for this process are provided in the Installing binaries with SAM-BA section.

Note

RedBoot is configured by default with both network interfaces active, and waits for both of them to come up before allowing user interaction. This may take up to a minute if either are not connected to a network. It is recommended to connect both interfaces.

Initializing RedBoot Flash Configuration

When RedBoot is loaded into the on-chip flash, it maintains a number of persistent settings in the configuration store on the NAND flash. The flash configuration area needs to be initialized with the following commands.

```
RedBoot> fconfig -i
Initialize non-volatile configuration - continue (y/n)? y
Run script at boot: false
Use BOOTP for network configuration: true
Default server IP address:
DNS domain name:
DNS server IP address:
Network hardware address [MAC] for eth0: 0x0E:0x00:0x00:0xEA:0x18:0xF0
Network hardware address [MAC] for eth1: 0x0E:0x00:0x00:0xEA:0x18:0xF0
GDB connection port: 9000
Force console for special debug messages: false
Network debug at boot time: false
Default network device: at91_eth1
Update RedBoot non-volatile configuration - continue (y/n)? y
... Erase from 0x00000000-0x00000fff: .
... Program from 0x2fdff000-0x2fe00000 to 0x00000000: .
RedBoot>
```

Rebuilding RedBoot

Note

Pre-built RedBoot binary images are supplied with the eCos release in the `loaders` sub-directory.

RedBoot will normally be a `ROMRAM` startup, since it will be loaded via the second-level bootloader, or loaded directly using a hardware JTAG debugger, into the DDR2-SDRAM memory for execution.

The following example illustrates the command-line steps needed to configure and build a `ROMRAM` RedBoot:

```
$ mkdir redboot_ROMRAM
$ cd redboot_ROMRAM
$ ecosconfig new sama5d3xpld redboot
$ ecosconfig import $ECOS_REPOSITORY/packages/hal/arm/cortexa/sama5d3/sama5d3xpld/current/misc/redboot_ROMRAM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

The RedBoot Location section of the generic SAMA5D3 variant documentation provides a graphical representation of the `ROMRAM` model.

Building BootUp

Full details are provided in the BootUp Integration section.

Building AT91Bootstrap

As an alternative to the eCosPro BootUp second-level loader, the Atmel AT91Bootstrap second-level loader can be used to load a `ROMRAM` startup type application from the NVM boot memory into RAM for execution.

A version of the AT91Bootstrap source, modified by eCosCentric, is available in the directory `$ECOS_REPOSITORY/packages/hal/arm/cortexa/sama5d3/var/current/misc/at91bootstrap`. The original (as of writing) is v3.6.2 obtained by:

```
git clone https://github.com/linux4sam/at91bootstrap
```

The modifications provide example configurations to allow RedBoot to be loaded by the SAMA5D3X-EK and SAMA5D3 Xplained platforms, and to allow compilation using the eCosCentric cross-compilation tools (e.g. 4.7.3e release).

The AT91Bootstrap binary can be built from the provided source with the following steps:

```
cd at91bootstrap
make mrproper
make sama5d3_xplainednf_redboot_defconfig
export CROSS_COMPILE=arm-eabi-
make
```

The result of this process is a second-level boot loader binary `binaries/sama5d3_xplained-nandflash-boot-uboot-3.6.2.bin` that can be installed into NAND using SAM-BA as described below.

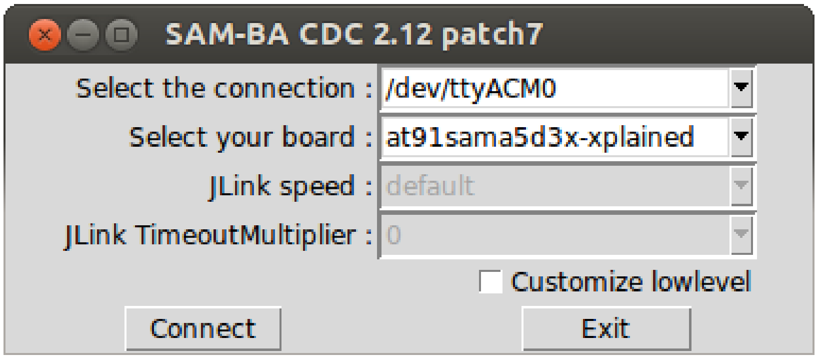
Installing binaries with SAM-BA

A version of the Atmel SAM-BA tool which supports the `at91sama5d3x-xplained` target (e.g. v2-12 patch level 7) can be used to install your choice of BootUp, AT91Bootstrap, RedBoot or any other `ROMRAM` eCos application.

1. Download and install the SAM-BA tool, plus any required patches, from http://www.atmel.com/tools/atmelsam-bain-system-programmer.aspx
2. Disconnect JP5 (`NAND CS`) and ensure no bootable SD card is installed
3. Connect host to J6 USB-A
4. Launch the SAM-BA application
   - Select the relevant COM or /dev/ttyACM port
   - Select board type “at91sama5d3x-xplained”

- Hit “Connect”

Figure 280.1. SAM-BA Board Connection



Select the connection : /dev/ttyACM0

Select your board : at91sama5d3x-xplained

JLink speed : default

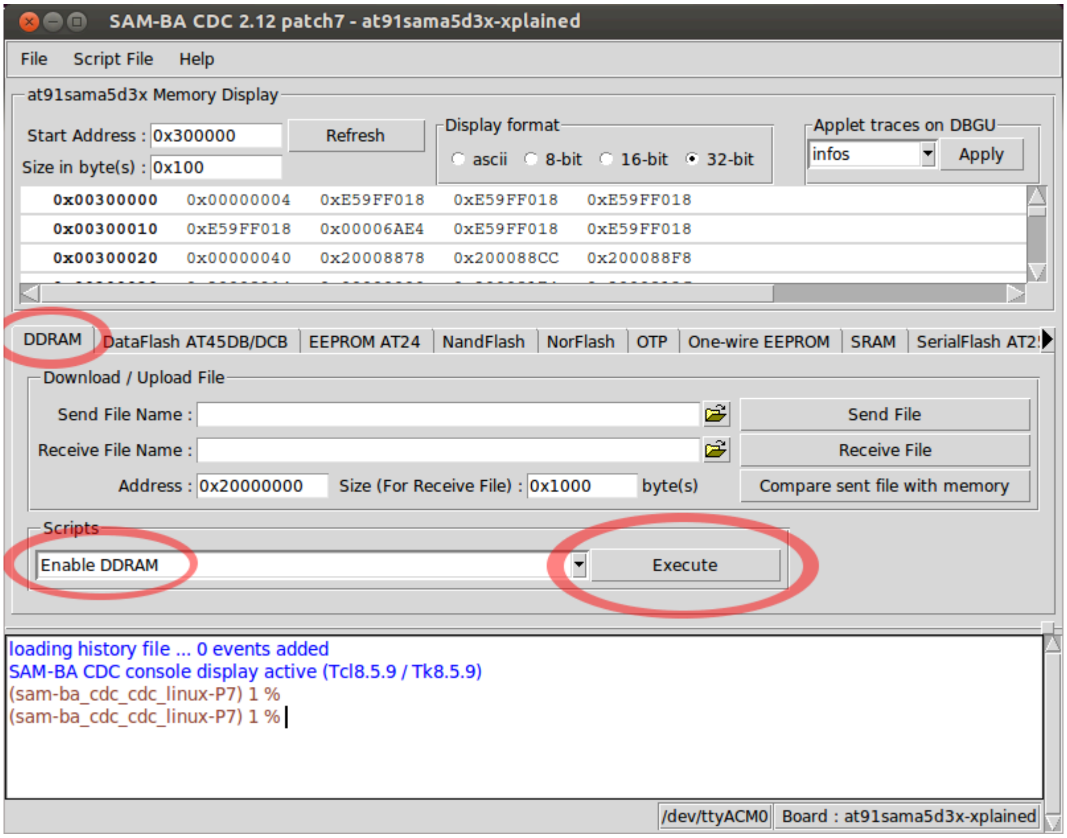
JLink TimeoutMultiplier : 0

□ Customize lowlevel

5. Re-connect (close) JP5 `NAND CS`
6. Select `DDRAM` tab

- In “Scripts” select `Enable DDRAM` and hit `Execute`

Figure 280.2. Enabling DDRAM



Start Address : 0x300000

Size in byte(s) : 0x100

○ ascii ○ 8-bit ○ 16-bit ● 32-bit

infos

```
0x00300000   0x00000004   0xE59FF018   0xE59FF018   0xE59FF018
0x00300010   0xE59FF018   0x00006AE4   0xE59FF018   0xE59FF018
0x00300020   0x00000040   0x20008878   0x200088CC   0x200088F8
```

Send File Name :

Receive File Name :

Address : 0x20000000   Size (For Receive File) : 0x1000 byte(s)

Enable DDRAM

```
loading history file ... 0 events added
SAM-BA CDC console display active (Tcl8.5.9 / Tk8.5.9)
(sam-ba_cdc_cdc_linux-P7) 1 %
(sam-ba_cdc_cdc_linux-P7) 1 %
```

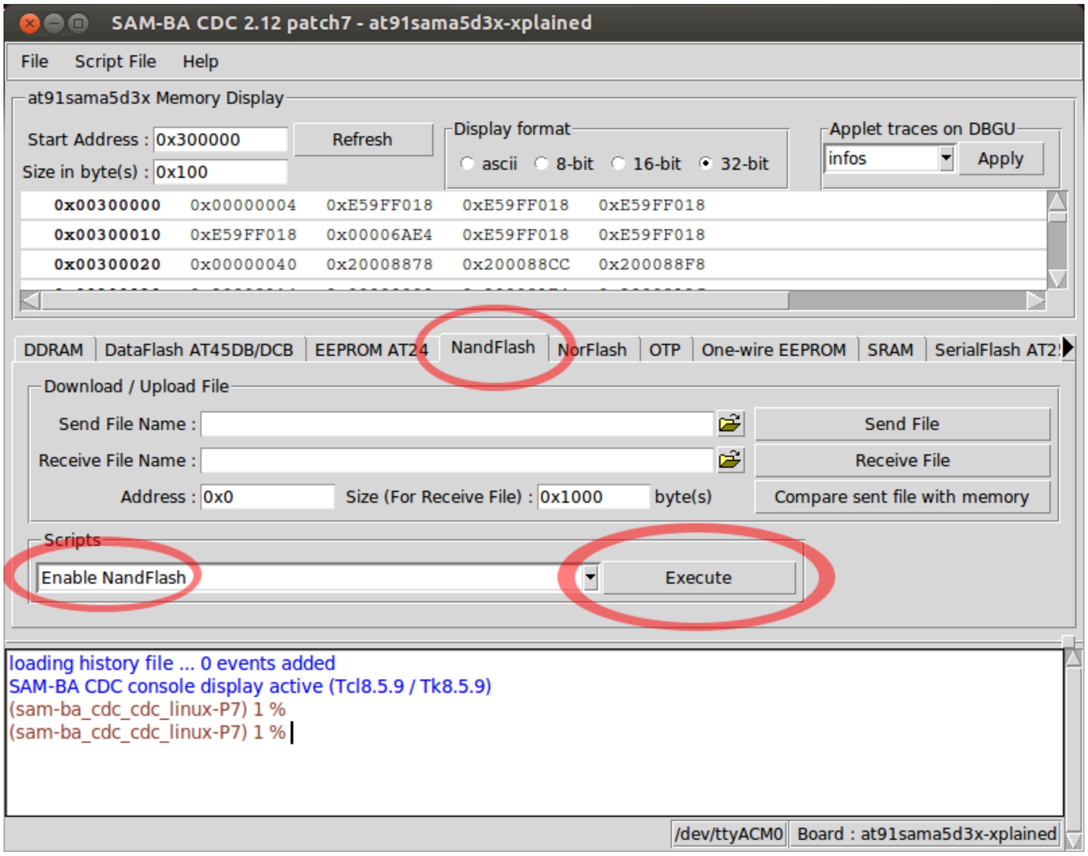
/dev/ttyACM0   Board : at91sama5d3x-xplained

7. Select `NandFlash` tab

- In “Scripts” select `Enable NandFlash` and hit `Execute`

- In “Scripts” select `Enable OS PMECC parameters` and hit `Execute`

Figure 280.3. Enabling NAND



Start Address : 0x300000

Size in byte(s) : 0x100

Display format: ○ ascii ○ 8-bit ○ 16-bit ● 32-bit

Applet traces on DBGU: infos

```
0x00300000   0x00000004   0xE59FF018   0xE59FF018   0xE59FF018
0x00300010   0xE59FF018   0x00006AE4   0xE59FF018   0xE59FF018
0x00300020   0x00000040   0x20008878   0x200088CC   0x200088F8
```

Address : 0x0   Size (For Receive File) : 0x1000 byte(s)

Scripts: Enable NandFlash

```
loading history file ... 0 events added
SAM-BA CDC console display active (Tcl8.5.9 / Tk8.5.9)
(sam-ba_cdc_cdc_linux-P7) 1 %
(sam-ba_cdc_cdc_linux-P7) 1 %
```

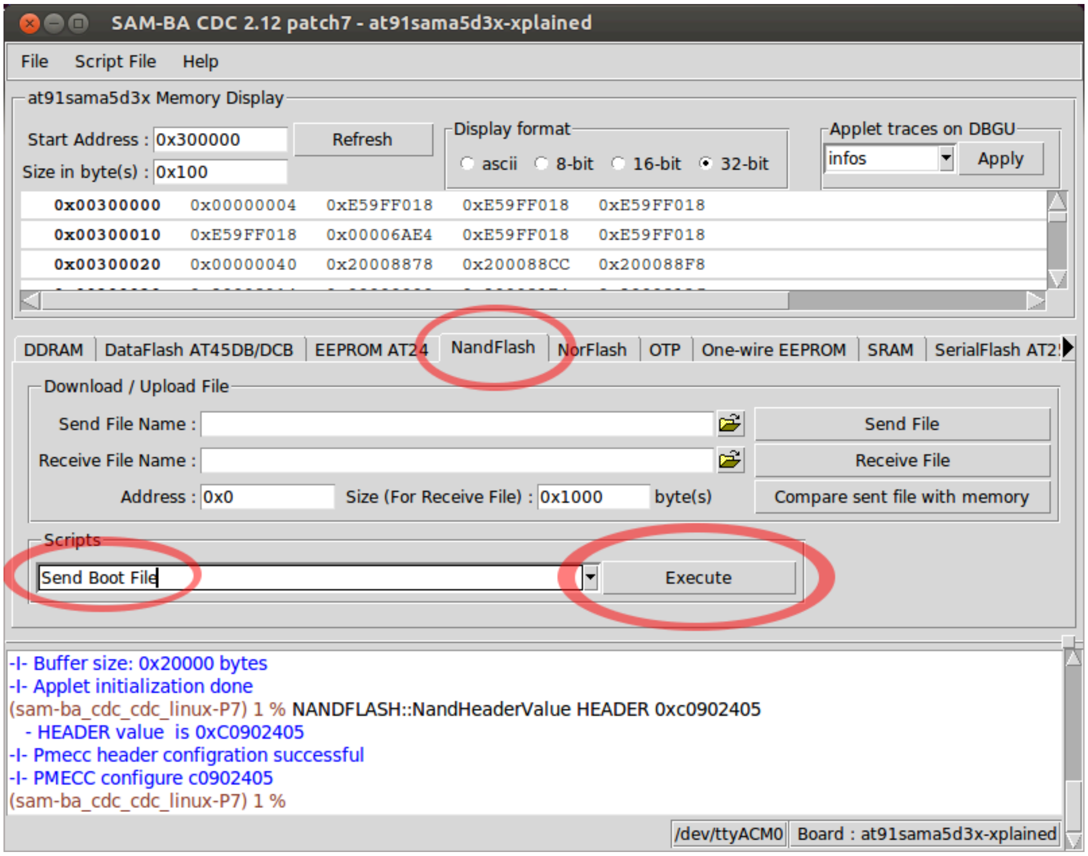
/dev/ttyACM0   Board : at91sama5d3x-xplained

8. Program your choice of second stage bootstrap (BootUp or AT91Bootstrap):

- Still within the `NandFlash` tab, in “Scripts” select `Send Boot File` and hit `Execute`

  - A file selection dialog opens. Find and select the relevant binary (e.g. `bootup.bin`) and hit `Open`.

Figure 280.4. Programming the Second-Stage bootstrap



Start Address : 0x300000

Size in byte(s) : 0x100

Applet traces on DBGU: infos

```
0x00300000  0x00000004  0xE59FF018  0xE59FF018  0xE59FF018
0x00300010  0xE59FF018  0x00006AE4  0xE59FF018  0xE59FF018
0x00300020  0x00000040  0x20008878  0x200088CC  0x200088F8
```

Send File Name :

Receive File Name :

Address : 0x0   Size (For Receive File) : 0x1000 byte(s)

Scripts: Send Boot File

```
-I- Buffer size: 0x20000 bytes
-I- Applet initialization done
(sam-ba_cdc_cdc_linux-P7) 1 % NANDFLASH::NandHeaderValue HEADER 0xc0902405
  - HEADER value  is 0xC0902405
-I- Pmecc header configration successful
-I- PMECC configure c0902405
(sam-ba_cdc_cdc_linux-P7) 1 %
```

/dev/ttyACM0   Board : at91sama5d3x-xplained

9. Program RedBoot (or other application)

   - Still within the `NandFlash` tab, in the “Download/Upload File” area, select the file selection icon 📂 beside the `Send File Name:` field.

     - A file selection dialog opens. Find and select the binary (e.g. `redboot.bin`).

Tip

The `objcopy` tool may be used to convert a built `ROMRAM` eCos application to the binary format required for programming.

```
arm-eabi-objcopy -O binary application.elf application.bin
```

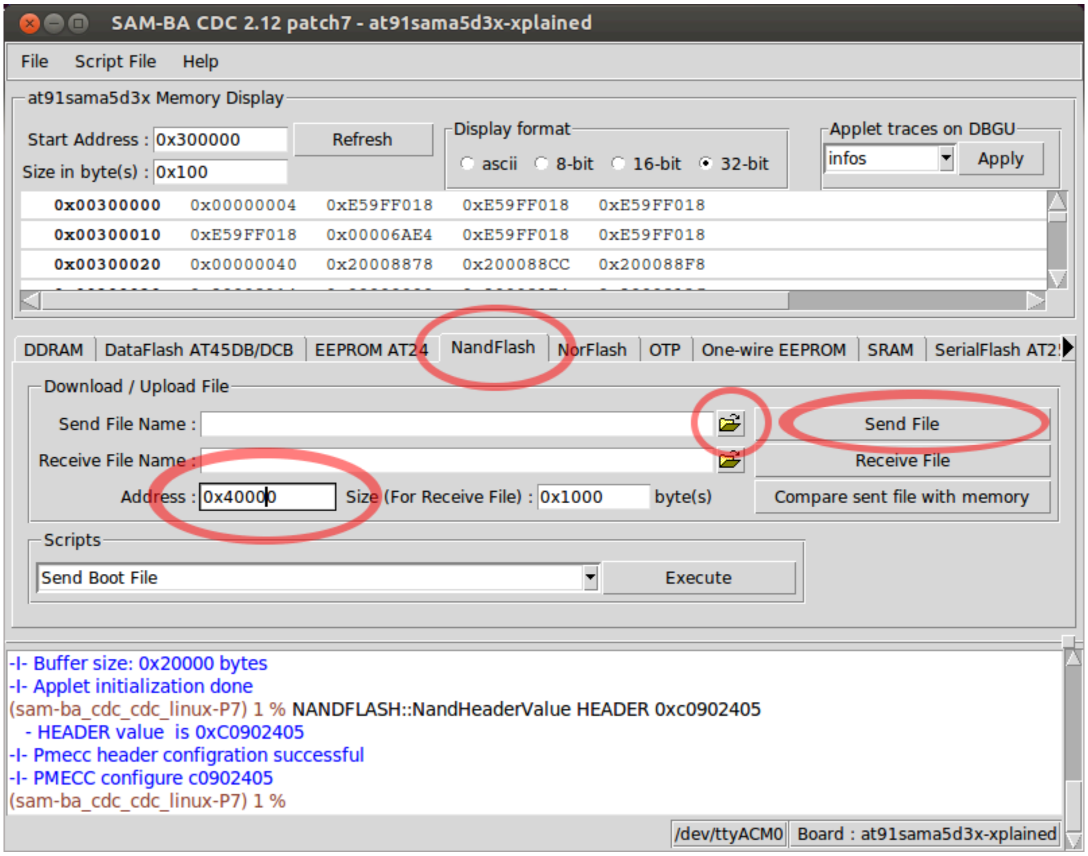
   - In the “Address” field change the address to `0x40000`.

Note

If you are using BootUp with a non-default setting of `CYGNUM_BOOTUP_SAMA5D3_SOURCE_OFFSET`, use that value instead of `0x40000`.

   - Hit the `Send File` button.

Figure 280.5. Programming the Application



Start Address : 0x300000

Size in byte(s) : 0x100

Display format: ○ ascii ○ 8-bit ○ 16-bit ● 32-bit

Applet traces on DBGU: infos

```
0x00300000    0x00000004    0xE59FF018    0xE59FF018    0xE59FF018
0x00300010    0xE59FF018    0x00006AE4    0xE59FF018    0xE59FF018
0x00300020    0x00000040    0x20008878    0x200088CC    0x200088F8
```

Send File Name :

Receive File Name :

Address : 0x40000

Size (For Receive File) : 0x1000 byte(s)

Scripts: Send Boot File

```
-I- Buffer size: 0x20000 bytes
-I- Applet initialization done
(sam-ba_cdc_cdc_linux-P7) 1 % NANDFLASH::NandHeaderValue HEADER 0xc0902405
  - HEADER value  is 0xC0902405
-I- Pmecc header configration successful
-I- PMECC configure c0902405
(sam-ba_cdc_cdc_linux-P7) 1 %
```

/dev/ttyACM0 Board : at91sama5d3x-xplained

10.The SAM-BA application can now be quit

11.Reset the board. The new images will be booted.

Note

With RedBoot, the board will typically take several seconds to become ready as it waits for the network interfaces to initialise. (This may take up to a minute if either or both network interfaces are not connected to a network.) During this time RedBoot will not respond, either to telnet on the configured gdb port (usually 9000), or to input on the TTL debug channel. However the TTL debug channel will emit diagnostic output along these lines:

```
RomBOOT
+NAND: onboard: 1 partition  configured
NAND: Read partition geometry from config store
NAND: onboard: 2 partitions configured
Ethernet eth1: MAC address 0e:00:00:ea:18:f0
IP: 172.20.45.204/255.255.255.0, Gateway: 172.20.45.1
Default server: 172.20.45.29
DNS server IP: 172.20.45.29, DNS domain name: null

RedBoot(tm) bootstrap and debug environment [ROMRAM]
Non-certified release, version UNKNOWN - built 11:43:08, Jan 16 2015
```

Name

Configuration — Platform-specific configuration options

Overview

The SAMA5D3 Xplained platform HAL package is loaded automatically when eCos is configured for a suitable target, e.g. `at-sama5d3xpld`. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The SAMA5D3-XPLD board platform HAL package supports three separate startup types, as documented in the variant Startup documentation. The `ROM` startup type is *not* supported on this platform since the BMS signal configuration does not allow for `EBI_CS#0` memory-mapped execution support.

Note

Normally the board has BootUp or AT91Bootstrap programmed into internal NAND flash within the first two non-factory-bad blocks. When using RedBoot as the loaded application then a `ROMRAM` build of RedBoot is programmed into the next four non-factory-bad blocks. On boot, RedBoot is copied to location 0x20008000 in external RAM. arm-eabi-gdb is then used to load a `RAM` startup application into memory from 0x20100000 and debug it. For such `RAM` applications it is assumed that the hardware has already been initialized by RedBoot. By default the application will use the eCos virtual vectors mechanism to obtain services from RedBoot, including diagnostic output.

Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building the GDB stub ROM (or RedBoot).

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then the serial port will be claimed for HAL diagnostic output.

UART Serial Driver

The SAMA5D3-XPLD board provides a two-pin UART within the on-chip debug unit. This UART is configured as virtual vector communications channel 0 and is typically claimed by RedBoot for use as a console and/or GDB communication.

If this UART is needed by the application, either directly or via the serial driver, then it cannot also be used for GDB communication using the HAL I/O support.

The UART manifests on the board as the TTL-level TXD and RXD pins on J23. Hardware flow control (RTS/CTS) is not supported.

SPI Driver

An SPI bus driver is available for the SAMA5D3 in the package "Atmel AT91 SPI device driver" (`CYGPKG_DEVS_SPI_ARM_AT91`).

There are no on-board SPI devices, so no SPI devices are instantiated by default.

Consult the generic SPI driver API documentation in the eCosPro Reference Manual for further details on SPI support in eCosPro, along with the configuration options in the AT91 SPI device driver.

I²C Driver

Support for SAM I²C (TWI) busses is provided by the "Atmel TWI (I2C) device driver" package (`CYGPKG_DEVS_I2C_ATMEL_TWI`). The SAM variant HAL causes the two buses to be instantiated. These have been tested using external I²C devices.

Onboard NAND

The HAL port includes a low-level driver to access the on-board Micron MT29F2G08 NAND flash memory chip. To enable the driver, add the CYGPKG_IO_NAND package to your eCos configuration.

If using the NAND library direct, see also the NAND port implementation details.

Note

The onboard NAND chip is usually the boot device on this board.

Warning

Before writing to the NAND device from an application, be sure you understand the partition scheme in use and how it relates to the system bootstrap code. Failure to do so risks overwriting the second stage boot loader and/or RedBoot, leaving your board unbootable until you reprogram it.

There are several ways to partition the NAND chip. The eCos configuration setting `CYGSEM_HAL_SAMA5D3_NAND_PARTITION_SCHEME` specifies which one to use. The default NAND configuration, which eCosCentric recommend for development, is "`Config_store`" which uses the eCos on-NAND partition table.

eCos on-NAND partition table

The eCos native partition scheme ("`Config_store`") for this board has the following contents:

- The first 6 non-factory-bad blocks (384k) are reserved for the system bootstrap, and normally contain AT91Bootstrap or BootUp, followed by RedBoot (or an application running in place of it). This number appears in the eCos configuration as `CYGNUM_HAL_SAMA5D3_NAND_RESERVE_BLOCKS_FOR_SYSTEM`.
- NAND partition 0 comprises the next 10 blocks. This is used for the NAND configuration store which records the geometry of the remaining NAND partitions and hosts the RedBoot flash config area. The size of the configuration store may be reconfigured as `CYGNUM_HAL_SAMA5D3_NAND_PARTITION_CONFIG_STORE_SIZE`.
- The remainder of the chip, and the remainder of the partition slots, are available for arbitrary partitioning and use by applications, with their geometry stored dynamically in the configuration store (see below). It is automatically set up as a single partition, numbered 1; up to three partitions are available in a default configuration, but moremay be provisioned if required by reconfiguring `CYGNUM_NAND_MAX_PARTITIONS`.

Warning

If you change `CYGNUM_HAL_SAMA5D3_NAND_RESERVE_BLOCKS_FOR_SYSTEM` or `CYGNUM_HAL_SAMA5D3_NAND_PARTITION_CONFIG_STORE_SIZE`, you must rebuild all images (including RedBoot) that use the changed value and reprogram them at the same time. You should consider erasing and reprogramming the entire NAND array in order to guard against configuration skew situations which are likely to cause data corruption.

The geometry of the remaining partitions is stored as integer entries in the config store, with config store keys named `nand.partition1.base`, `nand.partition1.size`, `nand.partition2.base`, `nand.partition2.size` and so on.

These may be changed from within RedBoot using the `nconfig` command. For example:

```
RedBoot> nand info
NAND device `onboard':
  2048 bytes/page, 64 pages/block, capacity 2048 blocks x 128 kB = 256 MB
  Partition Start Blocks
       0       6     10
       1      16   2036
RedBoot> nconfig put nand.partition1.size uint 200
Written OK
RedBoot> nconfig put nand.partition2.base uint 216
Written OK
RedBoot> nconfig put nand.partition2.size uint 1836
Written OK
RedBoot> reset
... Resetting. RomBOOT

. . . full boot-up messages omitted for brevity . . .

+NAND: onboard: 1 partition  configured
NAND: Read partition geometry from config store
NAND: onboard: 3 partitions configured

. . . full boot-up messages omitted for brevity . . .

NAND: onboard
RedBoot> nand info
NAND device `onboard':
  2048 bytes/page, 64 pages/block, capacity 2048 blocks x 128 kB = 256 MB
  Partition Start Blocks
       0       6     10
       1      16    200
       2     216   1836
RedBoot>
```

Applications may change these via the config store call `cyg_configstore_write_int`; refer to the configuration store documentation for details.

Note

After changing the partition geometry in the config store, it is necessary to reset the board for the changes to take effect - hence the reset command above.

Caution

When changing partition geometry, eCos makes no attempt to preserve any data on the affected partition(s); it is up to you to arrange this if it is important to you. If you do not need to keep the data, consider erasing the affected partition(s) before editing them. RedBoot provides a `nand erase` command for just this purpose.

Manual partitioning

A CDL script which allows the chip to be manually partitioned is provided (see `CYGSEM_HAL_SAMA5D3_NAND_PARTITION_MANUAL`); if you choose to use this, by setting `CYGSEM_HAL_SAMA5D3_NAND_PARTITION_SCHEME` to "`Manual`", the relevant data structures will automatically be set up for you when the device is initialised. By default, the manual config CDL script does not set up any partitions.

Note

The configuration store always uses partition 0 on this board. If you wish to use the configuration store (such as in RedBoot) you must leave a suitable space for it. Do not write any other data to partition 0; it is automatically managed by the config store and will likely be erased.

Warning

The manual partitioning scheme does NOT take account of the space required for system bootstrap. If you choose this option, be sure to allow sufficient space for these in your partition layout.

Other partitioning schemes

It is possible to configure the partitions in some other way, should it be appropriate for your setup, for example to read a Linux-style partition table from the chip. To do so you will have to add appropriate code to `sama5d3xpld_nand.c`, and a new option to `CYGSEM_HAL_SAMA5D3_NAND_PARTITION_SCHEME`.

Ethernet Driver

The SAMA5D3-XPLD board uses the SAMA5D3's internal EMAC and GMAC Ethernet devices attached to external Micrel KSZ8081RNB and KSZ9031RN PHYs respectively. The `CYGPKG_DEVS_ETH_ARM_AT91` package contains all the code necessary to support this device and the platform HAL package contains definitions that customize the driver to the board. This driver is not active until the generic Ethernet support package, `CYGPKG_IO_ETH_DRIVERS`, is included in the configuration.

Both the standard and direct (lwIP only) device drivers are supported. The standard driver is enabled by default; the direct driver can be enabled by setting `CYGOPT_IO_ETH_DRIVERS_LWIP_DRIVER_DIRECT` option. At the time of writing, the direct driver only supports the EMAC (ETH1), and not the GMAC (ETH0).

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the SAMA5D3-XPLD board hardware, and should be read in conjunction with that specification. The SAMA5D3-XPLD platform HAL package complements the ARM architectural HAL and the SAMA5D3 variant HAL. It provides functionality which is specific to the target board.

Startup

In the release view of the world the SAMA5D3 Xplained board boots into the BootUp “application loader” from a suitable on-chip `RomBOOT` supported memory.

When using the second-stage BootUp loader the main (`ROMRAM` startup type) application is then loaded from the configured non-volatile storage (e.g. NAND, SD card) into the DDR2-SDRAM for execution.

The CPU variant bootstrap overview should be read in conjunction with this documentation.

Following a hard or soft reset the HAL will initialize or reinitialize many of the on-chip peripherals. There is an exception for `RAM` startup applications which depend on a ROM monitor for certain services, and so will not attempt to re-initialize the underlying peripheral.

For `ROM`, `ROMRAM` and `SRAM` startups the HAL will perform additional initialization, programming the various internal registers including the PLL, peripheral clocks and GPIO pins as required. The details of the early platform hardware startup may be found in the `plf_hardware_init()` function within the source file `src/sama5d3xpld_misc.c`.

Memory Map

The SAMA5D3 Xplained HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

External RAM
: This is located at address 0x20000000 of the memory space, and is 256MiB long. For `ROM`, `ROMRAM` and `SRAM` applications the initial 32KiB is set aside, primarily for the first level MMU table and (depending on the startup type) the eCos VSR table. The rest of the RAM is then available for application use. For `RAM` startup applications the first 1MiB of RAM is reserved for the “debug” monitor (e.g. RedBoot), and the top `CYGNUM_HAL_COMMON_INTERRUPTS_STACK_SIZE` bytes are reserved for the interrupt stack. The remainder is then available for the `RAM` application.

Internal RAM
: This is located at address 0x00300000 of the memory space, and is 128KiB in size.

On-chip Peripherals
: The I/O is primarily accessible from location 0xF0000000 upwards, though some I/O is mapped into the initial 10MiB of the address space. Descriptions of the contents can be found in the Atmel SAMA5D3 Series Datasheet.

Linker Scripts

The platform linker script defines the following symbol:

hal_mmu_page_directory_base
: This symbol defines where the initialization code will place the level-1 table when initialising the MMU.

Diagnostic LEDs

Two LEDs are fitted onto the CPU Module for diagnostic purposes, one red and one blue.

The platform HAL header file at `<cyg/hal/plf_io.h>` defines the following convenience function to allow the LEDs to be controlled:

```
extern void hal_sama5d3xpld_led(cyg_uint32 bitmask);
```

The low-order 2-bits of the argument *`bitmask`* correspond to each of the 2 LEDs. The red LED is logically mapped to bit 0, with the blue LED mapped to bit 1.

Real-time characterization

The tm_basic kernel test gives statistics gathered about the real-time characterization and performance of the kernel. The sample output is shown here for information. The test was built for RAM startup, where both code and data are using the external DDR2-SDRAM.

Example 280.1. sama5d3xpld Real-time characterization

```
             Startup, main thrd : stack used   388 size  1792
             Startup :  Interrupt stack used  4096 size  4096
             Startup : Idlethread stack used    96 size  1280

eCos Kernel Timings
Notes: all times are in microseconds (.000001) unless otherwise stated

Reading the hardware clock takes 1 'ticks' overhead
... this value will be factored out of all other measurements
Clock interrupt took    2.97 microseconds (24 raw clock ticks)

Testing parameters:
   Clock samples:            32
   Threads:                  64
   Thread switches:         128
   Mutexes:                  32
   Mailboxes:                32
   Semaphores:               32
   Scheduler operations:    128
   Counters:                 32
   Flags:                    32
   Alarms:                   32
   Stack Size:             1088


                                 Confidence
     Ave     Min     Max     Var  Ave  Min  Function
  ======  ======  ======  ====== ========== ========
    1.60    1.09    5.09    0.29   65%  34% Create thread
    0.18    0.12    1.21    0.08   96%  65% Yield thread [all suspended]
    0.21    0.12    1.21    0.09   87%  53% Suspend [suspended] thread
    0.17    0.12    1.33    0.08   95%  75% Resume thread
    0.27    0.12    2.30    0.08   84%  10% Set priority
    0.02    0.00    0.48    0.04   87%  87% Get priority
    0.64    0.48    5.33    0.19   93%  82% Kill [suspended] thread
    0.18    0.12    1.09    0.07   98%  65% Yield [no other] thread
    0.36    0.24    1.82    0.09   48%  39% Resume [suspended low prio] thread
    0.17    0.12    1.33    0.07   96%  75% Resume [runnable low prio] thread
    0.25    0.12    1.82    0.06   75%  18% Suspend [runnable] thread
    0.18    0.12    1.09    0.08   96%  65% Yield [only low prio] thread
    0.15    0.12    0.97    0.05   82%  82% Suspend [runnable->not runnable]
    0.58    0.48    4.36    0.14   93%  93% Kill [runnable] thread
    0.56    0.48    3.15    0.11   95%  78% Destroy [dead] thread
    0.96    0.85    5.21    0.17   93%  89% Destroy [runnable] thread
    1.51    1.09    6.18    0.26   68%  48% Resume [high priority] thread
    0.40    0.36    1.21    0.05   74%  74% Thread switch

    0.02    0.00    0.24    0.03   87%  87% Scheduler lock
    0.11     0.00     0.24     0.01    92%   6% Scheduler unlock [0 threads]
    0.12     0.00     0.36     0.01    92%   6% Scheduler unlock [1 suspended]
    0.12     0.00     0.24     0.02    89%   7% Scheduler unlock [many suspended]
    0.12     0.00     0.36     0.01    92%   5% Scheduler unlock [many low prio]

    0.07     0.00     0.73     0.08    96%  59% Init mutex
    0.24     0.12     1.82     0.11    46%  46% Lock [unlocked] mutex
    0.27     0.12     2.06     0.14    56%  90% Unlock [locked] mutex
    0.22     0.12     1.21     0.12    90%  90% Trylock [unlocked] mutex
    0.19     0.12     1.45     0.11    90%  78% Trylock [locked] mutex
    0.03     0.00     0.24     0.04    81%  81% Destroy mutex
    1.25     1.09     2.79     0.10    90%   6% Unlock/Lock mutex

    0.19     0.00     1.09     0.11    78%  12% Create mbox
    0.03     0.00     0.73     0.05    90%  90% Peek [empty] mbox
    0.33     0.24     1.82     0.14    87%  87% Put [first] mbox
    0.02     0.00     0.61     0.04    96%  96% Peek [1 msg] mbox
    0.31     0.24     1.70     0.12    93%  93% Put [second] mbox
    0.02     0.00     0.48     0.03    96%  96% Peek [2 msgs] mbox
    0.29     0.12     1.82     0.12    81%  12% Get [first] mbox
    0.28     0.12     1.58     0.10    84%  12% Get [second] mbox
    0.28     0.12     1.33     0.10    81%  12% Tryput [first] mbox
    0.22     0.12     0.97     0.09    43%  43% Peek item [non-empty] mbox
    0.25     0.12     1.45     0.09    59%  31% Tryget [non-empty] mbox
    0.22     0.12     1.21     0.09    50%  46% Peek item [empty] mbox
    0.21     0.12     1.21     0.09    96%  50% Tryget [empty] mbox
    0.02     0.00     0.36     0.04    90%  90% Waiting to get mbox
    0.02     0.00     0.48     0.04    90%  90% Waiting to put mbox
    0.09     0.00     1.58     0.12    93%  68% Delete mbox
    0.95     0.73     3.15     0.17    90%  78% Put/Get mbox

    0.05     0.00     0.85     0.08    93%  84% Init semaphore
    0.17     0.12     1.09     0.09    93%  81% Post [0] semaphore
    0.23     0.12     1.21     0.10    46%  43% Wait [1] semaphore
    0.16     0.12     0.97     0.06    90%  90% Trywait [0] semaphore
    0.14     0.12     0.48     0.03    93%  93% Trywait [1] semaphore
    0.05     0.00     0.61     0.07    75%  75% Peek semaphore
    0.03     0.00     0.73     0.05    90%  90% Destroy semaphore
    0.87     0.73     2.55     0.11    68%  28% Post/Wait semaphore

    0.14     0.00     0.97     0.08    65%  18% Create counter
    0.04     0.00     0.48     0.06    75%  75% Get counter value
    0.02     0.00     0.24     0.03    90%  90% Set counter value
    0.22     0.12     0.73     0.07    56%  37% Tick counter
    0.03     0.00     0.73     0.06    87%  87% Delete counter

    0.03     0.00     0.61     0.06    84%  84% Init flag
    0.19     0.12     1.45     0.10    96%  78% Destroy flag
    0.15     0.12     0.73     0.05    87%  87% Mask bits in flag
    0.19     0.12     0.97     0.09    93%  62% Set bits in flag [no waiters]
    0.28     0.12     2.18     0.12    75%  93% Wait for flag [AND]
    0.30     0.24     1.70     0.10    96%  84% Wait for flag [OR]
    0.29     0.24     1.70     0.09    96%  96% Wait for flag [AND/CLR]
    0.28     0.12     1.94     0.10    84%  12% Wait for flag [OR/CLR]
    0.00     0.00     0.12     0.01    96%  96% Peek on flag

    0.17     0.00     1.21     0.12    75%  18% Create alarm
    0.31     0.12     2.42     0.15    84%  81% Initialize alarm
    0.16     0.12     0.97     0.06    90%  90% Disable alarm
    0.31     0.12     2.55     0.14    93%  93% Enable alarm
    0.19     0.12     1.09     0.09    93%  71% Delete alarm
    0.24     0.12     0.61     0.03    84%  12% Tick counter [1 alarm]
    1.02     0.97     1.70     0.07    81%  81% Tick counter [many alarms]
    0.40     0.24     1.70     0.09    90%   6% Tick & fire counter [1 alarm]
    5.85     5.70     7.15     0.09    78%  15% Tick & fire counters [>1 together]
    1.19     1.09     2.55     0.12    93%  56% Tick & fire counters [>1 separately]
    2.31     2.30     3.64     0.02    99%  99% Alarm latency [0 threads]
    2.54    2.30    3.27    0.12   50%  18% Alarm latency [2 threads]
    2.82    2.55    3.88    0.16   70%  35% Alarm latency [many threads]
    3.07    3.03    7.39    0.07   97%  97% Alarm -> thread resume latency

    1.02    0.97    2.18    0.00            Clock/interrupt latency

    0.68    0.48    1.45    0.00            Clock DSR latency

     221     136     264                    Worker thread stack used (stack size 1088)
            All done, main thrd : stack used   988 size  1792
            All done :  Interrupt stack used   156 size  4096
            All done : Idlethread stack used   240 size  1280

Timing complete - 29800 ms total

PASS:<Basic timing OK>
EXIT:<done>
```

Data integrity on the on-board NAND

The HAL port only provides ECC protection for the *main area* of the on-board MT29F2G08 NAND array. (This is achieved using the SAMA5D3 CPU's internal PMECC unit.)

Applications using the NAND library to store data in the *spare area* of the array should consider whether they ought to employ ECC or comparable protection for the data stored there. The recommended correction capacity for this part is 4 bits per quarter-page; refer to the MT29F2G08 datasheet for full details.

Caution

The consequences of insufficient ECC protection are difficult to predict but are likely to include data corruption, undetected by the driver, at a higher rate than expected.

Name

BootUp Integration — Detail

BootUp

The BootUp support for this platform is primarily implemented in the `sama5d3xpld_misc.c` file. The functions are only included when the `CYGPKG_BOOTUP` package is being used to construct the actual BootUp loader binary.

The BootUp code is designed to be very simple, and it is envisaged that once an implementation has been defined the binary will only need to be installed onto a device once. Its only purpose is to allow the startup of the main `ROMRAM` application.

This platform specific documentation should be read in conjunction with the generic BootUp package documentation.

The BootUp package provides a basic but fully functional implementation for the platform. It is envisaged that the developer will customize and further extend the platform side support to meet their specific application identification and update requirements.

The BootUp binary can be installed on *any* SAMA5D3x bootable media, and is not restricted to being placed into NAND flash.

On execution BootUp will copy the `ROMRAM` configured final application from its Non-Volatile-Memory (NVM) location. The configuration option `CYGIMP_BOOTUP_SAMA5D3_SOURCE` selects where the second-level BootUp code will look for the final application image. At present only NAND Flash is supported.

The SAMA5D3x-MB (MotherBoard) documentation provides more detail about the SAMA5D3 BootUp world, including secure boot functionality.

Building BootUp

Building a BootUp loader image is most conveniently done at the command line. The steps needed to rebuild the `SRAM` version of BootUp are:

```
$ mkdir bootup_SRAM
$ cd bootup_SRAM
$ ecosconfig new sama5d3xpld minimal
$ ecosconfig import $ECOS_REPOSITORY/packages/hal/arm/cortexa/sama5d3/sama5d3xpld/current/misc/bootup_SRAM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

The resulting `install/bin/bootup.bin` binary can then be programmed into a suitable non-volatile memory as supported by the SAMA5D3 on-chip `RomBOOT`. e.g. NAND Flash.

The example `bootup_SRAM.ecm` is configured to expect to find the `ROMRAM` application stored in the NAND Flash at offset `CYGNUM_BOOTUP_SAMA5D3_SOURCE_OFFSET`. It is also configured with application encryption support (`CYGFUN_BOOTUP_SAMA5D3_SOURCE_SECURE` option) to allow decryption of the stored application to its final RAM destination.

Note

The application image to be loaded does *not* need to be encrypted. The BootUp code uses binary identity markers to check for the presence of a plaintext (unencrypted) image; if not present, a check for encrypted markers is made.

Whether or not encryption is in use, the application image must be installed to offset `CYGNUM_BOOTUP_SAMA5D3_SOURCE_OFFSET` in the source NVM.

Installing BootUp

How the second-level BootUp loader is placed into bootable memory in a production environment is beyond the scope of this document. However, for the `sama5d3xpld` target platform, several solutions are available.

The simplest may be to use the on-chip SAM-BA support via a USB CDC-ACM host connnection to `J6`. This procedure is documented in the Setup section, which may also be used to install your choice of `ROMRAM` eCos application onto the target hardware.

Alternatively, a suitable hardware debugger (JTAG) configuration could directly update the relevant memories via suitable host-based debugger software features, or configuration script sequence. At its simplest the JTAG interface could be used to load a `ROMRAM` application that performs the necessary update of the boot NVM memory, either from embedding the required binaries in the application image, or (ideally) loading data from a suitable network location via the Ethernet interfaces `J12`/`J13`.

Chapter 281. Raspberry Pi Board Support

Name

eCos Support for the Raspberry Pi Board Family — Overview

Description

The eCosPro release for the Raspberry Pi family is free to use for non-commercial, educational and evaluation purposes. If you wish to use it within a commercial product then a license from eCosCentric is required. Please see the eCosPro license page for details.

This document covers the configuration and usage of eCos and RedBoot on the Raspberry Pi Family of Boards. eCos should be functional on all current members of the family and has been tested on the following variants: Pi Zero, Pi Zero W, Pi 1 Model B, Pi 2 Model B, Pi 3 Model B, Pi 3 Model B+, Pi 3 Model A+, Pi Compute Module 1 and Pi Compute Module 3 (standard, lite and plus versions). These boards are fitted with one of three variants of the Broadcom BCM283X System-on-Chip devices.

In addition to the BCM283X, the board contains 256MiB to 1GiB of SDRAM main memory, an SD card socket, and an optional LAN951X or LAN7515 USB hub/Ethernet device. Variants of the RPi may have a CYW43438 WiFi/Bluetooth device, and may have an eMMC device in place of the SD socket. All have a GPIO header giving access to various IO pins, including GPIO, UART, SPI and I^{2}C. For details of which devices are available on which board variants, see the Raspberry Pi Foundation website, or the Raspberry Pi Wikipedia page.

eCos applications can be developed either using a JTAG-based hardware debugger or by use of the RedBoot ROM monitor. eCos applications are deployed booting directly from the Raspberry Pi SD card, or in the case of the compute modules, from the eMMC.

For RedBoot-based development, the RedBoot image is programmed onto an SD card, or into eMMC, and the board boots directly into RedBoot from reset. RedBoot incorporates a gdb stub that enables eCos applications to be downloaded and debugged via the host-based gdb debugger. You can connect gdb to the board either via a serial line or over Ethernet. Note that applications that make use of USB-based peripherals (including Ethernet) are limited to the serial connection as the eCos USB functionality cannot be shared with RedBoot.

Support for SMP operation of the four CPUs in the RPi2 and RPi3 variants is available, although debugging support is restricted to use of an external JTAG debugger, such as the Lauterbach TRACE32. There is no SMP debug support in RedBoot.

This documentation is expected to be read in conjunction with the BCM283X variant HAL documentation and further device support and subsystems are described and documented there.

Note that the Raspberry Pi may be also be referred to using the abbreviation "RPi" within the documentation.

Supported Hardware

The following devices are currently supported by the Raspberry Pi port. These apply to all RPi boards unless stated otherwise.

UART
: Auxiliary mini UART only, RX and TX lines only connected to GPIO14 and GPIO15 on the GPIO header. Used by RedBoot for communication with the user and GDB. The PL011 UART1 is not currently supported and is reserved for future use with Bluetooth on those boards that are so equipped.

SPI
: SPI0 device only. The auxiliary SPI1 and SPI2 devices are not supported. SPI0 is mapped to GPIO7 to GPIO11 on the GPIO header.

I^{2}C
: I2C1 using the BSC1 controller only. I2C1 uses the GPIO2 and GPIO3 pins on the GPIO header. BSC0 is reserved for use by the GPU, and BSC2 is dedicated to the HDMI interface.

PWM
: Support for simple use of the PWM device is available via the PWM API package. The PWM device ("pwm0") has two channels. Channel 0 is connected to GPIO pins 12 and 18, pins 12 and 32 on the GPIO header. Channel 1 is connected to GPIO pins 13 and 19, pins 33 and 35 on the GPIO header.

USB
USB host support is provided by the USB protocol stack and the Synopsys DWC host controller driver. The stack also provides support for the hub component of the LAN951X/LAN7515 device. The stack currently only provides Control and Bulk endpoint support, at High, Fast and Low speeds.

Ethernet
For those boards equipped with a LAN951X/LAN7515, the Ethernet driver provides support for both the BSD and LWIP protocol stacks.

WiFi
For those RPi boards equipped with a suitable 802.11 chipset the hardware specific `CYGPKG_NET_WIFI_BROADCOM_WWD` driver package, in conjunction with the generic `CYGPKG_NET_WLAN` package, provides support for the LWIP protocol stack.

The wireless network support has been tested against the RPi3B (Raspberry Pi 3 Model B V1.2), RPi3B+ (Raspberry Pi 3 Model B+ 2017) and RPi0W (Raspberry Pi Zero W V1.1) platforms.

The majority of testing involves the standard `CYGPKG_NET_NETTEST` package network tests using the lwIP (`CYGPKG_NET_LWIP`) TCP/IP stack. The eCosPro WLAN (`CYGPKG_NET_WLAN`) provides the settings for the default wireless network connection settings used, as well as some wireless specific tests.

MMC/SD
A driver for the Broadcom SDHOST MMC/SD controller provides access to either cards in the SD socket or, in the compute modules, the eMMC device. The FAT filesystem package provides access to files on these devices.

GPIO
A set of macros in the BCM283X variant HAL provide support for controlling all GPIO pins, including setting pin modes and selecting alternate functions. GPIO interrupts are decoded to provide a separate vector for each pin.

JTAG
On booting RedBoot, and ROM startup eCos applications, the HAL sets a group of GPIO pins to their JTAG alternate functions. This allows an external JTAG debugger to connect and load an application for development. RedBoot also contains a JTAG command that allows the JTAG pin mappings to be changed at runtime.

In general, devices (Caches, GPIO, UARTs) are initialized only as far as is necessary for eCos to run. Other devices (I^{2}C, SPI, MMC/SD etc.) are not touched unless the appropriate driver is loaded, although in some cases, the HAL boot sequence may set up the appropriate pin alternate configuration.

Tools

The board support is intended to work with GNU tools configured for an arm-eabi target. The original port was undertaken using arm-eabi-gcc version 4.7.3, arm-eabi-gdb version 7.2, and binutils version 2.23.

Name

Setup — Preparing the Raspberry Pi for eCos Development

Overview

In a typical development environment, the board boots from the SD/eMMC and runs the RedBoot ROM monitor from SDRAM. eCos applications are configured for RAM startup and then downloaded and run on the board via the debugger **arm-eabi-gdb**. Preparing the board therefore usually involves programming a suitable RedBoot image into an SD card.

Initial Installation

A Raspberry Pi boots initially to the GPU which then reads and executes a second-level GPU bootstrap image from the SD card. This then reads an ARM executable image from the card into SDRAM and executes it. For eCos development we want RedBoot to be the ARM image loaded from the SD card.

The SD card used to boot RedBoot is prepared in exactly the same way as a card is prepared to boot Raspian or any other operating system for the RPi. A complete disk image must be written to the card, typically from a host system running either Windows or Linux. You will need a 2GB, or larger, SD card for this purpose.

A prebuilt bootable SD card image is supplied as part of the eCosPro release. This image is universal and will boot RedBoot on all Raspberry Pi variants. The `eCosPro_RedBoot_RPi_2Mb.img` image is located within your eCosPro installation in the `eCosPro/ecos-`*`VERSION`*`/loaders/rpi` directory. This RedBoot uses 2Mb/s as its default serial speed.

A few non-FTDI based USB to serial converters have been found to be incapable of reliably handling the high-speed 2Mb/s default baud rate used by the standard RedBoot image. For these systems an alternative `eCosPro_RedBoot_RPi_115k.img` image has been provided in the same location. This defaults to a baud rate of 115200, but is otherwise identical. This may also be useful if you encounter any communications issues with slow host machines, or with specific operations such as x/ymodem transfers. Alternatively you could dynamically change the communications speed used by RedBoot with its **baudrate** command.

Installation from Windows to an SD card

The following steps describe how to create a bootable RedBoot SD card for your Raspberry Pi. The best Windows-based tool for this purpose is the freely available **balenaEtcher** SD card image burning utility (previously and often still referred to as Etcher). This is also the recommended utility for writing Raspian and other RPi Linux images.

Visit www.balena.io/etcher to download and install the **balenaEtcher** SD Burner utility.

1. Insert the SD card to be programmed into the PC's card reader.
2. Start the **balenaEtcher** utility and click *Select image*, then browse to and select the RedBoot Raspberry Pi image from within the `loaders/rpi` directory.
3. Click *Select drive* and select your SD card drive. Note that **balenaEtcher** may have already pre-selected the SD drive. You can click on *Change* if this is not the correct drive.
4. Click *Flash!* to write the image to the SD card. You'll see a progress bar that tells you how much is left to do. Once complete, the utility will automatically unmount the SD card so it's safe to immediately remove it from the computer.

Installation from Linux to an SD card

Installation on Linux should be done using the **dd** command.

1. Locate a Linux PC with an SD card reader and insert the SD card to be programmed into the reader.

2. Identify the device name that Linux has assigned to the SD card. This can be done either by monitoring the system log, or by using lsblk before and after inserting the card. In this example we will use `/dev/sdX` as a placeholder.

3. Execute the following command in a shell:

   ```
   $ dd status=progress if=/path/to/eCosPro_RedBoot_RPi_2Mb.img of=/dev/sdX
   ```

   Depending on permissions and user group membership, it may be necessary to prefix this command with **sudo** or run it in a root shell.

4. Execute the **sync** command. This may not be strictly necessary, but it will ensure that all data is safely written to the card.

5. The card is now ready and may be removed.

Installation from Windows to Compute Module eMMC

Installation to the compute module eMMC requires the installation of a driver and a boot utility tool.

1. Download and run the Windows Installer to install the drivers and boot tool.

2. Follow the instructions on the Raspberry Pi website for installing the driver. Make sure the J4 jumper is in the EN position.

3. Run the **RPiBoot.exe** tool. The compute module will then appear as a USB mass storage disk drive under Windows.

4. You can now use the **balenaEtcher** tool to write the RedBoot image to the drive as described above for an SD card.

5. Once complete, the USB cable can be detached. Power off the CMIO board and move J4 back to its original position. On re-applying power, RedBoot will start.

Installation from Linux to Compute Module eMMC

To program an image from Linux it is necessary to download and compile the **rpiboot** utility program.

1. The utility program is in a github repository, so ensure that git is installed. The program also needs libusb-1, so ensure that this is also installed.

2. Clone the git repository into a convenient directory.

   ```
   $ git clone --depth=1 https://github.com/raspberrypi/usbboot
   Cloning into 'usbboot'...
   remote: Counting objects: 21, done.
   remote: Compressing objects: 100% (19/19), done.
   remote: Total 21 (delta 1), reused 14 (delta 0), pack-reused 0
   Unpacking objects: 100% (21/21), done.
   $
   ```

3. Build the tool:

   ```
   $ cd usbboot
   $ make
   cc -Wall -Wextra -g -o rpiboot main.c -lusb-1.0
   $
   ```

4. Move J4 on the compute module IO board to the EN position. Connect a USB cable between the USB SLAVE port and the host and power up the board.

5. Run the rpiboot just built from the build directory:

   ```
   $ sudo ./rpiboot
   Waiting for BCM2835/6/7
   ```

```
Sending bootcode.bin
Successful read 4 bytes
Waiting for BCM2835/6/7
Second stage boot server
File read: start.elf
Second stage boot server done
$
```

6. The CM board will now be visible to Linux as a new USB mass storage device. Identify the device name that Linux has assigned to the SD card. This can be done either by monitoring the system log, or by using lsblk before and after inserting the card. In this example we will use `/dev/sdx` as a placeholder.

7. Execute the following command in a shell:

   ```
   $ dd status=progress if=/path/to/eCosPro_RedBoot_RPi_2Mb.img of=/dev/sdx
   ```

   Depending on permissions and user group membership, it may be necessary to prefix this command with **sudo** or run it in a root shell.

8. Execute the **sync** command. This may not be strictly necessary, but it will ensure that all data is safely written to the eMMC.

9. Remove power from the compute module, move J4 back to its original position and detach the USB cable from the USB SLAVE port.

10. Restore power to the compute module and RedBoot should boot.

RedBoot Startup

Once the RedBoot image has been flashed, insert the SD card into your RPi board's SD socket. If the board has an Ethernet socket insert an Ethernet cable. Connect an RS232 TTL transceiver to the Raspberry Pi's UART pins. On the host run a terminal emulator and connect to the transceiver's serial port, setting it to 2Mb/s (2000000 baud), 8 data bits, no parity, one stop bit (8N1).

Note

Suitable low cost USB RS232 TTL transceivers include the FTDI Raspberry Pi Cable: TTL-232R-RPI or FTDI LC234X module. The FTDI cable can be connected directly to the Raspberry Pi, whereas connecting the module will reqire set of additional jumper cables. These FTDI products are fully supported by drivers on both Windows and Linux hosts. Whatever transceiver is used, it will need connecting to the Raspberry Pi's GND, GPIO14 TXD (transceiver RXD) and GPIO15 RXD (transceiver TXD) pins on the GPIO header. For the 40 way expansion header on standard RPi's these are pins 6, 8 and 10 respectively. We do NOT recommend connecting any other signals that the transceiver might provide, in particular 3V3 or 5V lines.

Note

There are a wide range of freely available terminal emulation programs to choose from - we can recommend Minicom on Linux and PuTTY on Windows.

When the above connections have been made, apply power to the board and similar output to the following should be seen on the terminal:

```
+USB LOG: HCD register DWC ports 1
USB LOG: hub 01 attach on port 1
USB LOG: hub attach 03
USB LOG: hub 03 attach on port 1
USB LOG: hub attach 04
USB LOG: hub 04 attach on port 1
USB LOG: device attach 05 [ff:00] [0424:7800]
... waiting for BOOTP information
Ethernet eth0: MAC address b8:27:eb:30:d0:7b
```

```
IP: 10.0.2.2/255.0.0.0, Gateway: 10.0.0.3
Default server: 0.0.0.0
DNS server IP: 10.0.0.5, DNS domain name: <null>

Mount /dev/mmcsd0/1 on /boot type fatfs:sync=write

RedBoot(tm) bootstrap and debug environment [ROM]
eCosCentric certified release, version v4_2_0 - built 17:03:10, Jun 12 2018

Copyright (C) 2000-2009 Free Software Foundation, Inc.
Copyright (C) 2003-2018 eCosCentric Limited
The RedBoot bootloader is a component of the eCos real-time operating system.
This is free software, covered by the eCosPro Non-Commercial Public License
and eCos Public License. You are welcome to change it and/or distribute copies
of it under certain conditions. Under the license terms, RedBoot's source code
and full license terms must have been made available to you.
Redboot comes with ABSOLUTELY NO WARRANTY.

Platform: Raspberry Pi B3 (BCM2837 - Cortex-A53)
RAM: 0x00000000-0x3e000000 [0x0004eeb0-0x3de00000 available]
  Arena: base 0x3de00000, size 0x200000, 95% free, maxfree 0x1eaf3c
== Executing boot script in 1.000 seconds - enter ^C to abort
RedBoot> fs exec /boot/redboot.txt
RedBoot> jtag 22 23 24 25 26 27
PIN FUNC
 22 ARM_TRST
 23 ARM_RTCK
 24 ARM_TDO
 25 ARM_TCK
 26 ARM_TDI
 27 ARM_TMS
RedBoot> fs exec
RedBoot>
```

Before loading RedBoot, the GPU configures the board according to the contents of the `config.txt` file in the boot partition. eCosPro currently makes minimal use of the settings in this file, the default `config.txt` for eCosPro just sets the GPU reserved memory to its minimum of 16MiB. No other configuration settings have been tested, so change them at your peril. Full details of the file contents can be found in the Raspberry Pi Foundation documentation.

Note

If a RPi board supports Ethernet and you do not connect an Ethernet cable, or the network does not provide a dhcp server, then you will experience a delay before the full RedBoot banner and prompt are displayed.

Resizing the Second SD Partition

The standard eCosPro disk image is sized to fit into a 2GB SD card with two partitions. The first partition is located 4MB from the beginning of the SD card and contains RedBoot and other system files necessary to boot the Raspberry Pi. The second partition follows the first and contains an initially empty VFAT filesystem of 1.23GB in size. If you have a larger card, and wish to utilise all the available storage capacity from eCos, you will need to expand the second partition to incorporate the unused capacity. As an embedded system eCosPro does not have the ability to repartition the disk itself. This needs to be done from a host system running either Windows or Linux.

Resizing from Windows

In Windows you need to run the Windows Disk Management tool to remove the second partition and create the new partition sized as required:

1. To start the tool you can click on the Start Menu and type "disk management", click on the "Create and format hard disk partitions control panel" entry that appears.

2. Remove the second partition (1.23GB FAT32) by right clicking on second partition and selecting "Delete Volume...". Click "Yes" to confirm.

3. Create a new volume of the required size (typically the remaining space on the SD) by right clicking on the "Unallocated" space and selecting "New Simple Volume...".

4. In the New Simple Volume Wizard's initial dialog click "Next >" to move to the "Specify Volume Size" dialog, then choose the disk size required (default is the remaining space on the SD) and click "Next >" again.

5. In the "Assign Drive Letter or Path" dialog choose the drive letter of your choice, typically the default provided and click "Next >".

6. In the final "Format Partition" dialog, choose the file system type - this should be the default "FAT32", then select the allocation size - choose "Default" or 4096 and then enter the "Volume label" of your choice. Leaving the "Perform a quick format" checkbox ticked, click "Next >". Review the settings and then click "Finish".

Your SD card should now contain two partitions. The first is the original boot partition containing RedBoot (or your own application, named as kernel*.img) and the second is a new partition expanded to incorporate the remaining SD card capacity.

Resizing from Linux

There are a variety of Linux tools in different distributions for resizing a partition, many are GUI based and similar to the Windows tool. The following example uses just command line tools and should work on all Linux distributions.

1. Depending on permissions, it may be best to perform the following actions as root. Either start a root shell, or prefix all commands with **sudo**.

2. Insert the SD card into the card reader and identify the device name assigned to it. This can be done either by monitoring the system log, or by using lsblk before and after inserting the card. In this example we will use `/dev/sdX` as a placeholder.

3. Run fdisk on the device and type `p` to print the partition table:

```
# fdisk /dev/sdX

Welcome to fdisk (util-linux 2.28).
Changes will remain in memory only, until you decide to write them.
Be careful before using the write command.


Command (m for help): p
Disk /dev/sdX: 14.9 GiB, 15931539456 bytes, 31116288 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0xec05ec05

Device     Boot  Start     End Sectors  Size Id Type
/dev/sdX1         8192  137215  129024   63M  c W95 FAT32 (LBA)
/dev/sdX2       137216 2715647 2578432  1.2G  c W95 FAT32 (LBA)

Command (m for help):
```

   Check that things like the disk identifier and the partition parameters match the above to ensure you have the right device.

4. Use the `d` command to delete partition 2:

```
Command (m for help): d
Partition number (1,2, default 2): 2

Partition 2 has been deleted.
```

```
Command (m for help):
```

5. Ensuring you set the first sector to 137216, since there is 3MB of free space before the first partition which should NOT be used, use the `n` command to create a new partition. Check that the new partition fills the remaining card space with the `p` command.

```
Command (m for help): n
Partition type
   p   primary (1 primary, 0 extended, 3 free)
   e   extended (container for logical partitions)
Select (default p): p
Partition number (2-4, default 2): 2
First sector (2048-31116287, default 2048): 137216
Last sector, +sectors or +size{K,M,G,T,P} (137216-31116287, default 31116287):

Created a new partition 2 of type 'Linux' and of size 14.8 GiB.

Command (m for help): p
Disk /dev/sdX: 14.9 GiB, 15931539456 bytes, 31116288 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0xec05ec05

Device     Boot  Start      End  Sectors  Size Id Type
/dev/sdX1         8192   137215   129024   63M  c W95 FAT32 (LBA)
/dev/sdX2       137216 31116287 30979072 14.8G 83 Linux

Command (m for help):
```

6. Fdisk has set the new partition's type to `Linux`, use the `t` command to change the type to FAT32:

```
Command (m for help): t
Partition number (1,2, default 2): 2
Partition type (type L to list all types): c

Changed type of partition 'Linux' to 'W95 FAT32 (LBA)'.

Command (m for help):
```

7. Finally use the `w` command to write the partition table back and exit fdisk.

8. The new partition now needs to be formatted with a FAT32 filesystem. This can be done using the **mkfs.fat** command:

```
# mkfs.fat -v -F 32 /dev/sdX2
mkfs.fat 3.0.26 (2014-03-07)
/dev/sdX2 has 64 heads and 32 sectors per track,
hidden sectors 0x21800;
logical sector size is 512,
using 0xf8 media descriptor, with 30979072 sectors;
drive number 0x80;
filesystem has 2 32-bit FATs and 16 sectors per cluster.
FAT size is 15112 sectors, and provides 1934301 clusters.
There are 32 reserved sectors.
Volume ID is e2a0eca8, no volume label.
#
```

9. Execute the **sync** command. This may not be strictly necessary, but it will ensure that all data is safely written to the card.

10. The card is now ready and may be removed.

Installing user applications onto an SD card

If you wish to install a ROM startup application onto an SD to be automatically booted instead of RedBoot, then you can do this by replacing the appropriate kernel image file in the SD card boot partition.

First prepare an SD card as per the Initial Installation section above. The card should then be remounted and your application copied over, replacing the relevant kernel image file(s) within the boot partition. Applications are compiled and linked to ELF format, and need to be converted to binary before copying to the the SD card. This can be done as follows using **objcopy**:

```
$ arm-eabi-objcopy -O binary app.elf kernel.img
```

If you have built the application for the BCM2835 (Pi1/Pi0) based boards, replace `kernel.img`, and for the BCM2836 (Pi2) or BCM2837 (Pi3) based boards, replace `kernel7.img`. Replacing the wrong file may result in the board failing to boot. For widest compatibility you can replace both `kernel.img` and `kernel7.img` with suitably built versions of your application.

Rebuilding RedBoot

Typical users should never need to rebuild RedBoot. If you do intend to modify RedBoot then please note that rebuilding it is currently only supported from the Linux command line.

The following RedBoot configurations are supported:

Configuration	**Description**	**Use**	**File**
ROM_2Mb	RedBoot loaded from SD card to SDRAM, Standard version with default baud rate of 2Mb/s	`redboot_ROM_2Mb.ecm`	`redboot_ROM_2Mb.bin`
ROM_115k	RedBoot loaded from SD card to SDRAM, alternative version with default baud rate of 115200	`redboot_ROM_115k.ecm`	`redboot_ROM_115k.bin`
JTAG	RedBoot loaded via a JTAG debugger, for RedBoot developers only	`redboot_JTAG.ecm`	`redboot_JTAG.bin`

The JTAG configuration is only required if you wish to build a version of RedBoot to load and debug over JTAG, before ultimately creating a final ROM version to be booted from SD.

Note that the use of the term ROM for the initial RedBoot configuration is a historical accident. RedBoot actually runs from SDRAM after being loaded there from the SD card by the GPU bootstrap. The use of ROM for this configuration is intended to indicate that it initializes the microprocessor and board peripherals, as opposed to a RAM configuration which assumes that this has already been done.

The ROM_115k configuration is provided as an alternative to the ROM_2Mb configuration, for compatibility with low performance USB to serial converters. It simply lowers the default baud rate to 115200 and is otherwise identical.

The final stage of the RedBoot build process will create a bootable RedBoot SD image. This is handled within the `eCosPro/ecos-`*`VERSION`*`/packages/hal/arm/cortexa/pi/current/host/buildimg.sh` script. In order for the script to complete successfully, the user must either have sudo privileges to **mount** and **cp**, or the user must be able to **mount** `/tmp/sd_pi` on `/tmp/sd`. For the latter case, the following entry in `/etc/fstab` is advised:

```
/tmp/sd_pi   /tmp/sd   vfat   user,noauto,rw,loop,offset=4194304  0   0
```

The steps needed to rebuild the ROM version of RedBoot are:

```
$ mkdir redboot_rpi2_rom
$ cd redboot_rpi2_rom
$ ecosconfig new raspberry_pi2 redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/arm/cortexa/pi/VERSION/misc/redboot_ROM_2Mb.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

At the end of the build the `install/bin` subdirectory should contain the file `sd_pi.img`. This file is the SD card image and is suitable at this point for booting Pi2 and Pi3 boards, including the CM3, since it only contains a `kernel7.img` file. To make it also capable of booting a Pi1 or Pi0 board repeat the above command sequence with the following changes:

```
$ mkdir redboot_rpi1b_rom
$ cd redboot_rpi1b_rom
$ ecosconfig new raspberry_pi1b redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/arm/cortexa/pi/VERSION/misc/redboot_ROM_2Mb.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

At the end of this build the `install/bin` subdirectory will contain a new `sd_pi.img` file. This should be a copy of the previous image with the addition of `kernel.img`. This image should then be written to an SD card as per the above Linux hosted setup instructions. e.g:

```
sudo dd status=progress if=install/bin/sd_pi.img of=/dev/sdX
```

The SD created should now be able to boot all RPi variants.

To build a RedBoot that uses a different default UART baud rate to the standard 2Mb/s, then edit the `ecos.ecc` file after each **ecosconfig import** stage as described above. Set user_value for both `cdl_option CYGNUM_HAL_VIRTUAL_VECTOR_DEBUG_CHANNEL_BAUD` and `cdl_option CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL_BAUD` to your desired baud rate. For example:

```
cdl_option CYGNUM_HAL_VIRTUAL_VECTOR_DEBUG_CHANNEL_BAUD {
    # Flavor: data
    # No user value, uncomment the following line to provide one.
    user_value 115200
    # value_source default
    # Default value: 2000000
    # Legal values: 9600 19200 38400 57600 115200 230400 460800 500000 576000 921600 1000000 1152000 1500000 2000000 2
};
```

You will then need to use this baud rate when connecting to the new RedBoot via a terminal or gdb debug session. The highest attainable reliable baud rate can depend upon the specific model Raspberry Pi, transceiver and/or host PC/OS used. The default 2MB/s has been found to work reliably across the Pi family members, using FTDI transceivers, connected to either Linux or Windows hosts. Use of other transceivers may require you to reduce the baud rate.

Name

JTAG-based Debugging — Usage

JTAG Hardware Connection

JTAG-based debugging requires direct wiring of the JTAG probe lines to the external GPIO pins of a Raspberry Pi board. The eCosCentric TAP-HAT provides a simple way of doing this. The GPIO pins are not enabled for JTAG operation by default and need to be initialised by RedBoot when the board is booted. See the RedBoot JTAG Command section below for details of how to select and enable the set of pins to be used.

Lauterbach TRACE32

eCosPro includes support for JTAG-based debugging using the Lauterbach **TRACE32** debugger with a Lauterbach Power Debug probe. This is currently the best solution for debugging of eCos SMP applications on the Raspberry Pi. You will need suitable licenses for ARM Architecture-v7, Architecture-v8 and multicore debugging in order to debug an SMP application, or an ARM11 license to debug the single core in an Raspberry Pi0 or Pi1.

The **TRACE32** debugger needs a startup script to initialize it for the Raspberry Pi. Some example scripts are present in the Raspberry Pi platform HAL (i.e. `packages/hal/arm/cortexa/pi/`*`VERSION`*`/misc/trace32`). The file `ecospropi2-qt.cmm` initializes TRACE32 to debug a Raspberry Pi2 using the QT based variant avaliable on some host operating systems. Similarly the file `ecospropi3-qt.cmm` initializes TRACE32 to debug a Raspberry Pi3, Pi2 v1.2, or CM3. Finally, `ecospropi1-qt.cmm` initializes TRACE32 to debug a Raspberry Pi1, Pi0, or CM1. Each of these files additionally loads `layout1s-qt.cmm` to define the initial window layout and `ecospromenu.cmm` to define an eCosPro menu.

The expectation in these scripts is that all files are present in the same directory, along with the application being debugged. It is recommended that these files be copied out of the source repository into a working directory to which the application can also be copied and that TRACE32 be started from the command line as follows:

```
$ cd /path/to/work/directory
$ t32marm64-qt -s ecospropi3-qt.cmm
```

Raspberry Pi boards cannot be reset from TRACE32, so the safest approach to debugging is to power cycle the board between runs. It is not necessary to exit TRACE32, so long as the eCosPro menu entries to re-attach or load a new application are used, TRACE32 can be re-attach to a reset board.

In addition to attaching to the target, these startup files define an additional `eCosPro` menu in the **TRACE32** GUI. It contains the following entries:

MMU Table List
: This entry causes a window showing the current state of the MMU tables for the current CPU to be displayed. In eCos, all CPUs should be using the same shared table.

Load eCos.t32
: This loads the **TRACE32** eCos RTOS specialization extension. This file should be copied out of the TRACE32 installation into the working directory. Note that depending on the version of TRACE32 in use, the eCos RTOS support may not be fully SMP-aware, so some information it displays in SMP application may be a little misleading. See the Lauterbach documentation on the RTOS debugger for eCos for more details of the functionality available.

Display Threads
: Displays a list of current threads. In recent versions of TRACE32 the current thread on each CPU will be marked RUNNING with the CPU number beside it in parentheses. However, CPU affinity is not currently displayed.

Display Scheduler
: Displays state of scheduler. Only those parts of the scheduler state common between single and multi-core systems will be displayed.

Display Thread Stacks

Displays a summary of stack usage for all threads. This includes the stack limits, current SP, and the maximum amount of stack each thread has used.

Reset Board

Reset the board. This requires an attachment between the SRST pin of the JTAG cable and the RUN pin on the Raspberry Pi board. If you are using an eCosCentric TAP-HAT adaptor, then the two pin header needs to be soldered in to the RUN-PEN holes on the Raspberry Pi board and the micro-probe lead connected between the centre SRST pin and the RUN pin.

(Re-)Attach

After the RPi has been reset, this entry will re-attach TRACE32 to the board. Typically it will be executing in RedBoot.

Disable MMU

This will disable the MMU and Caches, which may be useful during development if an unmapped view of memory is required. Any halted program cannot be restarted once this has been done.

Save Breakpoints

This entry saves the current set of breakpoints to `breakpoints.cmm`. The contents of this file is saved amd reloaded whenever an application file is loaded and allows breakpoint settings to be preserved across restarts of TRACE32. This menu entry provides an alternative way to save the current breakpoints. prior to exiting TRACE32 for example.

Load APP.ELF

This entry disables the caches and MMU, loads the file `app.elf` from the current directory, and loads any breakpoints from `breakpoints.cmm`. This is the menu entry that should be used to load and run SMP applications for development and testing. The `breakpoints.cmm` file allows a set of current breakpoints to be saved using the Store button on the breakpoint list window and have them reloaded automatically each time the application is reloaded.

Load APP.ELF Syms+Bkpts

This entry loads just the symbol tables and debug information from the application and also loads breakpoints from `breakpoints.cmm`. This is useful if the application is already running when TRACE32 is attached. For example if it is a ROM startup application that has been loaded from the SD card, or an SMP startup application that was loaded by RedBoot.

Load REDBOOT.ELF

This does exactly the same thing as the *Load APP.ELF* menu entry except that it loads `redboot.elf`. This has been useful in debugging RedBoot.

Load REDBOOT.ELF Syms +Bkpts

This does exactly the same thing as the *Load APP.ELF Syms+Bkpts* menu entry except that it loads symbold from `redboot.elf`.

Name

Configuration — Platform-specific Configuration Options

Overview

The Raspberry Pi platform HAL package is loaded automatically when eCos is configured for any RPi target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The platform HAL package supports four separate startup types:

RAM
: This startup type is normally used during application development when using the RedBoot ROM monitor. arm-eabi-gdb is used to load the RAM startup application into memory and debug it. It is assumed that the hardware has already been initialized by RedBoot. By default the application will use the eCos virtual vectors mechanism to obtain services from RedBoot, including diagnostic output.

ROM
: This startup type can be used for finished applications which will be booted direct from the SD card. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization. This startup type can also be used for applications loaded via JTAG for testing prior to committing to an SD card.

JTAG
: This startup type can be used for finished applications that are to be loaded by a JTAG debugger. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization. This startup type can also be used for applications loaded via JTAG.

SMP
: This startup type can be used for finished applications that can be loaded into RAM via RedBoot or via a JTAG debugger. The load address is set to the same as for RAM applications, however, the application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization. Once started, this application takes full control of the system and RedBoot will not be called again. This means that debugging via RedBoot will not be possible, only JTAG-based hardware debugging is supported. The intent of this startup type is to allow SMP test programs to be run from RedBoot, most SMP applications should use the ROM startup type.

The main difference between ROM, JTAG and SMP startup types is the address at which they are loaded. ROM applications load at 0x00008000, which is the default address used by the GPU boot loader. JTAG applications load at 0x00100000, leaving the lower 1MiB free for a ROM RedBoot. This is necessary because in multi-core systems, the secondary cores will be looping in code in the RedBoot executable, and we don't want to overwrite that, since it may cause these cores to run wild. SMP applications load at 0x00200000, leaving the lower 2MiB free. This allows either a ROM or JTAG RedBoot to be used to load it without affecting the secondary cores. For the same reasons, RAM applications also load at the 0x00200000 boundary.

RedBoot and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building RedBoot.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then the serial port will be claimed for HAL diagnostics.

UART Serial Driver

The auxiliary mini UART is 16550 compatible and is supported by the CYGPKG_IO_SERIAL_GENERIC_16X5X generic driver package which is modified by the CYGPKG_IO_SERIAL_ARM_BCM283X driver package for the Raspberry Pi family. By default the UART used only the TXD0 and RXD0 pins connected to GPIO14 and GPIO15 on the GPIO header. While RTS and CTS could be made available on GPIO16 and GPIO17, this is not currently supported since these pins are better left for other purposes. The mini UART is also lacking some 16550 functionality, it is only capable of 7 or 8 bit characters, and only supports one stop bit and no parity; attempts to select other settings will be accepted, but will have no effect. Since the TXD0 and RXD0 pins are not at RS232 levels it is necessary to either connect an RS232 transceiver to them or, more usually, an FTDI USB serial convertor module.

SPI Driver

SPI0 is supported by the CYGPKG_DEVS_SPI_ARM_BCM283X. The auxiliary SPI1 and SPI2 devices are not currently supported. By default the MISO, MOSI and SCLK signals are connected to GPIO9, GPIO10 and GPIO11 respectively. The chip select signals are connected to GPIO7 and GPIO8. These pins are managed in GPIO mode, so it is possible to add extra SPI devices by assigning additional GPIO pins to act as chip select lines.

I²C Driver

I2C1 is supported by the CYGPKG_DEVS_I2C_BSC driver. The SDA1 and SCL1 signals are connected to GPIO2 and GPIO3 pins. I2C0 is used by the GPU to operate the ID_SD and ID_SC pins as well as the GPIO extender in the Compute Modules; I2C2 is dedicated to the HDMI interface; so neither of these is available for use.

PWM Driver

PWM is supported by the CYGPKG_DEVS_PWM_BCM driver. The PWM device supports two channels connected to four pins available on the GPIO header. Channel 0 is connected to GPIO12 and GPIO18, pins 12 and 32 on the GPIO header. Channel 1 is connected to GPIO13 and GPIO19, pins 33 and 35 on the GPIO header. Note that some of these may clash with other uses of the same pins, particularly GPIO19 which is also SPI MISO.

USB Support

USB is supported by the CYGPKG_IO_USB package USB protocol stack together with the CYGPKG_DEVS_USB_DWC host controller driver. The stack will recognize and support the USB hub component of the LAN951X/LAN7515 when it is available on the board, together with any hubs that might be plugged in to that. The stack currently only supports CONTROL and BULK transfer types, INTERRUPT and ISOCHRONOUS transfers are not supported. The host controller driver currently supports High, Fast and Low speed devices. USB support is currently oriented towards supporting USB mass storage devices, serial devices including CDC/ACM and FTDI serial adaptors, and the LAN951X/LAN7515 Ethernet component.

Ethernet Driver

Support for the LAN951X/LAN7515 Ethernet component is provided by the CYGPKG_DEVS_ETH_LAN951X driver. This driver is dependent on the USB stack and can only be configured if the USB stack is also present. This driver is also not active until the generic Ethernet support package, CYGPKG_IO_ETH_DRIVERS, is included in the configuration.

The option CYGPKG_DEVS_ETH_LAN951X_SHORT_CIRCUIT controls whether the Ethernet driver uses the Raspberry Pi board type to decide whether the board contains an Ethernet device. If it does not, then initialization of the driver and the entire network stack is prevented, saving startup time. This option can be misled if, for example, a Compute Module is plugged in to a carrier board that has a LAN951X/LAN7515 installed. So by default the option is only enabled for RedBoot builds, and not for eCos builds, where the user is expected to know what they are doing. If the default is not as reqired, it can be changed in the ConfigTool.

WiFi Driver

Support for the wireless network is provided by the CYGPKG_NET_WIFI_BROADCOM_WWD driver. The majority of the wireless configuration is fixed based on the RPi platform configured with no user configuration required. It is enabled by default when the appropriate packages are included in the configuration.

The WiFi chipsets used in the Raspberry Pi platforms require 3rd-party binary firmware images for correct operation. See the WICED specific Chipset Firmware section for details covering the RPi specific images.

The option CYGPKG_HAL_ARM_CORTEXA_PI_WICED_DCT controls whether the WICED Device Configuration Tables (DCT) support is enabled. The enabled functionality can be used by the WICED SDK, and the application world, to hold persistent WiFi configurations.

MMC/SD Driver

Support for the SD/MMC socket is provided by the CYGPKG_DEVS_MMCSD_SDHOST driver. This uses the Broadcom SDHOST controller; the Arasan SDHCI controller is not currently supported, although in future it will be used for the CYW43438 WiFi device on the Pi3 and CM3. On the compute modules, this same driver provides access to the eMMC device.

The MMC/SD bus driver layer (CYGPKG_DEVS_DISK_MMC) is automatically included as part of the hardware-specific configuration for these targets. All that is required to enable MMC/SD support is to include the generic disk I/O infrastructure package (CYGPKG_IO_DISK), along with the intended filesystem, typically, the FAT filesystem (CYGPKG_FS_FAT) and any of its package dependencies (including CYGPKG_LIBC_STRING and CYGPKG_LINUX_COMPAT for FAT).

While earlier variants of the Raspberry Pi had a card detect signal from the SD card socket attached to a GPIO line, this has been dropped in later variants. It also makes no sense for the permanently connected eMMC devices on the compute modules. Therefore, card detection is not currently supported on any RPi boards, so the disk IO layer's removable media support cannot detect card insertion or removal, and the FILEIO layer's automounter cannot currently be used.

GPIO Support

GPIO support is provided by the BCM283X variant HAL , and is documented there. The GPIO API is built around descriptors that encode a GPIO pin number together with its function (IN, OUT or device function) and mode into a single 32 bit value. The mode includes edge and level interrupt recognition and pull up and down settings. The eCosPro subsystem also decoded all GPIO interrupts into separate vectors.

Frequency Control

eCos does not contain any support for dynamic frequency management in the same way that Linux does. The CPUs run at a single frequency throughout. Normally this is the maximum frequency permitted without overclocking. A different frequency may be selected in the configuration, or it may be set at runtime by calling hal_bcm283x_set_freqencies(). See the BCM283X HAL for details.

The option CYGHWR_HAL_ARM_CORTEXA_BCM283X_ARM_FREQ defines the frequency in MHz at which the ARM CPU will run. For RAM startup types, the default value of zero will cause the application to continue using the frequency set up by RedBoot. For other startup types, a value of zero will cause the HAL to select a default frequency based on the board type. A non zero value will force the CPU frequency to the given value.

The option CYGHWR_HAL_ARM_CORTEXA_BCM283X_CORE_FREQ defines the frequency in MHz at which the GPU will run. For RAM startup types, the default value of zero will cause the application to continue using the frequency set up by RedBoot. For other startup types, a value of zero will cause the HAL to select a default frequency based on the board type. A non zero value will force the CPU frequency to the given value.

Compiler Flags

The platform HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos. The following flags are specific to this port:

`-mcpu=arm1176jzf-s`
: The arm-eabi-gcc compiler supports many variants of the ARM architecture. A `-m` option should be used to select the specific variant in use. For the BCM2835 variants that contain a single ARM11 CPU, (Pi0, Pi1, CM1), this option should be selected.

`-mcpu=cortex-a7`
: For the BCM2836, containing 4 Cortex-A7 CPUs and the BCM2837, containing four Cortex-A53 CPUs, this option should be selected. While the Cortex-A53 is strictly a ARM64 CPU, we can only use it in ARM32 mode, where it supports the same instruction set as the Cortex-A7. So, for simplicity we use the same option for both.

`-mthumb`
: The arm-eabi-gcc compiler will compile C and C++ files into the Thumb2 instruction set when this option is used. The best way to build eCos in Thumb mode is to enable the configuration option `CYGHWR_THUMB`.

Name

SMP Support — Usage

Overview

Support is available for SMP operation of the four CPUs available in the Pi2, Pi3 and CM3 variants. However, debugging support is restricted to using an external SMP-aware JTAG debugger like ARM's DS-5 or a Lauterbach Power Debug probe. RedBoot does not have support for multi-core debugging.

A board intended to be used for SMP development should be initialized in the same way as a single core board by installing the same standard RedBoot SD card image.

SMP support is enabled by setting `CYGPKG_KERNEL_SMP_SUPPORT` to true. SMP applications should only be built using either ROM, JTAG or SMP startup types. ROM applications can be loaded from the SD card in place of RedBoot. The SMP startup is identical to a ROM startup except that the load address is set to allow the application to be loaded into a higher location in RAM from RedBoot. All application types may also be loaded via a JTAG debugger.

Loading an SMP startup application via RedBoot can be done from the RedBoot command line via serial. It may also be loaded via a GDB connection on serial. However, once started running the SMP application will take full control of the system, including redirecting all interrupt sources, exception vectors and virtual vector table entries. This means that RedBoot will no-longer be active. Any breakpoints planted by GDB will result in an exception to the application, Ctrl-C will not work, any Ethernet connections will be lost and serial output will come from the application in plain ASCII. Any GDB connection will be lost and GDB may start reporting packet errors.

It is possible to load an SMP startup program via GDB and have its output displayed on the GDB console. To do this set `CYGSEM_HAL_DIAG_MANGLER` to `"GDB"`, and `CYGDBG_HAL_DIAG_TO_DEBUG_CHAN` to `1`. The application will still not be controllable from GDB, but this does simplify the running of test code; avoiding the need to disconnect GDB and connect a terminal emulator to capture or view the output.

SMP development has been tested using the Lauterbach TRACE32 debugger. While DS-5 should also work, given appropriate configuration, this has never been tested. GDB working through a JTAG debugger such as OpenOCD may work, but support for debugging multiple cores is poor.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the Raspberry Pi hardware, and should be read in conjunction with that specification. The platform HAL package complements the ARM architectural HAL, the Cortex-A variant HAL and the BCM283X variant HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset, the HAL will initialize or reinitialize most of the on-chip peripherals. There is an exception for RAM startup applications which depend on a ROM monitor for certain services.

Linker Scripts and Memory Maps

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

SDRAM

This is located at address 0x00000000 of the physical memory space. The HAL configures the MMU to retain the SDRAM at virtual address 0x00000000 with caching enabled. The same memory is also accessible uncached and unbuffered at virtual location 0x80000000 for use by device drivers. ROM applications are loaded by the GPU starting at 0x000008000. Memory is required for the MMU tables, and must be aligned on a 16Kbyte boundary. These therefore occupy memory from 0x00004000 to 0x00008000, just below the load address. Interrupt and exception vectors are placed at 0x00000000 and the virtual vectors occupy 256 bytes at 0x00000050. For ROM startup, all remaining SDRAM above 0x00008000 is available. JTAG startup applications are loaded from 0x00100000, leaving the bottom 1MiB unused. RAM and SMP startup applications are loaded from location 0x00200000, reserving 2MiB.

Boards with 256MiB or 512MiB of SDRAM will only occupy the least significant portion of this region. Accesses beyond the limit of physical memory will raise an exception. On boards with 1GiB, the peripheral registers overlay the top 16MiB of SDRAM, so not all the SDRAM is available. In all boards, the GPU must reserve a portion of SDRAM. Since eCos does not currently make use of the GPU, the minimum possible 16MiB is reserved. So, the smaller boards have a maximum of 240MiB and 496MiB RAM available, and the 1GiB boards have a maximum of 992MiB available. The GPU reserved region depends on the contents of `config.txt` at boot time, so these figures should not be relied upon. Applications should rely on eCos to manage the quantity of RAM available, or can use the macro `HAL_MEM_REAL_REGION_TOP(0)` to discover the actual size of the SDRAM available.

Peripheral Registers

These occupy a 16MiB physical address space at either 0x20000000 for the BCM2835 based systems or at 0x3F000000 for BCM2836 and BCM2837 based systems. The MMU is used to unify these disparate mappings by relocating both, uncached, to 0x50000000.

Multicore Peripheral Registers

These are only present in the BCM2836 and BCM2837 based systems and contain hardware that is only applicable to these multicore devices. This 1MiB area is identity mapped uncached at 0x40000000.

The virtual address space visible to applications is summarized in the following table. Any address range not mentioned here should not be accessed and will raise an exception if it is.

Base	Size (MiB)	Cache	Description
0x00000000	1008	Enabled	Normal SDRAM access -- to limit of physical RAM, raises an exception beyond that.
0x40000000	1	Disabled	Multicore registers -- multicore systems only, raises an exception in single core systems.
0x50000000	16	Disabled	Peripheral registers.
0x80000000	1008	Disabled	Uncached SDRAM access -- to limit of physical RAM, raises an exception beyond that.

Real-time Characterization

The tm_basic kernel test gives statistics gathered about the real-time characterization and performance of the kernel. The sample output is shown here for information. The test was built in ARM32 mode and run in non-SMP mode on a Raspberry Pi3. The BCM283X is a high performance processor, but its general purpose timer is only clocked at 1Mhz. This results in microsecond level resolution which is insufficient precision for the timing of many eCos kernel operations.

Example 281.1. Raspberry Pi3 Real-time characterization

```
             Startup, main thrd : stack used   388 size  1792
             Startup :  Interrupt stack used  4096 size  4096
             Startup : Idlethread stack used    96 size  1280

eCos Kernel Timings
Notes: all times are in microseconds (.000001) unless otherwise stated

Reading the hardware clock takes 0 'ticks' overhead
... this value will be factored out of all other measurements
Clock interrupt took    3.00 microseconds (3 raw clock ticks)

Testing parameters:
   Clock samples:            32
   Threads:                  64
   Thread switches:         128
   Mutexes:                  32
   Mailboxes:                32
   Semaphores:               32
   Scheduler operations:    128
   Counters:                 32
   Flags:                    32
   Alarms:                   32
   Stack Size:             1088


                                 Confidence
     Ave     Min     Max     Var  Ave  Min  Function
  ======  ======  ======  ====== ========== ========
INFO:<Ctrl-C disabled until test completion>
    1.25    1.00    2.00    0.38   75%  75% Create thread
    0.44    0.00    1.00    0.49   56%  56% Yield thread [all suspended]
    0.53    0.00    1.00    0.50   53%  46% Suspend [suspended] thread
    0.47    0.00    1.00    0.50   53%  53% Resume thread
    0.52    0.00    1.00    0.50   51%  48% Set priority
    0.33    0.00    1.00    0.44   67%  67% Get priority
    0.69    0.00    1.00    0.43   68%  31% Kill [suspended] thread
    0.47    0.00    1.00    0.50   53%  53% Yield [no other] thread
    0.55    0.00    1.00    0.50   54%  45% Resume [suspended low prio] thread
    0.48    0.00    1.00    0.50   51%  51% Resume [runnable low prio] thread
    0.53    0.00    1.00    0.50   53%  46% Suspend [runnable] thread
    0.44    0.00    1.00    0.49   56%  56% Yield [only low prio] thread
    0.45    0.00    1.00    0.50   54%  54% Suspend [runnable->not runnable]
    0.69    0.00    1.00    0.43   68%  31% Kill [runnable] thread
    0.63    0.00    1.00    0.47   62%  37% Destroy [dead] thread
    0.94    0.00    1.00    0.12   93%   6% Destroy [runnable] thread
    1.47    1.00    2.00    0.50   53%  53% Resume [high priority] thread
    0.67    0.00    1.00    0.44   67%  32% Thread switch

    0.34    0.00    1.00    0.45   65%  65% Scheduler lock
    0.41    0.00    1.00    0.48   59%  59% Scheduler unlock [0 threads]
    0.43    0.00    1.00    0.49   57%  57% Scheduler unlock [1 suspended]
    0.38    0.00    1.00    0.47   61%  61% Scheduler unlock [many suspended]
    0.45    0.00    1.00    0.49   55%  55% Scheduler unlock [many low prio]

    0.34    0.00    1.00    0.45   65%  65% Init mutex
    0.47    0.00    1.00    0.50   53%  53% Lock [unlocked] mutex
    0.47    0.00    1.00    0.50   53%  53% Unlock [locked] mutex
    0.47    0.00    1.00    0.50   53%  53% Trylock [unlocked] mutex
    0.41    0.00    1.00    0.48   59%  59% Trylock [locked] mutex
    0.34    0.00    1.00    0.45   65%  65% Destroy mutex
    1.00    1.00    1.00    0.00  100% 100% Unlock/Lock mutex

    0.38    0.00    1.00    0.47   62%  62% Create mbox
    0.47    0.00    1.00    0.50   53%  53% Peek [empty] mbox
    0.47    0.00    1.00    0.50   53%  53% Put [first] mbox
    0.34    0.00    1.00    0.45   65%  65% Peek [1 msg] mbox
    0.50    0.00    1.00    0.50  100%  50% Put [second] mbox
    0.34    0.00    1.00    0.45   65%  65% Peek [2 msgs] mbox
    0.44    0.00    1.00    0.49   56%  56% Get [first] mbox
    0.50    0.00    1.00    0.50  100%  50% Get [second] mbox
    0.44    0.00    1.00    0.49   56%  56% Tryput [first] mbox
    0.50    0.00    1.00    0.50  100%  50% Peek item [non-empty] mbox
    0.44    0.00    1.00    0.49   56%  56% Tryget [non-empty] mbox
    0.50    0.00    1.00    0.50  100%  50% Peek item [empty] mbox
    0.41    0.00    1.00    0.48   59%  59% Tryget [empty] mbox
    0.34    0.00    1.00    0.45   65%  65% Waiting to get mbox
    0.34    0.00    1.00    0.45   65%  65% Waiting to put mbox
    0.38    0.00    1.00    0.47   62%  62% Delete mbox
    1.00    1.00    1.00    0.00  100% 100% Put/Get mbox

    0.34    0.00    1.00    0.45   65%  65% Init semaphore
    0.53    0.00    1.00    0.50   53%  46% Post [0] semaphore
    0.50    0.00    1.00    0.50  100%  50% Wait [1] semaphore
    0.53    0.00    1.00    0.50   53%  46% Trywait [0] semaphore
    0.44    0.00    1.00    0.49   56%  56% Trywait [1] semaphore
    0.34    0.00    1.00    0.45   65%  65% Peek semaphore
    0.34    0.00    1.00    0.45   65%  65% Destroy semaphore
    0.91    0.00    1.00    0.17   90%   9% Post/Wait semaphore

    0.38    0.00    1.00    0.47   62%  62% Create counter
    0.34    0.00    1.00    0.45   65%  65% Get counter value
    0.34    0.00    1.00    0.45   65%  65% Set counter value
    0.50    0.00    1.00    0.50  100%  50% Tick counter
    0.34    0.00    1.00    0.45   65%  65% Delete counter

    0.34    0.00    1.00    0.45   65%  65% Init flag
    0.47    0.00    1.00    0.50   53%  53% Destroy flag
    0.41    0.00    1.00    0.48   59%  59% Mask bits in flag
    0.44    0.00    1.00    0.49   56%  56% Set bits in flag [no waiters]
    0.50    0.00    1.00    0.50  100%  50% Wait for flag [AND]
    0.41    0.00    1.00    0.48   59%  59% Wait for flag [OR]
    0.53    0.00    1.00    0.50   53%  46% Wait for flag [AND/CLR]
    0.41    0.00    1.00    0.48   59%  59% Wait for flag [OR/CLR]
    0.34    0.00    1.00    0.45   65%  65% Peek on flag

    0.38    0.00    1.00    0.47   62%  62% Create alarm
    0.44    0.00    1.00    0.49   56%  56% Initialize alarm
    0.47    0.00    1.00    0.50   53%  53% Disable alarm
    0.50    0.00    1.00    0.50  100%  50% Enable alarm
    0.53    0.00    1.00    0.50   53%  46% Delete alarm
    0.50    0.00    1.00    0.50  100%  50% Tick counter [1 alarm]
    1.16    1.00    2.00    0.26   84%  84% Tick counter [many alarms]
    0.72    0.00    1.00    0.40   71%  28% Tick & fire counter [1 alarm]
    4.19    4.00    5.00    0.30   81%  81% Tick & fire counters [>1 together]
    1.50    1.00    2.00    0.50  100%  50% Tick & fire counters [>1 separately]
    2.80    2.00    3.00    0.31   80%  19% Alarm latency [0 threads]
    3.00    3.00    3.00    0.00  100% 100% Alarm latency [2 threads]
    3.00    3.00    3.00    0.00  100% 100% Alarm latency [many threads]
    3.00    3.00    3.00    0.00  100% 100% Alarm -> thread resume latency

    0.00    0.00    0.00    0.00            Clock/interrupt latency

    1.01    1.00    2.00    0.00            Clock DSR latency

     226     144     272                    Worker thread stack used (stack size 1088)
            All done, main thrd : stack used   820 size  1792
            All done :  Interrupt stack used   156 size  4096
            All done : Idlethread stack used   240 size  1280

Timing complete - 29830 ms total

PASS:<Basic timing OK>
EXIT:<done>
```

Other Issues

The platform HAL does not affect the implementation of other parts of the eCos HAL specification. The BCM283X processor HAL and the ARM architectural HAL documentation should be consulted for further details.

Name

RedBoot Extensions — Usage

Overview

The Raspberry Pi version of RedBoot provides a number of extensions to the standard RedBoot behaviour. These include the execution of a startup script and some extra commands.

Startup Script

Unlike most RedBoot instances, the Raspberry Pi always has access to a storage device containing a file system. It is also useful for JTAG debugging to be able to have some dynamic hardware initialization. So, on startup RedBoot attempts to mount the boot partition on the SD card and execute the contents of a script file.

The script file is named `redboot.txt` and consists of a sequence of RedBoot commands separated by newlines. Blank lines are ignored. Comments are introduced by the a hash character (#) and extend to the end of the current line.

RedBoot waits 1 second before executing the script, during which time the user may type a Ctrl-C character to abort execution of the script. This time may be changed by rebuilding RedBoot with a different `CYGNUM_REDBOOT_BOOT_SCRIPT_DEFAULT_TIMEOUT` value.

Other ports of RedBoot are typically flash memory resident and use the flash as a persistant store for RedBoot configuration information and files. You will find these areas referred to as the fconfig area and flash image system (FIS) within the RedBoot documentation. As the `redboot.txt` script file can contain any standard RedBoot command, as well as the additional Raspberry Pi commands described below, you can use it to configure any settings that would normally be stored in the fconfig area. For example, you could set a static IP address for the board with the **ip_address** command:

```
ip_address -l 192.168.1.100/24
```

To modify the `redboot.txt` file simply mount the SD card on a PC. You will find `redboot.txt` in the root directory of the boot partition.

INFO Command

The **info** command provides information about the current Raspberry Pi board. RedBoot, and eCos, discover the type of board and its properties at runtime. This command reports what has been found out. As an example, the output for a Pi3 would appear as follows:

```
RedBoot> info
Board Model    = 00000000
Board Revision = 00a02082
                 Model   = B3
                 Version = 1.2
                 CPU     = BCM2837 - Cortex-A53
                 RAM     = 1024MiB
                 Flags   = 07 Ethernet WiFi Bluetooth
Board Serial   = 000000009330d07b
MAC Address    = b8:27:eb:30:d0:7b
SDRAM Size     = 3e600000
DMA Channels   = 0 2 4 5 8 9 10 11 12 13 14
Temperatures:
       Current =  49.388 C
       Maximum =  85.000 C
Clocks:
       ARM     =  1200 MHz (min/orig/max: 600/600/1200 MHz)
       Core    =   400 MHz (min/orig/max: 250/250/400 MHz)
       Timer   =     1 MHz
       UART0   =    48 MHz
       EMMC    =   200 MHz
```

```
RedBoot>
```

And for a Pi0:

```
RedBoot> info
Board Model    = 00000000
Board Revision = 00900092
                 Model   = 0
                 Version = 1.2
                 CPU     = BCM2835 - ARM11
                 RAM     = 512MiB
                 Flags   = 00
Board Serial   = 00000000e868b4c6
MAC Address    = b8:27:eb:68:b4:c6
SDRAM Size     = 1f000000
DMA Channels   = 0 2 4 5 8 9 10 11 12 13 14
Temperatures:
       Current =  29.324 C
       Maximum =  85.000 C
Clocks:
       ARM     =  1000 MHz (min/orig/max: 700/700/1000 MHz)
       Core    =   400 MHz (min/orig/max: 250/250/400 MHz)
       Timer   =     1 MHz
       UART0   =    48 MHz
       EMMC    =   200 MHz
RedBoot>
```

FREQ Command

The **freq** command provides information and control over the ARM CPU frequency and the system CORE frequency. The `-a` option allows the ARM frequency to be changed and the `-c` option allows the system CORE frequency to be changed. In both cases the frequency is expressed in MHz. The command finishes by reporting the clock frequencies in the same format as the **info** command. For example on a Pi3:

```
RedBoot> freq

        Clocks:
       ARM     =  1200 MHz (min/orig/max: 600/600/1200 MHz)
       Core    =   400 MHz (min/orig/max: 250/250/400 MHz)
       Timer   =     1 MHz
       UART0   =    48 MHz
       EMMC    =   200 MHz
RedBoot> freq -a 800
Set ARM frequency to   800 MHz
Clocks:
       ARM     =   800 MHz (min/orig/max: 600/600/1200 MHz)
       Core    =   400 MHz (min/orig/max: 250/250/400 MHz)
       Timer   =     1 MHz
       UART0   =    48 MHz
       EMMC    =   200 MHz
RedBoot> freq -c 300
Set CORE frequency to   300 MHz
Clocks:
       ARM     =   800 MHz (min/orig/max: 600/600/1200 MHz)
       Core    =   300 MHz (min/orig/max: 250/250/400 MHz)
       Timer   =     1 MHz
       UART0   =    48 MHz
       EMMC    =   200 MHz
RedBoot>
```

GPIO Command

The **gpio** command provides information and control of the GPIO pins available in the BCM283X. The command supports a number of sub-commands:

gpio get [pin]

Print the state of a given GPIO pin, or if no pin is given, it prints the state of all pins. The information printed for each pin includes the alternate function value and the name of the function selected, mode setting bits, and the current input level.

gpio in <pin>

Reports the current input level of the pin, 0 or 1.

gpio monitor [-m <msec>] <pin>

Monitor a given pin for changes and print the value every second. The `-m` option allows the monitoring interval to be changed to the given number of milliseconds.

gpio out -0|-1 <pin>

Set the output of the given pin to 0 (`-0`) or 1 (`-1`). The pin may need to be set to output mode with **gpio set** before any effect can be seen.

gpio set [-i|-o|-a <alt>] <pin>

Set the function of a given pin. It can be set to INPUT with the `-i` option, OUTPUT with the `-o` option or to one of six alternate function with the `-a` option.

gpio table

Print a table of all GPIO pins and the names of the alternate functions they can take. The current setting of the pin is indicated by an asterisk against the relevant alternate setting. If no asterisk is present then the pin is in GPIO mode.

gpio toggle [-m <msec>] <pin>

Toggle a pin at a 1 second period with a 50% duty cycle. The `-m` option allows the toggling to occur with the given period in milliseconds. The pin may need to be set to output mode with **gpio set** before any effect can be seen.

JTAG Command

The **jtag** command provides direct support for setting up the JTAG debug pins and querying their state. The arguments to this command are a list of pin numbers. Each of these GPIO pins is put into its JTAG function. Only those pins that can be used for JTAG operation may be entered; at present these are GPIO4 to GPIO6, GPIO12, GPIO13 and GPIO22 to GPIO27. On completion the command lists the pins that are in JTAG mode. If no pins are given, then the current JTAG pin assignments are listed. Pins may be disabled by preceeding them with a hyphen; this puts the pin into GPIO input mode.

The selection of JTAG pins to use may depend on the functionality of any HATs or other hardware attached to the GPIO header. It may be necessary to mix and match pins to get a complete set. In some situations it may not be possible to find a set of pins that will work, in which case you may need to use serial debugging only.

The following table defines the possible GPIO pin alternatives for the JTAG signals, as well as their mapping on to the 40-way expansion bus used on standard RPi boards. Each JTAG signal has two alternative mappings, apart from TRST which can only be set to GPIO pin 22. Note that RTCK is not required or supported by all JTAG debuggers and may not need to be set.

JTAG Signal	Alt 4 GPIO pin	(Alt 4 40-way pin)	Alt 5 GPIO pin	(Alt 5 40-way pin)
TDI	26	37	4	7
TRST	22	15	n/a	n/a
TDO	24	18	5	29
TCK	25	22	13	33
TMS	27	13	12	32
RTCK	23	16	6	31

For example, the **jtag** command used in the default redboot.txt file, sets and reports the BCM pins as follows:

```
RedBoot> jtag 4 22 24 25 27
PIN FUNC
 22 ARM_TRST
 23 ARM_RTCK
 24 ARM_TDO
```

```
25 ARM_TCK
26 ARM_TDI
27 ARM_TMS
```

LED Command

The **led** command controls the behaviour of the board's ACT LED. By default RedBoot blinks the LED at 1 Hz with a 50% duty cycle; this serves as an indicator that RedBoot is up and running when the serial line is not connected. It takes one of three argument values. **`on`** disables blinking and switches the LED on. **`off`** also disables blinking and switches the LED off. **`blink`** re-enables the 1Hz blinking. Note that these setting only apply when RedBoot is running. Once and application has been loaded and is running, control of the LED passes to it.

USB Command

The **usb** command prints out some information from the USB stack. This includes some gathered statistics and information on each of the USB devices currently attached to the USB bus.

Chapter 282. Virtual Machine Support

Name

eCos Support for ARM Virtual Machines — Overview

Description

This document covers the configuration and usage of eCos and RedBoot on ARM Aarch32 Virtual Machines running under a variety of hypervizors. It supports Virtual Machines using a limited number of real device emulations as well as some VirtIO based devices. This package contains things that are common to all VM targets. An additional package is needed to customize support for a specific hypervizor. At present virtual machines running under QEMU and Xvisor are supported by these packages.

eCos applications are typically developed using the host-based gdb debugger, communicating with a "gdb stub" built-in to either RedBoot or the hypervisor. For RedBoot-based development, the VM is set up to boot into RedBoot either from a ROM image or from an image inserted into RAM from the hypervisor. RedBoot incorporates a gdb stub that enables eCos applications to be downloaded and debugged either via a serial line or over Ethernet. When running under QEMU, such applications may also be debugged using QEMU's GDB stub interface.

Support for SMP operation is available, although debugging support is limited. There is no SMP debug support in RedBoot. The emulated virtual GIC supports up to eight CPUs, eCos is therefore limited to a maximum of eight CPUs..

Finished eCos ROM startup applications can be deployed directly by a hypervisor, copying its image into a VM and booting the VM.

This documentation is expected to be read in conjunction with the ARM architecture HAL documentation and the hypervisor specific packages.

Supported Hardware

eCos currently runs in single processor or multi-processor modes on an ARM Aarch32 Virtual Machine. The following devices are currently supported by this and other generic packages.

Generic Interrupt Controller

The ARM GIC is used to handle interrupts. The GIC needs to be memory-mapped. The mapping of interrupt vectors is handled by a hypervisor specific package.

Generic Timer

The ARM architecture Generic Timer is used to supply the system timer. eCos used the virtual counter and timer in order to insulate it from hypervisor scheduling. The timer is accessed through the CP15 register set.

Emulated UART

ARM PL011 UART macrocell. Most ARM hypervisors support an emulation of this device and drivers are available in RedBoot and eCos to support it.

VirtIO Console

Support for the VirtIO console device class is present in both RedBoot and eCos.

VirtIO NET

Ethernet access is via a VirtIO NET device. The hypervisor is then responsible for routing packets to other VMs or an external network connection.

VirtIO RPMSG

Support is present for using a VirtIO RPMSG interface. This supports the RPMSG subsystem library to present a standard RPMSG API to user applications.

In general, devices are initialized only as far as is necessary for eCos to run. Other devices are not touched unless the appropriate driver is loaded. Further devices may be supported by the hypervisor specific HAL package and additional device drivers.

Tools

The board support is intended to work with GNU tools configured for an arm-eabi target. The original port was undertaken using arm-eabi-gcc version 7.3.0, arm-eabi-gdb version 8.1, and binutils version 2.30.

Name

Configuration — Platform-specific Configuration Options

Overview

The Virtual Machine platform HAL package is loaded automatically when eCos is configured for any virtual target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

RedBoot and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building RedBoot.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then the serial port will be claimed for HAL diagnostics.

SMP Support

SMP support is limited to a maximum of eight CPUs by the design of the ARM GIC. The exact number of CPUs that eCos supports is defined by the `CYGNUM_HAL_ARM_CORTEXA_VIRTUAL_CPU_COUNT` configuration option. The default value for this option is four, but each hypervisor package may set this to a value that corresponds to the number of CPUs configured by the hypervisor. It may also be changed by the user within certain constraints.

eCos does not handle dynamic configuration of CPUs and cannot detect the number of CPUs present at startup. So the value specified for `CYGNUM_HAL_ARM_CORTEXA_VIRTUAL_CPU_COUNT` should never exceed the number of CPUs configured by the hypervisor. It should be possible to run eCos with fewer CPUs than the hypervisor configures, since any extra CPUs will not be started. However, this is hypervisor dependent, see the documentation on each hypervisor for details.

UART Serial Driver

There are two common serial devices supported for the VM. The ARM PL011 macrocell is supported by a serial driver and as a command line input for RedBoot. The VirtIO console driver is similarly supported by a driver and as a RedBoot input; it is dependent on the VIRTIO driver and can only be configured if VIRTIO driver is also present.

Ethernet Driver

Support for the VirtIO NET device is provided by the `CYGPKG_DEVS_ETH_VIRTUAL` driver. This driver is dependent on the VIRTIO driver and can only be configured if the VIRTIO driver is also present. This driver is also not active until the generic Ethernet support package, `CYGPKG_IO_ETH_DRIVERS`, is included in the configuration.

RPMSG Driver

Support for a VirtIO RPMSG interface is present in package `CYGPKG_DEVS_RPMSG_VIRTUAL`. This driver is dependent on the VIRTIO driver and can only be configured if the VIRTIO driver is also present.

The RPMSG driver is accessed via the RPMSG API library, details of which may be found here.

VirtIO Driver

Support for generic VirtIO devices is provided by the VIRTIO driver (`CYGPKG_DEVS_VIRTIO`). This provides the generic initialization and management of queues common to all VirtIO devices.

The VirtIO driver is not normally accessed directly from applications, but only from client device drivers. For reference, details of the VirtIO driver may be found here.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the ARM Aarch32 Virtual Machine, and should be read in conjunction with that specification. The platform HAL package complements the ARM architectural HAL, the Cortex-A variant HAL and the VIRTUAL variant HAL. It provides functionality which is specific to the target board.

Startup

Following a reset, the HAL will initialize or reinitialize most of the on-chip peripherals. There is an exception for RAM startup applications which depend on a ROM monitor for certain services.

Chapter 283. QEMU Virtual Machine Support

Name

eCos Support for QEMU Virtual Machines — Overview

Description

This document covers the configuration and usage of eCos and RedBoot on ARM Aarch32 Virtual Machines running under QEMU. While QEMU is capable of emulating a variety of real board and devices, this package specifically supports running eCos using the generic "virt" target. As such it is similar to a VM running under a hypervisor on a real host and is treated as such here. It supports Virtual Machines using a limited number of real device emulations as well as some VirtIO based devices.

eCos applications are typically developed using the host-based gdb debugger, communicating with a "gdb stub" built-in to either RedBoot or QEMU. For RedBoot-based development, the VM is set up to boot into RedBoot either from a ROM image or from an image inserted into RAM when QEMU was started. RedBoot incorporates a gdb stub that enables eCos applications to be downloaded and debugged either via a serial line or over Ethernet. Such applications may also be debugged using QEMU's GDB stub interface.

Support for SMP operation is available, although debugging support is restricted to use of an external debugger connected to QEMU's GDB debug port. There is no SMP debug support in RedBoot. QEMU supports up to eight CPUs in the ARM virtual machine, by default eCos is configured to use four CPUs.

Finished eCos ROM startup applications can be deployed directly by replacing the RedBoot binary with the application binary when QEMU is started.

This documentation is expected to be read in conjunction with the ARM architecture HAL and Virtual Machine Support documentation; further device support and subsystems are described and documented there.

Supported Hardware

eCos currently runs in single processor mode on an ARM Aarch32 Virtual Machine. The following devices are currently supported by this port.

Generic Interrupt Controller
: The ARM GIC is used to handle interrupts. The GIC needs to be memory-mapped. Interrupt vector mapping is defined in this package.

Generic Timer
: The ARM architecture Generic Timer is used to supply the system timer. eCos uses the physical counter and timer. The timer is accessed through the CP15 register set.

Emulated UART
: ARM PL011 UART macrocell. QEMU supports an emulation of this device and drivers are available in RedBoot and eCos to support it.

VirtIO Console
: Support for the VirtIO console device class is present in both RedBoot and eCos.

VirtIO NET
: Ethernet access is via a VirtIO NET device. QEMU is then responsible for routing packets to other VMs or the wider network.

VirtIO RPMSG
: Support is present for using a VirtIO RPMSG interface. This supports the RPMSG subsystem library to present a standard RPMSG API to user applications.

In general, devices are initialized only as far as is necessary for eCos to run. Other devices are not touched unless the appropriate driver is loaded.

Tools

The board support is intended to work with GNU tools configured for an arm-eabi target. The original port was undertaken using arm-eabi-gcc version 7.3.0, arm-eabi-gdb version 8.1, and binutils version 2.30.

Name

Setup — Preparing for eCos Development

Overview

In a typical development environment, the VM boots into the RedBoot ROM monitor. eCos applications are configured for RAM startup and then downloaded and run on the board via the debugger **arm-eabi-gdb**. Preparing the VM therefore usually involves arranging for a suitable RedBoot image to be executed on startup.

QEMU Installation

This section describes how to run eCos under QEMU. The reader should be familiar with QEMU, how to install it onto their host and how to invoke it. The examples in this section describe running QEMU under Linux. This has been tested using QEMU version 4.2.0.

eCos applications can be run under QEMU either directly or by first loading RedBoot and then using that to load and execute the application. In both cases it is necessary to configure QEMU to emulate a virtual machine that matches the target for which eCos has been compiled. A typical QEMU invocation is shown below.

```
qemu-system-arm -M virt -m 32 -nographic -s -smp cpus=8 -kernel app \
          -netdev user,id=mynet,net=10.4.0.0/24,hostfwd=tcp::9000-:9000 \
          -device virtio-net-device,netdev=mynet -device virtio-serial-device \
          -chardev socket,id=vio,host=0.0.0.0,port=4322,server,telnet,nowait \
          -device virtserialport,chardev=vio
```

This will run an eCos application in an Aarch32 virtual machine with the main serial port routed to standard IO. Breaking that command line down into its components:

```
-M virt -m 32 -nographic -s -smp cpus=8
```

The `-M` option selects a generic ARMv7 virtual machine emulation. The `-m` option sets the main RAM size to 32MiB. The `-nographic` suppresses use of a graphical interface, confining QEMU to a command line interface. The `-s` option enables the GDB server. The `-smp cpus=8` option instantiates eight CPUs. Initially only CPU0 is running and will run non-SMP builds. If eCos is built to use more CPUs, then it will enable the additional CPUs when the scheduler starts. By default eCos is configured to use four CPUs.

```
-kernel app
```

This specifies the application to load. The application may be either an ELF executable or a binary file. Only applications built using the ROM startup type should be used here. This application executable should be RedBoot if it is intended to run RAM startup application via GDB.

```
-netdev user,id=mynet,net=10.4.0.0/24,hostfwd=tcp::9000-:9000
-device virtio-net-device,netdev=mynet
```

These options set up the networking interface. In this case we are using QEMU's user networking support, which avoids the need to set up additional interfaces on the host and does not need privileged access. This option runs the virtual machine in a private subnet, in this case `10.4.0.0/24`, which should be chosen so as not to clash with the existing network. The `hostfwd` option maps the host's TCP port 9000 to the virtual machine's port 9000; which is RedBoot's telnet/GDB port. Additional port mappings may be added if necessary.

```
-device virtio-serial-device
-chardev socket,id=vio,host=0.0.0.0,port=4322,server,telnet,nowait
-device virtserialport,chardev=vio
```

These options map the virtual machine's second serial port on to a telnet server on local port 4322.

The above command line uses standard IO for the main console. An alternative would be to attach the console to a Telnet socket. The following additional options will do this:

```
-serial tcp:0.0.0.0:4321,server,telnet,nowait
```

Use of `telnet` here provides support for the RedBoot command line interface. However, if the intention is to use GDB to load and run applications via the console, then the `telnet` option should be omitted since the telnet protocol does not interact well with the GDB protocol. The same consideration applies to the arguments for the second serial line given earlier.

For simplicity, a shell script, **ecos_qemu**, is available in the `etc` directory of the RedBoot install directory which runs an eCos application using the options above. It is recommended that this script and the executable to be run are copied out to a working directory. This script may be edited to adjust the arguments if a different configuration is needed. Running RedBoot using the script under Linux, will look something like this:

```
$ ./ecos_qemu redboot.elf
APP       redboot.elf
QEMU-ARGS
QEMU      qemu-system-arm
CMD       -M virt -m 32 -nographic -s -kernel redboot.elf -netdev user,id=mynet,net=10.4.0.0/24,
hostfwd=tcp::9000-:9000 -device virtio-net-device,netdev=mynet -device virtio-serial-device
-chardev socket,id=vio,host=0.0.0.0,port=4322,server,telnet,nowait -device virtserialport,
chardev=vio

Ctrl-A X to exit
Ctrl-A C for qemu monitor

+Ethernet eth0: MAC address 52:54:00:12:34:56
IP: 10.4.0.15/255.255.255.0, Gateway: 10.4.0.2
Default server: 10.4.0.2
DNS server IP: 10.4.0.3, DNS domain name: <null%gt;

RedBoot(tm) bootstrap and debug environment [ROM]
Non-certified release, version UNKNOWN - built 14:28:56, May 26 2020

Copyright (C) 2000-2009 Free Software Foundation, Inc.
Copyright (C) 2003-2019 eCosCentric Limited
The RedBoot bootloader is a component of the eCos real-time operating system.
Want to know more? Visit www.ecoscentric.com for everything eCos & RedBoot related.
This is free software, covered by the eCosPro Non-Commercial Public License
and eCos Public License. You are welcome to change it and/or distribute copies
of it under certain conditions. Under the license terms, RedBoot's source code
and full license terms must have been made available to you.
Redboot comes with ABSOLUTELY NO WARRANTY.

Platform: Virtual ARM (CORTEX-A)
RAM: 0x40000000-0x42000000 [0x40458000-0x42000000 available]
RedBoot>
```

Rebuilding RedBoot

Typical users should never need to rebuild RedBoot. If you do intend to modify RedBoot then please note that rebuilding it is currently only supported from the Linux command line.

The steps needed to rebuild the ROM version of RedBoot are:

```
$ mkdir redboot_qemu_rom
$ cd redboot_qemu_rom
$ ecosconfig new virtual_qemu redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/arm/cortexa/virrual/VERSION/misc/redboot_ROM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

At the end of the build the `install/bin` subdirectory should contain the files `redboot.elf` and `redboot.bin`. Either of these files can now be used to start RedBoot.

Name

Configuration — Platform-specific Configuration Options

Overview

The QEMU platform HAL package is loaded automatically when eCos is configured for a QEMU target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The platform HAL package supports three separate startup types:

RAM This startup type is normally used during application development when using the RedBoot ROM monitor. arm-eabi-gdb is used to load the RAM startup application into memory and debug it. It is assumed that the hardware has already been initialized by RedBoot. By default the application will use the eCos virtual vectors mechanism to obtain services from RedBoot, including diagnostic output.

This startup type is also used to for applications that are loaded from the RedBoot command line with the **load** command.

In QEMU VMs RAM applications are loaded at address 0x40500000, leaving the bottom 5MiB free for RedBoot.

ROM This startup type can be used for finished applications which will be booted direct into the VM from the hypervisor. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization.

In QEMU VMs ROM applications are loaded by QEMU at 0x40010000 in main RAM. The bottom 4MiB are treated as a read-only area. ROM applications will place exception vectors, DATA and BSS starting from 0x40400000.

SMP This startup type can be used for finished applications that can be loaded into RAM via RedBoot or from the QEMU command line. The load address is set to the same as for RAM applications, however, the application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization. Once started, this application takes full control of the system and RedBoot will not be called again. This means that debugging via RedBoot will not be possible, only external debugging via QEMU's debug port is supported.

RedBoot and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building RedBoot.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then the serial port will be claimed for HAL diagnostics.

UART Serial Driver

There are two serial devices supported for the VM. The ARM PL011 macrocell is supported by a serial driver and as a command line input for RedBoot. The VirtIO console driver is similarly supported by a driver and as a RedBoot input; it is dependent on the VIRTIO driver and can only be configured if VIRTIO driver is also present. In QEMU these serial interfaces may be routed to a variety of host devices including stdio, TCP streams or pseudo-terminals.

Ethernet Driver

Support for the VirtIO NET device is provided by the `CYGPKG_DEVS_ETH_VIRTUAL` driver. This driver is dependent on the VIRTIO driver and can only be configured if the VIRTIO driver is also present. This driver is also not active until the generic Ethernet support package, `CYGPKG_IO_ETH_DRIVERS`, is included in the configuration.

RPMSG Driver

Support for a VirtIO RPMSG interface is present in package `CYGPKG_DEVS_RPMSG_VIRTUAL`. This driver is dependent on the VIRTIO driver and can only be configured if the VIRTIO driver is also present.

The RPMSG driver is accessed via the RPMSG API library, details of which may be found here.

VirtIO Driver

Support for generic VirtIO devices is provided by the VIRTIO driver (`CYGPKG_DEVS_VIRTIO`). This provides the generic initialization and management of queues common to all VirtIO devices.

The VirtIO driver is not normally accessed directly from application, but only from client device drivers. For reference, details of the VirtIO driver may be found here.

Compiler Flags

The platform HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos. The following flags are specific to this port:

`-mcpu=cortex-a15`	The QEMU `virt` target contains a Cortex-A15 CPU emulation.

Name

SMP Support — Usage

Overview

Support is available for SMP operation. However, debugging support is restricted to using an external SMP-aware debugger connected to QEMU's GDB stub port. RedBoot does not have support for multi-core debugging.

SMP support is enabled by setting `CYGPKG_KERNEL_SMP_SUPPORT` to true. SMP applications should only be built using either ROM or SMP startup types. ROM applications can be loaded from the QEMU command line in place of RedBoot. The SMP startup is identical to a ROM startup except that the load address is set to allow the application to be loaded into a higher location in RAM from RedBoot. SMP startup applications can also be loaded from the command line if the ELF file is used, a binary file will not work.

By default eCos is configured to use four CPUs. However, QEMU can support up to eight CPUs and the `ecos_qemu` script instantiates all eight. CPUs are only enabled when eCos starts, so unused CPUs will not consume host cycles. The number of CPUs may be changed by setting `CYGNUM_HAL_ARM_CORTEXA_VIRTUAL_CPU_COUNT`.

Loading an SMP startup application via RedBoot can be done from the RedBoot command line via serial. It may also be loaded via a GDB connection on serial. However, once started running the SMP application will take full control of the system, including redirecting all interrupt sources, exception vectors and virtual vector table entries. This means that RedBoot will no longer be active. Any breakpoints planted by GDB will result in an exception to the application, Ctrl-C will not work, any Ethernet connections will be lost and serial output will come from the application in plain ASCII. Any GDB connection will be lost and GDB may start reporting packet errors.

It is possible to load an SMP startup program via GDB and have its output displayed on the GDB console. To do this set `CYGSEM_HAL_DIAG_MANGLER` to `"GDB"`, and `CYGDBG_HAL_DIAG_TO_DEBUG_CHAN` to `1`. The application will still not be controllable from GDB, but this does simplify the running of test code; avoiding the need to disconnect GDB and connect a terminal emulator to capture or view the output.

SMP applications may be debugged via the QEMU GDB debug port using a standard GDB. In this case the different CPUs are presented to GDB as separate threads. The command **info threads** can be used to list the CPUs and the **thread** command can be used to switch between them. eCos threads are not visible in this mode.

QEMU runs each virtual CPU in a separate thread. These in turn can be scheduled onto separate CPUs of the host system. As a result the virtual machine CPUs are subject to the scheduling decisions of the host. If the host is a typically-idle development workstation then this should not pose a problem. However, if the host is busy, a server or shared system, then the other workloads may affect the relative progress of the various virtual CPUs. If the eCos application makes strict use of synchronization primitives then there should be no problem. However, if it assumes that the CPUs all progress a the same rate, then it may see some issues. These issues can partly be alleviated by running QEMU at a higher priority, or on reserved CPUs.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the ARM Aarch32 Virtual Machine, and should be read in conjunction with that specification. This HAL package complements the ARM architectural HAL, the Cortex-A variant HAL and the Virtual variant HAL. It provides functionality which is specific to the target emulator.

Startup

On emulator startup or reset the HAL will initialize or reinitialize those peripherals that are to be used. There is an exception for RAM startup applications which depend on a ROM monitor for certain services.

QEMU VM Linker Scripts and Memory Maps

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

RAM

This is located at address 0x40000000 of the physical memory space and is 32MiB in size. The HAL configures the MMU to retain the RAM at virtual address 0x40000000 with caching enabled. The same memory is also accessible uncached and unbuffered at virtual address 0x50000000 for use by device drivers. ROM applications can use RAM starting at 0x4040000020. Interrupt and exception vectors are placed at 0x40400000 and the virtual vectors occupy 256 bytes at 0x40400050. For ROM startup, all remaining RAM is available. RAM startup applications are loaded from location 0x40500000, reserving 5MiB.

ROM applications are loaded into the bottom 4MiB of RAM, at 0x40010000, which is treated like a read-only memory for this purpose. This, for example, allows RedBoot to perform a reset by jumping to 0x40010000.

Peripheral Registers

These occupy regions of memory at 0x08000000, 0x09000000 and 0x0a000000 of varying sizes. These include the GIC registers, emulated peripherals such as the PL011 UART or VirtIO devices.

The virtual address space visible to applications is summarized in the following table. Any address range not mentioned here should not be accessed and will raise an exception if it is.

Base	**Size (MiB)**	**Cache**	**Description**
0x08000000	1	Disabled	GIC registers.
0x09000000	16	Disabled	Emulated device registers.
0x0A000000	16	Disabled	VirtIO device registers.
0x40000000	32	Enabled	Normal RAM access.
0x50000000	32	Disabled	Uncached access to RAM.

Real-time Characterization

The tm_basic kernel test gives statistics gathered about the real-time characterization and performance of the kernel. The sample output is shown here for information. The test was built in ARM32 mode and run in non-SMP mode under QEMU 4.2.0 running under Linux on a 6 processor Intel i7-8700K CPU running at up to 3.70GHz.

Example 283.1. VM Real-time characterization

```
             Startup, main thrd : stack used   404 size  1792
             Startup :  Interrupt stack used  4096 size  4096
             Startup : Idlethread stack used    96 size  1280

eCos Kernel Timings
Notes: all times are in microseconds (.000001) unless otherwise stated

Reading the hardware clock takes 25 'ticks' overhead
... this value will be factored out of all other measurements
Clock interrupt took  128.51 microseconds (8031 raw clock ticks)

Testing parameters:
   Clock samples:            32
   Threads:                  64
   Thread switches:         128
   Mutexes:                  32
   Mailboxes:                32
   Semaphores:               32
   Scheduler operations:    128
   Counters:                 32
   Flags:                    32
   Alarms:                   32
   Stack Size:             1088


                                 Confidence
     Ave     Min     Max     Var  Ave  Min  Function
  ======  ======  ======  ====== ========== ========
INFO:<Ctrl-C disabled until test completion>
    1.78    0.14   52.40    2.73   95%  93% Create thread
    0.69    0.00   44.16    1.36   98%  98% Yield thread [all suspended]
    0.67    0.00   42.62    1.31   98%  98% Suspend [suspended] thread
    0.65    0.00   41.49    1.28   98%  98% Resume thread
    0.70    0.00   44.24    1.36   98%  98% Set priority
    0.22    0.00   14.30    0.44   98%  98% Get priority
    1.77    0.00  112.58    3.46   98%  98% Kill [suspended] thread
    0.17    0.00   11.10    0.34   98%  98% Yield [no other] thread
    0.75    0.00   31.82    1.39   96%  96% Resume [suspended low prio] thread
    0.17    0.00   10.64    0.33   98%  98% Resume [runnable low prio] thread
    0.41    0.00   25.58    0.79   98%  98% Suspend [runnable] thread
    0.10    0.00    5.92    0.20   96%  96% Yield [only low prio] thread
    0.11    0.00    6.72    0.21   98%  98% Suspend [runnable->not runnable]
    0.14    0.00    8.46    0.27   96%  96% Kill [runnable] thread
    2.28    0.06  136.29    4.23   98%  98% Destroy [dead] thread
    0.59    0.29   17.97    0.55   98%  96% Destroy [runnable] thread
    2.47    1.01   45.74    2.50   96%  95% Resume [high priority] thread
    0.44    0.32   11.33    0.18   98%  93% Thread switch

    0.11    0.00   12.06    0.22   98%  98% Scheduler lock
    0.04    0.00    4.58    0.07   99%  99% Scheduler unlock [0 threads]
    0.04    0.00    4.62    0.07   99%  99% Scheduler unlock [1 suspended]
    0.04    0.00    4.98    0.08   99%  99% Scheduler unlock [many suspended]
    0.04    0.00    5.01    0.08   99%  99% Scheduler unlock [many low prio]

    0.70    0.00   22.56    1.37   96%  96% Init mutex
    1.93    0.00   61.71    3.74   96%  96% Lock [unlocked] mutex
    2.13    0.00   67.95    4.11   96%  96% Unlock [locked] mutex
    2.41    0.00   76.96    4.66   96%  96% Trylock [unlocked] mutex
    0.53    0.00   17.01    1.03   96%  96% Trylock [locked] mutex
    0.55    0.00   17.65    1.07   96%  96% Destroy mutex
    7.45    1.09  101.78    8.16   90%  90% Unlock/Lock mutex

    0.91    0.00   29.14    1.76   96%  96% Create mbox
    0.34    0.00   10.99    0.67   96%  96% Peek [empty] mbox
    2.14    0.00   68.35    4.14   96%  96% Put [first] mbox
    0.21    0.00    6.45    0.40   96%  96% Peek [1 msg] mbox
    0.95    0.00   30.26    1.83   96%  96% Put [second] mbox
    0.19    0.00    6.24    0.38   96%  96% Peek [2 msgs] mbox
    2.72    0.00   86.83    5.26   96%  96% Get [first] mbox
    0.23    0.00    7.50    0.45   96%  96% Get [second] mbox
    2.79    0.00   88.21    5.34   96%  96% Tryput [first] mbox
    1.50    0.00   47.94    2.90   96%  96% Peek item [non-empty] mbox
    2.39    0.00   76.43    4.63   96%  96% Tryget [non-empty] mbox
    0.23    0.00    7.25    0.44   96%  96% Peek item [empty] mbox
    0.36    0.00   11.14    0.67   96%  96% Tryget [empty] mbox
    0.46    0.00   14.77    0.89   96%  96% Waiting to get mbox
    0.65    0.00   20.66    1.25   96%  96% Waiting to put mbox
    0.78    0.00   24.88    1.51   96%  96% Delete mbox
    6.27    0.74   47.47    5.41   81%  78% Put/Get mbox

    0.29    0.00    9.34    0.57   96%  96% Init semaphore
    0.46    0.00   14.70    0.89   96%  96% Post [0] semaphore
    0.40    0.00    7.26    0.75   93%  93% Wait [1] semaphore
    1.20    0.00   38.26    2.32   96%  96% Trywait [0] semaphore
    0.19    0.00    6.19    0.37   96%  96% Trywait [1] semaphore
    0.78    0.00   24.96    1.51   96%  96% Peek semaphore
    0.65    0.00   20.86    1.26   96%  96% Destroy semaphore
    3.38    0.67   30.11    2.70   81%  75% Post/Wait semaphore

    1.01    0.00   32.35    1.96   96%  96% Create counter
    0.97    0.00   30.58    1.85   96%  96% Get counter value
    0.51    0.00   16.42    0.99   96%  96% Set counter value
    0.51    0.00   16.30    0.99   96%  96% Tick counter
    0.53    0.00   17.06    1.03   96%  96% Delete counter

    0.44    0.00   14.00    0.85   96%  96% Init flag
    1.41    0.00   44.91    2.72   96%  96% Destroy flag
    1.09    0.00   34.85    2.11   96%  96% Mask bits in flag
    1.49    0.00   47.54    2.88   96%  96% Set bits in flag [no waiters]
    2.56    0.00   81.68    4.95   96%  96% Wait for flag [AND]
    0.54    0.00    9.68    1.02   93%  93% Wait for flag [OR]
    0.74    0.00   23.62    1.43   96%  96% Wait for flag [AND/CLR]
    0.83    0.00   26.51    1.61   96%  96% Wait for flag [OR/CLR]
    0.38    0.00   12.18    0.74   96%  96% Peek on flag

    0.90    0.00   28.90    1.75   96%  96% Create alarm
    1.25    0.00   39.70    2.40   96%  96% Initialize alarm
    0.45    0.00   14.53    0.88   96%  96% Disable alarm
    2.17    0.00   69.17    4.19   96%  96% Enable alarm
    0.32    0.00   10.22    0.62   96%  96% Delete alarm
    0.26    0.00    8.46    0.51   96%  96% Tick counter [1 alarm]
    1.12    0.18   16.27    1.74   93%  93% Tick counter [many alarms]
    0.66    0.00   20.88    1.26   96%  96% Tick & fire counter [1 alarm]
    4.68    2.80   26.10    3.26   90%  90% Tick & fire counters [>1 together]
    0.51    0.27    7.26    0.42   96%  96% Tick & fire counters [>1 separately]
  259.37  101.76  523.44   60.66   58%  14% Alarm latency [0 threads]
  128.64   68.58  359.12   22.85   78%   2% Alarm latency [2 threads]
  124.96   74.43  290.21   23.04   74%  10% Alarm latency [many threads]
  251.84  111.49  415.10   61.60   53%  17% Alarm -> thread resume latency

  161.37   64.29  451.89    0.00            Clock/interrupt latency

   11.23    1.25  159.98    0.00            Clock DSR latency

     172     172     220                    Worker thread stack used (stack size 1088)
            All done, main thrd : stack used   884 size  1792
            All done :  Interrupt stack used   136 size  4096
            All done : Idlethread stack used   248 size  1280

Timing complete - 31030 ms total
```

```
PASS:<Basic timing OK>
EXIT:<done>
```

Other Issues

The platform HAL does not affect the implementation of other parts of the eCos HAL specification. The ARM architectural HAL documentation should be consulted for further details.

Chapter 284. Xvisor Virtual Machine Support

Name

eCos Support for Xvisor Virtual Machines — Overview

Description

This document covers the configuration and usage of eCos and RedBoot on ARM Aarch32 Virtual Machines running under the Xvisor hypervisor. Only devices that are emulated by Xvisor, or are accessed through VirtIO interfaces are used, so this should run under Xvisor on any ARM host.

eCos applications are typically developed using the host-based gdb debugger, communicating with a "gdb stub" built-in to RedBoot. The VM is set up to boot into RedBoot from a ROM image. RedBoot incorporates a gdb stub that enables eCos applications to be downloaded and debugged either via a serial line or over Ethernet.

Support for SMP operation is available, although debugging support is restricted to use diagnostic messages. There is no SMP debug support in RedBoot. The emulated virtual GIC supports up to eight CPUs, by default eCos is configured to use four CPUs.

Finished eCos ROM startup applications can be deployed directly by the hypervisor by replacing RedBoot with the application binary.

This documentation is expected to be read in conjunction with the ARM architecture HAL and Virtual Machine Support documentation; further device support and subsystems are described and documented there.

Supported Hardware

eCos currently runs in single processor mode on an ARM Aarch32 Virtual Machine. The following devices are currently supported by this port.

Generic Interrupt Controller
: The ARM GIC is used to handle interrupts. The GIC needs to be memory-mapped. Interrupt vector mapping is defined in this package.

Generic Timer
: The ARM architecture Generic Timer is used to supply the system timer. eCos used the virtual counter and timer in order to insulate it from hypervisor scheduling. The timer is accessed through the CP15 register set.

Emulated UART
: ARM PL011 UART macrocell. Xvisor supports an emulation of this device and drivers are available in RedBoot and eCos to support it.

VirtIO Console
: Support for the VirtIO console device class is present in both RedBoot and eCos.

VirtIO NET
: Ethernet access is via a VirtIO NET device. The hypervisor is then responsible for routing packets to other VMs or an external network connection.

VirtIO RPMSG
: Support is present for using a VirtIO RPMSG interface. This supports the RPMSG subsystem library to present a standard RPMSG API to user applications.

In general, devices are initialized only as far as is necessary for eCos to run. Other devices are not touched unless the appropriate driver is loaded.

Tools

The board support is intended to work with GNU tools configured for an arm-eabi target. The original port was undertaken using arm-eabi-gcc version 7.3.0, arm-eabi-gdb version 8.1, and binutils version 2.30.

Name

Setup — Preparing for eCos Development

Overview

In a typical development environment, the VM boots into the RedBoot ROM monitor. eCos applications are configured for RAM startup and then downloaded and run on the board via the debugger **arm-eabi-gdb**. Preparing the VM therefore usually involves arranging for a suitable RedBoot image to be executed on startup.

Xvisor VM Installation

This section describes how to install RedBoot into an Xvisor guest VM and start it running. The reader should be familiar with Xvisor, how to install it onto a target platform and how, in general, a guest VM is created.

To create an eCos VM under Xvisor, it is necessary to instantiate a virtual machine, populate it with the executable of RedBoot, and then set it running. To do this a number of files are copied to the `etc` directory of a RedBoot installation which are used set up a VM to run eCos. These files are as follows:

`evm.dts`
: This is a Linux device tree source file. It needs to be compiled into a device tree binary file using the **dtc** device tree compiler. This file is specific to use on Rockchip-based boards such as the Pine Rockpro64 or M2000.

`boot.xscript`
: This is a sample Xvisor boot script that instantiates a Linux guest together with a eCos VM running RedBoot. It may be used as it is, or used as a template for your own script.

`nor_flash.list`
: A list of addresses and file names used to populate the virtual flash in a VM. This defines the software that will run from reset in the VM. In this case the sole entry installs a binary of RedBoot at the start of flash.

`install`
: A shell script that automates the installation of a RedBoot executable into a Xvisor disk image. It creates the necessary sub-directory in the disk image, compiles and installs the device tree, RedBoot binary and flash list. It does not copy the `boot.xscript` file, which may be copied separately if required. Following this, a **make** should be run in the Xvisor build directory to rebuild the disk image. The image may then be transferred to the target system bootstrap medium.

See the next section, Rebuilding RedBoot, for an example of using the `install` script to install RedBoot into an Xvisor VM.

Once the files are installed, Xvisor may be started and the eCos VM will be executed. Depending on the default setting for the initial serial connection, and whether the `boot.xscript` file has been copied, you may see a startup banner from RedBoot similar to the following:

```
XVisor# vserial bind evm1/uart0
[evm1/uart0] +No network interfaces found

RedBoot(tm) bootstrap and debug environment [ROM]
Non-certified release, version UNKNOWN - built 14:58:47, Jan 30 2020

Copyright (C) 2000-2009 Free Software Foundation, Inc.
Copyright (C) 2003-2019 eCosCentric Limited
The RedBoot bootloader is a component of the eCos real-time operating system.
Want to know more? Visit www.ecoscentric.com for everything eCos & RedBoot related.
This is free software, covered by the eCosPro Non-Commercial Public License
and eCos Public License. You are welcome to change it and/or distribute copies
of it under certain conditions. Under the license terms, RedBoot's source code
and full license terms must have been made available to you.
Redboot comes with ABSOLUTELY NO WARRANTY.

Platform: Virtual ARM (CORTEX-A)
```

```
RAM: 0x40000000-0x42000000 [0x40030000-0x42000000 available]
RedBoot>
```

Rebuilding RedBoot

Typical users should never need to rebuild RedBoot. If you do intend to modify RedBoot then please note that rebuilding it is currently only supported from the Linux command line.

The steps needed to rebuild the ROM version of RedBoot are:

```
$ mkdir redboot_virtual_rom
$ cd redboot_virtual_rom
$ ecosconfig new virtual_xvisor redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/arm/cortexa/virtual/VERSION/misc/redboot_ROM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

At the end of the build the `install/bin` subdirectory should contain the file `redboot.bin`. This image can now be installed into an Xvisor boot disk image using the install script. Assuming the root directory for Xvisor is in `$XVISOR` the following commands will do this.

```
$ cd install
$ ./etc/install $XVISOR/disk

Install RedBoot in disk image

Done
$
```

If required, `boot.xscript` may also be copied into `$XVISOR/disk`.

For convenience, eCosPro releases include a prebuilt RedBoot image as well as the `install` subdirectory resulting from the creation of RedBoot. These are located within the `loaders` sub-directory of the eCosPro installation, and the install script may be run from within these subdirectories. For example:

```
$ cd ecos-4.4.0/loaders/virtual_xvisor/redboot_ROM.install
$ ./etc/install $XVISOR/disk

Install RedBoot in disk image

Done
$
```

It is now necessary to rebuild the Xvisor RAM disk. Exactly how this is done can depend on the target host, how it boots and how that RAM disk is loaded. Usually this involves using **genext2fs** to make a disk image; usually an Xvisor executable and a device tree are also created. Refer to the directions for installing Xvisor on the host for details.

Name

Configuration — Platform-specific Configuration Options

Overview

The xvisor platform HAL package is loaded automatically when eCos is configured for the xvisor target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The platform HAL package supports two separate startup types:

RAM This startup type is normally used during application development when using the RedBoot ROM monitor. arm-eabi-gdb is used to load the RAM startup application into memory and debug it. It is assumed that the hardware has already been initialized by RedBoot. By default the application will use the eCos virtual vectors mechanism to obtain services from RedBoot, including diagnostic output.

This startup type is also used to for applications that are loaded from the RedBoot command line with the **load** command.

In Xvisor VMs RAM applications are loaded at address 0x40200000, leaving the bottom 2MiB free for RedBoot's use.

ROM This startup type can be used for finished applications which will be booted direct into the VM from the hypervisor. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization.

In Xvisor VMs ROM application are loaded by the hypervisor at address 0x00000000 in the emulated flash memory. RedBoot uses RAM starting from 0x40000020. Interrupt and exception vectors are placed at 0x40000000 with the virtual vectors following.

SMP This startup type can be used for finished applications that can be loaded into RAM via RedBoot. The load address is set to the same as for RAM applications, however, the application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization. Once started, this application takes full control of the system and RedBoot will not be called again. This means that debugging via RedBoot will not be possible. The intent of this startup type is to allow SMP test programs to be run from RedBoot.

RedBoot and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building RedBoot.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then the serial port will be claimed for HAL diagnostics.

UART Serial Driver

There are two serial devices supported for the VM. The ARM PL011 macrocell is supported by a serial driver and as a command line input for RedBoot. The VirtIO console driver is similarly supported by a driver and as a RedBoot input; it is dependent on the VIRTIO driver and can only be configured if VIRTIO driver is also present. Both of these will typically be routed by the hypervisor to either a real external device or a serial interface of another VM.

Ethernet Driver

Support for the VirtIO NET device is provided by the `CYGPKG_DEVS_ETH_VIRTUAL` driver. This driver is dependent on the VIRTIO driver and can only be configured if the VIRTIO driver is also present. This driver is also not active until the generic Ethernet support package, `CYGPKG_IO_ETH_DRIVERS`, is included in the configuration.

RPMSG Driver

Support for a VirtIO RPMSG interface is present in package `CYGPKG_DEVS_RPMSG_VIRTUAL`. This driver is dependent on the VIRTIO driver and can only be configured if the VIRTIO driver is also present.

The RPMSG driver is accessed via the RPMSG API library, details of which may be found here.

VirtIO Driver

Support for generic VirtIO devices is provided by the VIRTIO driver (`CYGPKG_DEVS_VIRTIO`). This provides the generic initialization and management of queues common to all VirtIO devices.

The VirtIO driver is not normally accessed directly from application, but only from client device drivers. For reference, details of the VirtIO driver may be found here.

Compiler Flags

The platform HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos. The following flags are specific to this port:

`-mcpu=cortex-a15`
: The Xvisor virtual machine configures a generic ARMv7a CPU. This option selects a suitable Cortex-A CPU variant that will support the execution of Aarch32 code.

Name

SMP Support — Usage

Overview

Support is available for SMP operation. However, debugging support is restricted to using `diag_printf()` and related functions. RedBoot does not have support for multi-core debugging and Xvisor has no support for connecting a debugger to a virtual machine.

A board intended to be used for SMP development should be initialized in the same way as a single core board. RedBoot only uses a single CPU. The device tree for the VM should be defined with four CPUs. The example device tree files described in the Setup section show how this is done.

SMP support is enabled by setting `CYGPKG_KERNEL_SMP_SUPPORT` to true. SMP applications should only be built using either ROM or SMP startup types. ROM applications can be loaded from Xvisor in place of RedBoot. The SMP startup is identical to a ROM startup except that the load address is set to allow the application to be loaded into a higher location in RAM from RedBoot.

By default eCos is configured to use four CPUs. Xvisor can support up to eight CPUs. The example device tree currently only defines four CPUs. Extra CPUs may be added by editing this file. The number of CPUs supported by eCos may be changed by setting `CYGNUM_HAL_ARM_CORTEXA_VIRTUAL_CPU_COUNT`. The number of CPUs supported by eCos should never exceed the number configured in the device tree.

Loading an SMP startup application via RedBoot can be done from the RedBoot command line via serial. It may also be loaded via a GDB connection on serial. However, once started running the SMP application will take full control of the system, including redirecting all interrupt sources, exception vectors and virtual vector table entries. This means that RedBoot will no longer be active. Any breakpoints planted by GDB will result in an exception to the application, Ctrl-C will not work, any Ethernet connections will be lost and serial output will come from the application in plain ASCII. Any GDB connection will be lost and GDB may start reporting packet errors.

It is possible to load an SMP startup application via GDB and have its output displayed on the GDB console. To do this set `CYGSEM_HAL_DIAG_MANGLER` to `"GDB"`, and `CYGDBG_HAL_DIAG_TO_DEBUG_CHAN` to `1`. The application will still not be controllable from GDB, but this does simplify the running of test code; avoiding the need to disconnect GDB and connect a terminal emulator to capture or view the output.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the ARM Aarch32 Virtual Machine, and should be read in conjunction with that specification. This HAL package complements the ARM architectural HAL, the Cortex-A variant HAL and the Virtual variant HAL. It provides functionality which is specific to the target hypervisor and the board on which it is running.

Startup

Following a reset, the HAL will initialize or reinitialize most of the on-chip peripherals. There is an exception for RAM startup applications which depend on a ROM monitor for certain services.

Xvisor VM Linker Scripts and Memory Maps

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

Virtual Flash
: This is located at 0x00000000 and is 32MiB in size. The hypervisor is responsible for loading the initial image, typically RedBoot, in to the start of this region. When the VM is kicked into life execution starts here.

RAM
: This is located at address 0x40000000 of the physical memory space and is 32MiB in size. The HAL configures the MMU to retain the RAM at virtual address 0x40000000 with caching enabled. The same memory is also accessible uncached and unbuffered at virtual address 0x50000000 for use by device drivers. ROM applications can use RAM starting at 0x4000000020. Interrupt and exception vectors are placed at 0x40000000 and the virtual vectors occupy 256 bytes at 0x00000050. For ROM startup, all remaining RAM is available. RAM startup applications are loaded from location 0x40200000, reserving 2MiB.

Shared memory
: A shared memory region is allocated at address 0x38000000. This area can be mapped to memory shared between the eCos VM and any other VM. It may be used to load applications into the VM using the RedBoot **load** command and may also be used to enable communications between applications in the eCos VM and the other VM.

Peripheral Registers
: These occupy regions of memory at 0x08000000, 0x09000000 and 0x0a000000 of varying sizes. These include the GIC registers, emulated peripherals such as the PL011 UART or VirtIO devices.

The virtual address space visible to applications is summarized in the following table. Any address range not mentioned here should not be accessed and will raise an exception if it is.

Base	Size (MiB)	Cache	Description
0x00000000	32	Disabled	Virtual flash.
0x08000000	1	Disabled	GIC registers.
0x09000000	16	Disabled	Emulated device registers.
0x0A000000	16	Disabled	VirtIO device registers.
0x38000000	16	Disabled	Shared memory.
0x40000000	32	Enabled	Normal SDRAM access.
0x50000000	32	Disabled	Uncached access to RAM.

Real-time Characterization

The tm_basic kernel test gives statistics gathered about the real-time characterization and performance of the kernel. The sample output is shown here for information. The test was built in ARM32 mode and run in non-SMP mode on a VCPU bound to host CPU 3 of Xvisor running on a Pine Rockpro64.

Example 284.1. VM Real-time characterization

```
             Startup, main thrd : stack used   404 size  1792
             Startup :  Interrupt stack used  4096 size  4096
             Startup : Idlethread stack used    96 size  1280

eCos Kernel Timings
Notes: all times are in microseconds (.000001) unless otherwise stated

Reading the hardware clock takes 1 'ticks' overhead
... this value will be factored out of all other measurements
Clock interrupt took    5.35 microseconds (102 raw clock ticks)

Testing parameters:
   Clock samples:            32
   Threads:                  64
   Thread switches:         128
   Mutexes:                  32
   Mailboxes:                32
   Semaphores:               32
   Scheduler operations:    128
   Counters:                 32
   Flags:                    32
   Alarms:                   32
   Stack Size:             1088


                                 Confidence
     Ave     Min     Max     Var  Ave  Min  Function
  ======  ======  ======  ====== ========== ========
INFO:<Ctrl-C disabled until test completion>
    1.01    0.68    1.41    0.17   59%  25% Create thread
    0.20    0.00    0.31    0.15   62%  37% Yield thread [all suspended]
    0.35    0.00    0.68    0.09   81%   4% Suspend [suspended] thread
    0.19    0.00    0.31    0.15   59%  40% Resume thread
    0.24    0.00    0.31    0.11   76%  23% Set priority
    0.09    0.00    0.31    0.13   71%  71% Get priority
    0.46    0.31    1.04    0.18   60%  60% Kill [suspended] thread
    0.20    0.00    0.31    0.15   62%  37% Yield [no other] thread
    0.25    0.00    0.31    0.10   81%  18% Resume [suspended low prio] thread
    0.19    0.00    0.31    0.15   59%  40% Resume [runnable low prio] thread
    0.22    0.00    0.31    0.13   71%  28% Suspend [runnable] thread
    0.20    0.00    0.31    0.15   62%  37% Yield [only low prio] thread
    0.19    0.00    0.31    0.15   59%  40% Suspend [runnable->not runnable]
    0.38    0.31    0.68    0.11   81%  81% Kill [runnable] thread
    0.37    0.31    0.68    0.10   84%  84% Destroy [dead] thread
    0.66    0.31    1.04    0.04   92%   6% Destroy [runnable] thread
    1.16    1.04    1.77    0.16   68%  68% Resume [high priority] thread
    0.34    0.31    0.68    0.05   92%  92% Thread switch

    0.07    0.00    0.31    0.11   77%  77% Scheduler lock
    0.17    0.00    0.31    0.16   53%  46% Scheduler unlock [0 threads]
    0.17    0.00    0.31    0.16   53%  46% Scheduler unlock [1 suspended]
    0.17    0.00    0.31    0.16   53%  46% Scheduler unlock [many suspended]
    0.17    0.00    0.31    0.16   53%  46% Scheduler unlock [many low prio]

    0.19    0.00    0.31    0.15   59%  40% Init mutex
    0.23    0.00    0.68    0.16   59%  34% Lock [unlocked] mutex
    0.23    0.00    0.68    0.14   65%  31% Unlock [locked] mutex
    0.21     0.00     0.31     0.14    65%   34% Trylock [unlocked] mutex
    0.16     0.00     0.31     0.16   100%   50% Trylock [locked] mutex
    0.10     0.00     0.31     0.13    68%   68% Destroy mutex
    1.06     1.04     1.41     0.04    93%   93% Unlock/Lock mutex

    0.20     0.00     0.31     0.15    62%   37% Create mbox
    0.21     0.00     0.31     0.13    68%   31% Peek [empty] mbox
    0.24     0.00     0.68     0.13    68%   28% Put [first] mbox
    0.06     0.00     0.31     0.10    81%   81% Peek [1 msg] mbox
    0.20     0.00     0.31     0.15    62%   37% Put [second] mbox
    0.07     0.00     0.31     0.11    78%   78% Peek [2 msgs] mbox
    0.23     0.00     0.68     0.14    65%   31% Get [first] mbox
    0.21     0.00     0.31     0.13    68%   31% Get [second] mbox
    0.18     0.00     0.31     0.15    56%   43% Tryput [first] mbox
    0.21     0.00     0.31     0.13    68%   31% Peek item [non-empty] mbox
    0.21     0.00     0.68     0.16    59%   37% Tryget [non-empty] mbox
    0.20     0.00     0.31     0.15    62%   37% Peek item [empty] mbox
    0.20     0.00     0.31     0.15    62%   37% Tryget [empty] mbox
    0.08     0.00     0.31     0.12    75%   75% Waiting to get mbox
    0.08     0.00     0.31     0.12    75%   75% Waiting to put mbox
    0.09     0.00     0.31     0.13    71%   71% Delete mbox
    0.69     0.31     1.04     0.04    90%    3% Put/Get mbox

    0.16     0.00     0.31     0.16   100%   50% Init semaphore
    0.19     0.00     0.31     0.15    59%   40% Post [0] semaphore
    0.21     0.00     0.31     0.14    65%   34% Wait [1] semaphore
    0.19     0.00     0.31     0.15    59%   40% Trywait [0] semaphore
    0.19     0.00     0.31     0.15    59%   40% Trywait [1] semaphore
    0.09     0.00     0.31     0.13    71%   71% Peek semaphore
    0.08     0.00     0.31     0.12    75%   75% Destroy semaphore
    0.68     0.31     1.04     0.02    93%    3% Post/Wait semaphore

    0.16     0.00     0.31     0.16   100%   50% Create counter
    0.08     0.00     0.31     0.12    75%   75% Get counter value
    0.06     0.00     0.31     0.10    81%   81% Set counter value
    0.23     0.00     0.31     0.12    75%   25% Tick counter
    0.10     0.00     0.31     0.13    68%   68% Delete counter

    0.11     0.00     0.31     0.14    65%   65% Init flag
    0.21     0.00     0.31     0.14    65%   34% Destroy flag
    0.19     0.00     0.31     0.15    59%   40% Mask bits in flag
    0.20     0.00     0.31     0.15    62%   37% Set bits in flag [no waiters]
    0.22     0.00     0.31     0.13    71%   28% Wait for flag [AND]
    0.20     0.00     0.31     0.15    62%   37% Wait for flag [OR]
    0.21     0.00     0.31     0.13    68%   31% Wait for flag [AND/CLR]
    0.21     0.00     0.31     0.13    68%   31% Wait for flag [OR/CLR]
    0.09     0.00     0.31     0.13    71%   71% Peek on flag

    0.11     0.00     0.31     0.14    65%   65% Create alarm
    0.33     0.00     0.68     0.11    71%   12% Initialize alarm
    0.19     0.00     0.31     0.15    59%   40% Disable alarm
    0.24     0.00     0.31     0.11    78%   21% Enable alarm
    0.19     0.00     0.31     0.15    59%   40% Delete alarm
    0.21     0.00     0.31     0.13    68%   31% Tick counter [1 alarm]
    0.67     0.31     0.68     0.02    96%    3% Tick counter [many alarms]
    0.29     0.00     0.31     0.04    93%    6% Tick & fire counter [1 alarm]
    3.82     3.59     3.96     0.17    62%   37% Tick & fire counters [>1 together]
    0.77     0.68     1.04     0.14    75%   75% Tick & fire counters [>1 separately]
   11.19    10.52    12.45     0.33    59%   22% Alarm latency [0 threads]
    5.24     4.90     5.73     0.11    53%    0% Alarm latency [2 threads]
    5.82     5.42     6.46     0.23    56%   32% Alarm latency [many threads]
   11.70    10.89    12.66     0.35    60%   11% Alarm -> thread resume latency

    6.74     3.65    11.35     0.00            Clock/interrupt latency

    0.60     0.36     1.09     0.00            Clock DSR latency
```

```
     244     172     288                    Worker thread stack used (stack size 1088)
            All done, main thrd : stack used   884 size  1792
            All done :  Interrupt stack used   136 size  4096
            All done : Idlethread stack used   248 size  1280

Timing complete - 30920 ms total

PASS:<Basic timing OK>
EXIT:<done>
```

Other Issues

The platform HAL does not affect the implementation of other parts of the eCos HAL specification. The ARM architectural HAL documentation should be consulted for further details.

Part LXXVIII. Cortex-M Architecture

Table of Contents

Chapter 285. Cortex-M Architectural Support

Name

CYGPKG_HAL_CORTEXM — eCos Support for the Cortex-M Architecture

Description

The Cortex-M implements a new version of the ARM architecture intended for deeply embedded applications. The processor runs the Thumb-2 instruction set and implements an exception and interrupt model quite different from that supported by previous members of the ARM architecture. For this reason the Cortex-M HAL is implemented as an entirely new architecture within eCos.

In addition to the processor, the Cortex-M architecture also provides definitions of a Nested Vectored Interrupt Controller (NVIC) and a system tick timer. Both of these are used by eCos. If available on the target platform, and when suitably configured, the eCos Cortex-M HAL can make use of the Instrumentation Trace Macrocell (ITM) for diagnostics and instrumentation. Other features of the architecture such as the Memory Protection Unit, and the other debug units are not directly used by eCos at present.

Name

Options — Configuring the Cortex-M Architectural HAL Package

Description

The Cortex-M architectural HAL is included in all ecos.db entries for Cortex-M targets, so the package will be loaded automatically when creating a configuration. It should never be necessary to load the package explicitly or to unload it.

The Cortex-M architectural HAL contains a number of configuration points. Few of these should be altered by the user, they are mainly present for the variant and platform HALs to select different architectural features.

CYGINT_HAL_CORTEXM_BIGENDIAN

> This interface controls whether the CPU is run in big endian mode. It should be implemented by either the variant or platform HAL to reflect the setting of the hardware.

CYGHWR_HAL_CORTEXM_BIGENDIAN

> This option is active only if `CYGINT_HAL_CORTEXM_BIGENDIAN` is implemented. It provides the main test point for HAL, eCos and application code to test for a big endian target.

CYGHWR_HAL_CORTEXM

> The Cortex-M architecture has two main variants at present. The M1 is based on the ARMV6 architecture specification and executes an extended variant of the Thumb instruction set and has some differences in the interrupt controller. The M3 and M4 are based on the ARMV7 architecture specification and execute the Thumb2 instruction set. The M4 is an extended M3 family providing hardware FPU and DSP extensions. This option should be set using a `requires` command in the variant HAL to indicate which CPU variant is in use.

CYGINT_HAL_CORTEXM_FPU

> This interface controls whether the CPU is capable of supporting a hardware FPU (Floating Point Unit). It is the “common” FPU marker and is implemented whan either the variant or platform HAL in turn implements a supported FPU type.
>
> For example, a Cortex-M4F target may define `CYGINT_HAL_FPV4_SP_D16` when it provides the ARMv7 VFPv4-D16 architecture floating point unit.

CYGHWR_HAL_CORTEXM_FPU

> On targets which are capable of hardware FPU operation, this option is used to select whether soft or hard floating point operation is desired. It provides the main test point for HAL, eCos and application code to test for a hard-FP target. It is inactive if `CYGINT_HAL_CORTEXM_FPU` is not implemented.
>
> Even though an architecture may provide a hardware FPU, it is not always suitable for all applications. For example, there is the associated scheduler and RAM cost in preserving FPU contect for multi-threaded applications. If `CYGHWR_HAL_ARM_FPU` is enabled then some further configuration options are made available:

CYGNUM_HAL_CORTEXM_PRIORITY_MAX

> Most Cortex-M variants do not implement the full range of priorities defined by the architecture. Instead they only implement a few of the most significant bits of the 8 bit priority range. The option `CYGNUM_HAL_CORTEXM_PRIORITY_LEVEL_BITS` must be defined by the variant HAL to give this number. This option then uses that value to calculate the maximum allowable priority for interrupts.

CYGHWR_HAL_CORTEXM_DIAGNOSTICS_INTERFACE

> By default the architectural HAL does not implement diagnostic support, with the default `Serial` support being left to the variant or platform HAL.

However, if the variant provides the on-chip ITM then selecting `ITM` for this option will configure the system to use the generic architectural HAL ITM stimulus port diagnostic output. Accessing ITM diagnostic output will require corresponding support from the SWD host tools being used to connect to the hardware.

The `discard` option configures the system so that all diagnostic output is discarded. This can be used when no I/O channel is available for diagnostics.

CYGPKG_HAL_CORTEXM_SYSTEM_DEBUG_ITM

If the variant includes ITM support then this option can be enabled to allow configuration of the stimulus ports to be used for HAL diagnostics or instrumentation as required.

Compiler Flags

It is normally the responsibility of the platform HAL to define the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos. For all Cortex-M3 targets the options "-mcpu=cortex-m3" and "-mthumb" must always be defined.

When hardware floating point support is to be used, additional flags are required, as discussed later.

Linker Scripts

The linker script, supplied by either the variant or platform HALs, must define some symbols that the architecture HAL depends on:

hal_vsr_table

This defines the location of the VSR table. The address must obey the rules for locating the CPU vector table defined in the Cortex-M architecture specification. Usually it should be placed at the base of internal SRAM, at 0x20000000. The size of the table depends on the CPU variant in use.

hal_virtual_vector_table

This defines the location of the virtual vector table used to communicate between an ROM monitor and an eCos application. This table needs to be word aligned. It is usually placed in internal SRAM just after the VSR table, perhaps aligned to a convenient boundary.

hal_interrupt_stack

This defines the location of the interrupt stack, and is assigned to the CPU's MSP register. The stack grows down so it should be placed at the top of memory. It is usually placed at the top of internal SRAM. For RAM applications, which are loaded after initialization is complete, it can be placed in external RAM.

hal_startup_stack

This defines the location of the startup stack and is assiged to the CPU's PSP register. This value will be installed in slot zero of the initial vector table to be loaded automatically by the CPU on reset. The stack grows down so it should be placed near the top of memory. It is usually placed near top of internal SRAM, just below the interrupt stack. The default is to initially place this stack at the halfway point of the space allocated for the interrupt stack. This avoids allocating a unique space for it, and when the application starts it will usually move the PSP to another location, leaving all of the interrupt stack space available for the interrupts. For RAM applications, which are loaded after initialization is complete, it can be placed near the top of external RAM.

Name

Floating Point — Overview of use of floating point

Overview

The Cortex-M architectural HAL provides support for a hardware Floating Point Unit (FPU) if one is present, to provide accelerated floating point math operations.

Support is currently provided for the FPU designs found on the Cortex-M4 and Cortex-M7 architectural variants. However even with these variants, the FPU is an optional feature and a more specific classification of, for example, Cortex-M4F indicates the presence of the FPU.

Furthermore, even if an FPU is present, as indicated by a platform HAL package, it is not required to be used, and the default is not to use it (therefore defaulting to software FP) so that the developer must take the step of enabling hardware FPU support if it is desired. Equally the developer is still permitted to keep using software floating point, which may simplify and reduce code size and stack use (due to the larger register contexts required for FP) in some cases. This software floating point is provided by the compiler (GCC) runtime, based on the compiler flags in use.

Configuration

As described earlier, in order to enable hardware floating point support in this HAL package you must enable the configuration option `CYGHWR_HAL_CORTEXM_FPU` (Use hardware FPU) which can be found within the `CYGPKG_HAL_CORTEXM_FPU` (Floating Point Support) CDL component.

Configuration of the FPU support is an important step as the use of the FPU not only affects code generation and requires some initialization, but also an understanding of whether multiple kernel threads in the application may be using FP operations, in which case the method of saving/restoring the FPU register bank on context switches must be set appropriately.

Compile and link flags

Both the application and eCos must be built and linked with matching compiler/linker flags appropriate to the configuration selected for FPU support. It is usually easiest to examine the `CYGBLD_GLOBAL_CFLAGS` configuration option, or simply the build output, to see the relevant flags in use. These are the flags to look for, and a brief summary of their purpose:

`-mcpu=cortex-m4`
: No Cortex-M3 core supports FPU operations, so `-mcpu=cortex-m4` is required to allow the correct instructions to be generated. For the moment, use of this option also applies when using the Cortex-M7 although this will likely change in a future compiler update.

`-mfloat-abi=hard`
: This directs the compiler to generate FPU instructions for floating point operations. If this option is absent, the default of `-mfloat-abi=soft`, i.e. software FP is used.

`-mfpu=…`
: This option indicates which hardware FPU is present, covering the number of registers, their sizes, and so on. For the moment, only `-mfpu=fpv4-sp-d16`, as used on the Cortex-M4F and M7F, is supported. This corresponds to the VFPv4 specification with 32 single precision registers, also usable as 16 double precision registers.

Threads and context switching with FP

With the hardware FPU support enabled, it is then possible to configure the `CYGHWR_HAL_CORTEXM_FPU_SWITCH` (FPU context switch) configuration option in order to control how FPU registers are saved/restored in context switches. There are three settings: ALL, LAZY, and NONE.

ALL This mode is the most straightforward, and means that on every context switch, all FPU registers are saved and restored between threads.

This mode makes the most sense if you need determinism and/or most or all of your threads will use FP. However if few threads use FP, it can result in a lot of overhead due to saves and restores of unchanged registers.

Enabling the `ALL` mode also takes advantage of the Cortex-M lazy exception stacking feature in order to reduce interrupt/exception latency. This means that after an exception or interrupt, the core reserves space for the FPU register context on the stack, but does not actually save the FPU register contents onto the stack unless needed.

LAZY In this mode, if a thread has not used the FPU, the FPU context will not be saved or restored for it. The HAL installs exception handlers on the Cortex-M UsageFault and HardFault exceptions in order to detect the first time the FPU is accessed by that thread. Once the FPU is accessed, the fault handler enables the FPU for that thread, and from then on, the FPU context will be saved and restored when switching from or to that thread.

In a system where some or many threads do not use the FPU, this can greatly improve context switch time. However if the system spends most of its time swapping between two or more threads which do both use the FPU, then there may be additional overhead compared to the `ALL` mode (due to the need to check if the FPU was enabled for a particular thread on switch). This means the worst case context switch time is longer than with `ALL` mode. It also reduces determinism as there is an unavoidable latency at the point the thread first accesses the FPU, so that the fault handler can execute to enable the FPU; and determinism is further affected as context switch time depends on whether threads use the FPU.

The `LAZY` mode does not save on stack usage, as the number of registers which might need to be saved remains the same.

Unlike the `ALL` mode, there is not yet support for lazy exception stacking for those threads which have the FPU enabled, which means if the FPU is enabled at the time of interrupt or exception, much of the FPU register context (the FPSCR, and half the data registers) will be saved. Please contact eCosCentric if it would be of interest to enhance eCos by adding adding lazy exception stacking to the `LAZY` context switching mode.

NONE In this mode, the FPU is enabled, but no floating point context is stored at any point, which naturally means there is no overhead on context switch. However this means that only one thread or context may use the FPU at a time.

If using this mode, either all FP operations must be constrained to a single thread. Or there must be locking to ensure that multiple threads do not access the FPU registers simultaneously. But if you rely on locking, great care must be taken as the compiler has the potential to reorder floating point accesses outside of the critical region if it is still in the same function. The use of the `HAL_REORDER_BARRIER()` call from the `<cyg/hal/hal_arch.h>` HAL header can be useful to prevent reordering across a particular point in the code.

Floating point specific tests

The kernel package has a number of tests to exercise floating point operations, especially when switching threads. Some of these tests take particular account of the Cortex-M features in determining what to test. The relevant test names in the kernel package all have the prefix "fp".

FP in exception contexts

Floating point must not be used in an ISR or from kernel exception handlers. If used, FPU registers will not be restored correctly on the return from the ISR/exception.

However if the `ALL` context switch mode is in use, it is permitted to use floating point in DSR routines, including kernel alarm functions. They may also be used in `NONE` mode, but as expected, this could only be if no threads are using floating point; this can be ensured in threads by using the kernel scheduler lock to prevent DSRs from running temporarily, although clearly that has an impact on real-time behaviour. As mentioned earlier, it would also be advised to combine the lock with use of `HAL_REORDER_BARRIER()`.

Do not use floating point operations in DSRs when LAZY context switch mode is used. There is no guarantee of which thread context will be current when the DSR is run, meaning that if the interrupt occured while a thread that does not use FP was running, the DSR would cause the FPU to enabled for that thread from then on.

Name

HAL Port — Implementation Details

Description

This documentation explains how the eCos HAL specification has been mapped onto the Cortex-M hardware and should be read in conjunction with the Architecture Reference Manual and the Technical Reference Manual. It should be noted that the architectural HAL is usually complemented by a variant HAL and a platform HAL, and those may affect or redefine some parts of the implementation.

Exports

The architectural HAL provides header files `cyg/hal/hal_arch.h`, `cyg/hal/hal_intr.h` and `cyg/hal/hal_io.h`. These header files export the functionality provided by all the Cortex-M HALs for a given target, automatically including headers from the lower-level HALs as appropriate. For example the platform HAL may provide a header `cyg/hal/plf_io.h` containing additional I/O functionality, but that header will be automatically included by `cyg/hal/hal_io.h` so there is no need to include it directly.

Additionally, the architecture HAL provides the `cyg/hal/basetype.h` header, which defines the basic properties of the architecture, including endianness, data type sizes and alignment constraints.

Startup

The conventional bootstrap mechanism involves a table of exception vectors at the base of memory. The first two words of this table give the initial program counter and stack pointer. For ROM startup only these two words are defined at the beginning of the ROM image. The rest of the vector table is constructed at runtime in on-chip SRAM.

The architectural HAL provides a default implementation of the low-level startup code which will be appropriate in nearly all scenarios. For a ROM startup this includes copying initialized data from flash to RAM. For all startup types it will involve zeroing bss regions and setting up the general C environment. It will also set up the initial exception priorities, switches the CPU into the correct execution mode, enables the debug monitor and enables error exception handling.

In addition to the setup it does itself, the initialization code calls out to the variant and platform HALs to perform their own initialization. The first such function is `hal_system_init` which is called at the very start of initialization. This function is supplied by the platform HAL and should do minimal initialization to allow the rest of the initialization code to run. Typically it will set up GPIO lines, enable clocks and access to external RAM. This function runs before the data and bss sections have been initialized, so it cannot rely on global or static data. Full initialization is handled by `hal_variant_init` and `hal_platform_init`. The former should complete clock and GPIO initialization and switch from the startup clocking speed to the default rate, which may involve enabling PLLs etc. The platform initialization routine will complete any initialization needed for devices external to the microprocessor.

The architectural HAL also initializes the VSR and virtual vector tables, sets up HAL diagnostics, and invokes C++ static constructors, prior to calling the first application entry point `cyg_start`. This code resides in `src/hal_misc.c`.

The current code assumes that there is no memory management or MPU and hence will not perform any MPU initialization. Other functional units may be initialized by the variant or platform HALs.

Interrupts and Exceptions

The eCos interrupt and exception architecture is built around a table of pointers to Vector Service Routines that translate hardware exceptions and interrupts into the function calls expected by eCos. The Cortex-M vector table provides exactly this functionality, so it is used directly as the eCos VSR table. The `HAL_VSR_GET` and `HAL_VSR_SET` macros therefore manipulate the vector table directly. The `hal_intr.h` header provides definitions for all the standard Cortex-M exception vectors.

The vector table is constructed at runtime at the base of internal SRAM, which is always located at address 0x20000000, and the Vector Table Offset Register set to use it. For ROM and JTAG startup all entries are initialized. For RAM startup only the interrupt vectors are (re-)initialized to point to the VSR in the loaded code, the exception vectors are left pointing to the VSRs of the loading software, usually RedBoot or GDB stubs.

When an exception occurs it is delivered to a shared VSR, `hal_default_exception_vsr` in `vectors.S` This saves the CPU state and calls `hal_deliver_exception` in `hal_misc.c`, which passes the exception on to either the kernel or the GDB stub handler. If it returns then the CPU state is restored and the code continued.

Interrupts are numbered from zero starting at VSR table entry 15, which is the SysTick timer interrupt. The remaining interrupt numbers are defined by the variant HAL, and possibly the platform HAL. These definitions are used to declare interrupt handling tables in the architecture HAL.

When an interrupt occurs it is delivered to a shared VSR, `hal_default_interrupt_vsr`, which saves some state and calls `hal_deliver_interrupt`. This function is passed the interrupt number to be delivered, generated by subtracting 15 from the value of the IPSR register. It looks up the ISR in the interrupt tables and calls it. If the return value of the ISR has the `CYG_ISR_CALL_DSR` bit set then it calls `cyg_interrupt_post_dsr` to mark the DSR for execution and also sets the PENDSVSET bit in the NVIC ICSR register to set the PendSVC exception pending.

Interrupts are delivered onto the main or interrupt stack, which differs from the process stack that threads execute on. The interrupt priority mechanism allows interrupts to nest on the interrupt stack (irrespective of the CDL option `CYGSEM_HAL_COMMON_INTERRUPTS_ALLOW_NESTING`) and only when the last interrupt has been executed will the PendSVC exception be called. The PendSVC handler arranges for `interrupt_end` to be called by pushing a new exception frame on the process stack, preserving its own exception frame, and returning. This causes `interrupt_end` to be called in thread mode on the process stack, which will cause any pending DSRs to be called, and a context switch to a new thread if necessary. When execution resumes on this thread it returns to `hal_interrupt_end_done`, which uses a SWI to pop its own exception frame and use the preserved PendSVC frame to resume the interrupted thread where it left off.

The architectural HAL provides default implementations of `HAL_DISABLE_INTERRUPTS`, `HAL_RESTORE_INTERRUPTS`, `HAL_ENABLE_INTERRUPTS` and `HAL_QUERY_INTERRUPTS`. These involve manipulation of the CPU BASEPRI register. Similarly there are default implementations of the interrupt controller macros `HAL_INTERRUPT_MASK`, and `HAL_INTERRUPT_UNMASK` macros. These manipulate the NVIC interrupt mask registers, and the TICKINT bit of the SYSTICK CSR register. `HAL_INTERRUPT_ACKNOWLEDGE` and `HAL_INTERRUPT_CONFIGURE` are no-ops at the architectural level.

`HAL_INTERRUPT_SET_LEVEL` manipulates the NVIC interrupt priority registers. The valid range of interrupts supported depends on the number of interrupt priority bits supported by the CPU variant. Priority level 0 is reserved for exceptions and the debug monitor. Interrupts are only allowed to start at the first implemented priority below this: 0x10 if the CPU implements 4 priority bits, 0x20 if it implements 3, and 0x01 if it implements all 8. This macro shifts the priority level supplied to start at the implemented maximum and clamps the higher end to 0xFF. So on a CPU that implements 4 priority bits, level 0 will be mapped to 0x10, levels above 0xf0 will all be mapped to 0xFF.

For all of these macros, a variant specific version may also be defined: `HAL_VAR_INTERRUPT_MASK`, `HAL_VAR_INTERRUPT_UNMASK`, `HAL_VAR_INTERRUPT_ACKNOWLEDGE`, `HAL_VAR_INTERRUPT_SET_LEVEL` and `HAL_VAR_INTERRUPT_CONFIGURE`. These are each called by the architecture macros after any architecture defined operations are completed. These macros allow the variant HAL to modify the architecture HAL support, or implement further interrupts that are not directly supported by the NVIC. In support of this, the variant HAL must define `CYGNUM_HAL_INTERRUPT_NVIC_MAX` which separates interrupts handled by the NVIC from any extended vectors defined by the variant HAL.

Stacks and Stack Sizes

`cyg/hal/hal_arch.h` defines values for minimal and recommended thread stack sizes, `CYGNUM_HAL_STACK_SIZE_MINIMUM` and `CYGNUM_HAL_STACK_SIZE_TYPICAL`. These values depend on a number of configuration options.

The Cortex-M architecture HAL always uses a separate stack for startup and interrupt handling. This is usually allocated to uninitialized memory at the top of the available internal or external RAM, depending on the startup type. Thus the configuration option `CYGIMP_HAL_COMMON_INTERRUPTS_USE_INTERRUPT_STACK` has no effect.

Thread Contexts and setjmp/longjmp

`cyg/hal/hal_arch.h` defines a thread context data structure, the context-related macros, and the `setjmp`/`longjmp` support. The implementations can be found in `src/context.S`. The context structure is defined as a discriminated union with different layouts for thread, exception and interrupt saved states. This approach allows the most efficient code and layout to be used in each context. The only expense is that debug code must be slightly more careful in accessing a saved state.

Bit Indexing

The architectural HAL provides inline assembler implementations of `HAL_LSBIT_INDEX` and `HAL_MSBIT_INDEX` which use the CPU count-leading-zero instruction.

Idle Thread Processing

The architecture HAL provides a default `HAL_IDLE_THREAD_ACTION` implementation that executes a `WFI`, wait for interrupt, instruction. This puts the CPU into a low power mode ready to respond quickly to the next interrupt.

A potential problem can occur with this however, as this instruction is known to cause difficulties when debugging via a JTAG hardware debugger. A frequent symptom is a report from the debugger that it was unable to stop the target. Therefore if using a JTAG debugger, it is strongly recommended to disable the use of `WFI` by enabling the configuration option titled "Disable HAL-specific idle action" (`CYGIMP_KERNEL_THREADS_IDLE_NO_HAL_ACTION`) which can be found in the eCos kernel package, in the "Thread-related options" component. This automatically happens when additional eCos debugging support is enabled using `CYGPKG_INFRA_DEBUG`, or if the HAL startup type (`CYG_HAL_STARTUP`) is set to "JTAG". But it needs to be set manually for other startup types, notably when debugging an application installed into Flash, which would have "ROM" startup type.

When using Single Wire Debug (SWD) hardware debuggers this is not an issue. Since the `CYG_HAL_STARTUP` setting "JTAG" is used irrespective of the hardware debugger type this would normally disable the use of `WFI` within the idle thread. However, when ITM is configured as available, the `CYGHWR_HAL_CORTEXM_SYSTEM_DEBUG_ALLOW_IDLE` option can be enabled to override the disabling, allowing the normal `WFI` idle behaviour.

Clock Support

The architectural HAL provides a default implementation of the various system clock macros such as `HAL_CLOCK_INITIALIZE`. These macros use the architecture defined SysTick timer to implement the eCos system clock. The architecture HAL expects the variant HAL to define and initialize a variable named `hal_cortexm_systick_clock`, which should contain the frequency in Hz of the clock supplied to the SysTick timer input. To allow for varying CPU clock rates, the SysTick timer is always programmed to take a 1MHz input clock, and `CYGNUM_HAL_RTC_PERIOD` is then expressed in terms of this.

HAL I/O

The Cortex-M architecture does not have a separate I/O bus. Instead all hardware is assumed to be memory-mapped. Further it is assumed that all peripherals on the memory bus will switch endianness with the processor and that there is no need for any byte swapping. Hence the various HAL macros for performing I/O simply involve pointers to volatile memory.

The variant and platform files included by the `cyg/hal/hal_io.h` header will typically also provide details of some or all of the peripherals, for example register offsets and the meaning of various bits in those registers.

Cache Handling

The current Cortex-M implementations do not support caches, so no cache handling is currently included in the architecture port. Instead it is the responsibility of the variant HAL to supply the `cyg/hal/hal_cache.h` header.

Linker Scripts

The architectural HAL will generate the linker script for eCos applications. This involves the architectural file src/cortexm.ld and a .ldi memory layout file, typically provided by the platform HAL. It is the .ldi file which places code and data in the appropriate places for the startup type, but most of the hard work is done via macros in the cortexm.ld file.

Diagnostic Support

The architectural HAL implements diagnostic support for ITM stimulus port if available, or for discarding all output. However, by default, the diagnostics output is left to the variant or platform HAL, depending on whether suitable peripherals are available on-chip or off-chip. The CYGHWR_HAL_CORTEXM_DIAGNOSTICS_INTERFACE can be configured to direct the diagnostic output support used.

See Cortex-M Hardware Debug for more detail regarding using the ITM stimulus port.

SMP Support

The Cortex-M architectural HAL does not provide any SMP support.

Debug Support

The architectural HAL provides basic support for gdb stubs using the debug monitor exceptions. Breakpoints are implemented using a fixed-size list of breakpoints, as per the configuration option CYGNUM_HAL_BREAKPOINT_LIST_SIZE. When a JTAG device is connected to a Cortex-M device, it will steal breakpoints and other exceptions from the running code. Therefore debugging from RedBoot or the GDB stubs can only be done after detaching any JTAG debugger and power-cycling the board.

HAL_DELAY_US() Macro

cyg/hal/hal_intr.h provides a simple implementation of the HAL_DELAY_US macro based around reading the SysTick timer. The timer must therefore be initialized before this macro is used, and HAL_CLOCK_INITIALIZE() is called during initialization after the variant and platform initialization functions are called, but before constructors are invoked.

Profiling Support

The Cortex-M variant may support the Data Watchpoint and Trace (DWT) feature, and if available then it is possible to use the DWT to provide non-intrusive PC sampling without any eCos run-time configuration or support being needed. The SWD hardware debugger being used controls the enabling and processing of PC sample data for subsequent use by profiling tools.

When using local memory based profiling the Cortex-M architectural HAL implements the mcount function, allowing profiling tools like gprof to determine the application's call graph. It does not implement the profiling timer. Instead that functionality needs to be provided by the variant or platform HAL.

Name

Cortex-M Hardware Debug — Overview of hardware debug features

Introduction

Some Cortex-M designs may include the Instrumentation Trace Macrocell (ITM) and Data Watchpoint and Trace (DWT) features. The ITM allows for software generated “instrumentation” to be output via the Single Wire Debug (SWD) hardware interface in a relatively non-intrusive fashion. The SWD ITM support provides for much higher data rates than can normally be achieved via a UART, so they can provide for software controlled debug I/O support with a significantly reduced system overhead, which may be important for some eCos applications.

There is no explicit eCos support for making use of the DWT features, since the hardware debug tools being used can control the feature use independently of the eCos application. The specific hardware debug tools being used may provide support as required. For example the Ronetix PEEDI has built-in support for tracking DWT PC sample events for profiling.

Instrumentation Trace Macrocell (ITM)

If the Cortex-M ITM support is enabled in the CDL configuration then HAL diagnostics can be directed to a ITM stimulus port, and if instrumentation is enabled then event records directed to a different stimulus port. How the output data is processed or captured is dependent on the hardware tool being used to drive the SWD interface.

The following sections give a brief overview of using the Ronetix PEEDI hardware debugger, and the OpenOCD tool in conjunction with the STMicroelectronics ST-LINKv2 adapter. This is by no means an exhaustive list since many 3rd-party hardware tools exist, and any tool capable of handling ITM stimulus port output can normally be configured to accept console (HAL diagnostic) input or record captured data (instrumentation). More detail of specific variant or target SWD/JTAG setup and ITM stimulus port support may be found in the relevant eCos platform specific documentation.

Ronetix PEEDI

The documentation supplied with the Ronetix PEEDI provides an overview of using the Cortex-M ITM features. However, the example PEEDI configuration file *BASE_DIR*`/packages/hal/cortexm/arch/current/misc/peedi.cortexm3.swd.cfg` provided with eCos has comments on how the PEEDI can be configured to provide access to the data written to ITM stimulus ports on specific TCP network ports. It also provides an example of using the DWT PC sampling feature to gather non-intrusive application execution profiling.

The simplest use would be to use the telnet command to connect to the stimulus port connection being used for HAL diagnostics. When an application configured for ITM diagnostics is executed then the output will appear on the configured network port:

```
$ telnet peedi-0 2001
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
INFO:<code from 0x20000290 -> 0x2000e254, CRC 2a20>
…
```

Similarly the Linux nc command could be used to save data from an instrumentation port to a file for post-processing. For example, if the eCos application is built with Kernel instrumentation enabled and configured for ITM output, then the stimulus port output can be captured and processed using the example instdump tool provided in the Kernel package.

```
$ nc peedi-0 2003 > inst.bin
^C
$ instdump -r inst.bin elfapp
Threads:
 threadid 1 threadobj 200045D0 "idle_thread"

     0:[THREAD:CREATE][THREAD 4095][TSHAL 4][TSTICK 0][ARG1:200045D0] { ts 4 microseconds }
```

```
    1:[SCHED:LOCK][THREAD 4095][TSHAL 45][TSTICK 0][ARG1:00000002] { ts 45 microseconds }
    …
```

OpenOCD

The OpenOCD ITM support is limited to using the ST-LINKv2 hardware interface to the target board. For example, as built-in to the STM32F4DISCOVERY and STM32x0G-EVAL platforms. The ST-LINKv2 firmware on the boards should be at least `JTAG v17`, so older boards may need to have their ST-LINK firmware updated via the relevant STMicroelectronics tool prior to being used for ITM stimulus port support.

The *`BASE_DIR`*`/packages/hal/cortexm/stm32/stm32f4dis/current/misc/openocd.stm32f4dis.cfg` file provides an example OpenOCD configuration file that captures ITM stimulus port data and directs it to a file named `tpiu.out` in the current directory at the time **openocd** was invoked. This file can then be interpreted by a program **parseitm** which can be found in this package in the directory `packages/hal/cortexm/arch/`*`VERSION`*`/host` relative to the root of your eCos installation. It can be compiled simply with:

```
$ gcc -o parseitm parseitm.c
```

You simply run it with the desired ITM stimulus port and name of the file containing the ITM output, and it will echo all ITM stimulus for that port, continuing to read from the file until interrupted with Ctrl-C for example:

```
$ parseitm -p 31 -f tpiu.out
```

Note

Note that limited buffer space in debug hardware such as the ST-LINK can result in some ITM data becoming lost due to buffer overruns. eCosPro provides a workaround of throttling data within the `CYGHWR_HAL_CORTEXM_ITM_DIAGNOSTICS_THROTTLE` CDL configuration component of this package in order to reduce or avoid lost ITM data.

Two approaches are available: "Chunk" or "Per-Character" throttling. In the case of Chunk throttling, a configurable number of characters are output before a delay is applied to allow the buffer to be emptied. However this can result in a longer delay while waiting for the buffer to empty. Alternatively, Per-Character throttling imposes a small delay after every character, whether it is needed or not.

More detail on the behaviour of these configuration options can be found in the description of the "ITM Throttling" component (`CYGHWR_HAL_CORTEXM_ITM_DIAGNOSTICS_THROTTLE`) and its sub-options within the eCos Configuration Tool.

These are just default examples; it is also possible to configure the eCos kernel to send its instrumentation output via ITM, and applications may make use of other stimulus ports for application-specific functionality.

Chapter 286. Kinetis Variant HAL

Name

`CYGPKG_HAL_CORTEXM_KINETIS` — eCos Support for the Kinetis Microprocessor Family

Description

The Freescale Kinetis K series of Cortex-M microcontrollers is supported by eCos with an eCos processor variant HAL and a number of device drivers supporting some of the on-chip peripherals. These include device drivers for the on-chip flash, serial, I²C, SPI, Ethernet, RTC/wallclock and watchdog devices. In addition it provides common functionality and definitions that Kinetis based platform ports may require, as well as definitions useful to application developers.

This documentation covers the Kinetis functionality provided, but should be read in conjunction with the specific HAL documentation for the platform port. That documentation will cover issues that are platform-specific and are not covered here, and may also describe differences that override or supersede what the Kinetis variant HAL provides. The areas that are specific to platfom HALs and not the Kinetis variant HAL include:

- memory map and related configuration and setup
- Clock parameters
- GPIO setup
- Any special handling for external interrupts, or additional interrupts
- Diagnostic I/O baud rates
- Additional diagnostic I/O devices, if any
- LED/LCD control

Name

On-chip Subsystems and Peripherals — Hardware Support

Hardware support

On-chip memory

The Freescale Kinetis parts include on-chip SRAM, and on-chip FLASH. The SRAM and FLASH can vary in size depending on the model. There is also support in some models for external memory, which eCos may use where available.

Typically, an eCos platform HAL port will expect a GDB stub ROM monitor or RedBoot image to be programmed into the Kinetis on-chip FLASH memory for development, and the board would boot this image from reset. The stub ROM/RedBoot provides GDB stub functionality so it is then possible to download and debug stand-alone and eCos applications via the gdb debugger using serial interfaces or other debug channels. The JTAG interface may also be used for development if a suitable JTAG device is available. If RedBoot is present it may also be used to manage the on-chip and external flash memory. For production purposes, applications are programmed into external or on-chip FLASH and will be self-booting.

On-chip FLASH

The package `CYGPKG_DEVS_FLASH_KINETIS` ("Kinetis FLASH memory support") provides a driver for the on-chip flash. This driver conforms to the Version 2 flash driver API, and is automatically enabled if the generic "Flash device drivers" (`CYGPKG_IO_FLASH`) package is included in the eCos configuration. The driver will configure itself automatically for the size and parameters of the specific Kinetis variant which has been selected in the eCos configuration.

Cache Handling

The variant HAL supplies the `cyg/hal/hal_cache.h` header file to describe cache support to the generic eCos code. Some Kinetis K series parts have access to a Cortex-M4 core cache (the `CYGINT_HAL_CACHE` will be defined by the platform if appropriate). If the configured part does not provide a cache then the header will supply null macros for the required functions.

When a cache is available the HAL currently supports a 16KiB cache, split into a 8KiB data cache and a 8KiB instruction cache.

The architecture HAL also defines a macro `HAL_MEMORY_BARRIER()` which acts to synchronize the pipeline, delaying execution until all previous operations, including all pending writes, are complete. This will usually be necessary when interacting with devices that access memory directly.

Note

The use of the default Cortex-M architecture `HAL_MEMORY_BARRIER` macro requires that the CPU core supports the barrier instructions (`ID_ISAR4[19..16]` is not `0`). This is the case for the Freescale Kinetis family.

Serial I/O

The Kinetis variant HAL supports basic polled HAL diagnostic I/O over any of the on-chip serial devices. There is also a fully interrupt-driven serial device driver suitable for eCos applications for all on-chip serial devices. The serial driver consists of an eCos package: `CYGPKG_IO_SERIAL_FREESCALE_UART` which provides all support for the Kinetis on-chip serial devices. Using the HAL diagnostic I/O support, any of these devices can be used by the ROM monitor or RedBoot for communication with GDB. If a device is needed by the application, either directly or via the serial driver, then it cannot also be used for GDB communication using the HAL I/O support. An alternative serial port should be used instead.

The HAL defines CDL interfaces, `CYGINT_HAL_FREESCALE_UART0` to `CYGINT_HAL_FREESCALE_UART5` for each of the available UARTs. The platform HAL CDL should contain an **implements** directive for each such UART that is available for use on the board. This will enable use of the UART for diagnostic use.

The Kinetis UARTs provide `TX` and `RX` data lines plus hardware flow control using `RTS`/`CTS` for those UARTs that have the signals available on the platform hardware.

Note

The `UART0` and `UART1` devices are clocked from the main core clock, with the remaining UARTs clocked from the bus clock. This can affect baud rate accuracy at higher configured speeds depending on the main CPU clock configuration, and the physical UART being used.

Interrupts

The Kinetis HAL relies on the architectural HAL to provide support for the interrupts directly routed to the NVIC. The `cyg/hal/var_intr.h` header defines the vector mapping for these.

GPIO

The variant HAL provides support for packaging the configuration of a GPIO line into a single 32-bit descriptor that can then be used with macros to configure the pin and set and read its value. This is useful to drivers and other packages that need to configure and use different I/O lines for different devices.

A descriptor is created with the `CYGHWR_HAL_KINETIS_PIN(port,bit,mux,cnf)` macro, where the parameters required are:

`port`
: This identifies the GPIO port to which the pin is attached. Ports are identified by the letters from `A` to `F`.

`bit`
: This gives the bit or pin number within the port. These are numbered from `0` to `31`.

`mux`
: This paramater specifies how the pin should be used, and should be in the range `0` to `7`. The value `0` corresponds to the pin being disabled (Analog), with `1` used to specify GPIO control. The other values `2` to `7` are used to indicate alternative function mappings as defined by the CPU variant.

`cnf`
: This option allows explicit Port Control register configuration settings to be specified. The bits as held in the *`cnf`* value are defined as per the relevant Kinetis Reference Manual.

The macro `CYGHWR_HAL_KINETIS_PIN_NONE` may be used in place of a pin descriptor and has a value that no valid descriptor can take. It may therefore be used as a placeholder where no GPIO pin is present or to be used. This can be useful when defining pin configurations for a series of instances of a peripheral (e.g. UART ports), but where not all instances support all the same pins (e.g. hardware flow control lines).

The function `hal_set_pin_function(pin)` configures the pin according to the descriptor and must be called before other GPIO operations are performed on the pin.

The macros to manipulate GPIO state all take a suitably constructed GPIO pin descriptor as an argument. It is recommended to consult the header file `<cyg/hal/var_io_gpio.h>` (also present in the `include` subdirectory of the Kinetis variant HAL package within the eCos source repository), for the complete list if needed.

RTC/Wallclock

eCos includes a RTC (known in eCos as a wallclock) device driver for the on-chip RTC in the Kinetis family. This is located in the package `CYGPKG_DEVICES_WALLCLOCK_KINETIS_RTC` (“Real-time clock”).

Profiling Support

The Kinetis HAL contains support for **gprof**-based profiling using a sampling timer. The default timer used is `PIT0`, which is one of the basic periodic interrupt timers, leaving the more complex timers for application code. The timer used is selected by the CYGHWR_HAL_CORTEXM_KINETIS_PROFILE_PIT_CHANNEL configuration option. This timer is only enabled when the gprof profiling package (CYGPKG_PROFILE_GPROF) is included and enabled in the eCos configuration, otherwise it remains available for application use.

Clock Control

The CDL section CYGHWR_HAL_CORTEXM_KINETIS_CLOCKING contains many options to configure the various Kinetis on-chip clocks, based on platform supplied default values. The CDL will calculate the main CYGNUM_HAL_CORTEXM_KINETIS_MCGOUT_FREQ frequency value, which in turn is used in conjunction with the configured clock divider values to set the frequencies of the relevant subsystems.

The actual calculated values of the main clocks, in Hz, are stored in the global variables hal_kinetis_sysclk and hal_kinetis_busclk. The clock supplied to the Cortex-M SysTick timer, HCLK/8, is also assigned to the global variable hal_cortexm_systick_clock. These variables are used, rather than configuration options, in anticipation of future support for power management by varying the system clock rate.

Note that when changing or configuring any of these clock settings, you should consult the relevant processor datasheet as there may be both upper and lower constraints on the frequencies of some clock signals, including intermediate clocks. There are also some clocks where, while there is no strict constraint, clock stability is improved if values are chosen wisely. Finally, be aware that increasing clock speeds using this package may have an effect on platform specific properties, such as memory timings which may have to be adjusted accordingly.

Chapter 287. Freescale TWR-K60N512 and TWR-K60D100M Platform HAL

Name

CYGPKG_HAL_CORTEXM_KINETIS_TWR_K60N512 — eCos Support for the Freescale TWR-K60N512 and TWR-K60D100M boards

Description

The Freescale TWR-K60N512 and TWR-K60D100M boards are almost identical, so the CYGPKG_HAL_CORTEXM_KINETIS_TWR_K60N512 package provides support for both variants. The only functional differences are whether a revision 1 or revision 2 Kinetis K60 sub-family CPU is used, and a different on-board 3-axis accelerometer.

The TWR-K60N512 board has a MK60N512VMD100 revision 1 microcontroller and the TWR-K60D100M board has a MK60DN512VMD10 revision 2 microcontroller. Both microcontrollers incorporate 512KB of internal flash ROM and 128KB of internal SRAM.

In either case the stand-alone board may be targetted, but for access to some peripherals it is assumed that a TWR-ELEV setup, with a TWR-SER daughterboard is being used. For example, with a TWR-SER board present it provides a connector for Ethernet. The motherboards have limited I/O interfaces, with most of the I/O signals being propogated via multi-pin connectors.

The motherboards also provide an on-board JTAG debug circuit (OSJTAG) with virtual serial port, a 3-axis accelerometer, a potentiometer, a MicroSD card slot, and some LEDs and buttons.

The TWR-SER daughterboard also provides in addition to the Ethernet connector access to a RS232/484 DB9 connection, a USB Mini-AB connector and a 3-pin CAN connector.

For these boards, the expected eCos development model is that programs may be downloaded and debugged via the on-board OSJTAG USB interface, or via a hardware debugger (JTAG/SWD) attached to the JTAG socket. However, if required, it is possible to build and install RedBoot or a GDB stub image into the internal FLASH so that the CPU boots directly into that monitor, and that the GDB debugger accesses the monitor via serial or Ethernet.

This documentation describes platform-specific elements of the TWR-K60N512 and TWR-K60D100M board support within eCos. The Kinetis variant HAL documentation covers various topics including HAL support common to Kinetis variants, and on-chip device support. This document complements the Kinetis documentation.

Supported Hardware

The K60 parts used have two on-chip memory regions. There is a SRAM region of 128KiB present at 0x1FFF0000 and a 512KiB FLASH region present at 0x00000000.

The Kinetis variant HAL includes support for the six on-chip serial devices which are documented in the variant HAL. For these boards UART3 is connected to the J8 DB9 connector, with hardware flow control (RTS/CTS) lines available. Additionally UART5 is connected as a virtual serial through the on-board OSJTAG J13 USB connection, without support for hardware flow control lines.

Device drivers are provided for the Kinetis on-chip Ethernet MAC, I²C interface and SPI interface. Additionally support is provided for the on-chip watchdog, RTC (wallclock) and a flash driver exists to permit management of the Kinetis's on-chip flash.

The Kinetis K60 processor, and the TWR-K60N512/TWR-K60D100M platforms, provide a wide variety of peripherals, but unless support is specifically indicated it should be assumed that support is not included.

Tools

The board port is intended to work with GNU tools configured for an **arm-eabi** target. The original port was done using **arm-eabi-gcc** version 4.7.3a, **arm-eabi-gdb** version 7.6, and **binutils** version 2.23.2.

Name

Setup — Preparing the TWR-K60N512/TWR-K60D100M Board for eCos Development

Overview

Since the target motherboard provides a built-in hardware debug solution, it is expected that the most common development method when targeting the CPU is to use this hardware debug interface (instead of GDB stubs or RedBoot) for development. This will either be by loading smaller applications into on-chip SRAM, or by programming larger applications directly into on-chip flash. In the first case, eCos applications should be configured for the variant JTAG startup type, and in the second case for the variant ROM startup type.

Nevertheless, it is still possible to program RedBoot or a GDB stub ROM image into on-chip flash and download and debug via a serial UART, if pins for the UART are available. In that case, eCos applications are configured for a variant RAM startup and then downloaded and run on the board via the debugger **arm-eabi-gdb**, or via the Eclipse IDE. For serial communications, all versions run with 8 bits, no parity, and 1 stop bit at 38400 baud. This rate can be changed in the eCos configuration used for building the RedBoot or GDB stub ROM image.

HAL Startup Types

For the `twr_k60n512` and `twr_k60d100m` platforms the Kinetis variant HAL support provides some common startup types. The following variant HAL provided startup types may be selected for applications:

Configuration	Description
ROM	Stand-alone programs running from internal FLASH
SRAM	Programs loading via hardware debugger into on-chip SRAM, but expecting RedBoot or GDB stub ROM
RAM	Programs loading via RedBoot or GDB stub ROM into on-chip SRAM
JTAG	Stand-alone programs running from on-chip SRAM, loaded via hardware debugger

Further details are available later in this manual.

Preparing OSJTAG interface

The support for using the on-chip OSJTAG interface for hardware debugging and diagnostic output requires that the OSJTAG firmware is at least version `v30.21`. The firmware for the OSJTAG interface can be checked, and updated if needed, using the relevant firmware updater tool available for download via the Freescale website. Unfortunately the official firmware updater is only available for the Windows platform at the moment.

Programming ROM images

To program ROM startup applications into flash, including the GDB stub ROM or RedBoot, a hardware debugger that understands the Kinetis flash may be used. For example, the Ronetix PEEDI provides suitable support.

Warning

Due to a security feature of the Kinetis CPUs care should be taken to avoid completely erasing the flash to ensure the required `FSEC` value in the flash configuration field is not lost. For example when using the PEEDI the user should NOT use the “erase” suffix to the **flash program** command. The Kinetis aware **flash erase chip** should be executed to erase the flash prior to using the **flash program** command.

Programming ROM images with a Ronetix PEEDI

This section describes how to program ROM images using a Ronetix PEEDI debugger.

The PEEDI must be configured to allow communication with your local network, and configured with the parameters for interfacing with the target board. It must then be used to download and program the ROM image into the internal flash. The following steps give a typical outline for doing this. Consult the PEEDI documentation for alternative approaches, such as using FTP or HTTP instead of TFTP.

Preparing the Ronetix PEEDI JTAG debugger

1. Prepare a PC to act as a host and start a TFTP server on it.

2. Connect the PEEDI JTAG debugger via both serial and ethernet to the host PC and power it on. Use the serial cable supplied with the PEEDI (straight through, not null modem).

3. Verify the PEEDI is using up-to-date firmware, of version 12.3.0 or later. Older PEEDI firmware does not support the Kinetis family correctly, particularly if wishing to use the PEEDI's own '**flash**' commands to modify the on-chip flash. If the firmware is not recent enough, follow the PEEDI User Manual's instructions which describe how to update the PEEDI firmware.

4. Locate the PEEDI configuration file `peedi_twr_k60n512.cfg` within the eCos platform HAL package in the source repository. This will be in the directory `packages/hal/cortexm/kinetis/twr_k60n512/`*`VERSION`*`/misc` relative to the root of your eCos installation.

5. Place the PEEDI configuration file in a location on the PC accessible to the TFTP server. Later you will configure the PEEDI to load this file via TFTP as its configuration file.

6. Open `peedi_twr_k60n512.cfg` in an editor such as emacs or notepad and insert your own license information in the `[LICENSE]` section.

7. Install and configure the PEEDI in line with the PEEDI Quick Start Guide or User's Manual, especially configuring PEEDI's RedBoot with the network information. Configure it to use the `peedi_twr_k60n512.cfg` target configuration file on the TFTP server at the appropriate point of the **config** process, for example with a path such as: `tftp://192.168.7.9/peedi_twr_k60n512.cfg`

8. Reset the PEEDI.

9. Connect to the PEEDI's CLI interface via TCP/IP on the standard telnet port 23. The **telnet** application is suitable for this. You should see output similar to the following:

```
$ telnet 192.168.7.225
Trying 192.168.7.225...
Connected to 192.168.7.225.
Escape character is '^]'.

PEEDI - Powerful Embedded Ethernet Debug Interface
Copyright (c) 2005-2011 www.ronetix.at - All rights reserved
Hw:1.2, L:JTAG v1.6 Fw:13.3.0, SN: PD-XXXX-XXXX-XXXX
------------------------------------------------------------

twr_k60n512>
```

Preparing the TWR-K60N512/TWR-K60D100M board for programming with PEEDI

Follow the steps in this section in order to allow communication between the board and the host PC, and between the board and the PEEDI device.

If programming a RedBoot, GDB stub ROM, or an application, which uses serial output, you should first:

1. If OSJTAG firmware 30.21 or later is installed then the motherboard J13 USB provides a standard CDC-ACM virtual serial connection to the host computer. The alternative is to connect a null modem cable between the DB9 RS232 connector on the TWR-SER daughterboard and the host computer.

2. Start a suitable terminal emulator on the host computer such as **minicom** on Linux or PuTTY on Windows. Set the communication parameters to 38400 baud, 8 data bits, no parity bit and 1 stop bit with no flow control.

For all applications, you must:

1. Connect the board to the PEEDI using an appropriate 20-pin cable from the JTAG interface connector to the Target port on the PEEDI. This will normally be a PEEDI-CORTEX20 adapter.

2. Power up the `TWR-K60N512`/`TWR-K60D100M` board.

3. Connect to the PEEDI's telnet CLI on port 23 as before.

4. Confirm correct connection with the PEEDI with the **reset reset** command as follows:

   ```
   twr_k60n512> reset reset
   ++ info: user reset
   twr_k60n512>
   ++ info: RESET and TRST asserted
   ++ info: TRST released
   ++ info: TAP : IDCODE = 0x2BA01477, Cortex M3 SWD
   ++ info: RESET released
   ++ info: core connected

   CORE0 -> CortexM4 - stopped by breakpoint
            PC=0x1FFF0006, xPSR=0x01000000

   core #0 stopped
   ++ info: core 0: initialized

   twr_k60n512>
   ```

Installation into flash

The following describes the procedure for installing a ROM application into on-chip flash, using the GDB stub ROM image as an example of such an application.

1. Use **arm-eabi-objcopy** to convert the linked application, in ELF format, into binary format. For example:

   ```
   $ arm-eabi-objcopy -O binary programname programname.bin
   ```

2. Copy the binary file (.bin file) into a location on the host computer accessible to its TFTP server.

3. Connect to the PEEDI's telnet interface, and program the image into flash with the following commands, replacing *`TFTP_SERVER`* with the address of the TFTP server and *`/BINPATH`* with the location of the .bin file relative to the TFTP server root directory. For example for a RedBoot ROM image:

   ```
   twr_k60n512> flash erase chip

   erasing chip at 0x00000000
   done.

   ++ info: successfully erased 1 byte in 0.33 sec

   twr_k60n512> flash program tftp://TFTP_SERVER/BINPATH/redboot.bin bin 0x00000000
   ++ info: Programming image file: tftp://TFTP_SERVER/redboot.bin
   ++ info: Programming directly
   ++ info: At absolute address:    0x00000000
   programming at 0x00000000
   programming at 0x00001000
   programming at 0x00002000
   programming at 0x00003000
   programming at 0x00004000
   programming at 0x00005000
   ```

```
programming at 0x00006000
programming at 0x00007000
programming at 0x00008000
programming at 0x00009000
programming at 0x0000A000
programming at 0x0000B000
programming at 0x0000C000
programming at 0x0000D000
programming at 0x0000E000
programming at 0x0000F000
programming at 0x00010000
programming at 0x00011000
programming at 0x00012000
programming at 0x00013000
programming at 0x00014000
programming at 0x00015000

++ info: successfully programmed 88.00 KB in 0.74 sec

twr_k60n512>
```

The installation into flash is now complete. For applications which print output on startup to the USART3 RS232 serial port, such as the GDB stub ROM application, this can easily by tested by powering off the board, disconnecting the JTAG, and then powering on the board again. In the case of the GDB stub ROM image, output similar to the following should be visible (although specific numbers may differ):

```
$T050f:72250008;0d:f0ff0120;#8a
```

RedBoot Installation

Only the `ROM_VAR` RedBoot configuration is supported. This is for RedBoot running from internal FLASH, using on-chip SRAM. For serial communications this uses 8 bits, no parity, and 1 stop bit at 38400 baud. This rate can be changed using the RedBoot **baud** command, or in the eCos configuration used for building RedBoot.

Rebuilding RedBoot

Should it prove necessary to rebuild a RedBoot binary, this is done most conveniently at the command line. The steps needed to rebuild the ROM version of RedBoot for the `TWR-K60N512` are:

```
$ mkdir redboot_twr_k60n512_rom
$ cd redboot_twr_k60n512_rom
$ ecosconfig new twr_k60n512 redboot
[ … ecosconfig output elided … ]
$ ecosconfig import $ECOS_REPOSITORY/hal/cortexm/kinetis/twr_k60n512/VERSION/misc/redboot_ROM_VAR.ecm
$ ecosconfig tree
$ make
```

At the end of the build the `install/bin` subdirectory should contain the file `redboot.bin`. This may be programmed to the board using the above procedure, or by using RedBoot's own flash update mechanisms.

The other versions of RedBoot - RAM, ROM_FPU, ROM_VAR - may be similarly built by choosing the appropriate alternative .ecm file.

Initializing RedBoot flash Configuration

RedBoot manages the internal flash for the storage of application programs and configuration data. The flash needs to be initialized with the following commands:

```
RedBoot> fis init
About to initialize [format] FLASH image system - continue (y/n)? y
*** Initialize FLASH Image System
```

```
... Erase from 0x0007f800-0x0007ffff: .
... Program from 0x2000e800-0x2000f000 to 0x0007f800: .
RedBoot> fconfig -i
Initialize non-volatile configuration - continue (y/n)? y
Run script at boot: false
Console baud rate: 115200
Update RedBoot non-volatile configuration - continue (y/n)? y
... Erase from 0x0007f800-0x0007ffff: .
... Program from 0x2000e800-0x2000f000 to 0x0007f800: .
RedBoot>
```

Issue the **reset** command to RedBoot, and verify the the target board comes up as expected with the correct settings.

You may also need to run the **fconfig -i** command if you have updated your RedBoot from a previous version with a different configuration which might not have any new config fields used by the newly programmed RedBoot.

Name

Configuration — Platform-specific Configuration Options

Overview

The CYGPKG_HAL_CORTEXM_KINETIS_TWR_K60N512 platform HAL package is loaded automatically when eCos is configured for either the twr_k60n512 or twr_k60d100m targets. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The target board platform HAL package supports the standard variant HAL startup types:

ROM
: This startup type can be used for finished applications which will be programmed into internal flash ROM at location 0x00000000. Data and BSS will be put into on-chip SRAM starting from 0x1FFF0000. The application will be self-contained with no dependencies on services provided by other software. The program expects to boot from reset with ROM mapped at location zero. eCos startup code will perform all necessary hardware initialization.

RAM
: This is the startup type which is used if relying on either a RedBoot or a GDB stub ROM image programmed into internal flash to download and run applications into on-chip SRAM via arm-eabi-gdb. It is assumed that the hardware has already been initialized by the ROM monitor. By default the application will use the eCos virtual vectors mechanism to obtain services from the ROM monitor, including diagnostic output.

JTAG
: This is the startup type used to build applications that are loaded via a hardware debug interface into on-chip SRAM. The application will be self-contained with no dependencies on services provided by other software. For the variant JTAG startup type the eCos run-time will perform all necessary hardware initialization.

Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option CYGSEM_HAL_ROM_MONITOR should be set. Typically this option is set only when building the GDB stub ROM (or RedBoot).

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option CYGSEM_HAL_USE_ROM_MONITOR should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then the serial port will be claimed for HAL diagnostic output.

UART Serial Driver

The platform HAL uses the Kinetis's internal UART serial support. The HAL diagnostic interface, used for both polled diagnostic output and GDB stub communication, can be directed to either UART3 (default) or UART5 as required.

As well as the polled HAL diagnostic interface, there is also a CYGPKG_IO_SERIAL_FREESCALE_UART package which contains all the code necessary to support interrupt-driven operation with greater functionality. All six UARTs can be supported by this driver. Note that it is not recommended to use this driver with a port at the same time as using that port for HAL diagnostic I/O.

This driver is not active until the CYGPKG_IO_SERIAL_DEVICES configuration option (within the generic serial driver support package CYGPKG_IO_SERIAL) is enabled in the configuration. By default this will only enable support in the driver for the UART2 port (the same as the HAL diagnostic interface), but the default configuration can be modified to enable support for other serial ports.

Ethernet Driver

The TWR-SER daughterboard is fitted with an Ethernet port connected via a KSZ8041NL PHY to the K60 on-chip Ethernet MAC. This is supported in eCosPro with a driver contained in the package CYGPKG_DEVS_ETH_FREESCALE_ENET.

The driver will be inactive (not built and greyed out in the eCos Configuration Tool) unless the “Common Ethernet support” (CYGPKG_IO_ETH_DRIVERS) package is included in your configuration. As the Kinetis ethernet driver is likely to be used in deeply embedded, low footprint, applications it is most appropriate to choose the lwip_eth template as a starting point when choosing an eCos configuration, which will cause the necessary packages to be automatically included.

SPI Driver

A Kinetis SPI bus driver is available in the package “Freescale DSPI driver” (CYGPKG_DEVS_SPI_FREESCALE_DSPI).

No SPI devices are instantiated by default. Consult the generic SPI driver API documentation in the eCosPro Reference Manual for further details on SPI support in eCosPro, along with the configuration options in the Freescale DSPI device driver.

I²C Driver

A Kinetis I²C hardware driver is available in the package “Freescale I2C driver” (CYGPKG_DEVS_I2C_FREESCALE_I2C).

Flash Driver

The Kinetis's on-chip flash may be programmed and managed using the flash driver located in the “Kinetis FLASH memory support” (CYGPKG_DEVS_FLASH_KINETIS) package. This driver is enabled automatically if the generic “Flash device drivers” (CYGPKG_IO_FLASH) package is included in the eCos configuration.

The driver will configure itself automatically for the size and parameters of the specific Kinetis variant present on the TWR-K60N512/TWR-K60D100M board.

Name

JTAG/SWD support — Usage

Use of JTAG/SWD for debugging

JTAG/SWD can be used to single-step and debug loaded applications, or even applications resident in ROM, including the GDB stub ROM.

Debugging of ROM applications is only possible if using hardware breakpoints. The Cortex-M4 core of the K60-series only supports six such hardware breakpoints, so they may need to be used sparingly. If using a GDB front-end such as Eclipse, check it has not set unnecessary extra breakpoints such as at `main()`. Some JTAG/SWD devices give the option of whether to set hardware or software breakpoints by default. Be sure to configure your device appropriately.

When debugging via JTAG you are likely to need to disable the default HAL idle thread action, otherwise there may be issues where the target fails to halt and the debugging session is unreliable. More details can be found in the Cortex-M architectural HAL. This should *not* be necessary when using a SWD-based hardware debugger.

The default eCos configuration does not enable the use of ITM stimulus ports for the output of HAL diagnostics or Kernel instrumentation. If ITM output is required the architecture HAL package `CYGPKG_HAL_CORTEXM` provides options to enable such use. The host hardware debugger configuration may also need to be updated to provide the necessary ITM enable and capture support.

For HAL diagnostic (e.g. `diag_printf()`) output the architecture CDL option `CYGHWR_HAL_CORTEXM_DIAGNOSTICS_INTERFACE` should be updated to select `ITM` as the output destination. Once the `ITM` option has been configured the option `CYGHWR_HAL_CORTEXM_DIAGNOSTICS_ITM_PORT` allows the actual stimulus port used for the diagnostics to be selected.

When the Kernel instrumentation option `CYGPKG_KERNEL_INSTRUMENT` is enabled then the `CYGHWR_HAL_CORTEXM_ITM_INSTRUMENTATION` option can be enabled to direct instrumentation record output via an ITM stimulus port, rather than into a local memory buffer. The stimulus port used can be configured via the `CYGHWR_HAL_CORTEXM_ITM_INSTRUMENTATION_PORT_BASE` option.

Normally a notable disadvantage with JTAG/SWD debugging is that it does not allow thread-aware debugging, such as the ability to inspect different eCos threads or their stack backtraces, set thread-specific breakpoints, and so on. Fortunately the Ronetix PEEDI JTAG unit does support thread-aware debugging of eCos applications, however extra configuration steps are required. Similarly OpenOCD has support for interpreting eCos thread information. Consult the PEEDI or OpenOCD documentation for more details as usage is beyond the scope of this document.

OpenOCD notes

The OpenOCD debugger can be configured to support the on-board OSJTAG interface available via the USB J13 connection.

An example OpenOCD configuration file `openocd_twr_k60n512.cfg` is provided within the eCos platform HAL package in the source repository. This will be in the directory `packages/hal/cortexm/kinetis/twr_k60n512/`*`VERSION`*`/misc` relative to the root of your eCos installation.

This configuration file can be used with OpenOCD on the host as follows:

```
$ openocd -f openocd_twr_k60n512.cfg
Open On-Chip Debugger 0.8.0-dev-hg8c51ca8dbc00-dirty (2013-09-09-17:18)
Licensed under GNU GPL v2
For bug reports, read
    http://openocd.sourceforge.net/doc/doxygen/bugs.html
Info : only one transport option; autoselect 'jtag'
srst_only separate srst_gates_jtag srst_open_drain connect_deassert_srst
cortex_m3 reset_config sysresetreq
Info : add flash_bank kinetis k60.pflash
Info : OSBDM has opened
Info : This adapter doesn't support configurable speed
Info : JTAG tap: k60.cpu tap/device found: 0x4ba00477 (mfg: 0x23b, part: 0xba00, ver: 0x4)
START...
Info : k60.cpu: hardware has 6 breakpoints, 4 watchpoints
END...
Info : accepting 'gdb' connection from 3333
Halting CPU...
Info : JTAG tap: k60.cpu tap/device found: 0x4ba00477 (mfg: 0x23b, part: 0xba00, ver: 0x4)
START...
END...
```

By default **openocd** provides a telnet console on port `4444`, and this can be used to interact with the target system. This console interface can be used to perform debugging, program the flash, etc.

To debug the target from the command line, **arm-eabi-gdb** should be connected to OpenOCD using it's default GDB server port of `3333`. For example:

```
(gdb) target remote localhost:3333
Remote debugging using localhost:3333
0x1fff0006 in ?? ()
(gdb) monitor reset halt
JTAG tap: k60.cpu tap/device found: 0x4ba00477 (mfg: 0x23b, part: 0xba00, ver: 0x4)
target state: halted
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x00004e9c msp: 0x2000f800
(gdb)
```

The application can then be loaded and executed under GDB as normal. If you are using Eclipse then you may need to define a “preload” gdb macro to emit any necessary commands to OpenOCD prior to loading or executing. See the “Hardware Assisted Debugging” section of the “Eclipse/CDT for eCos application development” document's “Debugging eCos applications” chapter.

If you need to build your own copy of OpenOCD then when configuring the **openocd** tool build, the **configure** script can be given the option `--enable-osbdm` to provide for OSJTAG support.

Ronetix PEEDI notes

On the Ronetix PEEDI, the `peedi_twr_k60n512.cfg` file supplied in the platform HAL package should be used to setup and configure the hardware to an appropriate state to load programs.

The `peedi_twr_k60n512.cfg` file also contains an option to define whether hardware or software breakpoints are used by default, using the `CORE0_BREAKMODE` directive in the `[TARGET]` section. The supplied version of the file defaults to hardware breakpoints. If the breakpoint type is changed remember to use the **reboot** command on the PEEDI command line interface, or press the reset button to make the changes take effect.

The PEEDI provides a telnet console on the standard telnet port (`23`) for configuration and control, and gdb remote debugging access via port `9000`. For example, to use the command line **arm-eabi-gdb** to connect to the target via the PEEDI, you would issue the following gdb command:

```
(gdb) target remote 111.222.333.444:9000
```

By default when the PEEDI is powered up, the target will always run the initialization section of the `peedi_twr_k60n512.cfg` file, and halts the target. This behaviour is repeated with the **reset** command.

If the board is reset with the '**reset**' command, and then the '**go**' command is given, the board will boot from ROM as normal. A similar effect can be achieved in GDB by connecting with **target remote** and immediately typing **continue** or **c**.

It is also possible for the target to always run, without initialization, after reset. This mode is selected with the `CORE0_STARTUP_MODE` directive in the `[TARGET]` section of the `peedi_twr_k60n512.cfg` file. This conveniently allows the target to be connected to the JTAG debugger, and be able to reset and run the resident flash program without being required to always type '**go**' every time. Finally, it is also possible to set a temporary default (unless the PEEDI is reset) by giving an argument to the **reset** command, for example **reset run**. Use the command **help reset** at the PEEDI command prompt for more options.

Suitably configured applications can be loaded either via GDB, or directly via the telnet CLI into RAM for execution. For example:

```
twr_k60n512> memory load tftp://192.168.7.9/test.bin bin 0x20000000
++ info: Loading image file:  tftp://192.168.7.9/test.bin
++ info: At absolute address: 0x20000000
loading at 0x20000000
loading at 0x20004000

Successfully loaded 28KB (29064 bytes) in 0.1s
twr_k60n512> go 0x20000000
```

Consult the PEEDI documentation for information on other formats and loading mechanisms.

For Eclipse users wishing to debug ROM startup programs resident in flash, it is worth highlighting that it is possible to use the eCosCentric Eclipse plugin to automatically reprogram flash as the load sequence. To do so, you will need to install and use a TFTP server so that your application can be accessed from the PEEDI from there. You may then use a GDB command file, as described in more detail in the “Eclipse/CDT for eCos application development” manual. This file can then contain contents similar to the following example:

```
define doload
  shell arm-eabi-objcopy -O binary /path/to/eclipse/workspace/projectname/Debug/myapp /path/to/tftp/server/area/myapp.bin
  monitor flash program tftp://10.1.1.1/myapp.bin bin 0x08000000 erase
  set $pc=0x08000000
end
```

Obviously you will need to adjust the paths and names for your system and TFTP server requirements.

Configuration of JTAG/SWD applications

In order to configure the application to be downloaded and debugged via a hardware debugger, it is recommended to use one of the JTAG startup types (JTAG or ROM), which will implicitly cause two important settings to change. Firstly, `CYGSEM_HAL_USE_ROM_MONITOR` will be disabled. Secondly the `CYGDBG_HAL_DIAG_TO_DEBUG_CHAN` option will be enabled in order to prevent HAL diagnostic output being encoded into GDB ($O) packets. These configuration changes could also be made by hand, but use of one of the JTAG startup types will just work.

With these changes, any diagnostic output will appear out of the configured diagnostic channel. By default the K60's JTAG startup types output diagnostics via the USB CDC/ACM serial port on the J13 USB connector. Alternatively diagnostic output can be configured to appear in GDB instead. For this to work, you must enable the configuration option `CYGSEM_HAL_DIAG_TO_GDBFILEIO_CHAN` in the common HAL package. Then, after you load your application but before running it, you must give GDB the command:

```
(gdb) set hwdebug on
```

Eclipse users can do this by creating a GDB command file with the contents:

```
define postload
  set hwdebug on
end
```

This will be referenced from their Eclipse debug launch configuration. Using GDB command files is described in more detail in the "Eclipse/CDT for eCos application development" manual.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the Freescale TWR-K60N512 and TWR-K60D100M hardware, and should be read in conjunction with that specification. The `CYGPKG_HAL_CORTEXM_KINETIS_TWR_K60N512` platform HAL package complements the Cortex-M architectural HAL and the Kinetis variant HAL. It provides functionality which is specific to the target boards.

Startup

Following a hard or soft reset the HAL will initialize or reinitialize many of the on-chip peripherals. There is an exception for RAM startup applications which depend on a ROM monitor for certain services, and for the platform JTAG startup where the host debugger configuration is responsible for initialising the DDRMC world for the external SDRAM used to hold the application being loaded.

For ROM startup, the HAL will perform additional initialization, programming the various internal registers including the PLL, peripheral clocks and GPIO pins. The details of the early hardware startup may be found in the `src/twr_k60n512_misc.c` in both `hal_system_init` and `hal_platform_init`.

Memory Map

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

Internal RAM
: This is located at address 0x1FFF0000 of the memory space, and is 128KiB in size. The eCos VSR table occupies the bottom 512-bytes. The virtual vector table starts at 0x1FFF0200 and extends to 0x20000300. For ROM, and JTAG startups, the top `CYGNUM_HAL_COMMON_INTERRUPTS_STACK_SIZE` bytes are reserved for the interrupt stack. The remainder of internal RAM is available for use by applications.

Internal FLASH
: This is located at address 0x00000000 of the memory space. This region is 1MiB in size. ROM applications are by default configured to run from this memory.

On-Chip Peripherals
: These are accessible from locations 0x40000000 and 0xE0000000 upwards. Descriptions of the contents can be found in the Kinetis User Manual.

Linker Scripts

The platform linker scripts define the following symbols:

hal_vsr_table
: This defines the location of the VSR table. This is set to 0x1FFF0000 for all startup types, and space for 128 entries allocated, though the K60 sub-family in use may use less entries.

hal_virtual_vector_table
: This defines the location of the virtual vector table used to communicate between an ROM monitor and an eCos application. This is allocated right after the VSR table, at 0x20000200.

hal_interrupt_stack
: This defines the location of the interrupt stack. For all startup types this is allocated to the top of internal SRAM, 0x20010000.

hal_startup_stack
: This defines the location of the startup stack. For all startup types it is initially allocated at the half-way point of the interrupt stack.

Diagnostic LEDs

Four LEDs are fitted on the board for diagnostic purposes:

Platform HAL manifest	Colour	Board Label
`CYGHWR_IO_KINETIS_PIN_LED1`	Orange	`E1`
`CYGHWR_IO_KINETIS_PIN_LED2`	Yellow	`E2`
`CYGHWR_IO_KINETIS_PIN_LED3`	Green	`E3`
`CYGHWR_IO_KINETIS_PIN_LED4`	Blue	`E4`

The platform HAL header file at `<cyg/hal/plf_io.h>` defines the following convenience function to allow the LEDs to be set:

```
extern void hal_twr_k60n512_led(char c);
```

The lowest 4-bits of the argument `c` correspond to each of the 4 LEDs (with LED1 as the least significant bit).

The platform HAL will automatically light all of the LEDs when the platform initialisation is complete, however the LEDs are free for application use.

Real-time Characterization

The tm_basic kernel test gives statistics gathered about the real-time characterization and performance of the kernel. The sample output is shown here for information. The test was built for the variant ROM startup with optimization flag -O3.

Example 287.1. twr_k60n512 Real-time characterization

```
INFO:<code from 0x00000410 -> 0x000181dc, CRC 0356>
             Startup, main thrd : stack used   112 size  1536
             Startup : Idlethread stack used    84 size  1280

eCos Kernel Timings
Notes: all times are in microseconds (.000001) unless otherwise stated

Reading the hardware clock takes 0 'ticks' overhead
... this value will be factored out of all other measurements
Clock interrupt took    8.00 microseconds (8 raw clock ticks)

Testing parameters:
   Clock samples:            32
   Threads:                   6
   Thread switches:         128
   Mutexes:                  32
   Mailboxes:                32
   Semaphores:               32
   Scheduler operations:    128
   Counters:                 32
   Flags:                    32
   Alarms:                   32
   Stack Size:             1088


                                 Confidence
     Ave     Min     Max     Var  Ave  Min  Function
  ====== ====== ====== ====== ========== ========
    7.17    7.00    8.00    0.28   83%  83% Create thread
    1.33    1.00    2.00    0.44   66%  66% Yield thread [all suspended]
    1.83    1.00    2.00    0.28   83%  16% Suspend [suspended] thread
    2.00    2.00    2.00    0.00  100% 100% Resume thread
    3.33    3.00    4.00    0.44   66%  66% Set priority
    0.33    0.00    1.00    0.44   66%  66% Get priority
    5.50    5.00    6.00    0.50  100%  50% Kill [suspended] thread
    1.33    1.00    2.00    0.44   66%  66% Yield [no other] thread
    3.00    3.00    3.00    0.00  100% 100% Resume [suspended low prio] thread
    2.00    2.00    2.00    0.00  100% 100% Resume [runnable low prio] thread
    2.67    2.00    3.00    0.44   66%  33% Suspend [runnable] thread
    2.00    2.00    2.00    0.00  100% 100% Yield [only low prio] thread
    1.50    1.00    2.00    0.50  100%  50% Suspend [runnable->not runnable]
    5.33    5.00    6.00    0.44   66%  66% Kill [runnable] thread
    4.33    4.00    5.00    0.44   66%  66% Destroy [dead] thread
    8.33    8.00    9.00    0.44   66%  66% Destroy [runnable] thread
   10.83   10.00   12.00    0.56   50%  33% Resume [high priority] thread
    3.54    3.00    5.00    0.51   52%  46% Thread switch

    0.47    0.00    1.00    0.50   53%  53% Scheduler lock
    1.30    1.00    2.00    0.42   70%  70% Scheduler unlock [0 threads]
    1.34    1.00    2.00    0.45   66%  66% Scheduler unlock [1 suspended]
    1.28    1.00    2.00    0.40   71%  71% Scheduler unlock [many suspended]
    1.30    1.00    2.00    0.42   70%  70% Scheduler unlock [many low prio]

    0.50    0.00    1.00    0.50  100%  50% Init mutex
    2.34    2.00    3.00    0.45   65%  65% Lock [unlocked] mutex
    2.44    2.00    3.00    0.49   56%  56% Unlock [locked] mutex
    1.88    1.00    2.00    0.22   87%  12% Trylock [unlocked] mutex
    1.69    1.00    2.00    0.43   68%  31% Trylock [locked] mutex
    0.44    0.00    1.00    0.49   56%  56% Destroy mutex
   12.00   12.00   12.00    0.00  100% 100% Unlock/Lock mutex

    0.56    0.00    1.00    0.49   56%  43% Create mbox
    0.28    0.00    1.00    0.40   71%  71% Peek [empty] mbox
    2.53    2.00    3.00    0.50   53%  46% Put [first] mbox
    0.28    0.00    1.00    0.40   71%  71% Peek [1 msg] mbox
    2.69    2.00    3.00    0.43   68%  31% Put [second] mbox
    0.22    0.00    1.00    0.34   78%  78% Peek [2 msgs] mbox
    2.31    2.00    3.00    0.43   68%  68% Get [first] mbox
    2.28    2.00    3.00    0.40   71%  71% Get [second] mbox
    1.88    1.00    2.00    0.22   87%  12% Tryput [first] mbox
    1.72    1.00    2.00    0.40   71%  28% Peek item [non-empty] mbox
    1.69    1.00    2.00    0.43   68%  31% Tryget [non-empty] mbox
    1.53    1.00    2.00    0.50   53%  46% Peek item [empty] mbox
    1.94    1.00    2.00    0.12   93%   6% Tryget [empty] mbox
    0.31    0.00    1.00    0.43   68%  68% Waiting to get mbox
    0.31    0.00    1.00    0.43   68%  68% Waiting to put mbox
    0.44    0.00    1.00    0.49   56%  56% Delete mbox
    8.00    8.00    8.00    0.00  100% 100% Put/Get mbox

    0.28    0.00    1.00    0.40   71%  71% Init semaphore
    1.63    1.00    2.00    0.47   62%  37% Post [0] semaphore
    2.00    2.00    2.00    0.00  100% 100% Wait [1] semaphore
    1.69    1.00    2.00    0.43   68%  31% Trywait [0] semaphore
    2.00    2.00    2.00    0.00  100% 100% Trywait [1] semaphore
    0.44    0.00    1.00    0.49   56%  56% Peek semaphore
    0.38    0.00    1.00    0.47   62%  62% Destroy semaphore
    7.00    7.00    7.00    0.00  100% 100% Post/Wait semaphore

    0.72    0.00    1.00    0.40   71%  28% Create counter
    0.50    0.00    1.00    0.50  100%  50% Get counter value
    0.31    0.00    1.00    0.43   68%  68% Set counter value
    2.50    2.00    3.00    0.50  100%  50% Tick counter
    0.34    0.00    1.00    0.45   65%  65% Delete counter

    0.31    0.00    1.00    0.43   68%  68% Init flag
    2.00    2.00    2.00    0.00  100% 100% Destroy flag
    1.25    1.00    2.00    0.38   75%  75% Mask bits in flag
    2.00    2.00    2.00    0.00  100% 100% Set bits in flag [no waiters]
    3.00    3.00    3.00    0.00  100% 100% Wait for flag [AND]
    2.72    2.00    3.00    0.40   71%  28% Wait for flag [OR]
    2.88    2.00    3.00    0.22   87%  12% Wait for flag [AND/CLR]
    2.47    2.00    3.00    0.50   53%  53% Wait for flag [OR/CLR]
    0.38    0.00    1.00    0.47   62%  62% Peek on flag

    0.94    0.00    1.00    0.12   93%   6% Create alarm
    2.91    2.00    3.00    0.17   90%   9% Initialize alarm
    1.72    1.00    2.00    0.40   71%  28% Disable alarm
    2.78    2.00    3.00    0.34   78%  21% Enable alarm
    1.91    1.00    2.00    0.17   90%   9% Delete alarm
    2.66    2.00    3.00    0.45   65%  34% Tick counter [1 alarm]
    9.50    9.00   10.00    0.50  100%  50% Tick counter [many alarms]
    4.22    4.00    5.00    0.34   78%  78% Tick & fire counter [1 alarm]
   39.09   39.00   40.00    0.17   90%  90% Tick & fire counters [>1 together]
   11.00   11.00   11.00    0.00  100% 100% Tick & fire counters [>1 separately]
    7.00    7.00    7.00    0.00  100% 100% Alarm latency [0 threads]
    5.83    5.00    7.00    0.31   79%  18% Alarm latency [2 threads]
    6.01    5.00    7.00    0.64   36%  31% Alarm latency [many threads]
   12.02   12.00   14.00    0.03   99%  99% Alarm -> thread resume latency

     236     236     236                    Worker thread stack used (stack size 1088)
            All done, main thrd : stack used   760 size  1536
            All done : Idlethread stack used   188 size  1280

Timing complete - 27870 ms total

PASS:<Basic timing OK>
EXIT:<done>
```

Chapter 288. Freescale TWR-K70F120M Platform HAL

Name

CYGPKG_HAL_CORTEXM_KINETIS_TWR_K70F120M — eCos Support for the Freescale TWR-K70F120M board

Description

The Freescale TWR-K70F120M board has a MK70FN1M0VMJ12 microcontroller which incorporates 1MB of internal flash ROM and 128KB of internal SRAM. The stand-alone board may be targetted, but for access to some peripherals it is assumed that a TWR-ELEV setup, with a TWR-SER daughterboard is being used. For example, with a TWR-SER board present it provides a connector for Ethernet. The TWR-K70F120M motherboard has limited I/O interfaces, with most of the I/O signals being propogated via multi-pin connectors.

The TWR-K70F120M motherboard also provides an on-board JTAG debug circuit (OSJTAG) with virtual serial port, 128MB of DDR2 SDRAM, 256MB of SLC NAND, a MMA8451Q 3-axis accelerometer, a potentiometer, a MicroSD card slot, and some LEDs and buttons.

The TWR-SER daughterboard also provides in addition to the Ethernet connector access to a RS232/484 DB9 connection, a USB Mini-AB connector and a 3-pin CAN connector.

On this board, the expected eCos development model is that programs may be downloaded and debugged via the on-board OSJTAG USB interface, or via a hardware debugger (JTAG/SWD) attached to the JTAG socket. While it is possible to build and install RedBoot or a GDB stub image into the internal FLASH, this is not currently supported.

This documentation describes platform-specific elements of the TWR-K70F120M board support within eCos. The Kinetis variant HAL documentation covers various topics including HAL support common to Kinetis variants, and on-chip device support. This document complements the Kinetis documentation.

Supported Hardware

The MK70FN1M0VMJ12 has two on-chip memory regions. There is a SRAM region of 128KiB present at 0x1FFF0000 and a 1MiB FLASH region present at 0x00000000. The TWR-K70F120M motherboard has 128MiB of SDRAM mapped to 0x08000000.

The Kinetis variant HAL includes support for the six on-chip serial devices which are documented in the variant HAL. For the TWR-K70F120M UART2 is connected to the J8 DB9 connector and also as a virtual serial through the on-board OSJTAG J13 USB connection. There is no connection for hardware flow control (RTS/CTS) lines on this UART3 connection.

Device drivers are provided for the Kinetis on-chip Ethernet MAC, I²C interface and SPI interface. Additionally support is provided for the on-chip watchdog, RTC (wallclock) and a flash driver exists to permit management of the Kinetis's on-chip flash.

Also, whilst the board is fitted with a SLC NAND flash, this is not presently supported by the HAL port. The Kinetis MK70FN1M0VMJ12 processor, and the TWR-K70F120M platform, provide a wide variety of peripherals, but unless support is specifically indicated it should be assumed that support is not included.

Tools

The board port is intended to work with GNU tools configured for an **arm-eabi** target. The original port was done using **arm-eabi-gcc** version 4.7.3a, **arm-eabi-gdb** version 7.6, and **binutils** version 2.23.2.

Name

Setup — Preparing the TWR-K70F120M Board for eCos Development

Overview

Since the target motherboard provides a built-in hardware debug solution, it is expected that the most common development method when targeting the CPU is to use this hardware debug interface for development. This will either be by loading smaller applications into on-chip SRAM, or by programming larger applications directly into on-chip flash. In the first case, eCos applications should be configured for the variant JTAG startup type, and in the second case for the variant ROM startup type. Since the `TWR-K70F120M` motherboard provides 128MB of external SDRAM it is also possible to configure larger applications for hardware debug using that external memory using the platform JTAG startup type.

Note

When using the platform JTAG startup type to load application programs into the external SDRAM the host hardware debugger configuration is responsible for initialising the necessary CPU I/O to allow GDB to load the application into SDRAM.

HAL Startup Types

For the `twr_k70f120m` platform the Kinetis variant HAL support provides some common startup types, which are extended by the platform HAL. The following variant HAL provided startup types may be selected for applications:

Configuration	Description
ROM	Stand-alone programs running from internal FLASH
SRAM	Programs loading via hardware debugger into **on-chip SRAM**, but expecting RedBoot or GDB stub ROM
RAM	Currently not supported: Programs loading via RedBoot or GDB stub ROM into **on-chip SRAM**
JTAG	Stand-alone programs running from **on-chip SRAM**, loaded via hardware debugger

The following `TWR-K70F120M` platform specific startup types may also be selected for applications:

Configuration	Description
ByVariant	The variant defines the startup configuration
ROM	Stand-alone programs running from internal FLASH
RAM	Currently not supported: Programs loading via RedBoot or GDB stub ROM into **off-chip SDRAM**
JTAG	Stand-alone programs running from **off-chip SDRAM**, loaded via hardware debugger

Further details are available later in this manual.

Preparing OSJTAG interface

The support for using the on-chip OSJTAG interface for hardware debugging and diagnostic output requires that the OSJTAG firmware is at least version `v30.21`. The firmware for the OSJTAG interface can be checked, and updated if needed, using the relevant firmware updater tool available for download via the Freescale website. Unfortunately the official firmware updater is only available for the Windows platform at the moment.

Programming ROM images

To program ROM startup applications into flash a hardware debugger that understands the Kinetis flash may be used. For example, the Ronetix PEEDI provides suitable support.

Warning

Due to a security feature of the Kinetis CPUs care should be taken to avoid completely erasing the flash to ensure the required `FSEC` value in the flash configuration field is not lost. For example when using the PEEDI the user should NOT use the “erase” suffix to the **flash program** command. The Kinetis aware **flash erase chip** should be executed to erase the flash prior to using the **flash program** command.

Programming ROM images with a Ronetix PEEDI

This section describes how to program ROM images using a Ronetix PEEDI debugger.

The PEEDI must be configured to allow communication with your local network, and configured with the parameters for interfacing with the target board. It must then be used to download and program the ROM image into the internal flash. The following steps give a typical outline for doing this. Consult the PEEDI documentation for alternative approaches, such as using FTP or HTTP instead of TFTP.

Preparing the Ronetix PEEDI JTAG debugger

1. Prepare a PC to act as a host and start a TFTP server on it.
2. Connect the PEEDI JTAG debugger via both serial and ethernet to the host PC and power it on. Use the serial cable supplied with the PEEDI (straight through, not null modem).
3. Verify the PEEDI is using up-to-date firmware, of version 12.3.0 or later. Older PEEDI firmware does not support the Kinetis family correctly, particularly if wishing to use the PEEDI's own '**flash**' commands to modify the on-chip flash. If the firmware is not recent enough, follow the PEEDI User Manual's instructions which describe how to update the PEEDI firmware.
4. Locate the PEEDI configuration file `peedi_twr_k70f120m.cfg` within the eCos platform HAL package in the source repository. This will be in the directory `packages/hal/cortexm/kinetis/twr_k70f120m/`*`VERSION`*`/misc` relative to the root of your eCos installation.
5. Place the PEEDI configuration file in a location on the PC accessible to the TFTP server. Later you will configure the PEEDI to load this file via TFTP as its configuration file.
6. Open `peedi_twr_k70f120m.cfg` in an editor such as emacs or notepad and insert your own license information in the `[LICENSE]` section.
7. Install and configure the PEEDI in line with the PEEDI Quick Start Guide or User's Manual, especially configuring PEEDI's RedBoot with the network information. Configure it to use the `peedi_twr_k70f120m.cfg` target configuration file on the TFTP server at the appropriate point of the **config** process, for example with a path such as: `tftp://192.168.7.9/peedi_twr_k70f120m.cfg`
8. Reset the PEEDI.
9. Connect to the PEEDI's CLI interface via TCP/IP on the standard telnet port 23. The **telnet** application is suitable for this. You should see output similar to the following:

```
$ telnet 192.168.7.225
Trying 192.168.7.225...
Connected to 192.168.7.225.
Escape character is '^]'.

PEEDI - Powerful Embedded Ethernet Debug Interface
Copyright (c) 2005-2011 www.ronetix.at - All rights reserved
Hw:1.2, L:JTAG v1.6 Fw:13.3.0, SN: PD-XXXX-XXXX-XXXX
------------------------------------------------------------

twr_k70f120m>
```

Preparing the TWR-K70F120M board for programming with PEEDI

Follow the steps in this section in order to allow communication between the board and the host PC, and between the board and the PEEDI device.

If programming an application which uses serial output, you should first:

1. If OSJTAG firmware 30.21 or later is installed then the motherboard J13 USB provides a standard CDC-ACM virtual serial connection to the host computer. The alternative is to connect a null modem cable between the DB9 RS232 connector on the TWR-SER daughterboard and the host computer.

2. Start a suitable terminal emulator on the host computer such as **minicom** on Linux or PuTTY on Windows. Set the communication parameters to 38400 baud, 8 data bits, no parity bit and 1 stop bit with no flow control.

For all applications, you must:

1. Connect the board to the PEEDI using an appropriate 20-pin cable from the JTAG interface connector to the Target port on the PEEDI. This will normally be a PEEDI-CORTEX20 adapter.

2. Power up the `TWR-K70F120M` board.

3. Connect to the PEEDI's telnet CLI on port 23 as before.

4. Confirm correct connection with the PEEDI with the **reset reset** command as follows:

```
twr_k70f120m> reset reset
++ info: user reset
twr_k70f120m>
++ info: RESET and TRST asserted
++ info: TRST released
++ info: TAP : IDCODE = 0x2BA01477, Cortex M3 SWD
++ info: RESET released
++ info: core connected

CORE0 -> CortexM4 - stopped by breakpoint
         PC=0x1FFF0006, xPSR=0x01000000

core #0 stopped
++ info: core 0: initialized

twr_k70f120m>
```

Installation into flash

The following describes the procedure for installing a ROM application into on-chip flash, using the GDB stub ROM image as an example of such an application.

1. Use **arm-eabi-objcopy** to convert the linked application, in ELF format, into binary format. For example:

```
$ arm-eabi-objcopy -O binary programname programname.bin
```

2. Copy the binary file (.bin file) into a location on the host computer accessible to its TFTP server.

3. Connect to the PEEDI's telnet interface, and program the image into flash with the following commands, replacing *`TFTP_SERVER`* with the address of the TFTP server and *`/BINPATH`* with the location of the .bin file relative to the TFTP server root directory. For example for a RedBoot ROM image:

```
twr_k70f120m> flash erase chip

erasing chip at 0x00000000
done.
```

```
++ info: successfully erased 1 byte in 0.33 sec

twr_k70f120m> flash program tftp://TFTP_SERVER/BINPATH/redboot.bin bin 0x00000000
++ info: Programming image file: tftp://TFTP_SERVER/redboot.bin
++ info: Programming directly
++ info: At absolute address:    0x00000000
programming at 0x00000000
programming at 0x00001000
programming at 0x00002000
programming at 0x00003000
programming at 0x00004000
programming at 0x00005000
programming at 0x00006000
programming at 0x00007000
programming at 0x00008000
programming at 0x00009000
programming at 0x0000A000
programming at 0x0000B000
programming at 0x0000C000
programming at 0x0000D000
programming at 0x0000E000
programming at 0x0000F000
programming at 0x00010000
programming at 0x00011000
programming at 0x00012000
programming at 0x00013000
programming at 0x00014000
programming at 0x00015000

++ info: successfully programmed 88.00 KB in 0.74 sec

twr_k70f120m>
```

The installation into flash is now complete. For applications which print output on startup to the USART3 RS232 serial port, such as the GDB stub ROM application, this can easily by tested by powering off the board, disconnecting the JTAG, and then powering on the board again. In the case of the GDB stub ROM image, output similar to the following should be visible (although specific numbers may differ):

```
$T050f:72250008;0d:f0ff0120;#8a
```

RedBoot

RedBoot is currently unsupported.

Name

Configuration — Platform-specific Configuration Options

Overview

The Freescale TWR-K70F120M platform HAL package is loaded automatically when eCos is configured for the `twr_k70f120m` target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The TWR-K70F120M board platform HAL package supports separate variant and platform startup types:

ROM

> This startup type can be used for finished applications which will be programmed into internal flash ROM at location 0x00000000. Data and BSS will be put into internal SRAM starting from 0x1FFF0000. The application will be self-contained with no dependencies on services provided by other software. The program expects to boot from reset with ROM mapped at location zero. eCos startup code will perform all necessary hardware initialization.

RAM

> Currently unsupported, this is the startup type which is used if relying on either a RedBoot or a GDB stub ROM image programmed into internal flash to download and run applications into RAM via arm-eabi-gdb. It is assumed that the hardware has already been initialized by the ROM monitor. By default the application will use the eCos virtual vectors mechanism to obtain services from the ROM monitor, including diagnostic output.

JTAG

> This is the startup type used to build applications that are loaded via a hardware debug interface. The application will be self-contained with no dependencies on services provided by other software. For the variant JTAG startup type the eCos run-time will perform all necessary hardware initialization. When using the platform JTAG startup type the hardware debugger configuration is responsible for initialising the clocks and DDRMC necessary to allow access to the off-chip SDRAM where applications will be loaded.

Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building the GDB stub ROM (or RedBoot).

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then the serial port will be claimed for HAL diagnostic output.

UART Serial Driver

The TWR-K70F120M board uses the Kinetis's internal UART serial support. The HAL diagnostic interface, used for both polled diagnostic output and GDB stub communication, is only expected to be available to be used on the UART2 port.

As well as the polled HAL diagnostic interface, there is also a CYGPKG_IO_SERIAL_FREESCALE_UART package which contains all the code necessary to support interrupt-driven operation with greater functionality. All six UARTs can be supported by this driver. Note that it is not recommended to use this driver with a port at the same time as using that port for HAL diagnostic I/O.

This driver is not active until the CYGPKG_IO_SERIAL_DEVICES configuration option (within the generic serial driver support package CYGPKG_IO_SERIAL) is enabled in the configuration. By default this will only enable support in the driver for the UART2 port (the same as the HAL diagnostic interface), but the default configuration can be modified to enable support for other serial ports.

Ethernet Driver

The TWR-SER daughterboard is fitted with an Ethernet port connected via a KSZ8041NL PHY to the TWR-K70F120M on-chip Ethernet MAC. This is supported in eCosPro with a driver contained in the package CYGPKG_DEVS_ETH_FREESCALE_ENET.

The driver will be inactive (not built and greyed out in the eCos Configuration Tool) unless the “Common Ethernet support” (CYGPKG_IO_ETH_DRIVERS) package is included in your configuration. As the Kinetis ethernet driver is likely to be used in deeply embedded, low footprint, applications it is most appropriate to choose the lwip_eth template as a starting point when choosing an eCos configuration, which will cause the necessary packages to be automatically included.

SPI Driver

A Kinetis SPI bus driver is available in the package “Freescale DSPI driver” (CYGPKG_DEVS_SPI_FREESCALE_DSPI).

No SPI devices are instantiated by default. Consult the generic SPI driver API documentation in the eCosPro Reference Manual for further details on SPI support in eCosPro, along with the configuration options in the Freescale DSPI device driver.

I²C Driver

A Kinetis I²C hardware driver is available in the package “Freescale I2C driver” (CYGPKG_DEVS_I2C_FREESCALE_I2C).

Flash Driver

The Kinetis's on-chip flash may be programmed and managed using the flash driver located in the “Kinetis FLASH memory support” (CYGPKG_DEVS_FLASH_KINETIS) package. This driver is enabled automatically if the generic “Flash device drivers” (CYGPKG_IO_FLASH) package is included in the eCos configuration.

The driver will configure itself automatically for the size and parameters of the specific Kinetis variant present on the TWR-K70F120M board.

Name

JTAG/SWD support — Usage

Use of JTAG/SWD for debugging

JTAG/SWD can be used to single-step and debug loaded applications, or even applications resident in ROM, including the GDB stub ROM.

Debugging of ROM applications is only possible if using hardware breakpoints. The Cortex-M4 core of the K70-series only supports six such hardware breakpoints, so they may need to be used sparingly. If using a GDB front-end such as Eclipse, check it has not set unnecessary extra breakpoints such as at `main()`. Some JTAG/SWD devices give the option of whether to set hardware or software breakpoints by default. Be sure to configure your device appropriately.

When debugging via JTAG you are likely to need to disable the default HAL idle thread action, otherwise there may be issues where the target fails to halt and the debugging session is unreliable. More details can be found in the Cortex-M architectural HAL. This should **not** be necessary when using a SWD-based hardware debugger.

The default eCos configuration does not enable the use of ITM stimulus ports for the output of HAL diagnostics or Kernel instrumentation. If ITM output is required the architecture HAL package `CYGPKG_HAL_CORTEXM` provides options to enable such use. The host hardware debugger configuration may also need to be updated to provide the necessary ITM enable and capture support.

For HAL diagnostic (e.g. `diag_printf()`) output the architecture CDL option `CYGHWR_HAL_CORTEXM_DIAGNOSTICS_INTERFACE` should be updated to select `ITM` as the output destination. Once the `ITM` option has been configured the option `CYGHWR_HAL_CORTEXM_DIAGNOSTICS_ITM_PORT` allows the actual stimulus port used for the diagnostics to be selected.

When the Kernel instrumentation option `CYGPKG_KERNEL_INSTRUMENT` is enabled then the `CYGHWR_HAL_CORTEXM_ITM_INSTRUMENTATION` option can be enabled to direct instrumentation record output via an ITM stimulus port, rather than into a local memory buffer. The stimulus port used can be configured via the `CYGHWR_HAL_CORTEXM_ITM_INSTRUMENTATION_PORT_BASE` option.

Normally a notable disadvantage with JTAG/SWD debugging is that it does not allow thread-aware debugging, such as the ability to inspect different eCos threads or their stack backtraces, set thread-specific breakpoints, and so on. Fortunately the Ronetix PEEDI JTAG unit does support thread-aware debugging of eCos applications, however extra configuration steps are required. Similarly OpenOCD has support for interpreting eCos thread information. Consult the PEEDI or OpenOCD documentation for more details as usage is beyond the scope of this document.

OpenOCD notes

The OpenOCD debugger can be configured to support the on-board OSJTAG interface available via the USB J13 connection.

An example OpenOCD configuration file `openocd_twr_k70f120m.cfg` is provided within the eCos platform HAL package in the source repository. This will be in the directory `packages/hal/cortexm/kinetis/twr_k70f120m/`*`VERSION`*`/misc` relative to the root of your eCos installation.

This configuration file can be used with OpenOCD on the host as follows:

```
$ openocd -f openocd_twr_k70f120m.cfg
Open On-Chip Debugger 0.8.0-dev-hg8c51ca8dbc00-dirty (2013-09-09-17:18)
Licensed under GNU GPL v2
For bug reports, read
    http://openocd.sourceforge.net/doc/doxygen/bugs.html
Info : only one transport option; autoselect 'jtag'
srst_only separate srst_gates_jtag srst_open_drain connect_deassert_srst
cortex_m3 reset_config sysresetreq
Info : add flash_bank kinetis k70.pflash
Info : OSBDM has opened
Info : This adapter doesn't support configurable speed
Info : JTAG tap: k70.cpu tap/device found: 0x4ba00477 (mfg: 0x23b, part: 0xba00, ver: 0x4)
START...
Info : k70.cpu: hardware has 6 breakpoints, 4 watchpoints
END...
Info : accepting 'gdb' connection from 3333
Halting CPU...
Info : JTAG tap: k70.cpu tap/device found: 0x4ba00477 (mfg: 0x23b, part: 0xba00, ver: 0x4)
START...
END...
```

By default **openocd** provides a telnet console on port `4444`, and this can be used to interact with the target system. This console interface can be used to perform debugging, program the flash, etc.

To debug the target from the command line, **arm-eabi-gdb** should be connected to OpenOCD using it's default GDB server port of `3333`. For example:

```
(gdb) target remote localhost:3333
Remote debugging using localhost:3333
0x1fff0006 in ?? ()
(gdb) monitor reset halt
JTAG tap: k70.cpu tap/device found: 0x4ba00477 (mfg: 0x23b, part: 0xba00, ver: 0x4)
target state: halted
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x00004e9c msp: 0x2000f800
(gdb)
```

The application can then be loaded and executed under GDB as normal. If you are using Eclipse then you may need to define a “preload” gdb macro to emit any necessary commands to OpenOCD prior to loading or executing. See the “Hardware Assisted Debugging” section of the “Eclipse/CDT for eCos application development” document's “Debugging eCos applications” chapter.

If you need to build your own copy of OpenOCD then when configuring the **openocd** tool build, the **configure** script can be given the option `--enable-osbdm` to provide for OSJTAG support.

Ronetix PEEDI notes

On the Ronetix PEEDI, the `peedi_twr_k70f120m.cfg` file supplied in the platform HAL package should be used to setup and configure the hardware to an appropriate state to load programs.

The `peedi_twr_k70f120m.cfg` file also contains an option to define whether hardware or software breakpoints are used by default, using the `CORE0_BREAKMODE` directive in the `[TARGET]` section. The supplied version of the file defaults to hardware breakpoints. If the breakpoint type is changed remember to use the **reboot** command on the PEEDI command line interface, or press the reset button to make the changes take effect.

The PEEDI provides a telnet console on the standard telnet port (`23`) for configuration and control, and gdb remote debugging access via port `9000`. For example, to use the command line **arm-eabi-gdb** to connect to the target via the PEEDI, you would issue the following gdb command:

```
(gdb) target remote 111.222.333.444:9000
```

By default when the PEEDI is powered up, the target will always run the initialization section of the `peedi_twr_k70f120m.cfg` file, and halts the target. This behaviour is repeated with the **reset** command.

If the board is reset with the '**reset**' command, and then the '**go**' command is given, the board will boot from ROM as normal. A similar effect can be achieved in GDB by connecting with **target remote** and immediately typing **continue** or **c**.

It is also possible for the target to always run, without initialization, after reset. This mode is selected with the `CORE0_STARTUP_MODE` directive in the `[TARGET]` section of the `peedi_twr_k70f120m.cfg` file. This conveniently allows the target to be connected to the JTAG debugger, and be able to reset and run the resident flash program without being required to always type '**go**' every time. Finally, it is also possible to set a temporary default (unless the PEEDI is reset) by giving an argument to the **reset** command, for example **reset run**. Use the command **help reset** at the PEEDI command prompt for more options.

Suitably configured applications can be loaded either via GDB, or directly via the telnet CLI into RAM for execution. For example:

```
twr_k70f120m> memory load tftp://192.168.7.9/test.bin bin 0x20000000
++ info: Loading image file:  tftp://192.168.7.9/test.bin
++ info: At absolute address: 0x20000000
loading at 0x20000000
loading at 0x20004000

Successfully loaded 28KB (29064 bytes) in 0.1s
twr_k70f120m> go 0x20000000
```

Consult the PEEDI documentation for information on other formats and loading mechanisms.

For Eclipse users wishing to debug ROM startup programs resident in flash, it is worth highlighting that it is possible to use the eCosCentric Eclipse plugin to automatically reprogram flash as the load sequence. To do so, you will need to install and use a TFTP server so that your application can be accessed from the PEEDI from there. You may then use a GDB command file, as described in more detail in the “Eclipse/CDT for eCos application development” manual. This file can then contain contents similar to the following example:

```
define doload
  shell arm-eabi-objcopy -O binary /path/to/eclipse/workspace/projectname/Debug/myapp /path/to/tftp/server/area/myapp.bin
  monitor flash program tftp://10.1.1.1/myapp.bin bin 0x08000000 erase
  set $pc=0x08000000
end
```

Obviously you will need to adjust the paths and names for your system and TFTP server requirements.

Configuration of JTAG/SWD applications

In order to configure the application to be downloaded and debugged via a hardware debugger, it is recommended to use one of the JTAG startup types (JTAG or ROM), which will implicitly cause two important settings to change. Firstly, `CYGSEM_HAL_USE_ROM_MONITOR` will be disabled. Secondly the `CYGDBG_HAL_DIAG_TO_DEBUG_CHAN` option will be enabled in order to prevent HAL diagnostic output being encoded into GDB ($O) packets. These configuration changes could also be made by hand, but use of one of the JTAG startup types will just work.

With these changes, any diagnostic output will appear out of the configured diagnostic channel. By default the K60's JTAG startup types output diagnostics via the USB CDC/ACM serial port on the J13 USB connector. Alternatively diagnostic output can be configured to appear in GDB instead. For this to work, you must enable the configuration option `CYGSEM_HAL_DIAG_TO_GDBFILEIO_CHAN` in the common HAL package. Then, after you load your application but before running it, you must give GDB the command:

```
(gdb) set hwdebug on
```

Eclipse users can do this by creating a GDB command file with the contents:

```
define postload
  set hwdebug on
end
```

This will be referenced from their Eclipse debug launch configuration. Using GDB command files is described in more detail in the "Eclipse/CDT for eCos application development" manual.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the Freescale TWR-K70F120M hardware, and should be read in conjunction with that specification. The Freescale TWR-K70F120M platform HAL package complements the Cortex-M architectural HAL and the Kinetis variant HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset the HAL will initialize or reinitialize many of the on-chip peripherals. There is an exception for RAM startup applications which depend on a ROM monitor for certain services, and for the platform JTAG startup where the host debugger configuration is responsible for initialising the DDRMC world for the external SDRAM used to hold the application being loaded.

For ROM startup, the HAL will perform additional initialization, programming the various internal registers including the PLL, peripheral clocks and GPIO pins. The details of the early hardware startup may be found in the `src/twr_k70f120m_misc.c` in both `hal_system_init` and `hal_platform_init`.

Memory Map

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

Internal RAM
: This is located at address 0x1FFF0000 of the memory space, and is 128KiB in size. The eCos VSR table occupies the bottom 512-bytes. The virtual vector table starts at 0x1FFF0200 and extends to 0x20000300. For ROM, and JTAG startups, the top `CYGNUM_HAL_COMMON_INTERRUPTS_STACK_SIZE` bytes are reserved for the interrupt stack. The remainder of internal RAM is available for use by applications.

Internal FLASH
: This is located at address 0x00000000 of the memory space. This region is 1MiB in size. ROM applications are by default configured to run from this memory.

On-Chip Peripherals
: These are accessible from locations 0x40000000 and 0xE0000000 upwards. Descriptions of the contents can be found in the Kinetis User Manual.

Linker Scripts

The platform linker scripts define the following symbols:

hal_vsr_table
: This defines the location of the VSR table. This is set to 0x1FFF0000 for all startup types, and space for 128 entries allocated, though the K70 sub-family only use 121 entries.

hal_virtual_vector_table
: This defines the location of the virtual vector table used to communicate between an ROM monitor and an eCos application. This is allocated right after the VSR table, at 0x20000200.

hal_interrupt_stack
: This defines the location of the interrupt stack. For all startup types this is allocated to the top of internal SRAM, 0x20010000.

hal_startup_stack
: This defines the location of the startup stack. For all startup types it is initially allocated at the half-way point of the interrupt stack.

Diagnostic LEDs

Four LEDs are fitted on the board for diagnostic purposes:

Platform HAL manifest	Colour	Board Label
CYGHWR_IO_TWRK70F120M_PIN_LED1	Orange	E1
CYGHWR_IO_TWRK70F120M_PIN_LED2	Yellow	E2
CYGHWR_IO_TWRK70F120M_PIN_LED3	Green	E3
CYGHWR_IO_TWRK70F120M_PIN_LED4	Blue	E4

The platform HAL header file at `<cyg/hal/plf_io.h>` defines the following convenience function to allow the LEDs to be set:

```
extern void hal_twr_k70f120m_led(char c);
```

The lowest 4-bits of the argument `c` correspond to each of the 4 LEDs (with LED1 as the least significant bit).

The platform HAL will automatically light all of the LEDs when the platform initialisation is complete, however the LEDs are free for application use.

Real-time Characterization

The tm_basic kernel test gives statistics gathered about the real-time characterization and performance of the kernel. The sample output is shown here for information. The test was built for the variant ROM startup with optimization flag -O2.

Example 288.1. twr_k70f120m Real-time characterization

```
INFO:<code from 0x00000410 -> 0x00009d6c, CRC 3328>
             Startup, main thrd : stack used   240 size  2336
             Startup : Idlethread stack used   216 size  1440

eCos Kernel Timings
Notes: all times are in microseconds (.000001) unless otherwise stated

Reading the hardware clock takes 0 'ticks' overhead
... this value will be factored out of all other measurements
Clock interrupt took    4.00 microseconds (4 raw clock ticks)

Testing parameters:
   Clock samples:            32
   Threads:                   3
   Thread switches:         128
   Mutexes:                  32
   Mailboxes:                32
   Semaphores:               32
   Scheduler operations:    128
   Counters:                 32
   Flags:                    32
   Alarms:                   32
   Stack Size:             1888


                                 Confidence
     Ave     Min     Max     Var  Ave  Min  Function
  ======  ======  ======  ====== ========== ========
    6.67    6.00    8.00    0.89   66%  66% Create thread
    1.00    1.00    1.00    0.00  100% 100% Yield thread [all suspended]
    1.00    1.00    1.00    0.00  100% 100% Suspend [suspended] thread
    1.00    1.00    1.00    0.00  100% 100% Resume thread
    1.67    1.00    2.00    0.44   66%  33% Set priority
    0.00    0.00    0.00    0.00  100% 100% Get priority
    3.33    3.00    4.00    0.44   66%  66% Kill [suspended] thread
    1.00    1.00    1.00    0.00  100% 100% Yield [no other] thread
    2.00    2.00    2.00    0.00  100% 100% Resume [suspended low prio] thread
    1.33    1.00    2.00    0.44   66%  66% Resume [runnable low prio] thread
    1.33    1.00    2.00    0.44   66%  66% Suspend [runnable] thread
    1.00    1.00    1.00    0.00  100% 100% Yield [only low prio] thread
    1.00    1.00    1.00    0.00  100% 100% Suspend [runnable->not runnable]
    3.33    3.00    4.00    0.44   66%  66% Kill [runnable] thread
    2.67    2.00    3.00    0.44   66%  33% Destroy [dead] thread
    4.67    4.00    5.00    0.44   66%  33% Destroy [runnable] thread
    6.67    6.00    7.00    0.44   66%  33% Resume [high priority] thread
    2.13    2.00    3.00    0.23   86%  86% Thread switch

    0.24    0.00    1.00    0.37   75%  75% Scheduler lock
    0.81    0.00    1.00    0.31   81%  18% Scheduler unlock [0 threads]
    0.88    0.00    1.00    0.22   87%  12% Scheduler unlock [1 suspended]
    0.82    0.00    1.00    0.30   82%  17% Scheduler unlock [many suspended]
    0.88    0.00    1.00    0.22   87%  12% Scheduler unlock [many low prio]

    0.28    0.00    1.00    0.40   71%  71% Init mutex
    1.22    1.00    2.00    0.34   78%  78% Lock [unlocked] mutex
    1.34    1.00    2.00    0.45   65%  65% Unlock [locked] mutex
    1.09    1.00    2.00    0.17   90%  90% Trylock [unlocked] mutex
    0.97    0.00    1.00    0.06   96%   3% Trylock [locked] mutex
    0.31    0.00    1.00    0.43   68%  68% Destroy mutex
    5.94    5.00    7.00    0.18   87%   9% Unlock/Lock mutex

    0.38    0.00    1.00    0.47   62%  62% Create mbox
    0.25    0.00    1.00    0.38   75%  75% Peek [empty] mbox
    1.25    1.00    2.00    0.38   75%  75% Put [first] mbox
    0.22    0.00    1.00    0.34   78%  78% Peek [1 msg] mbox
    1.19    1.00    2.00    0.30   81%  81% Put [second] mbox
    0.28    0.00    1.00    0.40   71%  71% Peek [2 msgs] mbox
    1.28    1.00    2.00    0.40   71%  71% Get [first] mbox
    1.25    1.00    2.00    0.38   75%  75% Get [second] mbox
    1.13    1.00    2.00    0.22   87%  87% Tryput [first] mbox
    1.09    1.00    2.00    0.17   90%  90% Peek item [non-empty] mbox
    1.13    1.00    2.00    0.22   87%  87% Tryget [non-empty] mbox
    1.03    1.00    2.00    0.06   96%  96% Peek item [empty] mbox
    1.25    1.00    2.00    0.38   75%  75% Tryget [empty] mbox
    0.22    0.00    1.00    0.34   78%  78% Waiting to get mbox
    0.25    0.00    1.00    0.38   75%  75% Waiting to put mbox
    0.47    0.00    1.00    0.50   53%  53% Delete mbox
    4.13    4.00    5.00    0.22   87%  87% Put/Get mbox

    0.22    0.00    1.00    0.34   78%  78% Init semaphore
    1.00    1.00    1.00    0.00  100% 100% Post [0] semaphore
    1.09    1.00    2.00    0.17   90%  90% Wait [1] semaphore
    0.88    0.00    1.00    0.22   87%  12% Trywait [0] semaphore
    1.00    1.00    1.00    0.00  100% 100% Trywait [1] semaphore
    0.44    0.00    1.00    0.49   56%  56% Peek semaphore
    0.25    0.00    1.00    0.38   75%  75% Destroy semaphore
    3.69    3.00    5.00    0.47   62%  34% Post/Wait semaphore

    0.75    0.00    1.00    0.38   75%  25% Create counter
    0.38    0.00    1.00    0.47   62%  62% Get counter value
    0.25    0.00    1.00    0.38   75%  75% Set counter value
    1.25    1.00    2.00    0.38   75%  75% Tick counter
    0.25    0.00    1.00    0.38   75%  75% Delete counter

    0.28    0.00    1.00    0.40   71%  71% Init flag
    1.06    1.00    2.00    0.12   93%  93% Destroy flag
    0.97    0.00    2.00    0.12   90%   6% Mask bits in flag
    1.13    1.00    2.00    0.22   87%  87% Set bits in flag [no waiters]
    1.78    1.00    2.00    0.34   78%  21% Wait for flag [AND]
    1.50    1.00    2.00    0.50  100%  50% Wait for flag [OR]
    1.56    1.00    2.00    0.49   56%  43% Wait for flag [AND/CLR]
    1.59    1.00    2.00    0.48   59%  40% Wait for flag [OR/CLR]
    0.25    0.00    1.00    0.38   75%  75% Peek on flag

    0.81    0.00    1.00    0.31   81%  18% Create alarm
    1.78    1.00    2.00    0.34   78%  21% Initialize alarm
    0.94    0.00    1.00    0.12   93%   6% Disable alarm
    1.63    1.00    2.00    0.47   62%  37% Enable alarm
    1.13    1.00    2.00    0.22   87%  87% Delete alarm
    1.38    1.00    2.00    0.47   62%  62% Tick counter [1 alarm]
    6.63    6.00    7.00    0.47   62%  37% Tick counter [many alarms]
    2.22    2.00    3.00    0.34   78%  78% Tick & fire counter [1 alarm]
   36.91   36.00   37.00    0.17   90%   9% Tick & fire counters [>1 together]
    7.72    7.00    8.00    0.40   71%  28% Tick & fire counters [>1 separately]
    3.01    3.00    4.00    0.01   99%  99% Alarm latency [0 threads]
    3.00    3.00    3.00    0.00  100% 100% Alarm latency [2 threads]
    3.05    3.00    4.00    0.10   94%  94% Alarm latency [many threads]
    6.02    6.00    8.00    0.05   98%  98% Alarm -> thread resume latency

     344     344     344                    Worker thread stack used (stack size 1888)
            All done, main thrd : stack used   792 size  2336
            All done : Idlethread stack used   304 size  1440

Timing complete - 28960 ms total

PASS:<Basic timing OK>
EXIT:<done>
```

Chapter 289. LM3S Variant HAL

Name

CYGPKG_HAL_CORTEXM_LM3S — eCos Support for the LM3S Microprocessor Family

Description

The Luminary LM3Sxxxx series of Cortex-M microcontrollers is supported by eCos with an eCos processor variant HAL and a number of device drivers supporting some of the on-chip peripherals. These include device drivers for the on-chip flash, serial and watchdog devices. In addition it provides common functionality and definitions that LM3S based platform ports may require, as well as definitions useful to application developers.

This documentation covers the LM3S functionality provided but should be read in conjunction with the specific HAL documentation for the platform port. That documentation will cover issues that are platform-specific and are not covered here, and may also describe differences that override or supersede what the LM3S variant HAL provides. The areas that are specific to platform HALs and not the LM3S variant HAL include:

- memory map and related configuration and setup
- Clock parameters
- GPIO setup
- Any special handling for external interrupts, or additional interrupts
- Diagnostic I/O baud rates
- Additional diagnostic I/O devices, if any
- LED/LCD control

Name

On-chip Subsystems and Peripherals — Hardware Support

Hardware support

On-chip memory

The ST LM3S parts include on-chip SRAM, and on-chip FLASH. The RAM consists of up to 64KiB and the FLASH can be up to 512KiB in size depending on model.

Typically, an eCos platform HAL port will expect a GDB stub ROM monitor to be programmed into the LM3S on-chip ROM memory for development, and the board would boot this image from reset. The stub ROM provides GDB stub functionality so it is then possible to download and debug stand-alone and eCos applications via the gdb debugger using serial interfaces or other debug channels. The JTAG interface may also be used for development if a suitable JTAG device is available. For production purposes, applications are programmed into external or on-chip FLASH and will be self-booting.

On-Chip FLASH

The package `CYGPKG_DEVS_FLASH_LM3S` provides a driver for the on-chip flash. This driver conforms to the Version 2 flash driver API. It queries the microcontroller's device capabilities registers to determine the size and layout of the flash at runtime.

Cache Handling

The LM3S does not contain any caches, however, the variant HAL supplies the `cyg/hal/hal_cache.h` header to satisfy generic code. This header describes zero sized caches and provides null macros for the required functions.

Serial I/O

The LM3S variant HAL supports basic polled HAL diagnostic I/O over any of the on-chip serial devices. There is also a fully interrupt-driven serial device driver suitable for eCos applications for all on-chip serial devices. The serial driver consists of an eCos package: `CYGPKG_IO_SERIAL_CORTEXM_LM3S` which provides configuration for the generic `CYGPKG_IO_SERIAL_ARM_PL011` driver package. Using the HAL diagnostic I/O support, any of these devices can be used by the ROM monitor for communication with GDB. If a device is needed by the application, either directly or via the serial driver, then it cannot also be used for GDB communication using the HAL I/O support. An alternative serial port should be used instead.

The HAL defines CDL interfaces, `CYGINT_HAL_LM3S_UART0` to `CYGINT_HAL_LM3S_UART4` for each of the possible UARTs. At present no LM3S device has more that 3 UARTs, so these interfaces contain support for future expansion. The platform HAL CDL should contain an **implements** directive for each such UART that is available for use on the board. This will enable use of the UART for diagnostic use.

The LM3S UARTs provide only TX and RX data lines, although the PL011 macrocell is theoretically capable of RTS/CTS flow control.

Interrupts

The LM3S HAL relies on the architectural HAL to provide support for the interrupts directly routed to the NVIC. The `cyg/hal/var_intr.h` header defines the vector mapping for these.

GPIO

The variant HAL provides support for packaging the configuration of a GPIO line into a single 32-bit descriptor that can then be used with macros to configure the pin and set and read its value.

Clock Distribution

The variant HAL provides support for packaging the clock control parameters of a device into a single 32-bit descriptor that can then be used with macros to enable and disable the device's clock.

I2C Support

The variant HAL provides a driver for the I²C bus device. There is a configuration option, `CYGNUM_HAL_LM3S_I2C_BUS0_CLOCK`, that defines the clock speed at which the bus operates. The platform HAL must define the set of devices attached to the bus.

SPI Support

The SSI device is based on the ARM PL022 SSP primecell and SPI support is provided via the separate `CYGPKG_DEVS_SPI_ARM_PL022` driver. The platform HAL must define the bus instances and devices attached to them.

Profiling Support

The LM3S HAL contains support for **gprof**-base profiling using a sampling timer. The default timer used is Timer 0. The timer used is selected by a set of `#define`s in `src/lm3s_misc.c` which can be changed to refer to a different timer if required. This timer is only enabled when `CYGPKG_PROFILE_GPROF` is enabled, otherwise it remains available for application use.

Clock Control

The platform HAL must provide the input clock frequency (`CYGARC_HAL_CORTEXM_LM3S_INPUT_CLOCK`) in its CDL file. This is then combined with the following options defined in this package to define the default system clocks:

CYGHWR_HAL_CORTEXM_LM3S_CLOCK_SOURCE

> This defines the source of the main system clock. It can take one of six values: `INT` selects the internal ocillator, `INTby4` selects the internal ocillator divided by 4, `MAIN` selects the main ocillator, `PLL` selects the PLL, `30K` selects the 30KHz internal clock, `32K` selects the 32KHz internal clock. It defaults to `PLL`.

CYGHWR_HAL_CORTEXM_LM3S_CLOCK_SYSCLK_DIV

> This defines the divider applied to the 400MHz PLL output to generate the system clock. This can take values between 1 and 16. The default value is 4, giving a 50MHz system clock.

CYGHWR_HAL_CORTEXM_LM3S_CLOCK_PWM_DIV

> This defines the prescaler divider for the Pulse Width Modulator. It may take any power of 2 value between 1 and 64. The default is 1.

The actual values of the system clock, in Hz, is stored in the global variable `hal_lm3s_sysclk`. The clock supplied to the SysTick timer, SYSCLK/4, is also assigned to `hal_cortexm_systick_clock`. These variables are used, rather than configuration options, in anticipation of future support for power management by varying the system clock rate.

Name

GPIO Support — Details

Synopsis

#include <cyg/hal/hal_io.h>

pin = **CYGHWR_HAL_LM3S_GPIO**(*port*, *bit*, *drive*, *mode*);

CYGHWR_HAL_LM3S_GPIO_SET (*pin*);

CYGHWR_HAL_LM3S_GPIO_OUT (*pin*, *val*);

CYGHWR_HAL_LM3S_GPIO_IN (*pin*, **val*);

Description

The LM3S HAL provides a number of macros to support the encoding of GPIO pin identity and configuration into a single 32 bit descriptor. This is useful to drivers and other packages that need to configure and use different lines for different devices.

A descriptor is created with `CYGHWR_HAL_LM3S_GPIO(port, bit, mode)` which takes the following arguments:

port
: This identifies the GPIO port to which the pin is attached. Ports are identified by letters from A to F.

bit
: This gives the bit or pin number within the port. These are numbered from 0 to 8.

drive
: This defines the drive level for the external pad. It may be set to `2mA`, `4mA` or `8mA`. It may also be set to `8mAS` to add slew rate control.

mode
: This defines the mode in which the pin is to be used. The following values are currently defined: `IN_PULLUP` defines the pin as a GPIO input with a pull up resistor, `IN_PULLDOWN` defines the pin as a GPIO input with a pull down resistor, `OUT_OPENDRAIN` defines the pin as a GPIO output with an open drain, `ALT_DIGITAL` defines the pin as a digital line under the control of a peripheral. `ALT_OD` defines the pin as a digital open drain line under the control of a peripheral. `ALT_ODPU` defines the pin as a digital open drain line with pull up under the control of a peripheral. `ALT_ODPD` defines the pin as a digital open drain line with pull down under the control of a peripheral. `ALT_PP` defines the pin as a digital push-pull line under the control of a peripheral. `ALT_PPPU` defines the pin as a digital pushpull line with pull up under the control of a peripheral. `ALT_PPPD` defines the pin as a digital pushpull line with pull down under the control of a peripheral. This set may be extended as further requirements emerge, so check the sources for new definitions.

The following examples show how this macro may be used:

```
// Define port A pin 0 as a digital device pin with 2mA drive level
#define CYGHWR_HAL_LM3S_UART0_RX                CYGHWR_HAL_LM3S_GPIO( A,  0, 2mA, ALT_DIGITAL )
```

Additionally, the macro `CYGHWR_HAL_LM3S_GPIO_NONE` may be used in place of a pin descriptor and has a value that no valid descriptor can take. It may therefore be used as a placeholder where no GPIO pin is present or to be used.

The remaining macros all take a GPIO pin descriptor as an argument. `CYGHWR_HAL_LM3S_GPIO_SET` configures the pin according to the descriptor and must be called before any other macros. `CYGHWR_HAL_LM3S_GPIO_OUT` sets the output to the value of the least significant bit of the *val* argument. The *val* argument of `CYGHWR_HAL_LM3S_GPIO_IN` should be a pointer to an int, which will be set to 0 if the pin input is zero, and 1 otherwise.

Chapter 290. LM3S8962-EVAL Platform HAL

Name

CYGPKG_HAL_CORTEXM_LM3S_LM3S8962_EVAL — eCos Support for the LM3S8962-EVAL Board

Description

The LM3S8962-EVAL board contains a LM3S8962 microcontroller. It has connectors for one UART, MicroSD, USB, CAN, JTAG and various other devices.

For typical eCos development, a GDB stub image is programmed into internal FLASH and the CPU boots directly into that. It is then possible to download and debug stand-alone and eCos applications via the gdb debugger using UART0. Alternatively test programs may be downloaded and debugged via a JTAG debugger attached to the JTAG socket. Avaliable RAM is limited to 64KB, so development for larger applications may also consist of programming them to flash and using JTAG to debug them from there.

This documentation describes platform-specific elements of the LM3S8962-EVAL board support within eCos. The LM3S variant HAL documentation covers various topics including HAL support common to LM3S variants, and on-chip device support. This document complements the LM3S documentation.

Supported Hardware

The LM3S has two on-chip memory regions. A RAM region of 64KiB is present at 0x20000000. A FLASH region is present at 0x00000000.

The LM3S variant HAL includes support for the on-chip serial devices which are documented in the variant HAL. UART0 is connected to a USB adaptor, which also serves to provide the board with power.

The platform HAL contains configuration and definitions that allow the ARM PL022 primecell device to be used for SPI devices.

Tools

The board port is intended to work with GNU tools configured for an **arm-eabi** target. The original port was done using **arm-eabi-gcc** version 4.3.2, **arm-eabi-gdb** version 6.8, and **binutils** version 2.18.

Name

Setup — Preparing the LM3S8962-EVAL Board for eCos Development

Overview

In a typical development environment, the LM3S8962-EVAL board boots from internal flash into the GDB Stubrom. eCos applications are configured for RAM startup and then downloaded and run on the board via the debugger **arm-eabi-gdb**.

Stubrom Installation

For serial communications, the Stubrom runs with 8 bits, no parity, and 1 stop bit at 38400 baud. This rate can be changed in the platform HAL configuration. Under normal circumstances, the Stubrom runs in-place from the internal Flash.

Programming The Stubrom

To program the Stubrom into the internal flash either a JTAG debugger that understands the LM3S flash may be used, such as a Ronetix PEEDI or an Abatron BDI3000, or the TI **LM Flash Programmer** may be used. Configuration files for the PEEDI and BDI3000 are supplied in the LM3S8962-EVAL HAL package, and brief instructions for downloading the Stubrom are given in there. If no JTAG debugger is available, then the Stubrom must be downloading using the **LM Flash Programmer**. The following are brief instructions for doing this. The reader is referred to the documentation that comes with the loader for full details.

1. Download the **LM Flash Programmer** and Stellaris FTDI Windows driver either from TI or Luminary Micro websites, or from the CD supplied with the board and install it on a PC running Windows that has an available USB port.

2. Copy the file `stubrom.bin` to a suitable location on the Windows PC.

3. Connect the USB cable supplied with the board between the board and the PC. Follow any instructions to install the driver.

4. Start the **LM Flash Programmer** and in the Configuration tab select the LM3S8962 Ethernet an CAN evaluation board.

5. In the Program tab either type in or browse to the `stubrom.bin` file. Select *"Erase Entire Flash"* and *"Verify After Program"*. Ensure that the *"Program Address Offset"* is zero.

6. Press *"Program"* button and the loader should download and verify the binary file. The programming is now complete and the **LM Flash Programmer** can now be exited.

Whatever mechanism is used to program the Stubrom, something similar to the following output should be seen on the Windows USB virtual COM port when the reset button is pressed:

```
+$T050f:1a220000;0d:f0ff0020;#a7
```

Rebuilding The Stubrom

Should it prove necessary to rebuild the Stubrom binary, this is done most conveniently at the command line. The steps needed to rebuild the the ROM version of the Stubrom are:

```
$ mkdir stubs_lm3s8962_rom
$ cd stubs_lm3s8962_rom
$ ecosconfig new lm3s8962 stubs
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

At the end of the build the `install/bin` subdirectory should contain the file `stubrom.bin`.

Name

Configuration — Platform-specific Configuration Options

Overview

The LM3S8962-EVAL board platform HAL package is loaded automatically when eCos is configured for an `lm3s8962_eval` target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The LM3S8962-EVAL board platform HAL package supports three separate startup types:

RAM This is the startup type which is normally used during application development. The board has GDB stubs programmed into internal Flash at location 0x00000000 and uses internal RAM at location 0x20000000. arm-eabi-gdb is then used to load a RAM startup application into memory from 0x20001000 and debug it. It is assumed that the hardware has already been initialized by the stubs. By default the application will use the eCos virtual vectors mechanism to obtain services from the stubs, including diagnostic output.

ROM This startup type can be used for finished applications which will be programmed into internal ROM at location 0x00000000. Data and BSS will be put into internal RAM starting from 0x20000400. The application will be self-contained with no dependencies on services provided by other software. The program expects to boot from reset into ROM at location zero. eCos startup code will perform all necessary hardware initialization.

JTAG This is the startup type used to build applications that are loaded via a JTAG interface. The application will be self-contained with no dependencies on services provided by other software. The program expects to be loaded from 0x20000400 and entered at that address. eCos startup code will perform all necessary hardware initialization.

Monitors and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building the Stubrom.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then serial port 0 will be claimed for HAL diagnostics.

UART Serial Driver

The LM3S8962-EVAL board uses the LM3S's internal UART serial support. As well as the polled HAL diagnostic interface, there is also a `CYGPKG_IO_SERIAL_CORTEXM_LM3S` package which configures the `CYGPKG_IO_SERIAL_ARM_PL011` for use in the LM3S series. Both UARTs can be supported by this driver, although only UART0 is actually routed to an external connector. Note that it is not recommended to enable this driver on the port used for HAL diagnostic I/O. This driver is not active until the `CYGPKG_IO_SERIAL_DEVICES` configuration option within the generic serial driver support package `CYGPKG_IO_SERIAL` is enabled in the configuration.

Name

JTAG support — Usage

Use of JTAG for debugging

JTAG can be used to single-step and debug loaded applications, or even applications resident in ROM, including the Stubrom.

Debugging of ROM applications is only possible if using hardware breakpoints. The Cortex-M3 core of the LM3S only supports two such hardware breakpoints, so they should be used sparingly. If using a GDB front-end such as Eclipse, check it has not set unnecessary extra breakpoints. Some JTAG devices give the option of whether to set hardware or software breakpoints by default. Be sure to configure your device appropriately.

When debugging via JTAG, you are likely to need to disable the default HAL idle thread action, otherwise there may be issues where the target fails to halt and the debugging session is unreliable. More details can be found in the Cortex-M architectural HAL.

Ronetix PEEDI notes

On the Ronetix PEEDI, the `peedi.lm3s8962.cfg` file should be used to setup and configure the hardware to an appropriate state to load programs.

The `peedi.lm3s8962.cfg` file also contains an option to define whether hardware or software breakpoints are used by default, using the `CORE0_BREAKMODE` directive in the `[TARGET]` section. The supplied version of the file defaults to software breakpoints. With this default, hardware breakpoints can still be set from GDB using the **hbreak** command. The default can be changed to hardware breakpoints, and remember to use the **reboot** command on the PEEDI command line interface, or press the reset button to make the changes take effect.

On the PEEDI, debugging can be performed either via the telnet interface or using **arm-eabi-gdb**. In the case of the latter, **arm-eabi-gdb** needs to connect to TCP port 2000 on the PEEDI's IP address. For example:

```
(gdb) target remote 111.222.333.444:2000
```

By default when the PEEDI is powered up, the target will always run the initialization section of the `peedi.lm3s8962.cfg` file, and halts the target. This behaviour is repeated with the **reset** command.

If the board is reset with the '**reset**' command, or by pressing the reset button and the '**go**' command is then given, then the board will boot from ROM as normal. A similar effect can be achieved in GDB by connecting with **target remote** and immediately typing **continue** or **c**.

It is also possible for the target to always run, without initialization, after the reset button has been pressed. This mode is selected with the `CORE0_STARTUP_MODE` directive in the `[TARGET]` section of the `peedi.lm3s8962.cfg` file. This conveniently allows the target to be connected to the JTAG debugger, and be able to reset it with the reset button, without being required to always type '**go**' every time.

Suitably configured RAM applications can be loaded either via GDB, or directly via the telnet CLI. For example:

```
lm3s> memory load tftp://192.168.7.9/test.bin bin 0x20000000
++ info: Loading image file:  tftp://192.168.7.9/test.bin
++ info: At absolute address: 0x00000000
loading at 0x20000000
loading at 0x20004000

Successfully loaded 28KB (29064 bytes) in 0.1s
lm3s> go 0x20000000
```

Consult the PEEDI documentation for information on other formats and loading mechanisms.

Abatron BDI3000 notes

On the Abatron BDI3000, the `bdi3000.lm3s8962.cfg` file should be used to setup and configure the hardware to an appropriate state to load programs.

The `bdi3000.lm3s8962.cfg` file also contains an option to define whether hardware or software breakpoints are used by default, using the `BREAKMODE` directive in the `[TARGET]` section. The supplied version of the file defaults to software breakpoints. With this default, hardware breakpoints can still be set from GDB using the **hbreak** command. The default can be changed to hardware breakpoints, and remember to use the **boot** command on the BDI3000 command line interface.

On the BDI3000, debugging can be performed either via the telnet interface or using **arm-eabi-gdb**. In the case of the latter, **arm-eabi-gdb** needs to connect to TCP port 2001 on the BDI3000's IP address. For example:

```
(gdb) target remote 111.222.333.444:2001
```

By default when the BDI3000 is powered up, the target will always run the initialization section of the `bdi3000.lm3s8962.cfg` file, and halts the target. This behaviour is repeated with the **reset** command.

If the board is reset with the '**reset**' command, or by pressing the reset button and the '**go**' command is then given, then the board will boot from ROM as normal. A similar effect can be achieved in GDB by connecting with **target remote** and immediately typing **continue** or **c**.

It is also possible for the target to always run, without initialization, after the reset button has been pressed. This mode is selected with the **reset run** command. This conveniently allows the target to be connected to the JTAG debugger, and be able to reset it with the reset button, without being required to always type '**go**' every time. Thereafter, invoking the **reset** command will repeat the previous reset style. Also in this mode, exceptions will be handled by board software, rather than causing the JTAG debugger to halt the CPU.

Suitably configured RAM applications can be loaded either via GDB, or directly via the telnet CLI. For example:

```
LM3S> load 0x20000000 test.bin bin
Loading /test.bin , please wait ....
Loading program file passed
LM3S> go 0x20000000
```

Consult the BDI3000 documentation for information on other formats and loading mechanisms.

Configuration of JTAG applications

JTAG applications can be loaded directly into RAM without requiring a ROM monitor. This loading can be done directly through the JTAG device, or where supported by the JTAG device, through GDB.

In order to configure the application to support this mode, some configuration settings are required. Firstly `CYGSEM_HAL_USE_ROM_MONITOR` must be disabled. Secondly the `CYGDBG_HAL_DIAG_TO_DEBUG_CHAN` option should be enabled in order to prevent HAL diagnostic output being encoded into GDB ($O) packets. Both of these settings are made automatically if the JTAG startup type is selected.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the LM3S8962-EVAL board hardware, and should be read in conjunction with that specification. The LM3S8962-EVAL platform HAL package complements the ARM architectural HAL and the LM3S variant HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset the HAL will initialize or reinitialize many of the on-chip peripherals. There is an exception for RAM startup applications which depend on a ROM monitor for certain services.

For ROM and JTAG startup, the HAL will perform additional initialization, programming the various internal registers including the PLL, peripheral clocks and GPIO pins. The details of the early hardware startup may be found in the `src/lm3s8962_eval_misc.c` in both `hal_system_init` and `hal_platform_init`.

Memory Map

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

Internal RAM
: This is located at address 0x20000000 of the memory space, and is 64KiB in size. The eCos VSR table occupies the bottom 512 bytes. The virtual vector table starts at 0x00000200 and extends to 0x00000300. The top `CYGNUM_HAL_COMMON_INTERRUPTS_STACK_SIZE` bytes are reserved for the interrupt stack. The remainder of internal RAM is available for use by applications.

Internal FLASH
: This is located at address 0x00000000 of the memory space. This region is 256KiB in size. ROM applications are by default configured to run from this memory.

On-Chip Peripherals
: These are accessible at locations 0x40000000 and 0xE0000000 upwards. Descriptions of the contents can be found in the LM3S User Manual.

Linker Scripts

The platform linker scripts define the following symbols:

hal_vsr_table
: This defines the location of the VSR table. This is set to 0x20000000 for all startup types, and space for 128 entries is reserved.

hal_virtual_vector_table
: This defines the location of the virtual vector table used to communicate between an ROM monitor and an eCos application. This is allocated right after the VSR table, at 0x20000200. To permit expansion and possible addition of other tables, the linker scripts then allocate further sections from 0x20000400.

hal_interrupt_stack
: This defines the location of the interrupt stack. For all startups, this is allocated to the top of internal SRAM, 0x20010000.

hal_startup_stack
: This defines the location of the startup stack. For all startup types it is initially allocated at the half-way point of the interrupt stack.

Chapter 291. LPC1XXX Variant HAL

Name

`CYGPKG_HAL_CORTEXM_LPC1XXX` — eCos Support for the LPC1XXX Microprocessor Family

Description

The NXP LPC1XXX series of Cortex-M microcontrollers is supported by eCos with an eCos processor variant HAL and a number of device drivers supporting some of the on-chip peripherals. These include device drivers for the on-chip flash, serial, SPI and I²C devices. In addition it provides common functionality and definitions that LPC1XXX based platform ports may require, as well as definitions useful to application developers.

This documentation covers the LPC1XXX functionality provided but should be read in conjunction with the specific HAL documentation for the platform port. That documentation will cover issues that are platform-specific and are not covered here, and may also describe differences that override or supersede what the LPC1XXX variant HAL provides. The areas that are specific to platform HALs and not the LPC1XXX variant HAL include:

- memory map and related configuration and setup
- Clock parameters
- GPIO setup
- Any special handling for external interrupts, or additional interrupts
- Diagnostic I/O baud rates
- Additional diagnostic I/O devices, if any
- LED/LCD control

Name

On-chip Subsystems and Peripherals — Hardware Support

Hardware support

On-chip memory

The LPC1XXX parts include on-chip SRAM, and on-chip FLASH. The RAM consists of up to 64KiB in one or two disjoint blocks, and the FLASH can be up to 512KiB in size depending on model.

Typically, an eCos platform HAL port will expect a GDB stub ROM monitor to be programmed into the LPC1XXX on-chip ROM memory for development, and the board would boot this image from reset. The stub ROM provides GDB stub functionality so it is then possible to download and debug stand-alone and eCos applications via the gdb debugger using serial interfaces or other debug channels. The JTAG interface may also be used for development if a suitable JTAG device is available. For production purposes, applications are programmed into on-chip FLASH and will be self-booting.

On-Chip FLASH

The package `CYGPKG_DEVS_FLASH_LPC2XXX` provides a driver for the on-chip flash. This driver conforms to the Version 2 flash driver API. It queries the microcontroller's device capabilities registers to determine the size and layout of the flash at runtime. This driver is shared with the LPC2xxx microcontroller family, and its name reflects that.

Cache Handling

The LPC1XXX does not contain any caches, however, the variant HAL supplies the `cyg/hal/hal_cache.h` header to satisfy generic code. This header describes zero sized caches and provides null macros for the required functions.

Serial I/O

The LPC1XXX variant HAL supports basic polled HAL diagnostic I/O over any of the on-chip serial devices. There is also a fully interrupt-driven serial device driver suitable for eCos applications for all on-chip serial devices. The serial driver consists of an eCos package: `CYGPKG_IO_SERIAL_CORTEXM_LPC1XXX` which provides configuration for the generic `CYGPKG_IO_SERIAL_GENERIC_16X5X` driver package. Using the HAL diagnostic I/O support, any of these devices can be used by the ROM monitor for communication with GDB. If a device is needed by the application, either directly or via the serial driver, then it cannot also be used for GDB communication using the HAL I/O support. An alternative serial port should be used instead.

The HAL defines CDL interfaces, `CYGINT_HAL_LPC1XXX_UART0` to `CYGINT_HAL_LPC1XXX_UART3` for each of the possible UARTs. The platform HAL CDL should contain an **implements** directive for each such UART that is available for use on the board. This will enable use of the UART for diagnostic use.

UARTs 0, 2 and 3 only support TX and RX lines, however UART1 supports the full set of modem control lines.

Interrupts

The LPC1XXX HAL relies on the architectural HAL to provide support for the interrupts directly routed to the NVIC. The `cyg/hal/var_intr.h` header defines the vector mapping for these.

GPIO

The variant HAL provides support for packaging the configuration of a GPIO line into a single 32-bit descriptor that can then be used with macros to configure the pin and set and read its value.

Clock Distribution

The variant HAL provides support for packaging the clock control parameters of a device into a single 32-bit descriptor that can then be used with macros to enable and disable the device's clock.

I2C Support

A separate driver, `CYGPKG_DEVS_I2C_NXPI2C` provides support for I²C devices. The platform HAL must define the set of devices attached to each bus and must also configure the pins used for each I²C bus.

SPI Support

The SSP device is based on the ARM PL022 SSP primecell and SPI support is provided via the separate `CYGPKG_DEVS_SPI_ARM_PL022` driver. The platform HAL must define the bus instances and devices attached to them.

Profiling Support

The LPC1XXX HAL contains support for **gprof**-base profiling using a sampling timer. The default timer used is Timer 0. The timer used is selected by a set of `#define`s in `src/lpc1xxx_misc.c` which can be changed to refer to a different timer if required. This timer is only enabled when `CYGPKG_PROFILE_GPROF` is enabled, otherwise it remains available for application use.

Clock Control

The platform HAL must provide the input clock frequency (`CYGHWR_HAL_LPC1XXX_INPUT_CLOCK`) in its CDL file. This is then combined with the following options defined in this package to define the default system clocks:

CYGHWR_HAL_CORTEXM_LPC1XXX_CLOCK_SOURCE

> This defines the source of the main system clock. It can take one of three values: `IRC` selects the internal ocillator, `OSC` selects the main ocillator, `RTC` selects the 32KHz internal clock. It defaults to `OSC`.

CYGHWR_HAL_CORTEXM_LPC1XXX_CLOCK_SYSCLK_DIV

> This defines the divider applied to the 400MHz PLL output to generate the system clock. This can take values between 1 and 16. The default value is 4, giving a 50MHz system clock.

CYGHWR_HAL_CORTEXM_LPC1XXX_PLL0_MUL

> This defines the multiplier applied by PLL0 to the selected clock input. It can vary between 6 and 512. The default is 6.

CYGHWR_HAL_CORTEXM_LPC1XXX_PLL0_PREDIV

> This defines the pre-divider for PLL0. It may take any value betwen 1 and 32. The default is 1.

CYGHWR_HAL_CORTEXM_LPC1XXX_CCLK_SOURCE

> This defines the source of the main CPU clock, CCLK. The options are `PLL` to select PLL0 and `SYSCLK` to bypass the PLL and use the system clock directly. The default is `PLL`.

CYGHWR_HAL_CORTEXM_LPC1XXX_CCLK_DIV

> This defines the divider applied to the selected CCLK. It may range between 3 and 256. The default is 3.

The actual frequency of the system clock, in Hz, is stored in the global variable `hal_lpc1xxx_sysclk`. Similarly the frequency of the PLL output clock is stored in `hal_lpc1xxx_pllclk` and of CCLK in `hal_lpc1xxx_cclk`. The clock supplied to the

SysTick timer, CCLK, is also assigned to `hal_cortexm_systick_clock`. These variables are used, rather than configuration options, in anticipation of future support for power management by varying the system clock rate.

Name

GPIO Support — Details

Synopsis

#include <cyg/hal/hal_io.h>

`pin = ` **`CYGHWR_HAL_LPC1XXX_GPIO`**`(`*`port`*`, `*`bit`*`, `*`mode`*`);`

`CYGHWR_HAL_LPC1XXX_GPIO_SET`` (`*`pin`*`);`

`CYGHWR_HAL_LPC1XXX_GPIO_OUT`` (`*`pin`*`, `*`val`*`);`

`CYGHWR_HAL_LPC1XXX_GPIO_IN`` (`*`pin`*`, `*`*val`*`);`

Description

The LPC1XXX HAL provides a number of macros to support the encoding of GPIO pin identity and configuration into a single 32 bit descriptor. This is useful to drivers and other packages that need to configure and use different lines for different devices.

A descriptor is created with `CYGHWR_HAL_LPC1XXX_GPIO(port, bit, mode)` which takes the following arguments:

`port`
: This identifies the GPIO port to which the pin is attached. Ports are identified numerically between 0 and 4.

`bit`
: This gives the bit or pin number within the port. These are numbered from 0 to 31.

`mode`
: This defines the mode in which the pin is to be used. The following values are currently defined: `IN_PULLUP` defines the pin as a GPIO input with a pull up resistor, `IN_PULLDOWN` defines the pin as a GPIO input with a pull down resistor, `OUT_OPENDRAIN` defines the pin as a GPIO output with an open drain, `OUT_PULLUP` defines the pin as a GPIO out with a pull up resistor, `OUT_PULLDOWN` defines the pin as a GPIO output with a pull down resistor, `ALT1`, `ALT2`, `ALT3` define the pin as a line under the control of a peripheral. This set may be extended as further requirements emerge, so check the sources for new definitions.

The following examples show how this macro may be used:

```
// Define port 0 pin 2 as a peripheral pin for alternate peripheral 1
#define CYGHWR_HAL_LPC1XXX_UART0_TX             CYGHWR_HAL_LPC1XXX_GPIO( 0,  2, ALT1 )

// Define port 0 pin 16 as a GPIO output pin with a pull up resistor
#define CYGHWR_HAL_LPC1XXX_SPI_CS0              CYGHWR_HAL_LPC1XXX_GPIO(0, 16, OUT_PULLUP)
```

Additionally, the macro `CYGHWR_HAL_LPC1XXX_GPIO_NONE` may be used in place of a pin descriptor and has a value that no valid descriptor can take. It may therefore be used as a placeholder where no GPIO pin is present or to be used.

The remaining macros all take a GPIO pin descriptor as an argument. `CYGHWR_HAL_LPC1XXX_GPIO_SET` configures the pin according to the descriptor and must be called before any other macros. `CYGHWR_HAL_LPC1XXX_GPIO_OUT` sets the output to the value of the least significant bit of the *`val`* argument. The *`val`* argument of `CYGHWR_HAL_LPC1XXX_GPIO_IN` should be a pointer to an int, which will be set to 0 if the pin input is zero, and 1 otherwise.

Name

Peripheral Clock and Power Control — Description

Synopsis

#include <cyg/hal/hal_io.h>

pin = **CYGHWR_HAL_LPC1XXX_CLOCK**(*device*, *divider*);

CYGHWR_HAL_LPC1XXX_CLOCK_ENABLE (*desc*);

CYGHWR_HAL_LPC1XXX_CLOCK_DISABLE (*desc*);

Description

The LPC1XXX HAL provides a number of macros to support the management of peripheral power supply and clock frequencies. The macro `CYGHWR_HAL_LPC1XXX_CLOCK( device, divider )` encodes a control descriptor into a 32 bit value. The first argument is the name of the device to be described. The second argument gives the divider to be applied to CCLK for the peripheral clock; it can take the values 1, 2, 4 or 8 for most peripherals and 6 (instead of 8) for the CAN device.

The remaining functions all take a peripheral clock/power descriptor as an argument. `CYGHWR_HAL_LPC1XXX_CLOCK_ENABLE( desc )` turns on the power to the peripheral and sets the clock to the rate given in the descriptor. Likewise `CYGHWR_HAL_LPC1XXX_CLOCK_DISABLE( desc )` disables the power to the device.

Chapter 292. MCB1700 Platform HAL

Name

`CYGPKG_HAL_CORTEXM_LPC1XXX_MCB1700` — eCos Support for the MCB1700 Board

Description

The MCB1700 board contains a LPC1768 microcontroller. It has connectors for two UARTs, MicroSD, USB, CAN, a PHY connected to the on-chip MAC, JTAG and various other devices.

For typical eCos development, a GDB stub image is programmed into internal FLASH and the CPU boots directly into that. It is then possible to download and debug stand-alone and eCos applications via the gdb debugger using UART1. Alternatively test programs may be downloaded and debugged via a JTAG debugger attached to the JTAG socket. Avaliable RAM is limited to 64KiB, split into two disjoint 32KiB regions, so development for larger applications may also consist of programming them to flash and using JTAG to debug them from there.

This documentation describes platform-specific elements of the MCB1700 board support within eCos. The LPC1XXX variant HAL documentation covers various topics including HAL support common to LPC1XXX variants, and on-chip device support. This document complements the LPC1XXX documentation.

Supported Hardware

The LPC1768 has three on-chip memory regions. A RAM region of 32KiB is present at 0x0x10000000 and another is at 0x2007C000. A FLASH region is present at 0x00000000.

The LPC1XXX variant HAL includes support for the on-chip serial devices which are documented in the variant HAL. UART0 has reset and ISP control lines connected to the DTR and RTS lines. eCos uses UART1 for console and debug traffic, leaving UART0 free for ISP use.

The platform HAL contains configuration and definitions that allow the SPI and I²C device drivers to be used. This includes access to microSD cards via the on-board slot.

The on-chip Ethernet MAC and the DP83848 PHY are supported.

Tools

The board port is intended to work with GNU tools configured for an **arm-eabi** target. The original port was done using **arm-eabi-gcc** version 4.3.2, **arm-eabi-gdb** version 6.8, and **binutils** version 2.18.

Name

Setup — Preparing the MCB1700 Board for eCos Development

Overview

Applications can be developed on the MCB1700 board either by using a hardware JTAG-debugger based approach, or via a serial connection between the host and a GDB Stubrom installed on the board. In this latter case the board boots from internal flash into the Stubrom. eCos applications are configured for RAM startup and then downloaded and run on the board via the debugger **arm-eabi-gdb**.

Stubrom Installation

For serial communications, the Stubrom runs with 8 bits, no parity, and 1 stop bit at 38400 baud. This rate can be changed in the platform HAL configuration. Under normal circumstances, the Stubrom runs in-place from the internal Flash.

Programming The Stubrom

This process assumes that a Microsoft Windows host machine with the Embedded Systems Academy Free Flash Magic utility is available. You can install Flash Magic from http://www.flashmagictool.com. This machine also needs an RS232 serial port, either native or through a USB adaptor. In addition to bidirectional data transfer, control is also needed over the DTR and RTS lines. If necessary copy the `stubrom.hex` file to an easily accessible location. The stubrom file is located within your eCosPro installation in the `\eCosPro\ecos-<ver>loaders\mcb1700\gdbstub\ROM\` directory.

Supply the board with power via the USB DEVICE socket and connect a serial cable between the MCB1700 board COM0 port and the host.

Start Flash Magic and set the Communications section to select the host COM port connected to the board, 38400 baud, device LPC1768, Interface “None (ISP)” and 12MHz Oscillator Frequency. Test communication with the board by using the “ISP->Read Device Signature” menu entry. If communication is not successful, check that the serial cable is connected, and the correct COM port is being used.

Check “Erase blocks used by Hex File” under “Erase”. In the “Hex File” section, select the `stubrom.hex` file. Under “Options”, all boxes should be clear except “Verify after programming”. Now press the “Start” button. The utility should show the progress of the upload.

When the process completes, the utility should be closed. Verify the programming has been successful by connecting to the second (COM1) serial port on the board, start a terminal emulation application such as HyperTerminal or minicom on the host PC and set the serial communication parameters to 38400 baud, 8 data bits, no parity, 1 stop bit (8N1) and no flow control (handshaking). Reset the board and the Stubrom should start. The output should be similar to the following:

```
$T050f:6e220000;0d:f07f0010;#80
```

Rebuilding The Stubrom

Should it prove necessary to rebuild the Stubrom binary, this is done most conveniently at the command line. The steps needed to rebuild the the ROM version of the Stubrom are:

```
$ mkdir stubs_mcb1700_rom
$ cd stubs_mcb1700_rom
$ ecosconfig new mcb1700 stubs
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

At the end of the build the `install/bin` subdirectory should contain the file `stubrom.hex`.

Name

Configuration — Platform-specific Configuration Options

Overview

The MCB1700 board platform HAL package is loaded automatically when eCos is configured for an `mcb1700` target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The MCB1700 board platform HAL package supports three separate startup types:

RAM
: This is the startup type for application development using a GDB stubs based development approach. The Stubsrom is programmed into internal Flash at location 0x00000000 and uses internal RAM at location 0x10000000. arm-eabi-gdb is then used to load a RAM startup application into memory and debug it. The application code is loaded from 0x10001000 and its data and heap go into SRAM at 0x2007C000. It is assumed that the hardware has already been initialized by the stubs. By default the application will use the eCos virtual vectors mechanism to obtain services from the stubs, including diagnostic output.

ROM
: This startup type can be used for finished applications which will be programmed into internal ROM at location 0x00000000. Data and BSS will be put into internal RAM starting from 0x10000400. The remainder of SRAM at 0x10000000 and all of the SRAM at 0x2007C000 will be used for heap. The application will be self-contained with no dependencies on services provided by other software. The program expects to boot from reset into ROM at location zero. eCos startup code will perform all necessary hardware initialization. This startup type can also be used with JTAG debuggers, writing the application image into the flash and then using JTAG to debug the application. This approach makes the best use of the board's frugal memory resources, but does require the additional step of flashing the image onto the board each time.

JTAG
: This is the startup type used to build applications that are loaded via a JTAG interface. The application will be self-contained with no dependencies on services provided by other software. The program code expects to be loaded from 0x20000400 and entered at that address and its data and heap go into SRAM at 0x2007C000. eCos startup code will perform all necessary hardware initialization.

Monitors and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building the Stubrom.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then serial port 1 will be claimed for HAL diagnostics.

UART Serial Driver

The MCB1700 board uses the LPC1XXX's internal UART serial support. As well as the polled HAL diagnostic interface, there is also a `CYGPKG_IO_SERIAL_CORTEXM_LPC1XXX` package which configures the `CYGPKG_IO_SERIAL_GENERIC_16X5X` driver for use in the LPC1XXX series. Both UARTs can be supported by this driver. Note that it is not recommended to enable this driver on the port used for HAL diagnostic I/O. This driver is not active until the `CYGPKG_IO_SERIAL_DEVICES` configuration option within the generic serial driver support package `CYGPKG_IO_SERIAL` is enabled in the configuration.

Name

JTAG support — Usage

Use of JTAG for debugging

JTAG can be used to single-step and debug loaded applications, or even applications resident in ROM, including the Stubrom.

Debugging of ROM applications is only possible if using hardware breakpoints. The Cortex-M3 core of the LPC1XXX only supports two such hardware breakpoints, so they should be used sparingly. If using a GDB front-end such as Eclipse, check it has not set unnecessary extra breakpoints. Some JTAG devices give the option of whether to set hardware or software breakpoints by default. Be sure to configure your device appropriately.

When debugging via JTAG, you are likely to need to disable the default HAL idle thread action, otherwise there may be issues where the target fails to halt and the debugging session is unreliable. More details can be found in the Cortex-M architectural HAL.

Ronetix PEEDI notes

On the Ronetix PEEDI, the `peedi.mcb1700.cfg` file should be used to setup and configure the hardware to an appropriate state to load programs.

The `peedi.mcb1700.cfg` file also contains an option to define whether hardware or software breakpoints are used by default, using the `CORE0_BREAKMODE` directive in the `[TARGET]` section. The supplied version of the file defaults to software breakpoints. With this default, hardware breakpoints can still be set from GDB using the **hbreak** command. The default can be changed to hardware breakpoints, and remember to use the **reboot** command on the PEEDI command line interface, or press the reset button to make the changes take effect.

On the PEEDI, debugging can be performed either via the telnet interface or using **arm-eabi-gdb**. In the case of the latter, **arm-eabi-gdb** needs to connect to TCP port 2000 on the PEEDI's IP address. For example:

```
(gdb) target remote 111.222.333.444:2000
```

By default when the PEEDI is powered up, the target will always run the initialization section of the `peedi.mcb1700.cfg` file, and halts the target. This behaviour is repeated with the **reset** command.

If the board is reset with the '**reset**' command, or by pressing the reset button and the '**go**' command is then given, then the board will boot from ROM as normal. A similar effect can be achieved in GDB by connecting with **target remote** and immediately typing **continue** or **c**.

It is also possible for the target to always run, without initialization, after the reset button has been pressed. This mode is selected with the `CORE0_STARTUP_MODE` directive in the `[TARGET]` section of the `peedi.mcb1700.cfg` file. This conveniently allows the target to be connected to the JTAG debugger, and be able to reset it with the reset button, without being required to always type '**go**' every time.

Abatron BDI3000 notes

On the Abatron BDI3000, the `bdi3000.mcb1700.cfg` file should be used to setup and configure the hardware to an appropriate state to load programs.

The `bdi3000.mcb1700.cfg` file also contains an option to define whether hardware or software breakpoints are used by default, using the `BREAKMODE` directive in the `[TARGET]` section. The supplied version of the file defaults to software breakpoints. With this default, hardware breakpoints can still be set from GDB using the **hbreak** command. The default can be changed to hardware breakpoints, and remember to use the **boot** command on the BDI3000 command line interface.

On the BDI3000, debugging can be performed either via the telnet interface or using **arm-eabi-gdb**. In the case of the latter, **arm-eabi-gdb** needs to connect to TCP port 2001 on the BDI3000's IP address. For example:

```
(gdb) target remote 111.222.333.444:2001
```

By default when the BDI3000 is powered up, the target will always run the initialization section of the `bdi3000.mcb1700.cfg` file, and halts the target. This behaviour is repeated with the **reset** command.

If the board is reset with the '**reset**' command, or by pressing the reset button and the '**go**' command is then given, then the board will boot from ROM as normal. A similar effect can be achieved in GDB by connecting with **target remote** and immediately typing **continue** or **c**.

It is also possible for the target to always run, without initialization, after the reset button has been pressed. This mode is selected with the **reset run** command. This conveniently allows the target to be connected to the JTAG debugger, and be able to reset it with the reset button, without being required to always type '**go**' every time. Thereafter, invoking the **reset** command will repeat the previous reset style. Also in this mode, exceptions will be handled by board software, rather than causing the JTAG debugger to halt the CPU.

Suitably configured RAM applications can be loaded either via GDB, or directly via the telnet CLI. For example:

```
MCB1700> load test.srec srec
Loading test.srec , please wait ....
Loading program file passed
MCB1700> go 0x10000420
```

Consult the BDI3000 documentation for information on other formats and loading mechanisms.

Configuration of JTAG applications

JTAG applications can be loaded directly into RAM without requiring a ROM monitor. This loading can be done directly through the JTAG device, or where supported by the JTAG device, through GDB.

In order to configure the application to support this mode, some configuration settings are required. Firstly `CYGSEM_HAL_USE_ROM_MONITOR` must be disabled. Secondly the `CYGDBG_HAL_DIAG_TO_DEBUG_CHAN` option should be enabled in order to prevent HAL diagnostic output being encoded into GDB ($O) packets. Both of these settings are made automatically if the JTAG startup type is selected.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the MCB1700 board hardware, and should be read in conjunction with that specification. The MCB1700 platform HAL package complements the ARM architectural HAL and the LPC1XXX variant HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset the HAL will initialize or reinitialize many of the on-chip peripherals. There is an exception for RAM startup applications which depend on a ROM monitor for certain services.

For ROM and JTAG startup, the HAL will perform additional initialization, programming the various internal registers including the PLL, peripheral clocks and GPIO pins. The details of the early hardware startup may be found in the `src/mcb1700_misc.c` in both `hal_system_init` and `hal_platform_init`.

Memory Map

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

Internal RAM

This is located at addresses 0x10000000 and 0x2007C000 of the memory space, and is 64KiB in size, divided into two 32KiB blocks. The eCos VSR table occupies 512 bytes at 0x10000000. The virtual vector table starts at 0x10000200 and extends to 0x10000300. The top `CYGNUM_HAL_COMMON_INTERRUPTS_STACK_SIZE` bytes down from 0x10008000 are reserved for the interrupt stack.

The second 32KiB block is divided into two 16KiB parts, reflecting its organization in the hardware. The lower 16KiB, at 0x2007C000, are available for application use. The upper 16KiB, at 0x20080000, are used by the ethernet driver for packet buffers. If the ethernet device is not used, this memory may be used by applications.

Internal FLASH

This is located at address 0x00000000 of the memory space. This region is 512KiB in size. ROM applications are by default configured to run from this memory.

On-Chip Peripherals

These are accessible at locations 0x40000000 and 0xE0000000 upwards. Descriptions of the contents can be found in the LPC1XXX User Manual.

Linker Scripts

The platform linker scripts define the following symbols:

hal_vsr_table

This defines the location of the VSR table. This is set to 0x10000000 for all startup types, and space for 128 entries is reserved.

hal_virtual_vector_table

This defines the location of the virtual vector table used to communicate between an ROM monitor and an eCos application. This is allocated right after the VSR table, at 0x10000200. To permit expansion and possible addition of other tables, the linker scripts then allocate further sections from 0x10000400.

hal_interrupt_stack

This defines the location of the interrupt stack. For all startups, this is allocated to the top of internal SRAM, 0x10008000.

hal_startup_stack

This defines the location of the startup stack. For all startup types it is initially allocated at the half-way point of the interrupt stack.

Chapter 293. SAM3/4/x70 Variant HAL

Name

`CYGPKG_HAL_CORTEXM_SAM` — eCos Support for the SAM3/4 Microprocessor Family

Description

The Atmel SAM3, SAM4 and SAMX70 series of Cortex-M microcontrollers is supported by eCos with an eCos processor variant HAL and a number of device drivers supporting some of the on-chip peripherals. These include device drivers for the on-chip flash, serial, I²C, SPI, CAN, Ethernet, RTC/wallclock and watchdog devices. In addition it provides common functionality and definitions that SAM based platform ports may require, as well as definitions useful to application developers. Throughout this document this processor family will just be referred to as *SAM*, without any numerical designations.

This documentation covers the SAM functionality provided but should be read in conjunction with the specific HAL documentation for the platform port. That documentation will cover issues that are platform-specific and are not covered here, and may also describe differences that override or supersede what the SAM variant HAL provides. The areas that are specific to platform HALs and not the SAM variant HAL include:

- memory map and related configuration and setup
- Clock parameters
- Pin multiplexing and GPIO setup
- Any special handling for external interrupts, or additional interrupts
- Diagnostic I/O baud rates
- Additional diagnostic I/O devices, if any
- LED/LCD control

Name

On-chip Subsystems and Peripherals — Hardware Support

Hardware support

The SAM family contains many on-chip peripherals, many of which are compatible with devices on Atmel SAM3, SAM4, SAMX70, SAM7, SAM9 and SAMA5 parts. Where possible, the drivers are shared, and in places, package names and terminology show this.

On-chip memory

The Atmel SAM parts include on-chip SRAM, and on-chip FLASH. The RAM can vary in size from as little as 4KiB to 384KiB. The FLASH can be up to 2048KiB in size depending on model. There is also support in some models for external SRAM and Flash, which eCos may use where available.

Typically, an eCos platform HAL port will expect a GDB stub ROM monitor or RedBoot image to be programmed into the SAM on-chip ROM memory for development, and the board would boot this image from reset. The stub ROM/RedBoot provides GDB stub functionality so it is then possible to download and debug stand-alone and eCos applications via the gdb debugger using serial interfaces or other debug channels. The JTAG interface may also be used for development if a suitable JTAG adaptor is available. If RedBoot is present it may also be used to manage the on-chip and external flash memory. For production purposes, applications are programmed into external or on-chip FLASH and will be self-booting.

On-Chip FLASH

The package CYGPKG_DEVS_FLASH_AT91IAP ("FLASH memory support for Atmel AT91 IAP") provides a driver for the on-chip flash. This driver conforms to the Version 2 flash driver API, and is automatically enabled if the generic "Flash device drivers" (CYGPKG_IO_FLASH) package is included in the eCos configuration. The driver will configure itself automatically for the size and parameters of the specific SAM variant by querying the ChipID register.

Cache Handling

The SAM4 contains a small unified cache controller, the variant HAL supplies the cyg/hal/hal_cache.h header to implement cache control. This header describes the cache size and provides macros for enabling and disabling the cache as well as syncing and invalidating cache lines. The cache controller has limited functionality, so not all cache operations are supported.

The SAMX70 variants contain an instruction and data cache controller defined as part of the Cortex-M architectural specification. Support for this controller is supplied by the architecture HAL.

Serial I/O

The SAM variant HAL supports basic polled HAL diagnostic I/O over any of the on-chip serial devices. There is also a fully interrupt-driven serial device driver suitable for eCos applications for all on-chip serial devices. The serial driver consists of an eCos package: CYGPKG_IO_SERIAL_ARM_AT91 which provides support for the SAM on-chip serial devices. Using the HAL diagnostic I/O support, any of these devices can be used by the ROM monitor or RedBoot for communication with GDB. If a device is needed by the application, either directly or via the serial driver, then it cannot also be used for GDB communication using the HAL I/O support. An alternative serial port should be used instead.

The HAL defines CDL interfaces, CYGINT_HAL_CORTEXM_SAM_UART0, CYGINT_HAL_CORTEXM_SAM_UART1 and CYGINT_HAL_CORTEXM_SAM_USART0 to CYGINT_HAL_CORTEXM_SAM_USART3 for each of the available UARTs and USARTs. The platform HAL CDL should contain an **implements** directive for each such UART that is available for use on the board. This will enable use of the UART for diagnostic use.

Caution

For historical compatibility with the shared device drivers, hardware UART0 is mapped to the driver DEBUG UART, hardware USARTs 0 to 3 are mapped to driver UARTs 0 to 3 and hardware UART1 is mapped to driver UART5.

The SAM UARTs provide only TX and RX data lines while the USARTs also implement hardware flow control using RTS/CTS for those USARTs that have them connected.

Interrupts

The SAM HAL relies on the architectural HAL to provide support for the interrupts directly routed to the NVIC. The `cyg/hal/var_intr.h` header defines the vector mapping for these.

GPIO interrupts are currently not fully decoded, so all interrupts originating from a particular GPIO controller are routed to the single interrupt vector for that controller.

Pin Multiplexing and GPIO

The variant HAL provides support for packaging the configuration of a GPIO line into a single 32-bit descriptor that can then be used with macros to configure the pin and set and read its value. Details are supplied later.

RTC/Wallclock

eCos includes RTC (known in eCos as a wallclock) device drivers for the on-chip RTC in the SAM family. This is located in the package `CYGPKG_DEVICES_WALLCLOCK_ARM_AT91` ("AT91 wallclock driver").

Profiling Support

The SAM HAL contains support for **gprof**-base profiling using a sampling timer. The default timer used is Timer Counter 0. The timer used is selected by a set of `#define`s in `src/sam_misc.c` which can be changed to refer to a different timer if required. This timer is only enabled when the gprof profiling package (`CYGPKG_PROFILE_GPROF`) is included and enabled in the eCos configuration, otherwise it remains available for application use.

Clock Control

The platform HAL must provide the input oscillator frequency (`CYGHWR_HAL_CORTEXM_SAM_OSC_MAIN`) in its CDL file. This is then combined with the following options defined in this package to define the default system clocks:

CYGHWR_HAL_CORTEXM_SAM_CLOCK_PLL_DIVA

PLL input clock divider.

CYGHWR_HAL_CORTEXM_SAM_CLOCK_PLL_MULA

PLL input clock multiplier. The resulting PLL output clock will be the main oscillator frequency multiplied by this value and divided by `CYGHWR_HAL_CORTEXM_SAM_CLOCK_PLL_DIVA`.

CYGHWR_HAL_CORTEXM_SAM_CLOCK_PLLADIV2

This is an additional divide-by-two divider applied to the output from the PLL.

CYGHWR_HAL_CORTEXM_SAM_CLOCK_PRES

Processor clock prescaler. The PLL output clock is divided by this value to give the frequency of the master clock that is then distributed to the CPU and peripherals.

CYGHWR_HAL_CORTEXM_SAM_CLOCK_MDIV

> Master clock prescaler. In the SAMX70 variants, the processor clock is further divided by this prescaler to provide the master clock sent to the peripherals.

CYGHWR_HAL_CORTEXM_SAM_FLASH_WAIT

> This option defines the number of wait states applied to flash memory accesses. This will vary with the main clock frequency. The default is to set this to the maximum value.

The actual values of these clocks, in Hz, is stored in global variables `hal_sam_mainck`, `hal_sam_pllack`, `hal_sam_mclk` and `hal_sam_upllck`. The clock supplied to the SysTick timer, MCLK/8, is also assigned to `hal_cortexm_systick_clock`. These variables are set by examining the actual hardware register settings, rather than from the CDL, so they reflect settings made by any bootloader or JTAG adaptor.

Note that when changing or configuring any of these clock settings, you should consult the relevant processor datasheet as there may be both upper and lower constraints on the frequencies of some clock signals, including intermediate clocks. There are also some clocks where, while there is no strict constraint, clock stability is improved if values are chosen wisely. Finally, be aware that increasing clock speeds using this package may have an effect on platform specific properties, such as memory timings which may have to be adjusted accordingly.

Name

GPIO Support on SAM Processors — Details

Synopsis

#include <cyg/hal/hal_io.h>

cyg_uint32 pin = **CYGHWR_HAL_SAM_PIN**(*port*, *bit*, *mode*, *md*, *pupd*, *if*, *int*, *conf*);

cyg_uint32 pin = **CYGHWR_HAL_SAM_PIN_OUT**(*port*, *bit*, *md*, *pupd*);

cyg_uint32 pin = **CYGHWR_HAL_SAM_PIN_IN**(*port*, *bit*, *md*, *pupd*, *if*, *int*);

CYGHWR_HAL_SAM_PIN_SET (*pin*);

CYGHWR_HAL_SAM_GPIO_OUT (*pin*, *val*);

CYGHWR_HAL_SAM_GPIO_IN (*pin*, *val*);

Description

The SAM HAL provides a number of macros to support the encoding of the GPIO pin identity and I/O configuration into a single 32-bit descriptor. This is useful to drivers and other packages that need to configure and use different lines for different devices.

A descriptor is created with one of the 3 variants depending on how the pin is to be used. The support is implemented by the CYGHWR_HAL_SAM_PIN macro, with CYGHWR_HAL_SAM_PIN_IN and CYGHWR_HAL_SAM_PIN_OUT being shorthand helpers when direct GPIO control of a pin is required: CYGHWR_HAL_SAM_PIN_IN defines the pin as an input whose value can be accessed by the user using the macro CYGHWR_HAL_SAM_GPIO_IN (see later), CYGHWR_HAL_SAM_PIN_OUT defines the pin as an output where the user can set the pin output value with the macro CYGHWR_HAL_SAM_GPIO_OUT (see later).

The CYGHWR_HAL_SAM_PIN macro can be used when defining a pin that will be controlled by an on-chip peripheral.

Note

The HAL supplied header file var_io.h provides existing configuration definitions for the majority of the on-chip peripherals supported by eCos, thus obviating the need for the developer to provide their own pin definitions.

The macro variants take a subset of arguments from the following list:

port
This identifies the PIO controller to which the pin is attached. Ports are identified by letters from A to E.

bit
This gives the bit, or pin number, within the controller port. These are numbered from 0 to 31.

mode
This parameter indicates whether the pin is controlled by an on-chip peripheral, or is to be used as a GPIO pin under application control.

Table 293.1. Pin Mode

mode	Details
GPIOIN	The pin is to be configured as an INPUT, and after configuration the CYGHWR_HAL_SAM_GPIO_IN macro can be used to ascertain the pin state.

mode	Details
GPIOOUT	The pin is to be configured as an OUTPUT, and after configuration the CYGHWR_HAL_SAM_GPIO_OUT macro can be used to drive the pin level.
PER_A, PER_B, PER_C, PER_D	The required peripheral mapping when the pin is to be assigned to an on-chip peripheral. The multiplexing of peripheral signals is defined by the CPU variant being targeted, and is beyond the scope of this documentation. When creating pin configurations for on-chip peripherals the relevant Atmel datasheet or technical reference manual should be consulted.

md

This setting indicates whether the pin should be driven in open-drain mode (OPENDRAIN). If the pin is not to be configured as OPENDRAIN this value is unused, but for clarity can be given the setting NA.

pupd

If this is an input pin, or an output pin configured in open-drain mode (whether controlled by GPIO or a peripheral), this setting can be used to indicate whether a weak pull-up resistor (PU) is used, or a weak pull-down resistor (PD) is used. If neither are to be used, then a value of NONE can be given.

if

For input pins (GPIO or peripheral) a glitch (GLITCH) or debouncing (DEBOUNCE) filter can be configured for the pin. When no input filtering is required, or when the field is not relevant due to the other pin configuration fields, the value NONE can be specified.

int

This parameter indicates whether the pin should have an interrupt configuration defined.

Table 293.2. Interrupt Type

int	Details
EDGE_ANY	When an interrupt event should be raised on a pin edge event (rising or falling).
EDGE_RISE	An interrupt should only be raised on rising (LOW->HIGH) edge transitions.
EDGE_FALL	An interrupt should only be raised on falling (HIGH->LOW) edge transitions.
LEVEL_HIGH	Interrupts should be asserted when the pin is at a HIGH level.
LEVEL_LOW	Interrupts should be asserted when the pin is at a LOW level.
NA	This value can be used when an interrupt configuration is not required, or not applicable due to the other pin configuration parameters.

conf

This parameter provides a simple “extension” mechanism; and is treated as a 32-bit binary value that is OR-ed into the pin descriptor. Care must be taken to ensure that existing bitfields within the binary descriptor are not corrupted.

The following examples show how these macros may be used:

```
// Define port B pin 28 as being controlled by peripheral multiplex A,
// which for this pin on SAM devices is UART0 RX, without any
// pull-ups/pull-downs:
#define CYGHWR_HAL_SAM_USART1_RXD CYGHWR_HAL_SAM_PIN(A, 9,PER_A,OPENDRAIN,NONE,NONE,NA,(0))

// Define port D pin 20 as a GPIO output with a pull-down, for an
// active-low LED:
#define CYGHWR_HAL_SAM_LED_AMBER CYGHWR_HAL_SAM_PIN_OUT(D,20,NA,PD)
```

Additionally, the manifest CYGHWR_HAL_SAM_PIN_NONE may be used in place of a pin descriptor and has a value that no valid descriptor can take. It may therefore be used as a placeholder where no GPIO pin is present or to be used. This can be useful when defining pin configurations for a series of instances of a peripheral (e.g. USART ports), but where not all instances support all the same pins (e.g. hardware flow control lines).

The remaining macros all take a suitably constructed GPIO pin descriptor as an argument. The `CYGHWR_HAL_SAM_PIN_SET` macro configures the pin according to the descriptor and must be called before any other macros. `CYGHWR_HAL_SAM_GPIO_OUT` sets the output to the value of the least significant bit of the *`val`* argument. The *`val`* argument of `CYGHWR_HAL_SAM_GPIO_IN` should be a pointer to an int, which will be set to 0 if the pin input is zero, and 1 otherwise.

Further helper macros are available, and it is recommended to consult the header file `<cyg/hal/var_io.h>` (also present in the `include` subdirectory of the SAM variant HAL package within the eCos source repository), for the complete list if needed.

Name

Peripheral clock control — Details

Synopsis

#include <cyg/hal/hal_io.h>

```
cyg_uint32 CYGHWR_HAL_ATMEL_CLOCK_ENABLE(pid);
```

```
CYGHWR_HAL_ATMEL_CLOCK_DISABLE (pid);
```

Description

The HAL provides macros which may be used to enable or disable peripheral clocks. Effectively this indicates whether the peripheral is powered on (enabled) or powered down (disabled), and so may be used to ensure unused peripherals are turned off to save power. The `CYGHWR_HAL_ATMEL_CLOCK_ENABLE` macro will enforce the maximum frequency limitations for particular peripheral blocks, and will return the frequency of the clock used for the enabled peripheral. Such frequency information may be useful to device drivers if clock divider configuration is required.

It is important to remember that before a peripheral can be used, it must be enabled. It is safe to re-enable a peripheral that is already enabled, although usually a device driver will only do so once in its initialisation. eCos will automatically initialise some peripheral blocks where it needs to use the associated peripherals (such as memory controllers and some (but usually not all) PIO banks), and in eCos-supplied device drivers which are included in the eCos configuration. However this should not be relied on - it is always safest to enable the peripheral clocks anyway just in case. Finally, remember that each PIO bank must be enabled separately.

Each peripheral has a unique ID defined by the HAL, and these values are used as the *`pid`* parameter to the enable and disable macros.

Chapter 294. Atmel SAM4E-EK Platform HAL

Name

`CYGPKG_HAL_CORTEXM_SAM4E_EK` — eCos Support for the SAM4E-EK Board

Description

This document covers the configuration and usage of eCos on the Atmel SAM4E-EK evaluation kits. This board is fitted with a SAM4E16 variant of the SAM4 family of microcontrollers.

For typical eCos development it is expected that programs will be downloaded and debugged via a hardware debugger (JTAG/SWD) attached to the standard ARM 20-pin JTAG (J8) connector. Use of a hardware debugging interface avoids the requirement for a debug monitor application to be present on the platform.

Supported Hardware

The SAM variant HAL includes support for the six on-chip serial devices which are documented in the variant HAL. UART0 is connected to the external connector on the board marked DBGU/J7. There is no support for hardware flow control (RTS/CTS) lines on UART0. USART1 is connected to J5 but is configured to support RS485 rather than RS232; it needs a GPIO line to be pulled low and JP11 set to a non-default setting before RS232 is enabled. USART1 supports RTS/CTS flow control.

The SAM instantiates the SPI bus, and an AT25DF321A serial NOR flash is attached to chip select 3. The platform HAL instantiates this flash driver at the virtual address of 0x70000000.

Device drivers provide support for the two I²C (TWI) interfaces, which are instantiated by the platform HAL. These have been tested using external I²C devices, the only on-board I²C device, the QTouch controller, is not supported.

A driver is available for the CAN devices present on the chip, which are connected to external RJ12 sockets.

The board provides the 10/100 Ethernet (MII/RMII) Micrel KSZ8051MNL PHY providing support via the J20 (labelled “ETHERNET”) connector.

A driver for the AFE (Analog Front-End) Controller is available. A test program that uses the V1 potentiometer is present in the platform HAL.

Tools

The board port is intended to work with GNU tools configured for an **arm-eabi** target. The original port was done using **arm-eabi-gcc** version 4.7.3e, **arm-eabi-gdb** version 7.6.1, and **binutils** version 2.23.2.

Name

Setup — Preparing the SAM4E-EK Board for eCos Development

Overview

In a typical development environment the SAM4E-EK board is programmed via a JTAG/SWD interface. This will either be by loading applications into the on-chip SRAM, or into on-chip flash memory. The following sections deal with JTAG/SWD hardware based debugging approaches.

When debugging via JTAG, you are likely to need to disable the default HAL idle thread action, otherwise there may be issues where the target fails to halt and the debugging session is unreliable. More details can be found in the Cortex-M architectural HAL. If you are debugging via SWD this should not be necessary.

For debugging applications are loaded and then executed on the board via the debugger **arm-eabi-gdb**, or via the Eclipse IDE. The following describes setting up to use a Ronetix PEEDI debugger for use with GDB.

PEEDI

For the Ronetix PEEDI, the `peedi.sam4e_ek.cfg` file should be used to setup and configure the hardware to an appropriate state to load programs. This includes setting up the PLLs and SDRAM controller.

The `peedi.sam4e_ek.cfg` file also contains an option to define whether hardware or software breakpoints are used by default, using the `CORE0_BREAKMODE` directive in the `[PLATFORM_CortexM3]` section (NOTE: The PEEDI firmware identifies not just M3 CPUs with the `CortexM3` tag). Edit this file if you wish to use hardware break points, and remember to restart the PEEDI to make the changes take effect. However, hardware breakpoints only work below address 0x20000000, so should only be used for applications stored in flash.

On the PEEDI, debugging can be performed either via the telnet interface or using **arm-eabi-gdb** and the GDB interface. In the case of the latter, **arm-eabi-gdb** needs to connect to TCP port 2000 on the PEEDI's IP address. For example:

```
(gdb) target remote 111.222.333.444:2000
```

By default when the PEEDI is powered up, the target will always run the initialization section of the `peedi.sam4e_ek.cfg` file, and halts the target. This behaviour is repeated with the PEEDI **reset** command.

If the board is reset (either with the '**reset**', or by pressing the reset button) and the '**go**' command is then given, then the board will boot as normal and run from the contents of the flash.

Both JTAG and ROM startup types default to output of all diagnostics information via UART0 (DBGU). The default communications parameters are 115200 baud, no parity, 1 stop bit.

It is possible to arrange for diagnostics to be output via the JTAG connection and appear on the gdb console. This requires the configuration option `CYGFUN_HAL_GDB_FILEIO` in the common HAL package to be enabled. This has two sub-options, `CYGSEM_HAL_DIAG_TO_GDBFILEIO_CHAN` and `CYGSEM_HAL_DIAG_VIA_GDB_FILEIO_IMMEDIATE`, that are enabled by default when `CYGFUN_HAL_GDB_FILEIO` is enabled and both should remain enabled. In this case, when **arm-eabi-gdb** is attached to the PEEDI, the following gdb command must be issued:

```
(gdb) set hwdebug on
```

Eclipse users can do this by creating a GDB command file with the contents:

```
define preload
  set hwdebug on
end
```

This will be referenced from their Eclipse debug launch configuration. Using GDB command files is described in more detail in the "Eclipse/CDT for eCos application development" manual.

Consult the PEEDI documentation for information on other features.

Name

Configuration — Platform-specific Configuration Options

Overview

The SAM4E-EK motherboard platform HAL package is loaded automatically when eCos is configured for a suitable target, e.g. `sam4e_ek`. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The SAM4E_EK board platform HAL package supports three separate startup types:

JTAG

> This is the default startup type. It is used to build applications that are loaded via a JTAG interface. The application will be self-contained with no dependencies on services provided by other software. The program expects to be loaded from 0x20000000 and entered at 0x20000408. eCos startup code will perform all necessary hardware initialization.

ROM

> This startup type can be used for finished applications which will be programmed into internal flash at location 0x00400000. Data and BSS will be put into SRAM starting from 0x20000000. The application will be self-contained with no dependencies on services provided by other software. The program expects to boot from reset with flash mapped at location zero. It will then transfer control to the 0x00400000 region. eCos startup code will perform all necessary hardware initialization.

RAM

> This startup type can be used when a board has a GDB stub ROM resident in internal flash. It enables eCos application development without the necessity for a JTAG debugger, providing arm-eabi-gdb connectivity via UART0 (DBGU). The stub is programmed into the on-chip flash memory at 0x00400000 and uses SRAM at location 0x20000000. arm-eabi-gdb is then used to load RAM startup applications into memory from 0x20001000 and debug them. It is assumed that the hardware has already been initialized by the stub ROM. By default the application will use the eCos virtual vectors mechanism to obtain services from the ROM, including diagnostic output.

RedBoot and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building GDB Stubs.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then the serial port will be claimed for HAL diagnostic output.

UART Serial Driver

The SAM4E_EK board uses the SAM's internal UART serial support. The HAL diagnostic interface, used for both polled diagnostic output and GDB stub communication, is only expected to be available to be used on the UART0 (DBGU) port. This is because only UART0 is actually routed to an external RS232 connector.

As well as the polled HAL diagnostic interface, there is also a CYGPKG_IO_SERIAL_ARM_AT91 package which contains all the code necessary to support interrupt-driven operation with greater functionality.

It is not recommended to use the interrupt-driven serial driver with a port at the same time as using that port for HAL diagnostic I/O.

This driver is not active until the CYGPKG_IO_SERIAL_DEVICES configuration option within the generic serial driver support package CYGPKG_IO_SERIAL is enabled in the configuration. By default this will only enable support in the driver for the UART0 port (the same as the HAL diagnostic interface), but the default configuration can be modified to enable support for other serial ports.

USART1 is disabled by default since it requires pins and clocks to be configured and PA23 to be pulled low and a jumper to be changed to support RS232. The option CYGHWR_HAL_CORTEXM_SAM4E_EK_USART1 enables the hardware to be configured; the user must change JP11 to the 2-3 position in order to enable access to USART1 via J5.

SPI Driver

An SPI bus driver is available for the SAM in the package "Atmel AT91 SPI device driver" (CYGPKG_DEVS_SPI_ARM_AT91).

The only SPI device instantiated by default is for an external AT25DF321A NOR flash.

Consult the generic SPI driver API documentation in the eCosPro Reference Manual for further details on SPI support in eCosPro, along with the configuration options in the AT91 SPI device driver.

I²C Driver

Support for SAM I²C (TWI) busses is provided by the "Atmel TWI (I2C) device driver" package (CYGPKG_DEVS_I2C_ATMEL_TWI). The SAM variant HAL causes the two buses to be instantiated. These have been tested using external I²C devices. The only on-board I²C device, the QTouch controller, is not supported.

Ethernet Driver

The AT91SAM4E-EK board uses the AT91SAM4E's internal EMAC ethernet device attached to an external Micrel KSZ8051MNL PHY. The CYGPKG_DEVS_ETH_ARM_AT91 package contains all the code necessary to support this device and the platform HAL package contains definitions that customize the driver to the board. This driver is not active until the generic Ethernet support package, CYGPKG_IO_ETH_DRIVERS, is included in the configuration.

Both the standard and direct (lwIP only) device drivers are supported. The standard driver is enabled by default; the direct driver can be enabled by setting CYGOPT_IO_ETH_DRIVERS_LWIP_DRIVER_DIRECT option.

CAN Driver

The SAM4E has dual CAN devices for CAN support. Device support is via the Atmel SAM CAN Driver (CYGPKG_DEVS_CAN_SAM) package.

The board has two external CAN sockets, J13 and J14, which are RJ12 female sockets.

Consult the generic Chapter 90, *CAN Support* documentation for further details on use of the CAN API, CAN configuration and device drivers.

Flash Driver

The SAM4E's on-chip Flash may be programmed and managed using the Flash driver located in the "FLASH memory support for Atmel AT91 IAP" (CYGPKG_DEVS_FLASH_AT91IAP) package. This driver is enabled automatically if the generic "Flash device drivers" (CYGPKG_IO_FLASH) package is included in the eCos configuration. The driver will configure itself automatically for the size and parameters of the specific SAM variant present on the SAM4E-EK board.

A number of aspects of Flash driver behaviour can be configured within that driver, such as program/erase parallelism and program burst size. Consult the driver and Part XVIII, “NOR Flash Support” documentation for more details.

ADC Driver

ADC support is provided by the Atmel AFEC ADC Driver (CYGPKG_DEVS_ADC_ATMEL_AFEC) package. By default AFEC0 AD4 is connected to BNC C2 (jumper JP40 pins 1-2 connected). The jumper JP40 can be changed to select AFEC1 AD0 as the ADC input channel (NOTE: For JP40 2-3 the link J37-6 *MUST* be open). The CYGPKG_HAL_CORTEXM_SAM4E_EK_AFE_CN2_JP40 must be set to reflect the state of the jumper.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the SAM4E-EK board hardware, and should be read in conjunction with that specification. The SAM4E-EK platform HAL package complements the Cortex-M architectural HAL and the SAM variant HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset the HAL will initialize or reinitialize many of the on-chip peripherals. There is an exception for RAM startup applications which depend on a ROM monitor for certain services.

For ROM and JTAG startup, the HAL will perform additional initialization, programming the various internal registers including the PLL, peripheral clocks and pin multiplexing. The details of the early hardware startup may be found in the `src/sam4e_ek_misc.c` in both `hal_system_init()` and `hal_platform_init()`.

Memory Map

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory areas are as follows:

Internal SRAM

> This is located at address 0x20000000 of the memory space, and is 128KiB in size. The eCos VSR table occupies the bottom 256 bytes, with the virtual vector table starting at 0x20000100 and extending to 0x20000200. The top `CYGNUM_HAL_COMMON_INTERRUPTS_STACK_SIZE` bytes are reserved for the interrupt stack. The remainder of internal RAM is available for use by applications.

Internal FLASH

> This is located at address 0x004000000 of the memory space and will be mapped to 0x00000000 at reset. This region is 1024KiB in size. ROM applications are by default configured to run from this memory.

On-Chip Peripherals

> These are accessible at locations 0x40000000 and 0xE0000000 upwards. Descriptions of the contents can be found in the SAM4E User Manual.

Linker Scripts

The platform linker scripts define the following symbols:

hal_vsr_table
: This defines the location of the VSR table. This is set to 0x20000000 for all startup types, and space for 64 entries is reserved.

hal_virtual_vector_table
: This defines the location of the virtual vector table used to communicate between a ROM monitor and an eCos application. This is allocated right after the VSR table, at 0x20000100.

hal_interrupt_stack
: This defines the location of the interrupt stack. For all startup types this is allocated to the top of internal SRAM, 0x20020000.

hal_startup_stack
: This defines the location of the startup stack. For all startup types it is initially allocated at the half-way point of the interrupt stack.

Diagnostic LEDs

Three LEDs are fitted on the board for diagnostic purposes: D2 (blue), D3 (amber) and D4 (green).

The platform HAL header file at `<cyg/hal/plf_io.h>` defines the following convenience function to allow the LEDs to be set:

```
       extern void hal_sam_led(int val);
```

The lowest 3 bits of the argument `val` correspond to each of the 3 LEDs (with D2 as the least significant bit).

The variant HAL signals progress through clock initialization on the LEDs, leaving them all illuminated if initialization is successful. Following this, the LEDs are available for application use.

Real-time characterization

The tm_basic kernel test gives statistics gathered about the real-time characterization and performance of the kernel. The sample output is shown here for information. The test was built for JTAG startup.

Example 294.1. sam4e_ek Real-time characterization

```
             Startup, main thrd : stack used   352 size  1536
             Startup : Idlethread stack used    84 size  1280

eCos Kernel Timings
Notes: all times are in microseconds (.000001) unless otherwise stated

Reading the hardware clock takes 0 'ticks' overhead
... this value will be factored out of all other measurements
Clock interrupt took    9.00 microseconds (9 raw clock ticks)

Testing parameters:
   Clock samples:            32
   Threads:                   6
   Thread switches:         128
   Mutexes:                  32
   Mailboxes:                32
   Semaphores:               32
   Scheduler operations:    128
   Counters:                 32
   Flags:                    32
   Alarms:                   32
   Stack Size:             1088


                                 Confidence
     Ave     Min     Max     Var  Ave  Min  Function
  ======  ======  ======  ====== ========== ========
    7.50    7.00    8.00    0.50  100%  50% Create thread
    1.67    1.00    2.00    0.44   66%  33% Yield thread [all suspended]
    1.67    1.00    2.00    0.44   66%  33% Suspend [suspended] thread
    2.00    2.00    2.00    0.00  100% 100% Resume thread
    2.67    2.00    3.00    0.44   66%  33% Set priority
    0.33    0.00    1.00    0.44   66%  66% Get priority
    4.83    4.00    5.00    0.28   83%  16% Kill [suspended] thread
    2.00    2.00    2.00    0.00  100% 100% Yield [no other] thread
    2.33    2.00    3.00    0.44   66%  66% Resume [suspended low prio] thread
    2.00    2.00    2.00    0.00  100% 100% Resume [runnable low prio] thread
    2.33    2.00    3.00    0.44   66%  66% Suspend [runnable] thread
    1.17    1.00    2.00    0.28   83%  83% Yield [only low prio] thread
    2.00    2.00    2.00    0.00  100% 100% Suspend [runnable->not runnable]
    5.00    5.00    5.00    0.00  100% 100% Kill [runnable] thread
    4.00    4.00    4.00    0.00  100% 100% Destroy [dead] thread
    7.67    7.00    8.00    0.44   66%  33% Destroy [runnable] thread
    8.50    8.00    9.00    0.50  100%  50% Resume [high priority] thread
    2.87    2.00    5.00    0.26   84%  14% Thread switch

    0.34    0.00    1.00    0.45   65%  65% Scheduler lock
    1.35    1.00    2.00    0.46   64%  64% Scheduler unlock [0 threads]
    1.48    1.00    2.00    0.50   51%  51% Scheduler unlock [1 suspended]
    1.42    1.00    2.00    0.49   57%  57% Scheduler unlock [many suspended]
    1.18    1.00    2.00    0.29   82%  82% Scheduler unlock [many low prio]

    0.41    0.00    1.00    0.48   59%  59% Init mutex
    1.94    1.00    2.00    0.12   93%   6% Lock [unlocked] mutex
    2.28    2.00    3.00    0.40   71%  71% Unlock [locked] mutex
    1.81    1.00    2.00    0.31   81%  18% Trylock [unlocked] mutex
    1.00    1.00    1.00    0.00  100% 100% Trylock [locked] mutex
    0.56    0.00    1.00    0.49   56%  43% Destroy mutex
    9.13    9.00   10.00    0.22   87%  87% Unlock/Lock mutex

    0.72    0.00    1.00    0.40   71%  28% Create mbox
    0.31    0.00    1.00    0.43   68%  68% Peek [empty] mbox
    2.16    2.00    3.00    0.26   84%  84% Put [first] mbox
    0.28    0.00    1.00    0.40   71%  71% Peek [1 msg] mbox
    2.06    2.00    3.00    0.12   93%  93% Put [second] mbox
    0.31    0.00    1.00    0.43   68%  68% Peek [2 msgs] mbox
    2.03    2.00    3.00    0.06   96%  96% Get [first] mbox
    2.03    2.00    3.00    0.06   96%  96% Get [second] mbox
    1.75    1.00    2.00    0.38   75%  25% Tryput [first] mbox
    1.69    1.00    2.00    0.43   68%  31% Peek item [non-empty] mbox
    1.84    1.00    2.00    0.26   84%  15% Tryget [non-empty] mbox
    2.00    2.00    2.00    0.00  100% 100% Peek item [empty] mbox
    2.00    2.00    2.00    0.00  100% 100% Tryget [empty] mbox
    0.38    0.00    1.00    0.47   62%  62% Waiting to get mbox
    0.38    0.00    1.00    0.47   62%  62% Waiting to put mbox
    0.81    0.00    1.00    0.31   81%  18% Delete mbox
    6.19    6.00    7.00    0.30   81%  81% Put/Get mbox

    0.47    0.00    1.00    0.50   53%  53% Init semaphore
    1.50    1.00    2.00    0.50  100%  50% Post [0] semaphore
    1.78    1.00    2.00    0.34   78%  21% Wait [1] semaphore
    1.53    1.00    2.00    0.50   53%  46% Trywait [0] semaphore
    1.59    1.00    2.00    0.48   59%  40% Trywait [1] semaphore
    0.56    0.00    1.00    0.49   56%  43% Peek semaphore
    0.56    0.00    1.00    0.49   56%  43% Destroy semaphore
    5.41    5.00    6.00    0.48   59%  59% Post/Wait semaphore

    0.81    0.00    1.00    0.31   81%  18% Create counter
    0.56    0.00    1.00    0.49   56%  43% Get counter value
    0.34    0.00    1.00    0.45   65%  65% Set counter value
    2.00    2.00    2.00    0.00  100% 100% Tick counter
    0.47    0.00    1.00    0.50   53%  53% Delete counter

    0.41    0.00    1.00    0.48   59%  59% Init flag
    1.84    1.00    2.00    0.26   84%  15% Destroy flag
    1.63    1.00    2.00    0.47   62%  37% Mask bits in flag
    1.81    1.00    2.00    0.31   81%  18% Set bits in flag [no waiters]
    2.59    2.00    3.00    0.48   59%  40% Wait for flag [AND]
    2.50    2.00    3.00    0.50  100%  50% Wait for flag [OR]
    2.59    2.00    3.00    0.48   59%  40% Wait for flag [AND/CLR]
    2.50    2.00    3.00    0.50  100%  50% Wait for flag [OR/CLR]
    0.31    0.00    1.00    0.43   68%  68% Peek on flag

    1.19    1.00    2.00    0.30   81%  81% Create alarm
    2.56    2.00    3.00    0.49   56%  43% Initialize alarm
    1.56    1.00    2.00    0.49   56%  43% Disable alarm
    3.00    3.00    3.00    0.00  100% 100% Enable alarm
    1.81    1.00    2.00    0.31   81%  18% Delete alarm
    2.22    2.00    3.00    0.34   78%  78% Tick counter [1 alarm]
   11.59   11.00   12.00    0.48   59%  40% Tick counter [many alarms]
    3.94    3.00    4.00    0.12   93%   6% Tick & fire counter [1 alarm]
   65.00   65.00   65.00    0.00  100% 100% Tick & fire counters [>1 together]
   13.31   13.00   14.00    0.43   68%  68% Tick & fire counters [>1 separately]
    8.00    8.00    8.00    0.00  100% 100% Alarm latency [0 threads]
    7.34    7.00    8.00    0.45   65%  65% Alarm latency [2 threads]
    7.57    7.00    8.00    0.49   57%  42% Alarm latency [many threads]
   12.02   12.00   14.00    0.03   99%  99% Alarm -> thread resume latency

    0.00    0.00    0.00    0.00            Clock/interrupt latency

    3.68    3.00    4.00    0.00            Clock DSR latency

     190     180     212                    Worker thread stack used (stack size 1088)
            All done, main thrd : stack used   804 size  1536
            All done : Idlethread stack used   172 size  1280

Timing complete - 29810 ms total

PASS:<Basic timing OK>
EXIT:<done>
```

Chapter 295. Atmel SAMX70-EK Platform HAL

Name

`CYGPKG_HAL_CORTEXM_SAMX70_EK` — eCos Support for the SAMX70-EK Board

Description

This document covers the configuration and usage of eCos on the Atmel SAMX70 evaluation kits. This includes the SAME70 Xplained and the SAMV71 Xplained Ultra on which this support has been tested. Boards containing other devices in the SAM E70, S70 and V70 family should be supportable with minimal effort.

For typical eCos development it is expected that programs will be downloaded and debugged via a hardware debugger (JTAG/SWD) attached to either the standard ARM 20-pin JTAG connector or via the on-board EDBG USB socket. Use of a hardware debugging interface avoids the requirement for a debug monitor application to be present on the platform.

Supported Hardware

The SAM variant HAL includes support for the on-chip serial devices which are documented in the variant HAL. USART0 is connected to the EXT1 external connector. USART1 is connected to the EXT2 external connector and the EDBG USB socket where it is available as a CDC/ACM interface. There is no support for hardware flow control in either of these devices. USART1 is configured as the default diagnostics console.

Device drivers provide support for the two I²C (TWI) interfaces, which are instantiated by the platform HAL. These have been tested using external I²C devices.

A driver is available for the CAN devices present on the chip, which are available on pins on the various boards.

A driver for the AFE (Analog Front-End) Controller is available. A test program that uses an external potentiometer is present in the platform HAL.

Tools

The board port is intended to work with GNU tools configured for an **arm-eabi** target. The original port was done using **arm-eabi-gcc** version 4.7.3e, **arm-eabi-gdb** version 7.6.1, and **binutils** version 2.23.2.

Name

Setup — Preparing the SAMX70-EK Board for eCos Development

Overview

In a typical development environment the SAMX70-EK board is programmed via a JTAG/SWD interface. This will either be by loading applications into the on-chip SRAM, or into on-chip flash memory. The following sections deal with JTAG/SWD hardware based debugging approaches.

When debugging via JTAG, you may need to disable the default HAL idle thread action, otherwise there may be issues where the target fails to halt and the debugging session is unreliable. More details can be found in the Cortex-M architectural HAL. If you are debugging via SWD this should not be necessary.

For debugging applications are loaded and then executed on the board via the debugger **arm-eabi-gdb**, or via the Eclipse IDE. The following describes setting up to use a Ronetix PEEDI debugger for use with GDB.

PEEDI

For the Ronetix PEEDI, the `peedi.same70xpld.cfg` or `peedi.samv71xult.cfg` file should be used to setup and configure the hardware to an appropriate state to load programs. These files only perform basic initialization by default, leaving application code to initialize PLLs and other clocks. However, these files also contain an alternate initialization section that will initialize the clocks and SDRAM from JTAG.

The configuration files also contains an option to define whether hardware or software breakpoints are used by default, using the `CORE0_BREAKMODE` directive in the `[PLATFORM_CortexM3]` section (NOTE: The PEEDI firmware identifies not just M3 CPUs with the `CortexM3` tag). Edit this file if you wish to use hardware break points, and remember to restart the PEEDI to make the changes take effect. However, hardware breakpoints only work below address 0x20000000, so should only be used for applications stored in flash.

On the PEEDI, debugging can be performed either via the telnet interface or using **arm-eabi-gdb** and the GDB interface. In the case of the latter, **arm-eabi-gdb** needs to connect to TCP port 2000 on the PEEDI's IP address. For example:

```
(gdb) target remote 111.222.333.444:2000
```

By default when the PEEDI is powered up, the target will always run the initialization section of the configuration file, and halts the target. This behaviour is repeated with the PEEDI **reset** command.

If the board is reset (either with the '**reset**', or by pressing the reset button) and the '**go**' command is then given, then the board will boot as normal and run from the contents of the flash.

The JTAG, JTAGEXT, ROM and ROMEXT startup types default to output of all diagnostics information via USART1. The default communications parameters are 115200 baud, no parity, 1 stop bit. It is recommended that USART1 be accessed via the EDBG port.

It is possible to arrange for diagnostics to be output via the JTAG connection and appear on the gdb console. This requires the configuration option `CYGFUN_HAL_GDB_FILEIO` in the common HAL package to be enabled. This has two sub-options, `CYGSEM_HAL_DIAG_TO_GDBFILEIO_CHAN` and `CYGSEM_HAL_DIAG_VIA_GDB_FILEIO_IMMEDIATE`, that are enabled by default when `CYGFUN_HAL_GDB_FILEIO` is enabled and both should remain enabled. In this case, when **arm-eabi-gdb** is attached to the PEEDI, the following gdb command must be issued:

```
(gdb) set hwdebug on
```

Eclipse users can do this by creating a GDB command file with the contents:

```
define preload
  set hwdebug on
end
```

This will be referenced from their Eclipse debug launch configuration. Using GDB command files is described in more detail in the "Eclipse/CDT for eCos application development" manual.

Consult the PEEDI documentation for information on other features.

Name

Configuration — Platform-specific Configuration Options

Overview

The SAMX70-EK motherboard platform HAL package is loaded automatically when eCos is configured for a suitable target, e.g. `samv71_ek` or `same70_ek`. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The SAMX70_EK board platform HAL package supports four separate startup types:

JTAG

> This is the default startup type. It is used to build applications that are loaded via a JTAG interface into the on-chip SRAM. The application will be self-contained with no dependencies on services provided by other software. The program expects to be loaded from 0x20400000 and entered at 0x20400408. eCos startup code will perform all necessary hardware initialization.

JTAGEXT

> This startup type is used to build applications that are loaded via a JTAG interface, or a boot loader, into the off-chip SDRAM. The application will be self-contained with no dependencies on services provided by other software. The program expects to be loaded from 0x70000000 and entered at 0x70000008. eCos startup code will perform all necessary hardware initialization other than the SDRAM controller, since that is initialised via the H/W (JTAG/SWD) debugger or a flash based boot loader.

ROM

> This startup type can be used for finished applications which will be programmed into internal flash at location 0x00400000 using the on-chip SRAM as the main application RAM area. Data and BSS will be put into SRAM starting from 0x20400000. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization. To make ROM code run from reset, it is necessary to set the GPNVM1 bit to 1; this can be done via the Atmel SAM-BA utility.

ROMEXT

> This startup type can be used for finished applications which will be programmed into internal flash at location 0x00400000 using the off-chip SDRAM as the main application RAM area. Data and BSS will be put into SDRAM starting from 0x70000000. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization. To make ROM code run from reset, it is necessary to set the GPNVM1 bit to 1; this can be done via the Atmel SAM-BA utility.

The naming of the STARTUP types is for backwards compability with earlier releases that did not support the off-chip SDRAM.

UART Serial Driver

The SAMX70_EK board uses the SAM's internal UART serial support. The HAL diagnostic interface, used for both polled diagnostic output and GDB stub communication, is only expected to be available to be used on the UART0 (DBGU) port. This is because only UART0 is actually routed to an external RS232 connector.

As well as the polled HAL diagnostic interface, there is also a `CYGPKG_IO_SERIAL_ARM_AT91` package which contains all the code necessary to support interrupt-driven operation with greater functionality.

It is not recommended to use the interrupt-driven serial driver with a port at the same time as using that port for HAL diagnostic I/O.

This driver is not active until the CYGPKG_IO_SERIAL_DEVICES configuration option within the generic serial driver support package CYGPKG_IO_SERIAL is enabled in the configuration. By default this will only enable support in the driver for the UART0 port (the same as the HAL diagnostic interface), but the default configuration can be modified to enable support for other serial ports.

SPI Driver

An SPI bus driver is available in the package "Atmel AT91 SPI device driver" (CYGPKG_DEVS_SPI_ARM_AT91).

The only SPI device instantiated by default is for an external AT25DF321A NOR flash used for testing.

Consult the generic SPI driver API documentation in the eCosPro Reference Manual for further details on SPI support in eCosPro, along with the configuration options in the AT91 SPI device driver.

I²C Driver

Support for SAM I²C (TWI) busses is provided by the "Atmel TWI (I2C) device driver" package (CYGPKG_DEVS_I2C_ATMEL_TWI). The SAM variant HAL causes the two buses to be instantiated. These have been tested using external I²C devices.

CAN Driver

The SAMX70-EK has dual MCAN devices for CAN and CAN-FD support. Device support is via the Atmel MCAN CAN Driver (CYGPKG_DEVS_CAN_MCAN) package.

The SAMV71XULT board has an on-board CAN transceiver which is connected to MCAN1 and to CANH and CANL pins on the board. The SAME70XPLD board has no on-board transceiver and the CANTX and CANRX pins for both MCAN0 and MCAN1 are available on the Arduino connectors.

Consult the generic Chapter 90, *CAN Support* documentation for further details on use of the CAN API, CAN configuration and device drivers.

Flash Driver

The on-chip Flash may be programmed and managed using the Flash driver located in the "FLASH memory support for Atmel AT91 IAP" (CYGPKG_DEVS_FLASH_AT91IAP) package. This driver is enabled automatically if the generic "Flash device drivers" (CYGPKG_IO_FLASH) package is included in the eCos configuration. The driver will configure itself automatically for the size and parameters of the specific SAM variant present on the SAMX70-EK board.

Consult the driver and Part XVIII, “NOR Flash Support” documentation for more details.

ADC Driver

ADC support is provided by the Atmel AFEC ADC Driver (CYGPKG_DEVS_ADC_ATMEL_AFEC) package. Some ADC lines are available on the EXT1 and EXT2 headers.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the SAMX70-EK board hardware, and should be read in conjunction with that specification. The SAMX70-EK platform HAL package complements the Cortex-M architectural HAL and the SAM variant HAL. It provides functionality which is specific to the target board.

Startup

For ROM and JTAG startup, the HAL will perform initialization, programming the various internal registers including the PLL, peripheral clocks and pin multiplexing. The details of the early hardware startup may be found in the `src/samx70_ek_misc.c` in both `hal_system_init()` and `hal_platform_init()`.

Memory Map

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory areas are as follows:

Internal SRAM

> This is located at address 0x20400000 of the memory space, and is 384KiB in size. The eCos VSR table occupies the bottom 256 bytes, with the virtual vector table starting at 0x20000200 and extending to 0x20000300. The top of SRAM is reserved for the MCAN message buffer memory, whose size depends on the MCAN configuration, and below those `CYGNUM_HAL_COMMON_INTERRUPTS_STACK_SIZE` bytes are reserved for the interrupt stack. The remainder of internal RAM is available for use by applications.

Internal FLASH

> This is located at address 0x004000000 of the memory space. This region is 2048KiB in size. ROM applications are by default configured to run from this memory.

On-Chip Peripherals

> These are accessible at locations 0x40000000 and 0xE0000000 upwards. Descriptions of the contents can be found in the SAMX70 User Manual.

Linker Scripts

The platform linker scripts define the following symbols:

hal_vsr_table

> This defines the location of the VSR table. This is set to 0x20000000 for all startup types, and space for 128 entries is reserved.

hal_virtual_vector_table

> This defines the location of the virtual vector table used to communicate between a ROM monitor and an eCos application. This is allocated right after the VSR table, at 0x20400200.

hal_mcan_ram1

> This defines the location of the message RAM for MCAN1. It is allocated at the top of SRAM, and its size is specified by the MCAN package if present, or will be zero if it is not.

hal_mcan_ram0

> This defines the location of the message RAM for MCAN0. It is allocated just before the RAM for MCAN1, and its size is specified by the MCAN package if present, or will be zero if it is not.

hal_interrupt_stack — This defines the location of the interrupt stack. For all startup types this is allocated to the top of internal SRAM just before the MCAN message RAM.

hal_startup_stack — This defines the location of the startup stack. For all startup types it is initially allocated at the half-way point of the interrupt stack.

Real-time characterization

The tm_basic kernel test gives statistics gathered about the real-time characterization and performance of the kernel. The sample output is shown here for information. The test was built for JTAG startup on a SAMV71-XULT board.

Example 295.1. samv71-XULT Real-time characterization

```
Configured
Testing parameters:
   Clock samples:            32
   Threads:                  19
   Thread switches:         128
   Mutexes:                 873
   Mailboxes:               254
   Semaphores:             1528
   Scheduler operations:    128
   Counters:                509
   Flags:                  1018
   Alarms:                  436
   Stack Size:             1088

             Startup, main thrd : stack used   108 size  1536
             Startup : Idlethread stack used    76 size  1280

eCos Kernel Timings
Notes: all times are in microseconds (.000001) unless otherwise stated

Reading the hardware clock takes 0 'ticks' overhead
... this value will be factored out of all other measurements
Clock interrupt took    1.03 microseconds (1 raw clock ticks)

Testing parameters:
   Clock samples:            32
   Threads:                  19
   Thread switches:         128
   Mutexes:                  32
   Mailboxes:                32
   Semaphores:               32
   Scheduler operations:    128
   Counters:                 32
   Flags:                    32
   Alarms:                   32
   Stack Size:             1088


                                 Confidence
     Ave     Min     Max     Var  Ave  Min  Function
  ======  ======  ======  ====== ========== ========
INFO:<Ctrl-C disabled until test completion>
    1.16    1.00    2.00    0.27   84%  84% Create thread
    0.26    0.00    1.00    0.39   73%  73% Yield thread [all suspended]
    0.21    0.00    1.00    0.33   78%  78% Suspend [suspended] thread
    0.32    0.00    1.00    0.43   68%  68% Resume thread
    0.42    0.00    1.00    0.49   57%  57% Set priority
    0.11    0.00    1.00    0.19   89%  89% Get priority
    0.68    0.00    1.00    0.43   68%  31% Kill [suspended] thread
    0.32    0.00    1.00    0.43   68%  68% Yield [no other] thread
    0.42    0.00    1.00    0.49   57%  57% Resume [suspended low prio] thread
    0.32    0.00    1.00    0.43   68%  68% Resume [runnable low prio] thread
    0.42    0.00    1.00    0.49   57%  57% Suspend [runnable] thread
    0.32    0.00    1.00    0.43   68%  68% Yield [only low prio] thread
    0.32    0.00    1.00    0.43   68%  68% Suspend [runnable->not runnable]
    0.68    0.00    1.00    0.43   68%  31% Kill [runnable] thread
    0.63    0.00    1.00    0.47   63%  36% Destroy [dead] thread
    1.37    1.00    2.00    0.47   63%  63% Destroy [runnable] thread
    1.53    1.00    2.00    0.50   52%  47% Resume [high priority] thread
    0.52    0.00    1.00    0.50   51%  48% Thread switch

    0.06    0.00    1.00    0.12   93%  93% Scheduler lock
    0.20    0.00    1.00    0.32   79%  79% Scheduler unlock [0 threads]
    0.21    0.00    1.00    0.33   78%  78% Scheduler unlock [1 suspended]
    0.20    0.00    1.00    0.32   79%  79% Scheduler unlock [many suspended]
    0.22    0.00    1.00    0.34   78%  78% Scheduler unlock [many low prio]

    0.09    0.00    1.00    0.17   90%  90% Init mutex
    0.28    0.00    1.00    0.40   71%  71% Lock [unlocked] mutex
    0.34    0.00    1.00    0.45   65%  65% Unlock [locked] mutex
    0.28    0.00    1.00    0.40   71%  71% Trylock [unlocked] mutex
    0.28    0.00    1.00    0.40   71%  71% Trylock [locked] mutex
    0.06    0.00    1.00    0.12   93%  93% Destroy mutex
    3.00    3.00    3.00    0.00  100% 100% Unlock/Lock mutex

    0.22    0.00    1.00    0.34   78%  78% Create mbox
    0.06    0.00    1.00    0.12   93%  93% Peek [empty] mbox
    0.25    0.00    1.00    0.38   75%  75% Put [first] mbox
    0.09    0.00    1.00    0.17   90%  90% Peek [1 msg] mbox
    0.38    0.00    1.00    0.47   62%  62% Put [second] mbox
    0.06    0.00    1.00    0.12   93%  93% Peek [2 msgs] mbox
    0.38    0.00    1.00    0.47   62%  62% Get [first] mbox
    0.34    0.00    1.00    0.45   65%  65% Get [second] mbox
    0.25    0.00    1.00    0.38   75%  75% Tryput [first] mbox
    0.19    0.00    1.00    0.30   81%  81% Peek item [non-empty] mbox
    0.31    0.00    1.00    0.43   68%  68% Tryget [non-empty] mbox
    0.31    0.00    1.00    0.43   68%  68% Peek item [empty] mbox
    0.25    0.00    1.00    0.38   75%  75% Tryget [empty] mbox
    0.09    0.00    1.00    0.17   90%  90% Waiting to get mbox
    0.00    0.00    0.00    0.00  100% 100% Waiting to put mbox
    0.16    0.00    1.00    0.26   84%  84% Delete mbox
    1.09    1.00    2.00    0.17   90%  90% Put/Get mbox

    0.06    0.00    1.00    0.12   93%  93% Init semaphore
    0.25    0.00    1.00    0.38   75%  75% Post [0] semaphore
    0.25    0.00    1.00    0.38   75%  75% Wait [1] semaphore
    0.22    0.00    1.00    0.34   78%  78% Trywait [0] semaphore
    0.31    0.00    1.00    0.43   68%  68% Trywait [1] semaphore
    0.03    0.00    1.00    0.06   96%  96% Peek semaphore
    0.09    0.00    1.00    0.17   90%  90% Destroy semaphore
    1.75    1.00    2.00    0.38   75%  25% Post/Wait semaphore

    0.16    0.00    1.00    0.26   84%  84% Create counter
    0.09    0.00    1.00    0.17   90%  90% Get counter value
    0.06    0.00    1.00    0.12   93%  93% Set counter value
    0.25    0.00    1.00    0.38   75%  75% Tick counter
    0.13    0.00    1.00    0.22   87%  87% Delete counter

    0.09    0.00    1.00    0.17   90%  90% Init flag
    0.28    0.00    1.00    0.40   71%  71% Destroy flag
    0.19    0.00    1.00    0.30   81%  81% Mask bits in flag
    0.31    0.00    1.00    0.43   68%  68% Set bits in flag [no waiters]
    0.41    0.00    1.00    0.48   59%  59% Wait for flag [AND]
    0.38    0.00    1.00    0.47   62%  62% Wait for flag [OR]
    0.41    0.00    1.00    0.48   59%  59% Wait for flag [AND/CLR]
    0.47    0.00    1.00    0.50   53%  53% Wait for flag [OR/CLR]
    0.09    0.00    1.00    0.17   90%  90% Peek on flag
    0.31    0.00    1.00    0.43   68%  68% Create alarm
    0.44    0.00    1.00    0.49   56%  56% Initialize alarm
    0.25    0.00    1.00    0.38   75%  75% Disable alarm
    0.47    0.00    1.00    0.50   53%  53% Enable alarm
    0.28    0.00    1.00    0.40   71%  71% Delete alarm
    0.50    0.00    1.00    0.50  100%  50% Tick counter [1 alarm]
    1.38    1.00    2.00    0.47   62%  62% Tick counter [many alarms]
    0.53    0.00    1.00    0.50   53%  46% Tick & fire counter [1 alarm]
    7.81    7.00    8.00    0.31   81%  18% Tick & fire counters [>1 together]
    1.59    1.00    2.00    0.48   59%  40% Tick & fire counters [>1 separately]
    1.00    1.00    1.00    0.00  100% 100% Alarm latency [0 threads]
    1.00    1.00    1.00    0.00  100% 100% Alarm latency [2 threads]
    1.00    1.00    1.00    0.00  100% 100% Alarm latency [many threads]
    2.00    2.00    2.00    0.00  100% 100% Alarm -> thread resume latency

     192     168     212                    Worker thread stack used (stack size 1088)
            All done, main thrd : stack used   796 size  1536
            All done : Idlethread stack used   164 size  1280

Timing complete - 27860 ms total

PASS:<Basic timing OK>
EXIT:<done>
```

Chapter 296. STM32 Variant HAL

Name

CYGPKG_HAL_CORTEXM_STM32 — eCos Support for the STM32 Microprocessor Family

Description

The ST STM32Fxxxx series of Cortex-M microcontrollers is supported by eCos with an eCos processor variant HAL and a number of device drivers supporting most of the on-chip peripherals. These include device drivers for the on-chip flash, serial, I²C, SPI, MMC/SD, Ethernet, CAN, USB OTG HS and FS, ADC, RTC/wallclock and watchdog devices. In addition it provides common functionality and definitions that STM32 based platform ports may require, as well as definitions useful to application developers.

This documentation covers the STM32 functionality provided but should be read in conjunction with the specific HAL documentation for the platform port. That documentation will cover issues that are platform-specific and are not covered here, and may also describe differences that override or supersede what the STM32 variant HAL provides. The areas that are specific to platform HALs and not the STM32 variant HAL include:

- memory map and related configuration and setup
- Clock parameters
- GPIO setup
- Any special handling for external interrupts, or additional interrupts
- Diagnostic I/O baud rates
- Additional diagnostic I/O devices, if any
- LED/LCD control

Name

On-chip Subsystems and Peripherals — Hardware Support

Hardware support

On-chip memory

The ST STM32 parts include on-chip SRAM, and on-chip FLASH. The RAM can vary in size from as little as 4KiB to 128KiB. The FLASH can be up to 1024KiB in size depending on model. There is also support in some models for external SRAM and Flash, which eCos may use where available.

Typically, an eCos platform HAL port will expect a GDB stub ROM monitor or RedBoot image to be programmed into the STM32 on-chip ROM memory for development, and the board would boot this image from reset. The stub ROM/RedBoot provides GDB stub functionality so it is then possible to download and debug stand-alone and eCos applications via the gdb debugger using serial interfaces or other debug channels. The JTAG interface may also be used for development if a suitable JTAG device is available. If RedBoot is present it may also be used to manage the on-chip and external flash memory. For production purposes, applications are programmed into external or on-chip FLASH and will be self-booting.

On-Chip FLASH

The package `CYGPKG_DEVS_FLASH_STM32` ("STM32 Flash memory support") provides a driver for the on-chip flash. This driver conforms to the Version 2 flash driver API, and is automatically enabled if the generic "Flash device drivers" (`CYGPKG_IO_FLASH`) package is included in the eCos configuration. The driver will configure itself automatically for the size and parameters of the specific STM32 variant which has been selected in the eCos configuration.

Cache Handling

The STM32 does not contain any caches, however, the variant HAL supplies the `cyg/hal/hal_cache.h` header to satisfy generic code. This header describes zero sized caches and provides null macros for the required functions.

Serial I/O

The STM32 variant HAL supports basic polled HAL diagnostic I/O over any of the on-chip serial devices. There is also a fully interrupt-driven serial device driver suitable for eCos applications for all on-chip serial devices, and where appropriate for the target platform, DMA can be used to reduce the CPU load. The serial driver consists of an eCos package: `CYGPKG_IO_SERIAL_CORTEXM_STM32` which provides all support for the STM32 on-chip serial devices. Using the HAL diagnostic I/O support, any of these devices can be used by the ROM monitor or RedBoot for communication with GDB. If a device is needed by the application, either directly or via the serial driver, then it *cannot* also be used for GDB communication using the HAL I/O support. An alternative serial port should be used instead.

Depending on the STM32Fx series processor configured, the HAL defines CDL interfaces from `CYGINT_HAL_STM32_UART0` up to `CYGINT_HAL_STM32_UART7` for each of the available UARTs. For `F1` series devices the range is 0..4. For `F2` and `F4[01]` family members the range is 0..5 (4 USARTs and 2 UARTs). For `F4[23]` and `F7` devices the range is 0..7 (4 USARTs and 4 UARTs). The platform HAL CDL should contain an **implements** directive for each such UART that is available for use on the board. This will enable use of the UART for diagnostic and application use.

Caution

eCos uses a numbering scheme for UARTs which starts at 0, whereas STM32 documentation uses a numbering scheme which starts at 1. Be sure to check which numbering scheme is relevant when interpreting UART numbers in CDL options or eCos interfaces (eCos numbering) or hardware-specific definitions (ST numbering).

The STM32 USARTs provide TX and RX data lines plus hardware flow control using RTS/CTS for those UARTs that have them connected.

The `CYGPKG_IO_SERIAL_CORTEXM_STM32` package provides a number of CDL options that allow the size and location of buffers to be configured. These settings, in conjunction with platform specific manifests for DMA stream/channel configuration, are used to control how data is buffered by the STM32 UART driver. For the options listed below, the relevant UART controller is specified by replacing *`x`* with the logical eCos interface number.

`CYGNUM_IO_SERIAL_CORTEXM_STM32_SERIALx_BUFSIZE`

> This option specifies the size of both the input and output buffers for the common serial I/O driver layer, which sits above the low-level STM32 specific driver. When using high baud rates, to avoid overrun, the buffer size may need to be increased to cope with the application serial handler thread switching latency. If the device should only be used in polled mode then this option can be disabled by setting the value to `0`. Typically the device driver will run in interrupt or DMA mode (as configured by the platform), such that it will perform some buffering of both incoming and outgoing data.

`CYGIMP_IO_SERIAL_CORTEXM_STM32_SERIALx_BUFSRAM`

> By default the common serial I/O buffers of the size specified by the `CYGNUM_IO_SERIAL_CORTEXM_STM32_SERIALx_BUFSIZE` option are held in the main application data memory as defined by the active `CYG_HAL_STARTUP` configuration. For the majority of standard serial uses this will be sufficient, and this option can remain disabled. However, this option can be enabled to force the buffers to be placed into on-chip SRAM, which may be required when using very high communication rates with applications that use off-chip data memory. This option is ignored if the `CYG_HAL_STARTUP` configuration already selects the on-chip SRAM for application data.

`CYGNUM_IO_SERIAL_CORTEXM_STM32_SERIALx_RXBUFSIZE`

> This option controls the size of the low-level UART driver RX buffer.
>
> When interrupt transfers are being used then this option specifies the size of the RX buffer used to provide FIFO-alike operation to minimise the possibility for dropped characters when waiting for the UART DSR to be processed. A minimal buffer size is enforced since if the processor is using off-chip RAM then it may be too slow to handle higher baud rates.
>
> For DMA enabled RX this buffer is used to reduce the overall STM32 serial driver ISR and DSR system overhead. For DMA an interrupt will only be generated when 50% of the buffer size has been received, or when the receiver has been idle for some time. This reduces overall system load at the expense of making the driver slightly less “responsive” for non-continuous transfers. The DMA suport actually splits the buffer into two equal sized buffers and uses the hardware double-buffer support to automatically switch between buffers.

Serial overrun mitigation

Ideally it is prudent for UART connections to use hardware flow control where possible, since it provides a mechanism for both ends of a connection to throttle the data flow to avoid buffer overrun. This is especially true when using very high baud rates, since many characters could be received within the ISR, DSR and scheduling latencies of a running application.

If H/W flow control is *not* available then, especially when using higher baud rates, it is critical that the application developer understands the serial transfer requirements, and that the STM32 UART driver processing method and configured buffering is tuned to avoid (or at least minimise the possibility of) overrun. When H/W flow control is not being used then it may not be possible to avoid overrun completely, since the remote end can always transmit more data than the receiver has buffer space for. The possibility of overrun can be mitigated by ensuring a suitable software “throttling” handshake mechanism between the transmitted and receiver, or by ensuring that the receiver has sufficient buffer space at both the low-level driver reception layer (`CYGNUM_IO_SERIAL_CORTEXM_STM32_SERIALx_RXBUFSIZE`) and the higher-level common serial/application layer (`CYGNUM_IO_SERIAL_CORTEXM_STM32_SERIALx_BUFSIZE`) to cope with the worst-case scheduling latency of the application.

Caution

When DMA usage is configured by the specific STM32 platform support (implemented by provision of a suitably named manifest describing the stream/channel configuration), the UART DSR *MUST* have copied data out of the RX buffer into the common serial buffers BEFORE the second DMA buffer has been filled. For example, at 10.5MHz over 1000 8N1 characters can be received in 1-millisecond. The buffer depth should be tuned accordingly to cope with the worst-case UART driver DSR latency to ensure that the automatic hardware switch on DMA buffer fill does not cause data which has not yet been copied to the higher serial layer to be overwritten.

Further, when using DMA, the RX IDLE support is used to force data to be passed to the common serial layer in a timely fashion. This avoids the eCos receiver application having to wait for a complete DMA buffer fill for pending data, which is useful for “character” based serial protocols. The downside is that due to the IDLE implementation having to disable the DMA stream temporarily, if the remote end transmits multiple characters shortly after IDLE is detected by the STM32 receiver then the DMA ISR latency may affect whether RX overrun occurs before the handler can re-enable the DMA stream. The CYGNUM_IO_SERIAL_CORTEXM_STM32_SERIAL*x*_RXINTR_PRI configuration option can be tuned to ensure that the DMA interrupt priority is sufficiently high to minimise the possibility of overrun.

I²C

Note

The I/O controller found on newer STM32 designs (e.g. F7, H7 or L4 families) only supports divided transactions with repeated START conditions. The hardware is not capable of splitting a transaction without generating a START.

So all transaction calls between the cyg_i2c_transaction_begin() and cyg_i2c_transaction_end() calls should use true for the send_start parameter.

Interrupts

The STM32 HAL relies on the architectural HAL to provide support for the interrupts directly routed to the NVIC. The cyg/hal/var_intr.h header defines the vector mapping for these.

In addition support is present for interrupts EXTI5 to EXTI15 that share NVIC vectors. These are decoded by a springboard ISR routine that decodes the interrupt from the EXTI_PR register and calls hal_deliver_interrupt in the architectural HAL to deliver it to the real ISR. Variant specific macros are also defined to permit the masking and configuration of these interrupts.

GPIO

The variant HAL provides support for packaging the configuration of a GPIO line into a single 32-bit descriptor that can then be used with macros to configure the pin and set and read its value. Details are supplied later.

MMC/SD

The variant HAL provides support for MultiMediaCard (MMC) and Secure Digital (SD) media cards via the CDL component CYGPKG_HAL_STM32_MMCSD ("On-chip MMC/SD driver"). This makes use of the STM32 SDIO on-chip interface. NOTE: No support is yet in place for Secure Digital Input Output (I/O) cards.

If enabled this driver allows use of MultiMedia Cards (MMC cards) and Secure Digital (SD) flash storage cards within eCos, exported as block devices. This makes them suitable for use as the underlying devices for filesystems such as FAT.

This driver provides the necessary support for the generic MMC bus layer within the CYGPKG_DEVS_DISK_MMC package to export a disk block device. The disk block device is only available if the generic disk I/O layer found in the package CYGPKG_IO_DISK is included in the configuration.

The block device may then be used as the device layer for a filesystem such as FAT. Example devices are "/dev/mmcsd0/1" to refer to the first partition on the card, or "/dev/mmcsd0/0" to address the whole device including potentially the partition table at the start.

If the driver is enabled, the following options are available to configure it:

`CYGIMP_HAL_STM32_SDIO_INTMODE`

> This indicates that the driver should operate in interrupt-driven mode if possible. This is enabled by default if the eCos kernel is enabled. Note though that if the driver finds that global interrupts are off when running, then it will fall back to polled mode even if this option is enabled. This allows for use of the MMC/SD driver in an initialisation context.

`CYGNUM_HAL_STM32_SDIO_POWERSAVE`

> This option enables the power-saving feature that will disable the SD clock when the device is idle. This feature can normally be safely enabled on all STM32 platforms.

USB OTG Controllers

The STM32 series supports two different USB OTG controller implementations: OTG1 FS and OTG2 HS. If a specific STM32 microcontroller contains both controllers then configuration support is provided to enable you to select specific operation modes for each.

If only `CYGPKG_IO_USB_HOST` host support is enabled then both controllers will operate in host mode. Similarly, if only `CYGPKG_IO_USB_TARGET` is enabled then both controllers will operate in peripheral mode.

If both options are enabled then the controllers will default to host mode. To force a specific controller into peripheral mode then `CYGHWR_DEVS_USB_STM32_OTG_FS` or `CYGHWR_DEVS_USB_STM32_OTG_HS` should be set to `"TARGET"`.

If only one controller is in target mode, then `CYGHWR_IO_USB_TARGET_PCDI_DEFAULT`, and thus `CYGHWR_IO_USB_TARGET_PCDI`, will be set to that controller. If both controllers are in target mode then the FS controller will be used as the default. If you want to select the other controller then this will need to be set explicitly.

Note

The STM32 HS OTG2 controller support is currently limited to use in FS speed mode only. In addition, dynamically changing controller mode between host and target is not supported at this time.

RTC/Wallclock

eCos includes RTC (known in eCos as a wallclock) device drivers for the on-chip RTC in the STM32 family. For the F1 processor family this is located in the package `CYGPKG_DEVS_WALLCLOCK_STM32` ("STM32 RTC wallclock support"), and for the F2, F4, F7, L4 and H7 processor families this is located in the package `CYGPKG_DEVS_WALLCLOCK_STM32F2` ("STM32F2 RTC wallclock support").

On the F1 processor family, the wallclock driver can be configured with three different clock sources using the CDL configuration option `CYGHWR_DEVS_WALLCLOCK_STM32_RTC_SOURCE` ("RTC clock source"): `LSE` for the low-speed 32.768KHz external clock, `LSI` for the low-speed internal clock, or `HSE_128` for the high-speed external clock with a fixed divider of 128. The decision of which source to use depends on the desired clock stability and hardware configuration, and power requirements. Consult the STM32 documentation for more details.

The F2, F4, F7, L4 and H7 processor families are similar, except that the `LSI` clock is at 32KHz clock, and the CDL option name is `CYGHWR_DEVS_WALLCLOCK_STM32F2_RTC_SOURCE`.

Profiling Support

The STM32 HAL contains support for **gprof**-base profiling using a sampling timer. The default timer used is Timer 6, which is one of the basic timers, leaving the more complex timers for application code. The timer used is selected by a set of `#define`s in src/stm32_misc.c which can be changed to refer to a different timer if required. This timer is only enabled when the gprof profiling package (CYGPKG_PROFILE_GPROF) is included and enabled in the eCos configuration, otherwise it remains available for application use.

Clock Control

The platform HAL must provide the input clock frequency (CYGARC_HAL_CORTEXM_STM32_INPUT_CLOCK) in its CDL file. This is then combined with the following options defined in this package to define the default system clocks:

CYGHWR_HAL_CORTEXM_STM32_CLOCK_PLL_SOURCE

> This defines the source of the input to the PLL that generates the main system clock. There are two possible sources: the HSE selects the high speed external oscillator, and the HSI selects the high speed internal oscillator. This option defaults to the HSE.

CYGHWR_HAL_CORTEXM_STM32_CLOCK_NEED_HSE

> This indicates whether the HSE clock should be started at all. If the system clock is running solely from the HSI clock, there may be no need for an HSE clock, and no crystal connected to it. In that scenario, this option can be disabled. This is usually set by the platform HAL.

CYGHWR_HAL_CORTEXM_STM32_CLOCK_PLL_PREDIV

> This option specifies how much the clock from the input source defined by CYGHWR_HAL_CORTEXM_STM32_CLOCK_PLL_SOURCE is divided down by, before being used as an input for the PLL. On non-connectivity parts, you can only divide by 2 or 1. On other F1 parts, if using HSI as the clock source, then that is automatically divided by 2. If using HSE as the clock source, then this value corresponds to the PREDIV1 field of register RCC_CFGR2. On F2 and F4 parts, this value corresponds to the PLLM field of RCC_PLLCFGR.

CYGHWR_HAL_CORTEXM_STM32_CLOCK_PLL_MUL

> This defines the factor by which the PLL will multiply the selected input clock, after being divided by CYGHWR_HAL_CORTEXM_STM32_CLOCK_PLL_PREDIV. It ranges from 2 to 16 on F1 parts, and 2 to 432 on F2 and F4 parts, and defaults to 9, which is intended to give a 72MHz system clock on those boards with an 8MHz input. On the F1 it corresponds to the PLLMUL field of RCC_CFGR. On the F2/F4 it corresponds to the PLLN field of RCC_PLLCFGR.

CYGHWR_HAL_CORTEXM_STM32_CLOCK_SYSCLK_DIV

> This setting is only applicable to the F2 and F4 family of processors. This defines the divider applied to the PLL output for use as the SYSCLK. It can only have values 2, 4, 6 or 8 and defaults to 4. This setting corresponds to the PLLP field of RCC_PLLCFGR.

CYGHWR_HAL_CORTEXM_STM32_CLOCK_HCLK_DIV

> This defines the divider applied to the prescaler for all peripheral clocks. The HCLK fed to the AHB bus and other peripherals is taken directly from this output. Other peripheral clocks are derived by dividing it further. This can take values between 1 and 512 in powers of 2, the default value is 1.

CYGHWR_HAL_CORTEXM_STM32_CLOCK_PCLK1_DIV

> This defines the prescaler divider for the peripheral clock passed to peripherals on the APB1 bus. It can take values between 1 and 16 in powers of 2, the default value is 2.

CYGHWR_HAL_CORTEXM_STM32_CLOCK_PCLK2_DIV

> This defines the prescaler divider for the peripheral clock passed to peripherals on the APB2 bus. It can take values between 1 and 16 in powers of 2, the default value is 1.

CYGHWR_HAL_CORTEXM_STM32_CLOCK_PLLQ_DIV

> This setting only applies to the F2 and F4 family of processors. It defines the divider used to divide down the PLL output clock (VCO clock) for use by the USB OTG FS, SDIO and RNG peripherals. USB OTG FS requires a 48MHz clock and other peripherals require a clock no greater than 48MHz. The allowable values range from 4 to 15 and the default is 10.

The actual values of these clocks, in Hz, is stored in global variables `hal_stm32_sysclk`, `hal_stm32_hclk`, `hal_stm32_pclk1`, `hal_stm32_pclk2` and for the F2/F4/F7 processor families only, `hal_stm32_qclk`. The clock supplied to the SysTick timer, HCLK/8, is also assigned to `hal_cortexm_systick_clock`. These variables are used, rather than configuration options, in anticipation of future support for power management by varying the system clock rate.

Note that when changing or configuring any of these clock settings, you should consult the relevant processor datasheet as there may be both upper and lower constraints on the frequencies of some clock signals, including intermediate clocks. There are also some clocks where, while there is no strict constraint, clock stability is improved if values are chosen wisely. Finally, be aware that increasing clock speeds using this package may have an effect on platform specific properties, such as memory timings which may have to be adjusted accordingly.

Name

GPIO Support on STM32F processors — Details

Synopsis

#include <cyg/hal/hal_io.h>

`cyg_uint32 pin = `**`CYGHWR_HAL_STM32_PIN_OUT`**`(`*`port`*`, `*`bit`*`, `*`ppod`*`, `*`pupd`*`, `*`speed`*`);`

`cyg_uint32 pin = `**`CYGHWR_HAL_STM32_PIN_ALTFN_OUT`**`(`*`port`*`, `*`bit`*`, `*`af`*`, `*`ppod`*`, `*`pupd`*`, `*`speed`*`);`

`cyg_uint32 pin = `**`CYGHWR_HAL_STM32_PIN_IN`**`(`*`port`*`, `*`bit`*`, `*`pupd`*`);`

`cyg_uint32 pin = `**`CYGHWR_HAL_STM32_PIN_ALTFN_IN`**`(`*`port`*`, `*`bit`*`, `*`af`*`, `*`ppod`*`, `*`pupd`*`);`

`cyg_uint32 pin = `**`CYGHWR_HAL_STM32_PIN_ANALOG`**`(`*`port`*`, `*`bit`*`);`

`CYGHWR_HAL_STM32_GPIO_SET`` (`*`pin`*`);`

`CYGHWR_HAL_STM32_GPIO_OUT`` (`*`pin`*`, `*`val`*`);`

`CYGHWR_HAL_STM32_GPIO_IN`` (`*`pin`*`, `*`val`*`);`

Description

The STM32 HAL provides a number of macros to support the encoding of GPIO pin identity and configuration into a single 32 bit descriptor. This is useful to drivers and other packages that need to configure and use different lines for different devices.

A descriptor is created with one of the 5 variants depending on how the pin is to be used: `CYGHWR_HAL_STM32_PIN_IN` defines the pin as an input whose value can be accessed by the user using the macro `CYGHWR_HAL_STM32_GPIO_IN` (see later) `CYGHWR_HAL_STM32_PIN_OUT` defines the pin as an output where the user can set the pin output value with the macro `CYGHWR_HAL_STM32_GPIO_OUT` (see later); `CYGHWR_HAL_STM32_PIN_ALTFN_OUT` and `CYGHWR_HAL_STM32_PIN_ALTFN_IN` are used to define a pin that will be controlled by an on-chip peripheral; or `CYGHWR_HAL_STM32_PIN_ANALOG` which means it can be used as an input to the ADC peripheral or an output from the DAC peripheral.

The distinction between the `CYGHWR_HAL_STM32_PIN_ALTFN_OUT` and `CYGHWR_HAL_STM32_PIN_ALTFN_IN` is purely whether a speed parameter is required, since both macros define a pin that will be controlled by an on-chip peripheral. This distinction is required to provide simpler F1 family support for its different alternative I/O mapping implementation.

The 5 variants take a subset of arguments from the following list:

`port`
: This identifies the GPIO port to which the pin is attached. Ports are identified by letters from A to I.

`bit`
: This gives the bit or pin number within the port. These are numbered from 0 to 15.

`af`
: For the `ALTFN` macros this parameter indicates which on-chip peripheral is used to control the pin. Consult ST's documentation for the specific processor model to determine which peripheral number is used to select the peripheral for the required pin mapping.

  This field is not-relevant to, and hence ignored by, the F1 family of devices where the alternative AFIO configuration mechanism is used.

`ppod`
: If this is an output pin (either GPIO output or driven by a peripheral), this setting indicates whether it should be driven in push-pull mode (`PUSHPULL`) or open-drain mode (`OPENDRAIN`). If the pin is not an output pin, this value is unused, but for clarity can be given the setting `NA`.

pupd

If this is an input pin, or an output pin configured in open-drain mode (whether controlled by GPIO or a peripheral), this setting can be used to indicate whether a weak pull-up resistor (PULLUP) is used, or a weak pull-down resistor (PULLDOWN) is used. If neither are to be used, then a value of NONE can be given. For F1 family devices the value FLOATING can be used to identify a floating input, which is synonomous with NONE for F2/F4 family devices.

speed

This setting indicates the output speed for GPIO outputs. At time of writing, the speeds for F1 parts can be 2MHz, 10MHz or 50MHz and for F2/F4 parts can be 2MHz, 25MHz, 50Mhz, 100MHz with 30pF capacitance or 80MHz with 15pF capacitance. It is possible to indicate the desired speed by passing in the generic values LOW, MED, FAST, or HIGH.

For F1 family devices these are synonymous with using values of 2MHZ, 25MHZ, 50MHz or 50MHz respectively (FAST and HIGH being synonyms due to the 50MHz limit on F1 family devices).

For F2/F4 family devices these are synonymous with using values of 2MHZ, 25MHZ, or 50MHz respectively, with no corresponding value for HIGH due to its variable nature.

In order to make these definitions more future-proof and abstract, it is therefore strongly recommended to use this form of setting instead: AT_LEAST(*mhz*) to indicate the next speed rating above the supplied speed (in MHz); and analogously, AT_MOST(*mhz*) to indicate that the supplied speed in MHz must not be exceeded.

For non-GPIO output pins, a value of NA can be given.

The following examples show how these macros may be used:

```
// Define port A pin 10 as being controlled by a peripheral which, for
// this pin on F2/F4 devices, is alternate function 7 (and for this
// pin that means USART1), without any pull-ups/pull-downs.
#define CYGHWR_HAL_STM32_UART1_RX               CYGHWR_HAL_STM32_ALTFN_IN( A, 10, 7, NA, NONE )

// Define port B pin 10 as a push-pull output under the control of the
// peripheral which, for this pin on F2/F4 devices, is alternate
// function 7 (and for this pin that means USART3), with an output
// speed of 50MHz or greater.
#define CYGHWR_HAL_STM32_UART3_TX               CYGHWR_HAL_STM32_ALTFN_OUT( B, 10, 7, PUSHPULL, NONE, AT_LEAST(50) )

// Define port A pin 12 as a push-pull output under GPIO control, with no
// pull-ups/pull-downs, with an output speed of 50MHz or greater.
#define CYGHWR_HAL_STM32_UART1_RTS              CYGHWR_HAL_STM32_OUT( A, 12, PUSHPULL, NONE, AT_LEAST(50) )
```

Additionally, the macro CYGHWR_HAL_STM32_GPIO_NONE may be used in place of a pin descriptor and has a value that no valid descriptor can take. It may therefore be used as a placeholder where no GPIO pin is present or to be used. This can be useful when defining pin configurations for a series of instances of a peripheral (e.g. UART ports), but where not all instances support all the same pins (e.g. hardware flow control lines).

The remaining macros all take a suitably constructed GPIO pin descriptor as an argument. CYGHWR_HAL_STM32_GPIO_SET configures the pin according to the descriptor and must be called before any other macros. CYGHWR_HAL_STM32_GPIO_OUT sets the output to the value of the least significant bit of the *val* argument. The *val* argument of CYGHWR_HAL_STM32_GPIO_IN should be a pointer to an int, which will be set to 0 if the pin input is zero, and 1 otherwise.

Further helper macros are available, and it is recommended to consult the header file <cyg/hal/var_io_pins.h> (also present in the include subdirectory of the STM32 variant HAL package within the eCos source repository), for the complete list if needed. Ensure you only inspect the relevant sections of this low-level header file when investigating specific F1 and F2/F4 processor family variants.

EXTI wrapper

If a platform implements CYGINT_HAL_STM32_GPIO_EXTI_VECTOR then support for a simple EXTI interrupt handler wrapper is provided. This is used to “hide” the eCos interrupt implementation when supporting callback handlers for interrupt enabled

GPIO pins. For almost all STM32 platforms and eCos applications this wrapper functionality is *NOT* required, and the `CYGIMP_HAL_STM32_GPIO_EXTI_VECTOR` option should not be enabled.

The interface is exported via manifests that are only defined when the relevant functionality is available. The *`pin`* argument is a standard eCos GPIO pin descriptor as described above.

The `CYGHWR_HAL_VAR_GPIO_IRQ_INIT()` referenced code should be called at a suitable initialisation point either by the platform or the eCos application, prior to enabling a specific source.

When enabling a callback for a GPIO EXTI source the *`trigger`* parameter encodes whether the handler is called for rising (`CYGHWR_HAL_VAR_IRQ_EDGE_RISE`), falling (`CYGHWR_HAL_VAR_IRQ_EDGE_FALL`), or either (`CYGHWR_HAL_VAR_IRQ_EDGE_BOTH`), edge events. When the configured event occurs the referenced *`handler`* function is called with the supplied *`arg`* parameter.

An example use of this functionality is when building eCos WICED applications against an unmodified WICED-SDK source-tree,

The `HAL_VAR_EXTI_EVENT_CONFIGURE()` code allows for control of EXTI events. The *`vector`* is the corresponding EXTI interrupt identifier. The *`up`* parameter encodes whether the event is triggered from a rising signal (`1`) or falling signal (`0`). The *`enable`* parameter defines whether the event is being enabled (`1`) or disabled (`0`).

EXTI wrapper API

#include <cyg/hal/hal_io.h>

`void` **`CYGHWR_HAL_VAR_GPIO_IRQ_INIT`**`();`

`cyg_bool enabled =` **`CYGHWR_HAL_VAR_GPIO_IRQ_ENABLE`**`(`*`pin`*`,` *`trigger`*`,` *`handler`*`,` *`arg`*`);`

`void` **`CYGHWR_HAL_VAR_GPIO_IRQ_DISABLE`**`(`*`pin`*`);`

`void` **`HAL_VAR_EXTI_EVENT_CONFIGURE`**`(`*`vector`*`,` *`up`*`,` *`enable`*`);`

Name

Peripheral clock control — Details

Synopsis

#include <cyg/hal/hal_io.h>

cyg_uint32 clkdesc = **CYGHWR_HAL_STM32_CLOCK**(*clk*);

CYGHWR_HAL_STM32_CLOCK_ENABLE (*clkdesc*);

CYGHWR_HAL_STM32_CLOCK_DISABLE (*clkdesc*);

Description

The HAL provides macros which may be used to enable or disable peripheral clocks. Effectively this indicates whether the peripheral is powered on (enabled) or powered down (disabled) and so may be used to ensure unused peripherals are turned off, to save power.

It is important to remember that before a peripheral can be used, it must be enabled. It is safe to re-enable a peripheral that is already enabled, although usually a device driver will only do so once in its initialisation. eCos will automatically initialise some peripheral blocks where it needs to use the associated peripherals (such as memory controllers and some (but usually not all) GPIO banks), and in eCos-supplied device drivers which are included in the eCos configuration. However this should not be relied on - it is always safest to enable the peripheral clocks anyway just in case. Finally, remember that each GPIO bank must be enabled separately.

The `CYGHWR_HAL_STM32_CLOCK` macro can be used to create a descriptor used to specify the clock. This takes the following parameter:

clk
: This parameter selects the specific clock. Definitions for each clock can be found in the file `<cyg/hal/var_io.h>` which can also be found as the file `include/var_io.h` within the STM32 processor variant HAL package in the eCos source repository, ensuring you are looking at the correct section for the relevant processor family.

Once a descriptor has been created, it may be used as the parameter to the `CYGHWR_HAL_STM32_CLOCK_ENABLE` or `CYGHWR_HAL_STM32_CLOCK_DISABLE` macros in order to enable or disable the peripheral clock.

The following example shows how these macros may be used:

```
#define CYGHWR_HAL_STM32_ETH_MAC_CLOCK          CYGHWR_HAL_STM32_CLOCK( ETHMAC )

int init_eth(...)
{
      CYGHWR_HAL_STM32_CLOCK_ENABLE(CYGHWR_HAL_STM32_ETH_MAC_CLOCK);
      ... /* Rest of initialisation */
}

void stop_eth(...)
{
     ... /* Free up resources */
     CYGHWR_HAL_STM32_CLOCK_DISABLE(CYGHWR_HAL_STM32_ETH_MAC_CLOCK);
}
```

Name

DMA Support — Details

Synopsis

#include <cyg/hal/var_dma.h>

`pin = CYGHWR_HAL_STM32_DMA(ctlr, stream, chan, mode);`

`pin = CYGHWR_HAL_STM32_DMA(dmareq_id, dmamux_channel, mode);`

`hal_stm32_dma_init (hal_stm32_dma_stream *stream, int priority);`

`hal_stm32_dma_delete (hal_stm32_dma_stream *stream);`

`hal_stm32_dma_disable (hal_stm32_dma_stream *stream);`

`hal_stm32_dma_configure (hal_stm32_dma_stream *stream, int tfr_size, cyg_bool no_minc, cyg_bool polled);`

`hal_stm32_dma_configure_circular (hal_stm32_dma_stream *stream, cyg_bool enable);`

`hal_stm32_dma_configure_doublebuffer (hal_stm32_dma_stream *stream, cyg_bool enable, void *memory1);`

`hal_stm32_dma_configure_flow (hal_stm32_dma_stream *stream, cyg_bool enable);`

`hal_stm32_dma_start (hal_stm32_dma_stream *stream, void *memory, CYG_ADDRESS peripheral, cyg_uint32 size);`

`hal_stm32_dma_stop (hal_stm32_dma_stream *stream);`

`hal_stm32_dma_poll (hal_stm32_dma_stream *stream);`

Description

The HAL provides support for access to the DMA controllers. This support is not intended to expose the full functionality of these devices and is mainly limited to supporting peripheral DMA, currently ADC, SPI, I²C and MMC/SD.

The user is referred to the ST documentation for a full description of the DMA devices, and to the sources of the ADC, I²C, SPI and MMC/SD drivers for examples of the use of this API. This documentation only gives a brief description of the functions available.

A DMA stream is defined by a controller number (0 or 1), a stream number (0 to 8), a channel number (0 to 7) and a mode defining transfer direction. The macro `CYGHWR_HAL_STM32_DMA()` combines these into a 32-bit descriptor that may be stored with a device driver and used to initialize the stream.

NOTE: the DMA terminology has changed between F1 and F2/F4 versions of the STM32 family. In F1 devices each DMA controller has a number of channels, each of which can be driven by a subset of the on-chip devices; there is no way to select which device drives the channel, and care must be taken to allocate channels so that devices don't trigger the wrong channel. In F2/F4 devices each DMA controller has a number of streams; each stream can explicitly select one of a number of driving devices by means of a channel selection field in a control register. F1 channels and F2/F4 streams are essentially the same thing, and the problems of false triggering in F1 devices is solved in F2/F4 by adding the explicit channel selection. To make things more complicated, F1 channels are numbered from one while F2/F4 streams are numbered from zero. HAL support for DMA largely follows the F2/F4 terminology, but the original channel numbering for the F1 is preserved when defining channels.

The following examples show how definitions should be made:

```
    // F1 definition: controller 1, channel 6, memory-to-peripheral
    #define CYGHWR_HAL_STM32_I2C1_DMA_TX            CYGHWR_HAL_STM32_DMA( 1, 6, 0, M2P )


    // F2/F4 definition: controller 2, stream 0, channel 3, peripheral-to-memory
    #define CYGHWR_HAL_STM32_SPI1_DMA_RX            CYGHWR_HAL_STM32_DMA( 2, 0, 3, P2M )
```

The special manifest `CYGHWR_HAL_STM32_DMA_NONE` can be used when the code does not require DMA support for a specific stream. For example:

```
    // I2C2 RX should NOT use DMA
    #define CYGHWR_HAL_STM32_I2C2_DMA_RX            CYGHWR_HAL_STM32_DMA_NONE
```

Note

Not all device drivers support the ability of using `CYGHWR_HAL_STM32_DMA_NONE` to disable DMA use for specific stream mappings. For example, the STM32 I²C driver does allow for individual streams to be configured for interrupt-driven or DMA transfers as required.

Some later variants such as the L4+ or H7 implement a DMA multipexor which routes peripheral DMA requests to DMA controller channels. This means that the way in which DMA channels are associated with peripherals is somewhat different and DMA descriptors have a different format. Instead of selecting a DMA controller and channel, a DMAMUX channel is selected which automatically selects the DMA controller and channel. The following examples show how these descriptors are initialized:

```

// L4+
// I2C1 RX uses DMAMUX channel 0, which maps to DMAC 1, channel 0, peripheral-to-memory
#define CYGHWR_HAL_STM32_I2C1_DMA_RX     CYGHWR_HAL_STM32_DMA(CYGHWR_HAL_STM32_DMAMUX_I2C1_RX,0,P2M)

// H7
// SPI5 TX uses DMAMUX channel 10, which maps to DMAC 2, channel 3, memory-to-peripheral
#define CYGHWR_HAL_STM32_SPI5_DMA_TX    CYGHWR_HAL_STM32_DMA(SPI5_TX,10,M2P)

```

Before use a DMA stream must be initialized. This is done by calling `hal_stm32_dma_init()`. The first argument to this is a hal_stm32_dma_stream structure in which the *`desc`* field should have been initialized to a DMA descriptor; the *`callback`* field set to a callback function; and the *`data`* field set to any user defined data. The *`priority`* argument defines both the interrupt level assigned to the stream interrupt, and the DMA channel arbitration priority level (defined by the top two bits). This function initializes the hardware and the stream structure and needs only to be called once during driver initialization.

By default a stream is initialized to perform 8 bit transfers under interrupt control and to advance the memory address pointer. If a different configuration is required, then the driver should call `hal_stm32_dma_configure()` which will allow these options to be varied. The *`tfr_size`* argument defines the transfer size and may be 8, 16 or 32 bits. The *`no_minc`* argument disables memory increments if true. The *`polled`* argument configures the stream for polled mode if true, otherwise it will be interrupt driven. This function may either be called once to set up the stream permanently, or on a transfer-by-transfer basis, or not at all if the defaults are what is required.

If the driver needs circular mode DMA processing then it can call the function `hal_stm32_dma_configure_circular()` with *`enable`* set to true. This allows circular buffers and continuous data flows (e.g. ADC scan mode as used by the STM32F ADC driver). Calling the function with *`enable`* set to false will disable circular mode.

If a F2/F4 driver wants to use the continuous double-buffer support then it can call the `hal_stm32_dma_configure_doublebuffer()` with *`enable`* set to true. This configures the DMA to automatically switch between buffers at the end of a transfer. The *`memory1`* is the second buffer to be used in conjunction with the buffer passed as the *`memory`* parameter to the `hal_stm32_dma_start()` function.

Note

The second buffer *must* be at least the same *`size`* as the (first) buffer subsequently passed to the `hal_stm32_dma_start()` function.

Calling the function with *enable* set to false will disable the double buffer mode, with the passed *memory1* parameter being ignored.

Warning

The DMA controller hardware when using double buffer mode will automatically switch buffers on a buffer fill event.

When using double buffer mode the developer should ensure that the buffer size used is large enough to cope with the processing code associated with a completed transaction being able to complete to avoid overrun. The size of the buffers will depend on the DMA transfer rate for the peripheral being used and the application DSR latency, plus the actual callback buffer processing code time.

If a F2/F4 driver needs to use peripheral controlled DMA flow then it can call the function `hal_stm32_dma_configure_flow()` with *enable* set to true. This configures the DMA to allow the peripheral to control the flow of DMA transfers, instead of the DMA controller (e.g. the STM32 SDIO device signals the end of data transfers). Calling the function with *enable* set to false will disable the peripheral flow control mode.

A transfer is defined and started by calling `hal_stm32_dma_start()`. The *memory* argument defines the memory address to/from which the transfer will be made. The *peripheral* argument is the address of the data register of the peripheral involved. The *size* argument defines the number of data items to be transferred, or in the case of peripheral flow control configurations the number of items expected, as defined by *tfr_size*. Once this call completes, the channel is operational and will transfer data once the peripheral starts triggering transfers.

If the stream is configured for interrupt control then when a transfer completes an interrupt is raised. This will disable the stream and cause the callback in the stream structure to be called from DSR mode. The prototype of the callback is as follows:

```
    typedef void hal_stm32_dma_callback( hal_stm32_dma_stream *stream, cyg_uint32 count, CYG_ADDRWORD data );
```

The *stream* argument is the stream structure initialized by the user. The *count* argument is a count of the number of data items that remain to be transferred, and will be zero for a successful transfer. The *data* argument is a copy of the *data* field from the stream structure.

The configuration option `CYGIMP_HAL_STM32_DMA_CALLBACK_ISR` can be used to enable the optional support for an ISR level callback when the DMA TransferComplete (TC) interrupt is triggered. This support is not normally required, but some STM32 family variants may require some collusion between the DMA system and the peripheral H/W controller to ensure correct operation. As such it is not expected that the developer should ever need to manually enable `CYGIMP_HAL_STM32_DMA_CALLBACK_ISR` since it will be automatically enabled if any of the configured packages require the functionality. The callback is executed within the ISR of the DMA TC processing and so should not block.

```
    typedef void hal_stm32_dma_callback_isr( hal_stm32_dma_stream *stream, CYG_ADDRWORD data );
```

As with the normal DSR callback the *stream* argument is the stream structure initialized by the user and the *data* argument is a copy of the *data* field from the stream structure.

If the stream is configured for polled mode, then the driver must call `hal_stm32_dma_poll()` frequently. When the transfer has completed the callback function will be called from within the poll routine. The driver needs to detect this and terminate the polling loop.

Most drivers will initialize a DMA stream and keep it enabled throughout the system lifetime. However, if it is necessary to share a stream, or otherwise disable use of a stream, the driver may call `hal_stm32_dma_delete()` to return a stream to unused state. It will be necessary to call `hal_stm32_dma_init()` before it can be used again.

Alternatively for circular mode configured streams the `hal_stm32_dma_disable()` can be used to disable the stream DMA without clearing the state. The function `hal_stm32_dma_start()` can then be used to re-enable the DMA stream with the previous configured state.

The `hal_stm32_dma_stop()` function allows a stream to be disabled *without* clearing the transfer state. The normal callback handler is subsequently called with a non-zero *count* indicating a partial transfer.

Note

The `hal_stm32_dma_stop()` function will return immediately, however the stream may remain active until any active “item” transfer has completed (*not* the full *`size`* amount). When appropriate the stream will be disabled by the DMA controller and the relevant callback handler function called asynchronously.

Name

Test Programs — Details

Test Programs

The STM32 variant HAL contains some test programs which allow various aspects of the microcontroller or the architecture to be tested.

Timers Test

The `timers` test checks the functionality of the microcontroller timers and in particular the interrupt priority and nesting mechanisms. The test programs all the available timers to interrupt at a variety of different rates and records various parameters. The timers are programmed to interrupt at higher rates for lower numbered timers, and higher rate timers are given higher priority. The test outputs a sequence of tables of the following format:

```
ISRs max_nesting 6 max_nesting_seen 7
 T      Ticks         0         1         2         3         4         5         6         7         8
 1:    937883    468079         0     84264     78541     73773     66846     60741     55480     50159
 2:    337228    186324         0         0     30306     28894     26156     24077     21771     19700
 3:    164018     99600         0         0         0     15444     14024     12693     11667     10590
 4:    120064     80417         0         0         0         0     11341     10349      9408      8549
 5:     82581     60965         0         0         0         0         0      7833      7181      6602
 6:     62154     50107         0         0         0         0         0         0      5953      6094
 7:     59684     53864         0         0         0         0         0         0         0      5820
 8:     51186     51186         0         0         0         0         0         0         0         0
DSRs
 T:                   0         1         2         3         4         5         6         7         8
 1:  preempt:    465351      1226       525       296       240       152       103        97        89
       count:         0     93768        10         0         0         0         0         0         0
 2:  preempt:    185237       606       127       116        75        60        41        27        35
       count:         0     33722         0         0         0         0         0         0         0
 3:  preempt:     99002       354        95         7        55        31        23        18        15
       count:         0     16401         0         0         0         0         0         0         0
 4:  preempt:     79892       331        83        39         3        23        21        12        13
       count:         0     12006         0         0         0         0         0         0         0
 5:  preempt:     60628       184        65        33        27         0         9        13         6
       count:         0      8258         0         0         0         0         0         0         0
 6:  preempt:     49798       180        53        22        21        15         0        12         6
       count:         0      6215         0         0         0         0         0         0         0
 7:  preempt:     53583       158        48        25        18        11        16         0         5
       count:         0      5968         0         0         0         0         0         0         0
 8:  preempt:     50937       140        46        22        18         6         9         8         0
       count:         0      5118         0         0         0         0         0         0         0
```

The first line shows the depth of ISR nesting seen since the last report, plus the maximum seen throughout the run.

The first table contains a row for each timer. The *Ticks* column shows the total number of ISRs called for this timer. The *0* column shows how many ISR calls interrupted thread state. The remaining columns show how many ISR calls preempted the ISR for the given timer. For example, the ISR for timer 1 preempted the ISR for timer 6 60741 times.

The second table contains two rows for each timer. The *preempt:* row shows how many times the ISR preempted the DSR for the given timer. The zero column correspond to thread state as before. For example the ISR for timer 2 preempted the DSR for timer 4 75 times. The *count:* row shows the range of `count` values passed to the DSR and indicate the number of DSR calls not matched exactly to ISR calls. The ISR calls the DSR every 10 ticks, so the total counts should be one tenth of the ISR Ticks value.

Chapter 297. STM3210C-EVAL Platform HAL

Name

CYGPKG_HAL_CORTEXM_STM32_STM3210C_EVAL — eCos Support for the STM3210C-EVAL Board

Description

The STM3210C-EVAL board contains a STM32F107VCT microcontroller. It has connectors for one UART, I²C, MicroSD, USB, CAN, JTAG and various other devices.

For typical eCos development, a GDB stub image is programmed into internal FLASH and the CPU boots directly into that. It is then possible to download and debug stand-alone and eCos applications via the gdb debugger using USART1. Alternatively test programs may be downloaded and debugged via a JTAG debugger attached to the JTAG socket. Available RAM is limited to 64KB, so development for larger applications may also consist of programming them to flash and using JTAG to debug them from there.

This documentation describes platform-specific elements of the STM3210C-EVAL board support within eCos. The STM32 variant HAL documentation covers various topics including HAL support common to STM32 variants, and on-chip device support. This document complements the STM32 documentation.

Supported Hardware

The STM32 has two on-chip memory regions. A RAM region of 64KiB is present at 0x20000000. A FLASH region is present at 0x08000000 and is aliased to 0x0000000 during normal execution.

The STM32 variant HAL includes support for the five on-chip serial devices which are documented in the variant HAL. USART1 is connected to an external connector on the board marked "CN6". The UART does not have any RTS/CTS lines, but the CTS and DSR pins are connected to bootloader reset and select lines. Care must therefore be taken when connecting this board to a host to ensure that these lines are not pulled in the wrong direction. A three wire cable is recommended.

The STM32 variant HAL also includes support for the I²C bus. A single I²C device is instantiated as part of the platform port, which is for the M24C64 serial EEPROM connected via I²C. It is exported to `<cyg/io/i2c.h>` with the name `hal_stm32_i2c_eeprom` in the normal way.

Device drivers are provided for the STM32 on-chip ADC interfaces, SPI interface and Ethernet MAC. Additionally, support is provided for the on-chip watchdog, RTC (wallclock) and a Flash driver exists to permit management of the STM32's on-chip Flash.

Also, while the board is fitted with a CAN interface, this is not presently supported by the HAL port; nor is the microSD card (or SDIO interface) at the present time. The STM32F1 processor and the STM3210C-EVAL board provide an extremely wide variety of peripherals, but unless support is specifically indicated, it should be assumed that it is not included.

Tools

The board port is intended to work with GNU tools configured for an **arm-eabi** target. The original port was done using **arm-eabi-gcc** version 4.3.2, **arm-eabi-gdb** version 6.8, and **binutils** version 2.18.

Name

Setup — Preparing the STM3210C-EVAL Board for eCos Development

Overview

In a typical development environment, the STM3210C-EVAL board boots from internal flash into the GDB Stubrom. eCos applications are configured for RAM startup and then downloaded and run on the board via the debugger **arm-eabi-gdb**.

Stubrom Installation

For serial communications, the Stubrom runs with 8 bits, no parity, and 1 stop bit at 38400 baud. This rate can be changed in the platform HAL configuration. Under normal circumstances, the Stubrom runs in-place from the internal Flash.

Programming The Stubrom

To program the Stubrom into the internal flash either a JTAG debugger that understands the STM32 flash may be used, such as a Ronetix PEEDI or an Abatron BDI3000, or the ST **Flash Loader Demonstrator** may be used. Configuration files for the PEEDI and BDI3000 are supplied in the STM3210C-EVAL HAL package, and brief instructions for downloading the Stubrom are given in there. If no JTAG debugger is available, then the Stubrom must be downloading using the **Flash Loader Demonstrator** which can be downloaded from the page for the STM32F101ZE CPU, in the *Design Support->Demo SW* section of ST's website at http://www.st.com/internet/mcu/product/164506.jsp

The documentation for this may be found at http://www.st.com/internet/com/TECHNICAL_RESOURCES/TECHNICAL_LITERATURE/USER_MANUAL/CD00171488.pdf

The following are brief instructions for downloading the Stubrom using the **Flash Loader Demonstrator**.

1. Download the **Flash Loader Demonstrator** from the ST website and install it on a PC running Windows that has an available serial port.

2. Copy the file `stubrom.bin` from the `loaders/stm3210c_eval/gdbstub/ROM` sub-directory within your eCosPro installation to a suitable location on the Windows PC.

3. Connect a null-modem serial cable between the USART1 serial port and a serial port on the Windows machine.

4. Move the *BOOT0/SW2* switch to the 1 position, ensure the *BOOT1/SW1* switch remains in the 0 position, and press the reset button.

5. Start the **Flash Loader Demonstrator** and in the first screen select the COM port connected to the board under *Port Name*. Ensure that the baud rate is also set at *115200*, with *Even* parity and *Echo* disabled. Press *Next*. If you are succesfully connected to the board, you should see the message *Target is readable. Please press "Next" to proceed* . Press *Next* and you will be prompted to select a target. Select *STM32_Connectivity-line_256K*, then *Next* to go to the Operation choice page. If any of these steps fail, follow the direction given by the loader to recover.

6. On the operation choice page select *"Download to device"* and under *"Download from file"* either type the location of the `stubrom.bin` file, or browse to it. Ensure that the *"@"* field is set to *8000000*, the *"Global Erase"* radio button is selected and that all other options are clear except *"Verify after download"*.

7. Press *"Next"* and the loader should download and verify the binary file. The download or verify may fail if the flash has been previously locked, in which case you should select the *"Enable/Disable Flash protection"* radio button, *"Disable"* and *"Write Protection"* drop-down items, press *"Next"* and retry the download. Upon successful download press *"Close"* to exit the loader.

8. Move the *BOOT0/SW2* switch back to the 0 position and press the reset button.

The behaviour of the **Flash Loader Demonstrator** documented here is that of version 2.2.0. The actual behaviour of newer or older versions may vary slightly.

Whatever mechanism is used to program the Stubrom, something similar to the following output should be seen on USART1 when the reset button is pressed:

```
+$T050f:e2230008;0d:f0ff0020;#b5
```

Rebuilding the Stubrom

Should it prove necessary to rebuild a Stubrom binary, this is done most conveniently at the command line. The steps needed to rebuild the the ROM version of the Stubrom are:

```
$ mkdir stubrom_stm3210c_rom
$ cd stubrom_stm3210c_rom
$ ecosconfig new stm3210c stubs
$ ecosconfig import $ECOS_REPOSITORY/hal/cortexm/stm32/stm3210c_eval/VERSION/misc/stubs_ROM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

At the end of the build the `install/bin` subdirectory should contain the file `stubrom.bin`.

Name

Configuration — Platform-specific Configuration Options

Overview

The STM3210C-EVAL board platform HAL package is loaded automatically when eCos is configured for an `stm3210c_eval` target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The STM3210C-EVAL board platform HAL package supports three separate startup types:

RAM

> This is the startup type which is normally used during application development. The board has GDB stubs programmed into internal Flash at location 0x08000000 and uses internal RAM at location 0x20000000. arm-eabi-gdb is then used to load a RAM startup application into memory from 0x20001000 and debug it. It is assumed that the hardware has already been initialized by the stubs. By default the application will use the eCos virtual vectors mechanism to obtain services from the stubs, including diagnostic output.

ROM

> This startup type can be used for finished applications which will be programmed into internal ROM at location 0x08000000. Data and BSS will be put into internal RAM starting from 0x20000400. The application will be self-contained with no dependencies on services provided by other software. The program expects to boot from reset with ROM mapped at location zero. It will then transfer control to the 0x08000000 region. eCos startup code will perform all necessary hardware initialization.

JTAG

> This is the startup type used to build applications that are loaded via a JTAG interface. The application will be self-contained with no dependencies on services provided by other software. The program expects to be loaded from 0x20000400 and entered at that address. eCos startup code will perform all necessary hardware initialization.

Monitors and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building the Stubrom.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then serial port 0 will be claimed for HAL diagnostics.

UART Serial Driver

The STM3210C-EVAL board uses the STM32's internal UART serial support. As well as the polled HAL diagnostic interface, there is also a `CYGPKG_IO_SERIAL_CORTEXM_STM32` package which contains all the code necessary to support interrupt-driven operation with greater functionality. All five UARTs can be supported by this driver, although only USART1 is actually routed to an external connector. Note that it is not recommended to enable this driver on the port used for HAL diagnostic I/O. This driver is not active until the `CYGPKG_IO_SERIAL_DEVICES` configuration option within the generic serial driver support package `CYGPKG_IO_SERIAL` is enabled in the configuration.

Ethernet Driver

The STM3210C-EVAL board is fitted with an Ethernet port connected via a DP83848 PHY to the STM32's on-chip Ethernet MAC. This is supported in eCosPro with a driver for the lwIP networking stack, contained in the package `CYGPKG_DEVS_ETH_CORTEXM_STM32`. At the present time it only supports the lwIP networking stack, and cannot be used for either the BSD networking stack, nor RedBoot.

The driver will be inactive (not built and greyed out in the eCos Configuration Tool) unless the platform HAL option "STM32 Ethernet Support" (`CYGPKG_HAL_CORTEXM_STM32_STM3210C_EVAL_ETH0`) is enabled. This option in turn is only active if the "Common Ethernet support" (`CYGPKG_IO_ETH_DRIVERS`) package is included in your configuration. As the STM32 ethernet driver is an lwIP-only driver, it is most appropriate to choose the `lwip_eth` template as a starting point when choosing an eCos configuration, which will cause the necessary packages to be automatically included.

The STM32 ethernet driver defines a further configuration option "Use MCO as PHY clock" (`CYGHWR_DEVS_ETH_CORTEXM_STM32_PHY_CLK_MCO`) which indicates whether the MCO1 clock signal is used as the 25MHz clock for the Ethernet PHY. With this option disabled, the on-board 25MHz crystal oscillator located at X1 can be used instead, although in that case the board jumper JP4 must be changed. You may wish to make this change if you wish to use the MCO1 clock output for another purpose. Consult the STM32 clock and ethernet documentation for more details.

I²C Driver

The STM32 variant HAL provides the main I²C hardware driver itself, configured at `CYGPKG_HAL_STM32_I2C`. But the platform I²C support can also be configured separately at `CYGPKG_HAL_CORTEXM_STM32_STM3210C_EVAL_I2C`. This ensures that the M24C64 serial EEPROM is instantiated and becomes available for applications from `<cyg/io/i2c.h>`, and also ensures the STM32's I²C bus 1 is enabled for it. This option also allows the m24c64.c test is built. In order to allow writes to the serial EEPROM, you must ensure that the STM3210C-EVAL jumper JP17 is fitted. The m24c64.c test will not pass otherwise.

SPI Driver

An SPI bus driver is available for the STM32 in the package "ST STM32 SPI driver" (`CYGPKG_DEVS_SPI_CORTEXM_STM32`).

The only SPI device instantiated by default is for an external Aardvark SPI test board, connected to SPI bus 3 with the SD card attached to chip select 0 on PA4 and the Aardvark AT25080 to chip select 1 on PD2 (CN8 pin 24). To disable the Aardvark device support, the platform HAL contains an option "SPI devices" (`CYGPKG_HAL_CORTEXM_STM32_STM3210C_EVAL_SPI`) which can be disabled. No other SPI devices are instantiated.

Consult the generic SPI driver API documentation in the eCosPro Reference Manual for further details on SPI support in eCosPro, along with the configuration options in the STM32 SPI device driver.

ADC Driver

The STM32 processor variant HAL provides an ADC driver. The STM3210C-EVAL platform HAL enables the support for the devices ADC1, ADC2 and ADC3 and for configuration of the respective ADC device input channels.

Consult the generic ADC driver API documentation in the eCosPro Reference Manual for further details on ADC support in eCosPro, along with the configuration options in the STM32 ADC device driver.

Name

JTAG support — Usage

Use of JTAG for debugging

JTAG can be used to single-step and debug loaded applications, or even applications resident in ROM, including the Stubrom.

Debugging of ROM applications is only possible if using hardware breakpoints. The Cortex-M3 core of the STM32 only supports two such hardware breakpoints, so they should be used sparingly. If using a GDB front-end such as Eclipse, check it has not set unnecessary extra breakpoints. Some JTAG devices give the option of whether to set hardware or software breakpoints by default. Be sure to configure your device appropriately.

When debugging via JTAG, you are likely to need to disable the default HAL idle thread action, otherwise there may be issues where the target fails to halt and the debugging session is unreliable. More details can be found in the Cortex-M architectural HAL.

Ronetix PEEDI notes

On the Ronetix PEEDI, the `peedi.stm32.cfg` file should be used to setup and configure the hardware to an appropriate state to load programs. This includes setting up the clocks, GPIO lines, SRAM and flash memory controller.

The `peedi.stm32c.cfg` file also contains an option to define whether hardware or software breakpoints are used by default, using the `CORE0_BREAKMODE` directive in the `[TARGET]` section. The supplied version of the file defaults to software breakpoints. With this default, hardware breakpoints can still be set from GDB using the **hbreak** command. The default can be changed to hardware breakpoints, and remember to use the **reboot** command on the PEEDI command line interface, or press the reset button to make the changes take effect.

On the PEEDI, debugging can be performed either via the telnet interface or using **arm-eabi-gdb**. In the case of the latter, **arm-eabi-gdb** needs to connect to TCP port 2000 on the PEEDI's IP address. For example:

```
(gdb) target remote 111.222.333.444:2000
```

By default when the PEEDI is powered up, the target will always run the initialization section of the `peedi.stm32c.cfg` file, and halts the target. This behaviour is repeated with the **reset** command.

If the board is reset with the '**reset**' command, or by pressing the reset button and the '**go**' command is then given, then the board will boot from ROM as normal. A similar effect can be achieved in GDB by connecting with **target remote** and immediately typing **continue** or **c**.

It is also possible for the target to always run, without initialization, after the reset button has been pressed. This mode can be selected in two ways. Either with the `CORE0_STARTUP_MODE` directive in the `[TARGET]` section of the `peedi.stm32c.cfg` file. Or by using the **reset run** command. Either approach conveniently allows the target to be connected to the JTAG debugger, and be able to reset it with the reset button, without being required to always type '**go**' every time. Although using **reset run** will not be persistent across reboots of the PEEDI itself.

Suitably configured RAM applications can be loaded either via GDB, or directly via the telnet CLI. For example:

```
stm3210c> memory load tftp://192.168.7.9/test.bin bin 0x02000000
++ info: Loading image file:  tftp://192.168.7.9/test.bin
++ info: At absolute address: 0x20000400
loading at 0x20000400

Successfully loaded 23KB (24560 bytes) in 0.1s
stm3210c> go
```

Consult the PEEDI documentation for information on other formats and loading mechanisms.

Abatron BDI3000 notes

On the Abatron BDI3000, the `bdi3000.stm32.cfg` file should be used to setup and configure the hardware to an appropriate state to load programs. This includes setting up the clocks, GPIO lines, SRAM and flash memory controller.

The `bdi3000.stm32.cfg` file also contains an option to define whether hardware or software breakpoints are used by default, using the `BREAKMODE` directive in the `[TARGET]` section. The supplied version of the file defaults to software breakpoints. With this default, hardware breakpoints can still be set from GDB using the **hbreak** command. The default can be changed to hardware breakpoints, and remember to use the **boot** command on the BDI3000 command line interface.

On the BDI3000, debugging can be performed either via the telnet interface or using **arm-eabi-gdb**. In the case of the latter, **arm-eabi-gdb** needs to connect to TCP port 2001 on the BDI3000's IP address. For example:

```
(gdb) target remote 111.222.333.444:2001
```

By default when the BDI3000 is powered up, the target will always run the initialization section of the `bdi3000.stm32.cfg` file, and halts the target. This behaviour is repeated with the **reset** command.

If the board is reset with the '**reset**' command, or by pressing the reset button and the '**go**' command is then given, then the board will boot from ROM as normal. A similar effect can be achieved in GDB by connecting with **target remote** and immediately typing **continue** or **c**.

It is also possible for the target to always run, without initialization, after the reset button has been pressed. This mode is selected with the **reset run** command. This conveniently allows the target to be connected to the JTAG debugger, and be able to reset it with the reset button, without being required to always type '**go**' every time. Thereafter, invoking the **reset** command will repeat the previous reset style. Also in this mode, exceptions will be handled by board software, rather than causing the JTAG debugger to halt the CPU.

Suitably configured RAM applications can be loaded either via GDB, or directly via the telnet CLI. For example:

```
STM3210C> load 0x20000400 test.bin bin
Loading /test.bin , please wait ....
Loading program file passed
STM3210C> go 0x20000400
```

Consult the BDI3000 documentation for information on other formats and loading mechanisms.

Configuration of JTAG applications

JTAG applications can be loaded directly into RAM without requiring a ROM monitor. This loading can be done directly through the JTAG device, or where supported by the JTAG device, through GDB.

In order to configure the application to support this mode, some configuration settings are required, and a memory layout selected which will take advantage of the extra RAM space available. All necessary settings are made automatically if the JTAG startup type is selected.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the STM3210C-EVAL board hardware, and should be read in conjunction with that specification. The STM3210C-EVAL platform HAL package complements the ARM architectural HAL and the STM32 variant HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset the HAL will initialize or reinitialize many of the on-chip peripherals. There is an exception for RAM startup applications which depend on a ROM monitor for certain services.

For ROM and JTAG startup, the HAL will perform additional initialization, programming the various internal registers including the PLL, peripheral clocks and GPIO pins. The details of the early hardware startup may be found in the `src/stm3210c_eval_misc.c` in both `hal_system_init` and `hal_platform_init`.

Memory Map

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

Internal RAM

> This is located at address 0x20000000 of the memory space, and is 64KiB in size. The eCos VSR table occupies the bottom 512 bytes. The virtual vector table starts at 0x00000200 and extends to 0x00000300. The top `CYGNUM_HAL_COMMON_INTERRUPTS_STACK_SIZE` bytes are reserved for the interrupt stack. The remainder of internal RAM is available for use by applications.

Internal FLASH

> This is located at address 0x08000000 of the memory space and will be mapped to 0x00000000 at reset. This region is 512KiB in size. ROM applications are by default configured to run from this memory.

On-Chip Peripherals

> These are accessible at locations 0x40000000 and 0xE0000000 upwards. Descriptions of the contents can be found in the STM32 User Manual.

Linker Scripts

The platform linker scripts define the following symbols:

hal_vsr_table	This defines the location of the VSR table. This is set to 0x20000000 for all startup types, and space for 128 entries is reserved.
hal_virtual_vector_table	This defines the location of the virtual vector table used to communicate between an ROM monitor and an eCos application. This is allocated right after the VSR table, at 0x20000200. To permit expansion and possible addition of other tables, the linker scripts then allocate further sections from 0x20000400.
hal_interrupt_stack	This defines the location of the interrupt stack. For all startups, this is allocated to the top of internal SRAM, 0x20010000.

hal_startup_stack

This defines the location of the startup stack. For all startup types it is initially allocated at the half-way point of the interrupt stack.

Name

Test Programs — Details

Test Programs

The STM3210C platform HAL contains some test programs which allow various aspects of the board to be tested.

ADC test

This program tests the ADC driver for the STM32. The only device connected to the ADC on the board is the potentiometer connected to ADC1 logical channel 14. Therefore this test primarily tests that. However, in addition it also report the values of the temperature sensor and Vrefint inputs that are sourced on-chip. The option `CYGBLD_HAL_CORTEXM_STM3210C_EVAL_TEST_ADC` must be enabled to run this test since it needs human interaction.

M24C64 I²C serial EEPROM test

This program tests I²C access to the on-board M24C64 serial EEPROM. This test is only built if the CDL configuration option `CYGPKG_HAL_CORTEXM_STM32_STM3210C_EVAL_I2C` is enabled. The STM3210C-EVAL board prevents writes to the EEPROM unless jumper JP17 is closed. Therefore to allow this test to pass, ensure JP17 is fitted with a jumper.

Chapter 298. STM3210E-EVAL Platform HAL

Name

`CYGPKG_HAL_CORTEXM_STM32_STM3210E_EVAL` — eCos Support for the STM3210E-EVAL Board

Description

The STM3210E-EVAL board consists of a STM32F103Z microcontroller, 1MiB of external SRAM, 16MiB NOR flash and 64MiB NAND flash. It has connectors for two UARTs, I²C, MicroSD, USB, CAN, JTAG and various other devices. Early versions of the board contain STM32F103ZET6 MCUs and later boards from RevD-03 onwards are populated with the STM32F103ZGT6. The former has 512KiB of internal flash and the latter 1MiB.

For typical eCos development, RedBoot or a GDB stub image is programmed into internal FLASH and the CPU boots directly into that. It is then possible to download and debug stand-alone and eCos applications via the gdb debugger using UART0. Alternatively test programs may be downloaded and debugged via a JTAG debugger attached to the JTAG socket.

This documentation describes platform-specific elements of the STM3210E-EVAL board support within eCos. The STM32 variant HAL documentation covers various topics including HAL support common to STM32 variants, and on-chip device support. This document complements the STM32 documentation.

We have found that some revisions of the STM3210E-EVAL board and its attached LCD are unreliable unless the LCD is removed. If the board suffers from lockups or other reliability issues then we recommend disconnecting the LCD to determine if that is the cause.

Supported Hardware

The STM32 has two on-chip memory regions. A RAM region of 64KiB is present at 0x20000000. A FLASH region is present at 0x08000000 and is aliased to 0x0000000 during normal execution. On-board memory consists of 1MiB of SRAM mapped to 0x68000000 and 16MiB of NOR FLASH mapped to 0x64000000.

The STM32 variant HAL includes support for the five on-chip serial devices which are documented in the variant HAL. UART0 and UART1 are connected to external connectors on the board marked "CN12" and "CN8" respectively. Only UART1 on CN8 is equipped with RTS/CTS lines.

Device drivers are provided for the STM32 on-chip ADC interfaces, I²C interface, and SPI interface. Additionally, support is provided for the on-chip watchdog, RTC (wallclock) and a Flash driver exists to permit management of the STM32's on-chip Flash.

Due to a silicon errata with the STM32F103Zx device the I2C1 bus cannot be used if FSMC is enabled. Therefore the I²C test example for the on-board STLM75 Temperature sensor explicitly disables FSMC prior to accessing I2C1.

Also, while the board is fitted with a CAN interface, this is not presently supported by the HAL port. The STM32F1 processor and the STM3210E-EVAL board provide an extremely wide variety of peripherals, but unless support is specifically indicated, it should be assumed that it is not included.

Tools

The board port is intended to work with GNU tools configured for an **arm-eabi** target. The original port was done using **arm-eabi-gcc** version 4.3.2, **arm-eabi-gdb** version 6.8, and **binutils** version 2.18.

Name

Setup — Preparing the STM3210E-EVAL Board for eCos Development

Overview

In a typical development environment, the STM3210E-EVAL board boots from internal flash into either the GDB stubrom or RedBoot. eCos applications are configured for RAM startup and then downloaded and run on the board via the debugger **arm-eabi-gdb**.

RedBoot Installation

The following RedBoot configurations are supported:

Configuration	Description	Use	File
ROM	RedBoot running from internal FLASH	`redboot_ROM.ecm`	`redboot_ROM.bin`
RAM	RedBoot running from external RAM	`redboot_RAM.ecm`	`redboot_RAM.bin`
JTAG	RedBoot running from external RAM, loaded via JTAG	`redboot_JTAG.ecm`	`redboot_JTAG.bin`

For serial communications, all versions run with 8 bits, no parity, and 1 stop bit at 115200 baud. This rate can be changed using the RedBoot **baud** command.

Under normal circumstances, RedBoot runs in-place from the internal Flash. The RAM version is provided to allow for updating the resident RedBoot image in Flash. The JTAG version is only used if loading RedBoot into RAM via a JTAG debugger. It is similar to the RAM version, but loads at a lower address within RAM, and so can be used to load eCos applications, as if it is the normal resident boot monitor. The ELF format image of this JTAG version of RedBoot can also be loaded and executed from GDB using a JTAG device, to allow it to be debugged.

Programming RedBoot

To program RedBoot into the internal flash either a JTAG debugger that understands the STM32 flash may be used, such as a Ronetix PEEDI or an Abatron BDI3000, or the ST **Flash Loader Demonstrator** may be used. Configuration files for the PEEDI and BDI3000 are supplied in the STM3210E-EVAL HAL package, and brief instructions for downloading RedBoot are given in there. If no JTAG debugger is available, then RedBoot must be downloading using the **Flash Loader Demonstrator** which can be downloaded from the page for the STM32F101ZE CPU, in the *Design Support->Demo SW* section of ST's website at http://www.st.com/internet/mcu/product/164506.jsp

The documentation for this may be found at http://www.st.com/internet/com/TECHNICAL_RESOURCES/TECHNICAL_LITERATURE/USER_MANUAL/CD00171488.pdf

The following are brief instructions for downloading RedBoot using the **Flash Loader Demonstrator**.

1. Download the **Flash Loader Demonstrator** from the ST website and install it on a PC running Windows that has an available serial port.

2. Copy the file `redboot_ROM.bin` from the `loaders/stm3210e_eval` sub-directory within your eCosPro installation to a suitable location on the Windows PC.

3. Connect a null-modem serial cable between the USART1 serial port and a serial port on the Windows machine.

4. Move the *BOOT0/SW2* switch to the 1 position, ensure the *BOOT1/SW2* switch remains in the 0 position, and press the reset button.

5. Start the **Flash Loader Demonstrator** and in the first screen select the COM port connected to the board under *Port Name*. Ensure that the baud rate is also set at *115200*, with *Even* parity and *Echo* disabled. Press *Next*. If you are succesfully connected to the board, you should see the message *Target is readable. Please press "Next" to proceed* . It should also note the size of the internal flash memory. Press *Next* and you will be prompted to select a target. Depending on the CPU model on your STM3210E-EVAL board and the size of its internal flash, either select the *STM32_High-density_512K* or *STM32_XL-density_1024K*, then *Next* to go to the Operation choice page. If any of these steps fail, follow the direction given by the loader to recover.

6. On the operation choice page select *"Download to device"* and under *"Download from file"* either type the location of the `redboot_ROM.bin` file, or browse to it. Ensure that the *"@"* field is set to *8000000*, the *"Global Erase"* radio button is selected and that all other options are clear except *"Verify after download"*.

7. Press *"Next"* and the loader should download and verify the binary file. The download or verify may fail if the flash has been previously locked, in which case you should select the *"Enable/Disable Flash protection"* radio button, *"Disable"* and *"Write Protection"* drop-down items, press *Next* and retry the download. Upon successful download press *"Close"* to exit the loader.

8. Move the *BOOT0/SW2* switch back to the 0 position and press the reset button.

The behaviour of the **Flash Loader Demonstrator** documented here is that of version 2.5. The actual behaviour of newer or older versions may vary slightly.

Whatever mechanism is used to program RedBoot, something similar to the following output should be seen on USART1 when the reset button is pressed:

```
RedBoot(tm) bootstrap and debug environment [ROM]
eCosCentric certified release, version v3_1_25 - built 15:36:19, May  8 2012

Copyright (C) 2000-2009 Free Software Foundation, Inc.
Copyright (C) 2003-2012 eCosCentric Limited
RedBoot is free software, covered by the eCos license, derived from the
GNU General Public License. You are welcome to change it and/or distribute
copies of it under certain conditions. Under the license terms, RedBoot's
source code and full license terms must have been made available to you.
Redboot comes with ABSOLUTELY NO WARRANTY.

Platform: ST STM3210E EVAL (Cortex-M3)
RAM: 0x68000000-0x68100000 [0x680036d0-0x680dd000 available]
     0x20000000-0x2000f000 [0x20000000-0x2000f000 available]
FLASH: 0x08000000-0x0807ffff, 256 x 0x800 blocks
FLASH: 0x64000000-0x64ffffff, 128 x 0x20000 blocks
RedBoot>
```

Initializing RedBoot Flash Configuration

While RedBoot is loaded into the on-chip flash, it manages the external flash for the storage of application programs and configuration data. The flash needs to be initialized with the following commands.

```
RedBoot> fis init
About to initialize [format] FLASH image system - continue (y/n)? y
*** Initialize FLASH Image System
... Erase from 0x64fe0000-0x64ffffff: .
... Program from 0x680e0000-0x68100000 to 0x64fe0000: .
RedBoot> fconfig -i
Initialize non-volatile configuration - continue (y/n)? y
Run script at boot: false
Console baud rate: 115200
Update RedBoot non-volatile configuration - continue (y/n)? y
... Erase from 0x64fe0000-0x64ffffff: .
... Program from 0x680e0000-0x68100000 to 0x64fe0000: .
```

```
RedBoot>
```

Rebuilding RedBoot

Should it prove necessary to rebuild a RedBoot binary, this is done most conveniently at the command line. The steps needed to rebuild the the ROM version of RedBoot are:

```
$ mkdir redboot_stm3210e_rom
$ cd redboot_stm3210e_rom
$ ecosconfig new stm3210e redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/cortexm/stm32/stm3210e_eval/VERSION/misc/redboot_ROM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

At the end of the build the `install/bin` subdirectory should contain the file `redboot.bin`.

The other versions of RedBoot - RAM or JTAG - may be similarly built by choosing the appropriate alternative .ecm file.

Name

Configuration — Platform-specific Configuration Options

Overview

The STM3210E-EVAL board platform HAL package is loaded automatically when eCos is configured for an `stm3210e_eval` target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The STM3210E-EVAL board platform HAL package supports five separate startup types:

RAM

> This is the startup type which is normally used during application development. The board has RedBoot programmed into internal Flash at location 0x08000000 and uses external RAM at location 0x68000000. arm-eabi-gdb is then used to load a RAM startup application into memory from 0x68008000 and debug it. It is assumed that the hardware has already been initialized by RedBoot. By default the application will use the eCos virtual vectors mechanism to obtain services from RedBoot, including diagnostic output.

ROM

> This startup type can be used for finished applications which will be programmed into internal ROM at location 0x08000000. Data and BSS will be put into external RAM starting from 0x68000000. The application will be self-contained with no dependencies on services provided by other software. The program expects to boot from reset with ROM mapped at location zero. It will then transfer control to the 0x08000000 region. eCos startup code will perform all necessary hardware initialization.

ROMINT

> This startup type can be used for finished applications which will be programmed into internal ROM at location 0x08000000. Data and BSS will be put into internal SRAM starting from 0x20000400. The application will be self-contained with no dependencies on services provided by other software. The program expects to boot from reset with ROM mapped at location zero. It will then transfer control to the 0x08000000 region. eCos startup code will perform all necessary hardware initialization.

JTAG

> This is the startup type used to build applications that are loaded via a JTAG interface. The application will be self-contained with no dependencies on services provided by other software. The program expects to be loaded from 0x68000000 and entered at that address. eCos startup code will perform all necessary hardware initialization.

SRAM

> This is a variation of the JTAG type that only uses internal memory. The application will be self-contained with no dependencies on services provided by other software. The program expects to be loaded from 0x20000400 and entered at that address. eCos startup code will perform all necessary hardware initialization.

RedBoot and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building RedBoot.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for

a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then serial port 0 will be claimed for HAL diagnostics.

UART Serial Driver

The STM3210E-EVAL board uses the STM32's internal UART serial support. As well as the polled HAL diagnostic interface, there is also a `CYGPKG_IO_SERIAL_CORTEXM_STM32` package which contains all the code necessary to support interrupt-driven operation with greater functionality. All five UARTs can be supported by this driver, although only UARTs 0 and 1 are actually routed to external connectors. Note that it is not recommended to enable this driver on the port used for HAL diagnostic I/O. This driver is not active until the `CYGPKG_IO_SERIAL_DEVICES` configuration option within the generic serial driver support package `CYGPKG_IO_SERIAL` is enabled in the configuration.

SPI Driver

An SPI bus driver is available for the STM32 in the package "ST STM32 SPI driver" (`CYGPKG_DEVS_SPI_CORTEXM_STM32`).

The only SPI device present on the board is the M25PXX device connected to SPI bus 1 using PB2 as the chip-select. To enable support for this device the platform HAL "External SPI M25PXX flash support" (`CYGHWR_HAL_CORTEXM_STM32_FLASH_M25PXX`) option should be enabled.

Consult the generic SPI driver API documentation in the eCosPro Reference Manual for further details on SPI support in eCosPro, along with the configuration options in the STM32 SPI device driver.

I²C Driver

The STM32 processor variant HAL provides an I²C driver. The STM3210E-EVAL platform HAL enables the support for bus 1 with the on-board STLM75 temperature sensor instantiated by default. It is exported to `<cyg/io/i2c.h>` with the name `hal_stm32_i2c_temperature` in the normal way.

Consult the generic I²C driver API documentation in the eCosPro Reference Manual for further details on I²C support in eCosPro, along with the configuration options in the STM32 on-chip I²C driver.

ADC Driver

The STM32 processor variant HAL provides an ADC driver. The STM3210E-EVAL platform HAL enables the support for the devices ADC1, ADC2 and ADC3 and for configuration of the respective ADC device input channels.

Consult the generic ADC driver API documentation in the eCosPro Reference Manual for further details on ADC support in eCosPro, along with the configuration options in the STM32 ADC device driver.

Onboard NAND

The HAL port includes a low-level driver to access the on-board NAND flash memory chip. To enable the driver, ensure that the `CYGPKG_DEVS_NAND_ST_NANDXXXX3A` and `CYGPKG_IO_NAND` packages are present in your eCos configuration. The driver is capable of operating with the NAND_RB jumper (JP7) in either position.

`CYGHWR_HAL_CORTEXM_STM3210E_EVAL_RB_ON_INT2`

Jumper 7 on the board - labelled NAND_RB - controls the wiring of the NAND chip's Ready/Busy line. You must set this option correspondingly. If the jumper is in position 1-2, this option must be 0; in position 2-3, this option must be 1. This setting changes the behaviour of the driver when waiting for slow NAND operations to complete. In position 1-2, the memory controller halts accesses until the chip is ready; in 2-3, memory access is unimpeded but the CPU polls for the chip to signal ready before attempting to read from it. The latter case may yield an efficiency boost in multi-threaded applications.

Partitioning the NAND chip

The NAND chip must be partitioned before it can become available to applications.

A CDL script which allows the chip to be manually partitioned is provided (see `CYGSEM_DEVS_NAND_STM3210E_EVAL_PARTITION_MANUAL_CONFIG`); if you choose to use this, the relevant data structures will automatically be set up for you when the device is initialised. By default, the manual config CDL script sets up a single partition (number 0) encompassing the entire device.

It is possible to configure the partitions in some other way, should it be appropriate for your setup. To do so you will have to add appropriate code to `stm3210e_eval_nand.c`.

Name

JTAG support — Usage

Use of JTAG for debugging

JTAG can be used to single-step and debug loaded applications, or even applications resident in ROM, including RedBoot.

Debugging of ROM applications is only possible if using hardware breakpoints. The Cortex-M3 core of the STM32 only supports two such hardware breakpoints, so they should be used sparingly. If using a GDB front-end such as Eclipse, check it has not set unnecessary extra breakpoints. Some JTAG devices give the option of whether to set hardware or software breakpoints by default. Be sure to configure your device appropriately.

When debugging via JTAG, you are likely to need to disable the default HAL idle thread action, otherwise there may be issues where the target fails to halt and the debugging session is unreliable. More details can be found in the Cortex-M architectural HAL.

Ronetix PEEDI notes

On the Ronetix PEEDI, the `peedi.stm32e.cfg` file, supplied in the platform HAL package's misc directory, should be used to setup and configure the hardware to an appropriate state to load programs. This includes setting up the clocks, GPIO lines, SRAM and flash memory controller.

The `peedi.stm32e.cfg` file also contains an option to define whether hardware or software breakpoints are used by default, using the `CORE0_BREAKMODE` directive in the `[TARGET]` section. The supplied version of the file defaults to software breakpoints. With this default, hardware breakpoints can still be set from GDB using the **hbreak** command. The default can be changed to hardware breakpoints, and remember to use the **reboot** command on the PEEDI command line interface, or press the reset button to make the changes take effect.

On the PEEDI, debugging can be performed either via the telnet interface or using **arm-eabi-gdb**. In the case of the latter, **arm-eabi-gdb** needs to connect to TCP port 2000 on the PEEDI's IP address. For example:

```
(gdb) target remote 111.222.333.444:2000
```

By default when the PEEDI is powered up, the target will always run the initialization section of the `peedi.stm32e.cfg` file, and halts the target. This behaviour is repeated with the **reset** command.

If the board is reset with the '**reset**' command, or by pressing the reset button and the '**go**' command is then given, then the board will boot from ROM as normal. A similar effect can be achieved in GDB by connecting with **target remote** and immediately typing **continue** or **c**.

It is also possible for the target to always run, without initialization, after the reset button has been pressed. This mode is selected with the `CORE0_STARTUP_MODE` directive in the `[TARGET]` section of the `peedi.stm32e.cfg` file. This conveniently allows the target to be connected to the JTAG debugger, and be able to reset it with the reset button, without being required to always type '**go**' every time.

Suitably configured RAM applications can be loaded either via GDB, or directly via the telnet CLI. For example:

```
stm32> memory load tftp://192.168.7.9/test.bin bin 0x68000000
++ info: Loading image file:  tftp://192.168.7.9/test.bin
++ info: At absolute address: 0x68000000
loading at 0x68000000
loading at 0x68004000

Successfully loaded 28KB (29064 bytes) in 0.1s
stm32> go 0x68000000
```

Consult the PEEDI documentation for information on other formats and loading mechanisms.

Abatron BDI3000 notes

On the Abatron BDI3000, the `bdi3000.stm32e.cfg` file should be used to setup and configure the hardware to an appropriate state to load programs. This includes setting up the clocks, GPIO lines, SRAM and flash memory controller.

The `bdi3000.stm32e.cfg` file also contains an option to define whether hardware or software breakpoints are used by default, using the `BREAKMODE` directive in the `[TARGET]` section. The supplied version of the file defaults to software breakpoints. With this default, hardware breakpoints can still be set from GDB using the **hbreak** command. The default can be changed to hardware breakpoints, and remember to use the **boot** command on the BDI3000 command line interface.

On the BDI3000, debugging can be performed either via the telnet interface or using **arm-eabi-gdb**. In the case of the latter, **arm-eabi-gdb** needs to connect to TCP port 2001 on the BDI3000's IP address. For example:

```
(gdb) target remote 111.222.333.444:2001
```

By default when the BDI3000 is powered up, the target will always run the initialization section of the `bdi3000.stm32e.cfg` file, and halts the target. This behaviour is repeated with the **reset** command.

If the board is reset with the '**reset**' command, or by pressing the reset button and the '**go**' command is then given, then the board will boot from ROM as normal. A similar effect can be achieved in GDB by connecting with **target remote** and immediately typing **continue** or **c**.

It is also possible for the target to always run, without initialization, after the reset button has been pressed. This mode is selected with the **reset run** command. This conveniently allows the target to be connected to the JTAG debugger, and be able to reset it with the reset button, without being required to always type '**go**' every time. Thereafter, invoking the **reset** command will repeat the previous reset style. Also in this mode, exceptions will be handled by board software, rather than causing the JTAG debugger to halt the CPU.

Suitably configured RAM applications can be loaded either via GDB, or directly via the telnet CLI. For example:

```
STM32> load 0x68000000 test.bin bin
Loading /test.bin , please wait ....
Loading program file passed
STM32> go 0x68000000
```

Consult the BDI3000 documentation for information on other formats and loading mechanisms.

Configuration of JTAG applications

JTAG applications can be loaded directly into RAM without requiring a ROM monitor. This loading can be done directly through the JTAG device, or where supported by the JTAG device, through GDB.

In order to configure the application to support this mode, some configuration settings are required. Firstly `CYGSEM_HAL_USE_ROM_MONITOR` must be disabled. Secondly the `CYGDBG_HAL_DIAG_TO_DEBUG_CHAN` option should be enabled in order to prevent HAL diagnostic output being encoded into GDB ($O) packets. Both of these settings are made automatically if the JTAG startup type is selected.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the STM3210E-EVAL board hardware, and should be read in conjunction with that specification. The STM3210E-EVAL platform HAL package complements the ARM architectural HAL and the STM32 variant HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset the HAL will initialize or reinitialize many of the on-chip peripherals. There is an exception for RAM startup applications which depend on a ROM monitor for certain services.

For ROM and JTAG startup, the HAL will perform additional initialization, programming the various internal registers including the PLL, peripheral clocks and GPIO pins. The details of the early hardware startup may be found in the `src/stm3210e_eval_misc.c` in both `hal_system_init` and `hal_platform_init`.

Memory Map

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

External RAM

> This is located at address 0x68000000 of the memory space, and is 1MiB long. For ROM applications, all of RAM is available for use. For RAM startup applications, RAM below 0x68008000 is reserved for RedBoot and the top `CYGNUM_HAL_COMMON_INTERRUPTS_STACK_SIZE` bytes are reserved for the interrupt stack, the remainder is available for the application.

External FLASH

> This is located at address 0x64000000 of the memory space and is 4MiB in size. Since RedBoot is normally programmed into the internal flash this memory is entirely available for application use and may be managed by the FIS flash file system.

Internal RAM

> This is located at address 0x20000000 of the memory space, and is 64KiB in size. The eCos VSR table occupies the bottom 512 bytes. The virtual vector table starts at 0x00000100 and extends to 0x00000200. For ROM and JTAG startups, the top `CYGNUM_HAL_COMMON_INTERRUPTS_STACK_SIZE` bytes are reserved for the interrupt stack. The remainder of internal RAM is available for use by applications.

Internal FLASH

> This is located at address 0x08000000 of the memory space and will be mapped to 0x00000000 at reset. This region is 512KiB in size. ROM applications are by default configured to run from this memory. This memory is not managed by RedBoot's FIS system, but it can be written using the **fis write** command and erased using the **fis erase** command.

On-Chip Peripherals

> These are accessible at locations 0x40000000 and 0xE0000000 upwards. Descriptions of the contents can be found in the STM32 User Manual.

Linker Scripts

The platform linker scripts define the following symbols:

hal_vsr_table

This defines the location of the VSR table. This is set to 0x20000000 for all startup types, and space for 128 entries is reserved.

hal_virtual_vector_table

This defines the location of the virtual vector table used to communicate between an ROM monitor and an eCos application. This is allocated right after the VSR table, at 0x20000100. To permit expansion and possible addition of other tables, the linker scripts then allocate further sections from 0x20000400.

hal_interrupt_stack

This defines the location of the interrupt stack. For ROM and JTAG startups, this is allocated to the top of internal SRAM, 0x20010000. For RAM startups, it is allocated to the top of external SRAM, 0x68100000.

hal_startup_stack

This defines the location of the startup stack. For all startup types it is initially allocated at the half-way point of the interrupt stack.

Name

Test Programs — Details

Test Programs

The STM3210E platform HAL contains some test programs which allow various aspects of the board to be tested.

ADC Test

This program tests the ADC driver for the STM32. The only device connected to the ADC on the board is the potentiometer connected to ADC1 logical channel 14. Therefore this test primarily tests that. However, in addition it also report the values of the temperature sensor and Vrefint inputs that are sourced on-chip. The option `CYGBLD_HAL_CORTEXM_STM3210E_EVAL_TEST_ADC` must be enabled to run this test since it needs human interaction.

SPI flash test

This program tests the MPC25Pxx serial flash connected to SPI bus 1. It erases, programs and reads a number of sectors in the flash, and should therefore not be run if the flash contains data that should be retained. The `CYGPKG_IO_FLASH` package must be present and `CYGHWR_HAL_CORTEXM_STM32_FLASH_M25PXX` enabled to allow this test to be built.

I²C Test

This program tests the I²C driver for the STM32. The only device connected to I²C on the board is the STLM75 temperature sensor on I2C1. The option `CYGBLD_HAL_CORTEXM_STM3210E_EVAL_TEST_STLM75` must be enabled to run this test.

NAND ECC walk test

This program is an adapted version of the `sweccwalk` test in the NAND Flash library which tests the hardware ECC support provided by the on-chip FSMC. It does not affect any data currently stored on the on-board NAND flash chip.

Chapter 299. STM32X0G-EVAL Platform HAL

Name

`CYGPKG_HAL_CORTEXM_STM32_STM32X0G_EVAL` — eCos Support for the STM3220G-EVAL, STM3240G-EVAL and STM3241G-EVAL Boards

Description

There are currently three variants of ST's STM32 "G" evaluation board, the STM3220G-EVAL. STM3240G-EVAL and STM3241G-EVAL. This documentation uses "STM32X0G-EVAL" to refer generically to these boards. The boards have "STM32 20-21-45-46 G-EVAL" silk screened on the PCB, with the specific board model only identifiable by the CPU markings.

The STM3220G-EVAL board has a STM32F207IG microcontroller that incorporates 1MiB of internal flash and 128KiB of internal SRAM.

The STM3240G-EVAL board has a STM32F407IG microcontroller that incorporates 1MiB of internal flash and 192KiB of internal SRAM.

The STM3241G-EVAL board has a STM32F417IG microcontroller that incorporates 1MiB of internal flash and 192KiB of internal SRAM.

The boards have an additional 2MiB of external SRAM, and connectors for UART, Ethernet, MicroSD, USB, CAN, JTAG and various other devices.

For typical eCos development, RedBoot or a GDB stub image is programmed into internal FLASH and the CPU boots directly into that. It is then possible to download and debug stand-alone and eCos applications via the gdb debugger using UART4. Alternatively test programs may be downloaded and debugged via a JTAG debugger attached to the JTAG socket.

This documentation describes platform-specific elements of the STM32X0G-EVAL board support within eCos. The STM32 variant HAL documentation covers various topics including HAL support common to STM32 variants, and on-chip device support. This document complements the STM32 documentation.

Supported Hardware

The STM32F207IG has two on-chip memory regions, with the STM32F407IG and STM32F417IG having three on-chip memory regions. The F2 and F4 devices have a RAM region of 128KiB present at 0x20000000 and a 1MiB FLASH region present at 0x08000000 which is aliased to 0x00000000 during normal execution. The STM32F407IG and STM32F417IG devices have another RAM region of 64KiB present at 0x10000000. For all platform variants on-board memory consists of 2MiB of SRAM mapped to 0x64000000.

The STM32 variant HAL includes support for the six on-chip serial devices which are documented in the variant HAL. UART4 is connected to the external connector on the board marked "CN16". There is no connection for hardware flow control (RTS/CTS) lines.

The STM32 variant HAL also includes support for the I²C bus. A single I²C device is instantiated as part of the platform port, which is for the M24C64 serial EEPROM connected via I²C. It is exported to `<cyg/io/i2c.h>` with the name `hal_stm32_i2c_eeprom` in the normal way.

Device drivers are provided for the STM32 on-chip Ethernet MAC, ADC interfaces, I²C interface, SPI interface and SDIO interface. Additionally, support is provided for the on-chip watchdog, RTC (wallclock) and a Flash driver exists to permit management of the STM32's on-chip Flash.

The STM32 variant HAL support for the SDIO interface is currently limited to supporting MMC/SD cards. Due to the GPIO availability on the platform by default only 1-bit MMC/SD data accesses are enabled, since 4-bit data access would clash with the UART4 debug channel. If 4-bit mode is required then the platform configuration would need to be changed to avoid the use of UART3/UART4. For 4-bit mode the use of on-chip SRAM for the transfer buffers is recommended to avoid RX overrun and TX underrun errors due to the slow external RAM access speed.

A driver is available for the BXCAN devices present on the chip. Note that BXCAN1 is incompatible with the use of off-chip SRAM. Under normal circumstances ensure jumpers JP3 and JP10 are not fitted so that neither CAN device is enabled, or fitted to enable CAN2 only. Also that JP1 and JP2 are set with pins 2-3 connected to allow SRAM use. These are the factory-supplied defaults for these jumpers.

Also, while the board is fitted with OneNAND Flash, this is not presently supported by the HAL port. The STM32F2 processor and the STM32X0G-EVAL board provide an extremely wide variety of peripherals, but unless support is specifically indicated, it should be assumed that it is not included.

Tools

The board port is intended to work with GNU tools configured for an **arm-eabi** target. The original port was done using **arm-eabi-gcc** version 4.4.5, **arm-eabi-gdb** version 7.2, and **binutils** version 2.20.1.

Name

Setup — Preparing the STM32X0G-EVAL Board for eCos Development

Overview

In a typical development environment, the STM32X0G-EVAL board boots from internal flash into either the GDB stubrom or RedBoot. eCos applications are configured for RAM startup and then downloaded and run on the board via the debugger **arm-eabi-gdb**, or via the Eclipse IDE.

RedBoot Installation

The following RedBoot configurations are supported:

Configuration	Description	Use	File
ROM	RedBoot running from internal FLASH	`redboot_ROM.ecm`	`redboot_ROM.bin`
RAM	RedBoot running from external RAM	`redboot_RAM.ecm`	`redboot_RAM.bin`
JTAG	RedBoot running from external RAM, loaded via JTAG	`redboot_JTAG.ecm`	`redboot_JTAG.elf`

For serial communications, all versions run with 8 bits, no parity, and 1 stop bit at 115200 baud. This rate can be changed using the RedBoot **baud** command, or in the eCos configuration used for building RedBoot.

Under normal circumstances, RedBoot runs in-place from the internal Flash. The RAM version is provided to allow for updating the resident RedBoot image in Flash. The JTAG version is only used if loading RedBoot into RAM via a JTAG debugger. It is similar to the RAM version, but loads at a lower address within RAM. The ELF format image of this JTAG version of RedBoot can also be loaded and executed from GDB using a JTAG device, to allow it to be debugged.

Programming RedBoot with ST Flash Loader Demonstrator

To program RedBoot into the internal flash either a JTAG debugger that understands the STM32 flash may be used, such as a Ronetix PEEDI , or the ST **Flash Loader Demonstrator** may be used. Configuration files for the PEEDI have been supplied in the STM32X0G-EVAL HAL package. If no JTAG debugger is available, then RedBoot must be downloaded using the **Flash Loader Demonstrator** which can be downloaded from the page for the STM32F207IG CPU, in the *Design Support->Software & Development Tools->Software Demos* section of ST's website at http://www.st.com/internet/mcu/product/245085.jsp#SOFTWARE_AND_DEVELOPMENT_TOOLS. It must be at least version 2.4.0 in order to support the STM32F2xx devices. At time of writing, a direct link to this version is available at http://www.st.com/internet/com/SOFTWARE_RESOURCES/SW_COMPONENT/SW_DEMO/stm32-stm8_flash_loader_demo.zip.

The following are brief instructions for downloading RedBoot using the **Flash Loader Demonstrator**. For more complete usage instructions, consult the ST documentation for this tool at http://www.st.com/internet/com/TECHNICAL_RESOURCES/TECHNICAL_LITERATURE/USER_MANUAL/CD00171488.pdf

1. Download the **Flash Loader Demonstrator** from the ST website and install it on a PC running Windows that has an available RS232 serial port, or with a USB to RS232 adaptor.

2. Copy the file `redboot_ROM.bin` from the `loaders/stm32x0g_eval` sub-directory within your eCosPro installation to a suitable location on the Windows PC.

3. Connect a null-modem serial cable between the USART serial port (CN16) and the RS232 serial port on the Windows machine.

4. Move the *BOOT0/SW2* switch to the 1 position, ensuring the *BOOT1/SW1* switch remains in the 0 position, and press the reset button.

5. Start the **Flash Loader Demonstrator** and in the first screen select the COM port connected to the board under *Port Name*. Ensure that the baud rate is also set at *115200*, with *Even* parity, *Echo* disabled and *Flow control* off. Press *Next*. If you are succesfully connected to the board, you should see the message *Target is readable. Please click "Next" to proceed* . Press *Next* and you will be prompted to select a target. Select *STM32F2_1024K*, then *Next* to go to the Operation choice page. If any of these steps fail, follow the direction given by the loader to recover.

6. On the operation choice page select *"Download to device"* and under *"Download from file"* either type the location of the `redboot_ROM.bin` file, or browse to it. Ensure that the *"@"* field is set to *8000000*, the *"Global Erase"* radio button is selected and that all other options are clear except *"Verify after download"*.

7. Press *"Next"* and the loader should download and verify the binary file. The download or verify may fail if the flash has been previously locked, in which case you should select the *"Enable/Disable Flash protection"* radio button, *"Disable"* and *"Write Protection"* drop-down items, press *Next* and retry the download. Upon successful download press *"Close"* to exit the loader.

8. Move the *BOOT0/SW2* switch back to the 0 position and press the reset button.

The behaviour of the **Flash Loader Demonstrator** documented here is that of version 2.4.0. The actual behaviour of newer versions may vary slightly.

Whatever mechanism is used to program RedBoot, something similar to the following output should be seen on the RS232 serial port when the reset button is pressed:

```
+**Warning** FLASH configuration checksum error or invalid key
Use 'fconfig -i' to [re]initialize database

RedBoot(tm) bootstrap and debug environment [ROM]
eCosCentric certified release, version v3_1_X - built 04:32:11, Feb  3 2012

Copyright (C) 2000-2009 Free Software Foundation, Inc.
Copyright (C) 2003-2012 eCosCentric Limited
RedBoot is free software, covered by the eCos license, derived from the
GNU General Public License. You are welcome to change it and/or distribute
copies of it under certain conditions. Under the license terms, RedBoot's
source code and full license terms must have been made available to you.
Redboot comes with ABSOLUTELY NO WARRANTY.

Platform: ST STM3220G EVAL (Cortex-M3)
RAM: 0x64000000-0x64200000 [0x64003668-0x641dd000 available]
     0x20000000-0x2001f000 [0x20000000-0x2001f000 available]
FLASH: 0x08000000-0x080fffff, 4 x 0x4000 blocks, 1 x 0x10000 blocks, 7 x 0x20000 blocks
RedBoot>
```

You should now proceed to the section describing how to initialize RedBoot's flash configuration.

Programming RedBoot with Ronetix PEEDI

This section describes how to install RedBoot using a Ronetix PEEDI JTAG debugger.

The PEEDI must be configured to allow communication with your local network, and configured with the parameters for interfacing with the target board. It must then be used to download and program a RedBoot image into the internal flash. The following steps give a typical outline for doing this. Consult the PEEDI documentation for alternative approaches, such as using FTP or HTTP instead of TFTP.

Preparing the Ronetix PEEDI JTAG debugger

1. Prepare a PC to act as a host and start a TFTP server on it.

2. Connect the PEEDI JTAG debugger via both serial and ethernet to the host PC and power it on. Use the serial cable supplied with the PEEDI (straight through, not null modem).

3. Verify the PEEDI is using up-to-date firmware, of version 11.10.1 or later. Older PEEDI firmware does not support the STM32 F2 family correctly, particularly if wishing to use the PEEDI's own '**flash**' commands to modify the on-chip Flash. If the firmware is not recent enough, follow the PEEDI User Manual's instructions which describe how to update the PEEDI firmware.

4. Locate the appropriate file `peedi.stm3220g.cfg` (STM3220G-EVAL) or `peedi.stm3240g.cfg` (STM3240G-EVAL and STM3241G-EVAL) within the eCos platform HAL package in the source repository. This will be in the directory `packages/hal/cortexm/stm32/stm32x0g/`*`VERSION`*`/misc` relative to the root of your eCos installation.

5. Place the respective configuration file in a location on the PC accessible to the TFTP server. Later you will configure the PEEDI to load this file via TFTP as its configuration file.

6. Open `peedi.stm3220g.cfg` or `peedi.stm3240g.cfg` (as appropriate) in an editor such as emacs or notepad and insert your own license information in the `[LICENSE]` section.

7. Install and configure the PEEDI in line with the PEEDI Quick Start Guide or User's Manual, especially configuring PEEDI's RedBoot with the network information. Configure it to use the appropriate `peedi.stm3220g.cfg` or `peedi.stm3240g.cfg` target configuration file on the TFTP server at the appropriate point of the **config** process, for example with a path such as: `tftp://192.168.7.9/peedi.stm3220g.cfg`

8. Reset the PEEDI.

9. Connect to the PEEDI's CLI interface via TCP/IP on the standard telnet port 23. The **telnet** application is suitable for this. You should see output similar to the following:

```
$ telnet 192.168.7.225
Trying 192.168.7.225...
Connected to 192.168.7.225.
Escape character is '^]'.

PEEDI - Powerful Embedded Ethernet Debug Interface
Copyright (c) 2005-2011 www.ronetix.at - All rights reserved
Hw:1.2, L:JTAG v1.5 Fw:11.10.1, SN: PD-0000-XXXX-XXXX
------------------------------------------------------------

stm3220g>
```

Preparing the STM32X0G board for programming with PEEDI

Follow the steps in this section in order to allow communication between the board and the host PC, and between the board and the JTAG device.

1. Connect a null modem DB9 RS232 serial cable between the serial port on the board and a serial port on the host computer.

2. Start a suitable terminal emulator on the host computer such as **minicom** on Linux or PuTTY on Windows. Set the communication parameters to 115200 baud, 8 data bits, no parity bit and 1 stop bit with no flow control.

3. Connect the board to the PEEDI using a 20-pin ARM/Xscale cable from the JTAG interface connector to the Target port on the PEEDI.

4. Power up the STM32X0G board.

5. Connect to the PEEDI's telnet CLI on port 23 as before.

6. Confirm correct connection with the PEEDI with the **reset reset** command as follows:

```
stm3220g> reset reset
++ info: RESET and TRST asserted
++ info: TRST released
++ info: BYPASS check passed
++ info: 2 TAP controller(s) detected
++ info: TAP 0 : IDCODE = 0x06411041, unknown ID
++ info: TAP 1 : IDCODE = 0x4BA00477, Cortex M3 JTAG
++ info: RESET released
++ info: core #0 connected
++ info: core 0: initialized

stm3220g>
```

Installation into Flash

The following describes the procedure for installing the RedBoot ROM monitor into on-chip Flash. It can also be adapted for installing user applications into Flash.

1. Locate the `redboot_ROM.bin` image within the `loaders` subdirectory of the base of the eCos installation. For applications, use **arm-eabi-objcopy -O binary** to convert the linked application, in ELF format, into binary format.

2. Copy the `redboot_ROM.bin` file into a location on the host computer accessible to its TFTP server.

3. Connect to the PEEDI's telnet interface, and program the RedBoot image into Flash with the following command, replacing *`TFTP_SERVER`* with the address of the TFTP server and *`/RBPATH`* with the location of the redboot_ROM.bin file relative to the TFTP server root directory:

   ```
   stm3220g> flash program tftp://TFTP_SERVER/RBPATH/redboot_ROM.bin bin 0x08000000 erase
   ++ info: Programming image file: tftp://TFTP_SERVER/RBPATH/redboot_ROM.bin
   ++ info: Programming using agent, buffer = 4096 bytes
   ++ info: At absolute address:    0x08000000
   erasing     at 0x08000000 (sector #0)
   programming at 0x08000000
   programming at 0x08001000
   programming at 0x08002000
   programming at 0x08003000
   erasing     at 0x08004000 (sector #1)
   programming at 0x08004000
   programming at 0x08005000
   programming at 0x08006000
   programming at 0x08007000
   erasing     at 0x08008000 (sector #2)
   programming at 0x08008000
   programming at 0x08009000
   programming at 0x0800A000
   programming at 0x0800B000
   erasing     at 0x0800C000 (sector #3)
   programming at 0x0800C000
   programming at 0x0800D000
   programming at 0x0800E000
   programming at 0x0800F000

   ++ info: successfully programmed 64.00 KB in 2.39 sec

   stm3220g>
   ```

The RedBoot installation is now complete. This can be tested by powering off the board, disconnecting the JTAG, and then powering on the board again. The RedBoot banner should be visible on the serial port:

```
+**Warning** FLASH configuration checksum error or invalid key
Use 'fconfig -i' to [re]initialize database

RedBoot(tm) bootstrap and debug environment [ROM]
eCosCentric certified release, version v3_1_X - built 04:32:11, Feb  3 2012

Copyright (C) 2000-2009 Free Software Foundation, Inc.
Copyright (C) 2003-2012 eCosCentric Limited
RedBoot is free software, covered by the eCos license, derived from the
```

```
GNU General Public License. You are welcome to change it and/or distribute
copies of it under certain conditions. Under the license terms, RedBoot's
source code and full license terms must have been made available to you.
Redboot comes with ABSOLUTELY NO WARRANTY.

Platform: ST STM3220G EVAL (Cortex-M3)
RAM: 0x64000000-0x64200000 [0x64003668-0x641dd000 available]
     0x20000000-0x2001f000 [0x20000000-0x2001f000 available]
FLASH: 0x08000000-0x080fffff, 4 x 0x4000 blocks, 1 x 0x10000 blocks, 7 x 0x20000 blocks
RedBoot>
```

Note

An alternative approach would be to use **arm-eabi-gdb** to load and run the `redboot_JTAG.elf` file supplied as a prebuilt in the release. Once that instance of RedBoot is running, standard RedBoot commands such as **'load'** and **'fis'** can be used via the serial RS232 interface to program images to Flash, e.g.:

```
$ arm-eabi-gdb /path/to/redboot_JTAG.elf
GNU gdb (eCosCentric GNU tools 4.4.5c) 7.2
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "--host=i686-pc-linux-gnu --target=arm-eabi".
For bug reporting instructions, please see:
<http://bugzilla.ecoscentric.com/>...
Reading symbols from /path/to/redboot_JTAG.elf...done.
(gdb) target remote PEEDI-IP-ADDRESS:2000
Remote debugging using PEEDI-IP-ADDRESS:2000
0x080035a0 in ?? ()
(gdb) load
Loading section .rom_vectors, size 0x8 lma 0x64000000
Loading section .text, size 0xb314 lma 0x64000008
Loading section .rodata, size 0x372c lma 0x6400b320
Loading section .data, size 0x754 lma 0x6400ea50
Start address 0x64000009, load size 61852
Transfer rate: 236 KB/sec, 10308 bytes/write.
(gdb) c
Continuing.
```

Initializing RedBoot Flash Configuration

RedBoot manages the internal flash for the storage of application programs and configuration data. The flash needs to be initialized with the following commands:

```
RedBoot> fis init
About to initialize [format] FLASH image system - continue (y/n)? y
*** Initialize FLASH Image System
... Erase from 0x080e0000-0x080fffff: .
... Program from 0x641e0000-0x64200000 to 0x080e0000: .
RedBoot> fconfig -i
Initialize non-volatile configuration - continue (y/n)? y
Run script at boot: false
Console baud rate: 115200
Update RedBoot non-volatile configuration - continue (y/n)? y
... Erase from 0x080e0000-0x080fffff: .
... Program from 0x641e0000-0x64200000 to 0x080e0000: .
RedBoot>
```

Issue the **reset** command to RedBoot, and verify the the target board comes up as expected with the correct settings.

You may also need to run the **fconfig -i** command if you have updated your RedBoot from a previous version with a different configuration which might not have any new config fields used by the newly programmed RedBoot.

Rebuilding RedBoot

Should it prove necessary to rebuild a RedBoot binary, this is done most conveniently at the command line. The steps needed to rebuild the ROM version of RedBoot are:

```
$ mkdir redboot_stm3220g_rom
$ cd redboot_stm3220g_rom
$ ecosconfig new stm3220g redboot
[ … ecosconfig output elided … ]
$ ecosconfig import $ECOS_REPOSITORY/hal/cortexm/stm32/stm32x0g_eval/VERSION/misc/redboot_ROM.ecm
$ ecosconfig tree
$ make
```

At the end of the build the `install/bin` subdirectory should contain the file `redboot.bin`. This may be programmed to the board using the above procedure, or by using RedBoot's own flash update mechanisms.

The other versions of RedBoot - RAM or JTAG - may be similarly built by choosing the appropriate alternative .ecm file.

Name

Configuration — Platform-specific Configuration Options

Overview

The STM32X0G-EVAL board platform HAL package is loaded automatically when eCos is configured for an `stm3220g`, `stm3240g` or `stm3241g` target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The STM32X0G-EVAL board platform HAL package supports five separate startup types:

RAM

> This is the startup type which is normally used during application development. The board has RedBoot programmed into internal Flash at location 0x08000000 and uses external RAM at location 0x64000000. arm-eabi-gdb is then used to load a RAM startup application into memory from 0x64008000 and debug it. It is assumed that the hardware has already been initialized by RedBoot. By default the application will use the eCos virtual vectors mechanism to obtain services from RedBoot, including diagnostic output.

ROM

> This startup type can be used for finished applications which will be programmed into internal ROM at location 0x08000000. Data and BSS will be put into external RAM starting from 0x64000000. The application will be self-contained with no dependencies on services provided by other software. The program expects to boot from reset with ROM mapped at location zero. It will then transfer control to the 0x08000000 region. eCos startup code will perform all necessary hardware initialization.

ROMINT

> This startup type can be used for finished applications which will be programmed into internal ROM at location 0x08000000. Data and BSS will be put into internal SRAM starting from 0x20000284 (F2) or 0x20000288 (F4). Internal SRAM below this address is reserved for vector tables. The application will be self-contained with no dependencies on services provided by other software. The program expects to boot from reset with ROM mapped at location zero. It will then transfer control to the 0x08000000 region. eCos startup code will perform all necessary hardware initialization.

JTAG

> This is the startup type used to build applications that are loaded via a JTAG interface. The application will be self-contained with no dependencies on services provided by other software. The program expects to be loaded from 0x64000000 and entered at that address. eCos startup code will perform all necessary hardware initialization.

SRAM

> This is a variation of the JTAG type that only uses internal memory. The application will be self-contained with no dependencies on services provided by other software. The program expects to be loaded from 0x20000284 (F2) or 0x20000288 (F4) and entered at that address. eCos startup code will perform all necessary hardware initialization.

RedBoot and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building RedBoot.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for

a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then the serial port will be claimed for HAL diagnostic output.

UART Serial Driver

The STM32X0G-EVAL board uses the STM32's internal UART serial support. The HAL diagnostic interface, used for both polled diagnostic output and RedBoot / GDB stub communication, is only expected to be available to be used on the UART 4 port (counting the first UART as UART1). This is because only UART 4 is actually routed to an external connector.

As well as the polled HAL diagnostic interface, there is also a `CYGPKG_IO_SERIAL_CORTEXM_STM32` package which contains all the code necessary to support interrupt-driven operation with greater functionality. All six UARTs can be supported by this driver Note: while it is possible for USART3 to be configured to use this connector, the only advantage of USART3 would be flow control lines, which are not routed to the connector on the STM32X0G board.

It is not recommended to use the interrupt-driven serial driver with a port at the same time as using that port for HAL diagnostic I/O.

This driver is not active until the `CYGPKG_IO_SERIAL_DEVICES` configuration option within the generic serial driver support package `CYGPKG_IO_SERIAL` is enabled in the configuration. By default this will only enable support in the driver for the UART4 port (the same as the HAL diagnostic interface), but the default configuration can be modified to enable support for other serial ports. Note that in this package, serial port numbering starts at 0, rather than 1. So for example, to enable the serial driver for ports USART1 and USART2, enable the configuration options "ST STM32 serial port 0 driver" (`CYGPKG_IO_SERIAL_CORTEXM_STM32_SERIAL0`) and "ST STM32 serial port 1 driver" (`CYGPKG_IO_SERIAL_CORTEXM_STM32_SERIAL1`).

Ethernet Driver

The STM32X0G-EVAL board is fitted with an Ethernet port connected via a DP83847 PHY to the STM32's on-chip Ethernet MAC. This is supported in eCosPro with a driver for the lwIP networking stack, contained in the package `CYGPKG_DEVS_ETH_CORTEXM_STM32`. At the present time it only supports the lwIP networking stack, and cannot be used for either the BSD networking stack, nor RedBoot.

The driver will be inactive (not built and greyed out in the eCos Configuration Tool) unless the platform HAL option "STM32 Ethernet Support" (`CYGPKG_HAL_CORTEXM_STM32_STM32X0G_EVAL_ETH0`) is enabled. This option in turn is only active if the "Common Ethernet support" (`CYGPKG_IO_ETH_DRIVERS`) package is included in your configuration. As the STM32 ethernet driver is an lwIP-only driver, it is most appropriate to choose the `lwip_eth` template as a starting point when choosing an eCos configuration, which will cause the necessary packages to be automatically included.

The platform HAL defines a further configuration option "Use MCO as PHY clock" (`CYGHWR_HAL_CORTEXM_STM32X0G_ETH_PHY_CLOCK_MCO`) which indicates whether the MCO1 clock signal is used as the 25MHz clock for the Ethernet PHY. With this option disabled, the on-board 25MHz crystal oscillator located at X1 can be used instead, although in that case the board jumper J5 must be changed. You may wish to make this change if you wish to use the MCO1 clock output for another purpose. Consult the STM32 clock and ethernet documentation for more details.

SPI Driver

An SPI bus driver is available for the STM32 in the package "ST STM32 SPI driver" (`CYGPKG_DEVS_SPI_CORTEXM_STM32`).

The only SPI device instantiated by default is for an external Aardvark SPI test board, connected to SPI bus 2 with pin PI0 selected as the chip select. To disable the Aardvark device support, the platform HAL contains an option "SPI devices" (`CYGPKG_HAL_CORTEXM_STM32_STM32X0G_EVAL_SPI`) which can be disabled. No other SPI devices are instantiated.

Consult the generic SPI driver API documentation in the eCosPro Reference Manual for further details on SPI support in eCosPro, along with the configuration options in the STM32 SPI device driver.

I²C Driver

The STM32 variant HAL provides the main I²C hardware driver itself, configured at `CYGPKG_HAL_STM32_I2C`. But the platform I²C support can also be configured separately at `CYGPKG_HAL_CORTEXM_STM32_STM32X0G_EVAL_I2C`. This ensures that the M24C64 serial EEPROM is instantiated and becomes available for applications from `<cyg/io/i2c.h>`, and also ensures the STM32's I²C bus 1 is enabled for it. This option also allows the m24c64.c test is built. In order to allow writes to the serial EEPROM, you must ensure that the STM32X0G-EVAL jumper JP24 is fitted. The m24c64.c test will not pass otherwise.

ADC Driver

The STM32 processor variant HAL provides an ADC driver. The STM32X0G-EVAL platform HAL enables the support for the devices ADC1, ADC2 and ADC3 and for configuration of the respective ADC device input channels.

Consult the generic ADC driver API documentation in the eCosPro Reference Manual for further details on ADC support in eCosPro, along with the configuration options in the STM32 ADC device driver.

CAN Driver

The STM32 has a dual BXCAN device for CAN support. This consists of a master device, BXCAN1, and a slave device, BXCAN2. If BXCAN2 is to be used, BXCAN1 must be powered and clocked, regardless of whether it is to be used for CAN traffic. BXCAN1 shares IO pins with the FSMC, which controls the external SRAM. Therefore, BXCAN1 can only be used in ROMINT startup mode.

The board has a single external CAN socket, to which either BXCAN1 or BXCAN2 can be routed, based on the settings of JP3 and JP10. By default these jumpers are set to route neither CAN device to the socket, and jumpers must be fitted to route the desired CAN device as appropriate. See the board reference manual for details.

Consult the generic CAN driver API documentation in the eCosPro Reference Manual for further details on CAN support in eCosPro, along with the documentation and configuration options in the BXCAN device driver.

Flash Driver

The STM32's on-chip Flash may be programmed and managed using the Flash driver located in the "STM32 Flash memory support" (`CYGPKG_DEVS_FLASH_STM32`) package. This driver is enabled automatically if the generic "Flash device drivers" (`CYGPKG_IO_FLASH`) package is included in the eCos configuration. The driver will configure itself automatically for the size and parameters of the specific STM32 variant present on the STM32X0G board.

A number of aspects of Flash driver behaviour can be configured within that driver, such as program/erase parallelism and program burst size. Consult the driver for more details.

Name

JTAG support — Usage

Use of JTAG for debugging

JTAG can be used to single-step and debug loaded applications, or even applications resident in ROM, including RedBoot.

Debugging of ROM applications is only possible if using hardware breakpoints. The Cortex-M3 core of the STM32 only supports two such hardware breakpoints, so they should be used sparingly. If using a GDB front-end such as Eclipse, check it has not set unnecessary extra breakpoints such as at `main()`. Some JTAG devices give the option of whether to set hardware or software breakpoints by default. Be sure to configure your device appropriately.

When debugging via JTAG, you are likely to need to disable the default HAL idle thread action, otherwise there may be issues where the target fails to halt and the debugging session is unreliable. More details can be found in the Cortex-M architectural HAL.

Normally, a notable disadvantage with JTAG debugging is that it does not allow thread-aware debugging, such as the ability to inspect different eCos threads or their stack backtraces, set thread-specific breakpoints, and so on. Fortunately the Ronetix PEEDI JTAG unit does support thread-aware debugging of eCos applications, however extra configuration steps are required. Consult the PEEDI documentation for more details as usage is beyond the scope of this document.

Ronetix PEEDI notes

On the Ronetix PEEDI, the `peedi.stm3220g.cfg` or `peedi.stm3240g.cfg` file supplied in the platform HAL package should be used to setup and configure the hardware to an appropriate state to load programs. This includes setting up the clocks, GPIO lines, and FSMC memory controller for off-chip SRAM.

The `peedi.stm3220g.cfg` and `peedi.stm3240g.cfg` files also contain an option to define whether hardware or software breakpoints are used by default, using the `CORE0_BREAKMODE` directive in the `[TARGET]` section. The supplied version of the file defaults to software breakpoints. With this default, hardware breakpoints can still be set from GDB using the **hbreak** command, or in the eCosPro version of Eclipse by setting the Breakpoint Type - consult the “eCosPro CDT plug-in user's guide” manual for details. The default can be changed to hardware breakpoints, and remember to use the **reboot** command on the PEEDI command line interface, or press the reset button to make the changes take effect.

On the PEEDI, debugging can be performed either via the telnet interface or using **arm-eabi-gdb**. In the case of the latter, **arm-eabi-gdb** needs to connect to TCP port 2000 on the PEEDI's IP address. For example:

```
(gdb) target remote 111.222.333.444:2000
```

By default when the PEEDI is powered up, the target will always run the initialization section of the relevant `peedi.stm3220g.cfg` or `peedi.stm3240g.cfg` file, and halts the target. This behaviour is repeated with the **reset** command.

If the board is reset either with the '**reset**' command, or by pressing the reset button, and then the '**go**' command is given, the board will boot from ROM as normal. A similar effect can be achieved in GDB by connecting with **target remote** and immediately typing **continue** or **c**.

It is also possible for the target to always run, without initialization, after the reset button has been pressed. This mode is selected with the `CORE0_STARTUP_MODE` directive in the `[TARGET]` section of the appropriate `peedi.stm3220g.cfg` or `peedi.stm3240g.cfg` file. This conveniently allows the target to be connected to the JTAG debugger, and be able to reset it with the reset button, without being required to always type '**go**' every time. Finally, it is also possible to set a temporary default (unless the PEEDI is reset) by giving an argument to the **reset** command, for example **reset run**. Use the command **help reset** at the PEEDI command prompt for more options.

Suitably configured applications can be loaded either via GDB, or directly via the telnet CLI into RAM for execution. For example:

```
stm3220g> memory load tftp://192.168.7.9/test.bin bin 0x64000000
```

```
++ info: Loading image file:  tftp://192.168.7.9/test.bin
++ info: At absolute address: 0x64000000
loading at 0x64000000
loading at 0x64004000

Successfully loaded 28KB (29064 bytes) in 0.1s
stm3220g> go 0x64000000
```

Consult the PEEDI documentation for information on other formats and loading mechanisms.

For Eclipse users wishing to debug ROM startup programs resident in Flash, it is worth highlighting that it is possible to use the eCosCentric Eclipse plugin to automatically reprogram Flash as the load sequence. To do so, you will need to install and use a TFTP server so that your application can be accessed from the PEEDI from there. You may then use a GDB command file, as described in more detail in the “eCosPro CDT plug-in user's guide” manual. This file can then contain contents similar to the following example:

```
define doload
  shell arm-eabi-objcopy -O binary /path/to/eclipse/workspace/projectname/Debug/myapp /path/to/tftp/server/area/myapp.bin
  monitor flash program tftp://10.1.1.1/myapp.bin bin 0x08000000 erase
  set $pc=0x08000000
end
```

Obviously you will need to adjust the paths and names for your system and TFTP server requirements.

Configuration of JTAG applications

JTAG applications can be loaded directly into RAM without requiring a ROM monitor. This loading can be done directly through the JTAG device, or where supported by the JTAG device, through GDB.

In order to configure the application to support this mode, it is recommended to use the JTAG startup type which will implicitly cause two important settings to change. Firstly, `CYGSEM_HAL_USE_ROM_MONITOR` must be disabled. Secondly the `CYGDBG_HAL_DIAG_TO_DEBUG_CHAN` option should be enabled in order to prevent HAL diagnostic output being encoded into GDB ($O) packets. These configuration changes could be made by hand, but use of the JTAG startup type will just work.

With these changes, any diagnostic output will appear out of the configured diagnostic channel, usually a serial port. An eCosCentric extension allows diagnostic output to appear in GDB instead. For this to work, you must enable the configuration option `CYGSEM_HAL_DIAG_TO_GDBFILEIO_CHAN` in the common HAL package.

For details of using hardware debug with the Eclipse IDE see the “eCos Hardware Debugging” section of the “eCosPro CDT plug-in user's guide” manual.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the STM32X0G-EVAL board hardware, and should be read in conjunction with that specification. The STM32X0G-EVAL platform HAL package complements the Cortex-M architectural HAL and the STM32 variant HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset the HAL will initialize or reinitialize many of the on-chip peripherals. There is an exception for RAM startup applications which depend on a ROM monitor for certain services.

For ROM and JTAG startup, the HAL will perform additional initialization, programming the various internal registers including the PLL, peripheral clocks and GPIO pins. The details of the early hardware startup may be found in the `src/stm32x0g_eval_misc.c` in both `hal_system_init` and `hal_platform_init`.

Memory Map

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

External RAM

> This is located at address 0x64000000 of the memory space, and is 1MiB long. For ROM applications, all of RAM is available for use. For RAM startup applications, RAM below 0x64008000 is reserved for RedBoot and the top `CYGNUM_HAL_COMMON_INTERRUPTS_STACK_SIZE` bytes are reserved for the interrupt stack, the remainder is available for the application.

Internal RAM

> This is located at address 0x20000000 of the memory space, and is 128KiB in size. On the STM3220G, the eCos VSR table occupies the bottom 388 bytes, with the virtual vector table starting at 0x20000184 and extending to 0x20000284. On the STM3240G, the eCos VSR table occupies the bottom 392 bytes on the STM3220G, with the virtual vector table starting at 0x20000188 and extending to 0x20000288. For ROM, ROMINT, SRAM and JTAG startups, the top `CYGNUM_HAL_COMMON_INTERRUPTS_STACK_SIZE` bytes are reserved for the interrupt stack. The remainder of internal RAM is available for use by applications.
>
> The STM32F407IG based systems have a block of (close-coupled) SRAM located at address 0x10000000 of the memory space, and 64KiB in size.

Internal FLASH

> This is located at address 0x08000000 of the memory space and will be mapped to 0x00000000 at reset. This region is 1024KiB in size. ROM applications are by default configured to run from this memory. This memory is managed by RedBoot's FIS system.

On-Chip Peripherals

> These are accessible at locations 0x40000000 and 0xE0000000 upwards. Descriptions of the contents can be found in the STM32 User Manual.

Linker Scripts

The platform linker scripts define the following symbols:

hal_vsr_table

This defines the location of the VSR table. This is set to 0x20000000 for all startup types, and space for either 97 (F2 processors) or 98 (F4 processors) entries is reserved.

hal_virtual_vector_table

This defines the location of the virtual vector table used to communicate between an ROM monitor and an eCos application. This is allocated right after the VSR table, at 0x20000184 (F2) or 0x20000188 (F4).

hal_interrupt_stack

This defines the location of the interrupt stack. For ROM and JTAG startups, this is allocated to the top of internal SRAM, 0x20020000. For RAM startups, it is allocated to the top of external SRAM, 0x64200000.

hal_startup_stack

This defines the location of the startup stack. For all startup types it is initially allocated at the half-way point of the interrupt stack.

Diagnostic LEDs

Four LEDs are fitted on the board for diagnostic purposes: LED1 (green), marked as LD1; LED2 (orange) marked as LD2; LED3 (red) marked as LD3; and LED4 (blue) marked as LD4.

The platform HAL header file at `<cyg/hal/plf_io.h>` defines the following convenience function to allow the LEDs to be set:

```
       extern void hal_stm32x0_led(char c);
```

The lowest 4 bits of the argument `c` correspond to each of the 4 LEDs (with LED1 as the least significant bit).

The platform HAL will automatically light LED1 when the platform initialisation is complete, however the LEDs are free for application use.

Flash wait states

The STM32X0G-EVAL platform HAL provides a configuration option to set the number of Flash read wait states to use: `CYGNUM_HAL_CORTEXM_STM32_FLASH_WAIT_STATES`. It is important to verify and if necessary update this value if changing the CPU clock (HCLK) frequency or CPU voltage. Consult the STM32 Flash programming manual (PM0059) for appropriate values for different clock speeds or voltages. The default of 3 reflects a supply voltage of 3.3V and HCLK of 120MHz.

Real-time characterization

The tm_basic kernel test gives statistics gathered about the real-time characterization and performance of the kernel. The sample output is shown here for information. The test was built for ROMINT startup, which provided the best performance as both code and data could remain on-chip.

Example 299.1. stm32x0g_eval Real-time characterization

```
            Startup, main stack : stack used   348 size  3920
             Startup : Idlethread stack used    84 size  2048

eCos Kernel Timings
Notes: all times are in microseconds (.000001) unless otherwise stated

Reading the hardware clock takes 0 'ticks' overhead
... this value will be factored out of all other measurements
Clock interrupt took    6.03 microseconds (6 raw clock ticks)

Testing parameters:
   Clock samples:            32
   Threads:                   5
   Thread switches:         128
   Mutexes:                  32
   Mailboxes:                32
```

```
  Semaphores:               32
  Scheduler operations:    128
  Counters:                 32
  Flags:                    32
  Alarms:                   32


                                Confidence
    Ave     Min     Max     Var  Ave  Min  Function
 ======  ======  ======  ====== ========== ========
   4.20    4.00    5.00    0.32   80%  80% Create thread
   1.20    1.00    2.00    0.32   80%  80% Yield thread [all suspended]
   1.20    1.00    2.00    0.32   80%  80% Suspend [suspended] thread
   1.00    1.00    1.00    0.00  100% 100% Resume thread
   1.40    1.00    2.00    0.48   60%  60% Set priority
   0.40    0.00    1.00    0.48   60%  60% Get priority
   3.20    3.00    4.00    0.32   80%  80% Kill [suspended] thread
   1.20    1.00    2.00    0.32   80%  80% Yield [no other] thread
   2.00    2.00    2.00    0.00  100% 100% Resume [suspended low prio] thread
   1.00    1.00    1.00    0.00  100% 100% Resume [runnable low prio] thread
   1.20    1.00    2.00    0.32   80%  80% Suspend [runnable] thread
   1.00    1.00    1.00    0.00  100% 100% Yield [only low prio] thread
   1.00    1.00    1.00    0.00  100% 100% Suspend [runnable->not runnable]
   3.20    3.00    4.00    0.32   80%  80% Kill [runnable] thread
   2.40    2.00    3.00    0.48   60%  60% Destroy [dead] thread
   4.60    4.00    6.00    0.72   80%  60% Destroy [runnable] thread
   5.80    5.00    7.00    0.64   40%  40% Resume [high priority] thread
   2.16    2.00    4.00    0.27   85%  85% Thread switch

   0.40    0.00    1.00    0.48   60%  60% Scheduler lock
   1.00    1.00    1.00    0.00  100% 100% Scheduler unlock [0 threads]
   0.99    0.00    1.00    0.02   99%   0% Scheduler unlock [1 suspended]
   0.98    0.00    1.00    0.03   98%   1% Scheduler unlock [many suspended]
   0.98    0.00    1.00    0.05   97%   2% Scheduler unlock [many low prio]

   0.38    0.00    1.00    0.47   62%  62% Init mutex
   1.38    1.00    2.00    0.47   62%  62% Lock [unlocked] mutex
   1.53    1.00    2.00    0.50   53%  46% Unlock [locked] mutex
   1.28    1.00    2.00    0.40   71%  71% Trylock [unlocked] mutex
   1.13    1.00    2.00    0.22   87%  87% Trylock [locked] mutex
   0.41    0.00    1.00    0.48   59%  59% Destroy mutex
   7.66    7.00    8.00    0.45   65%  34% Unlock/Lock mutex

   0.44    0.00    1.00    0.49   56%  56% Create mbox
   0.22    0.00    1.00    0.34   78%  78% Peek [empty] mbox
   1.38    1.00    2.00    0.47   62%  62% Put [first] mbox
   0.22    0.00    1.00    0.34   78%  78% Peek [1 msg] mbox
   1.34    1.00    2.00    0.45   65%  65% Put [second] mbox
   0.31    0.00    1.00    0.43   68%  68% Peek [2 msgs] mbox
   1.44    1.00    2.00    0.49   56%  56% Get [first] mbox
   1.44    1.00    2.00    0.49   56%  56% Get [second] mbox
   1.22    1.00    2.00    0.34   78%  78% Tryput [first] mbox
   1.16    1.00    2.00    0.26   84%  84% Peek item [non-empty] mbox
   1.13    1.00    2.00    0.22   87%  87% Tryget [non-empty] mbox
   1.09    1.00    2.00    0.17   90%  90% Peek item [empty] mbox
   1.13    1.00    2.00    0.22   87%  87% Tryget [empty] mbox
   0.06    0.00    1.00    0.12   93%  93% Waiting to get mbox
   0.19    0.00    1.00    0.30   81%  81% Waiting to put mbox
   0.47    0.00    1.00    0.50   53%  53% Delete mbox
   5.00    5.00    5.00    0.00  100% 100% Put/Get mbox

   0.31    0.00    1.00    0.43   68%  68% Init semaphore
   1.06    1.00    2.00    0.12   93%  93% Post [0] semaphore
   1.31    1.00    2.00    0.43   68%  68% Wait [1] semaphore
   1.13    1.00    2.00    0.22   87%  87% Trywait [0] semaphore
   1.13    1.00    2.00    0.22   87%  87% Trywait [1] semaphore
   0.41    0.00    1.00    0.48   59%  59% Peek semaphore
```

```
    0.41    0.00    1.00    0.48   59%  59% Destroy semaphore
    4.22    4.00    5.00    0.34   78%  78% Post/Wait semaphore

    0.44    0.00    1.00    0.49   56%  56% Create counter
    0.31    0.00    1.00    0.43   68%  68% Get counter value
    0.19    0.00    1.00    0.30   81%  81% Set counter value
    1.28    1.00    2.00    0.40   71%  71% Tick counter
    0.28    0.00    1.00    0.40   71%  71% Delete counter

    0.34    0.00    1.00    0.45   65%  65% Init flag
    1.19    1.00    2.00    0.30   81%  81% Destroy flag
    1.06    1.00    2.00    0.12   93%  93% Mask bits in flag
    1.19    1.00    2.00    0.30   81%  81% Set bits in flag [no waiters]
    1.75    1.00    2.00    0.38   75%  25% Wait for flag [AND]
    1.72    1.00    2.00    0.40   71%  28% Wait for flag [OR]
    1.66    1.00    2.00    0.45   65%  34% Wait for flag [AND/CLR]
    2.00    2.00    2.00    0.00  100% 100% Wait for flag [OR/CLR]
    0.28    0.00    1.00    0.40   71%  71% Peek on flag

    0.63    0.00    1.00    0.47   62%  37% Create alarm
    1.78    1.00    2.00    0.34   78%  21% Initialize alarm
    1.03    1.00    2.00    0.06   96%  96% Disable alarm
    1.63    1.00    2.00    0.47   62%  37% Enable alarm
    1.22    1.00    2.00    0.34   78%  78% Delete alarm
    1.44    1.00    2.00    0.49   56%  56% Tick counter [1 alarm]
    8.22    8.00    9.00    0.34   78%  78% Tick counter [many alarms]
    2.38    2.00    3.00    0.47   62%  62% Tick & fire counter [1 alarm]
   40.66   40.00   41.00    0.45   65%  34% Tick & fire counters [>1 together]
    9.16    9.00   10.00    0.26   84%  84% Tick & fire counters [>1 separately]
    6.00    6.00    6.00    0.00  100% 100% Alarm latency [0 threads]
    5.45    5.00    6.00    0.49   55%  55% Alarm latency [2 threads]
    5.39    5.00    6.00    0.48   60%  60% Alarm latency [many threads]
   10.01   10.00   11.00    0.01   99%  99% Alarm -> thread resume latency

    0.00    0.00    0.00    0.00            Clock/interrupt latency

    2.23    1.00    3.00    0.00            Clock DSR latency

  224    224     224  (main stack:   877)  Thread stack used (1360 total)
           All done, main stack : stack used   877 size  3920
            All done : Idlethread stack used   172 size  2048

Timing complete - 29800 ms total

PASS:<Basic timing OK>
EXIT:<done>
```

Name

Test Programs — Details

Test Programs

The STM32X0G platform HAL contains some test programs which allow various aspects of the board to be tested.

ADC Test

This program tests the ADC driver for the STM32. The only device connected to the ADC on the board is the potentiometer connected to ADC3 logical channel 7. Therefore this test primarily tests that. However, in addition it also report the values of the temperature sensor, Vrefint and Vbat inputs that are sourced on-chip. The option `CYGBLD_HAL_CORTEXM_STM32X0G_EVAL_TEST_ADC` must be enabled to run this test since it needs human interaction.

Chapter 300. STM32F429I-DISCO Platform HAL

Name

`CYGPKG_HAL_CORTEXM_STM32_STM32F429I_DISCO` — eCos Support for the STM32F429I-DISCO Board

Description

The STM32F429I-DISCO board has a STM32F429ZIT6U microcontroller that incorporates 2MiB of internal flash and 256KiB of internal SRAM. The board also has an additional 8MiB of external SDRAM, plus an I^{2}C touch-panel peripheral, a SPI motion sensor peripheral, and a USB connector (CN6 “USB USER”).

Since the board is equipped with an on-board ST-LINK/V2 hardware debugger interface (via the CN1 “USB ST-LINK” connector), and there are no UART or Ethernet connections, for typical eCos development test programs are downloaded and debugged via the SWD hardware debugger in conjunction with the relevant host-side tools.

This documentation describes platform-specific elements of the STM32F429I-DISCO board support within eCos. The STM32 variant HAL documentation covers various topics including HAL support common to STM32 variants, and on-chip device support. This document complements the STM32 documentation.

Supported Hardware

The STM32F429ZI has three main on-chip memory regions. The device has a SRAM region of 192KiB present at 0x20000000, and a 2MiB FLASH region present at 0x08000000 (which is aliased to 0x00000000 during normal execution). There is also has another on-chip RAM region of 64KiB present at 0x10000000 that is only accessible via the CPU core. Also, the STM32F429I-DISCO motherboard has 8MiB of SDRAM memory mapped to address 0x90000000.

The STM32 variant HAL includes support for the eight on-chip serial devices which are documented in the variant HAL. However, the STM32F429I-DISCO motherboard does not provide a standard UART connector. To make use of serial devices suitable transceiver hardware and connectors would need to be attached via the relevant motherboard P1 or P2 expansion connectors.

The STM32 variant HAL also includes support for the I^{2}C buses. A single I^{2}C device is instantiated as part of the platform port, which is for the STMPE811 touch-panel sensor connected via bus I^{2}C3. It is exported to `<cyg/io/i2c.h>` with the name `hal_stm32f429i_disco_touchpanel` in the normal way.

Similarly the STM32 variant HAL includes support for the SPI buses. A single SPI device is instantiated as part of the platform port, which is for the L3GD20 MEMS (motion sensor) connected via bus SPI5. It is exported via `<cyg/io/spi.h>` with the name `cyg_stm32f429i_disco_mems` in the normal way.

Device drivers are also provided for the STM32 on-chip Ethernet MAC, ADC, BXCAN and SDIO interfaces, but similarly suitable hardware support via the motherboard P1 and P2 expansion connectors would be needed to utilise the drivers. Additionally, support is provided for the on-chip watchdog, RTC (wallclock) and a Flash driver exists to permit management of the STM32's on-chip Flash.

Note

The STM32 variant HAL support for the SDIO interface is currently limited to supporting MMC/SD cards. If the multi-bit MMC/SD support is used it is recommended that on-chip SRAM transfer buffers are used to avoid RX overrun or TX underrun due to the slow external SDRAM access speed.

The STM32F4 processor and the STM32F429I-DISCO board provide a wide variety of peripherals, but unless support is specifically indicated, it should be assumed that it is not included.

Tools

The board port is intended to work with GNU tools configured for an **arm-eabi** target. The original port was done using **arm-eabi-gcc** version 4.7.3e, **arm-eabi-gdb** version 7.6.1, and **binutils** version 2.23.2.

Name

Setup — Preparing the STM32F429I-DISCO Board for eCos Development

Overview

Typically, since the STM32F429I-DISCO motherboard has a built-in ST-LINK/V2 interface providing hardware debug support, eCos applications are loaded and run via the debugger **arm-eabi-gdb** or via the Eclipse IDE. The debugger then communicates with the “GDB server” provided by the relevant host ST-LINK/V2 support tool being used (e.g. OpenOCD).

Normally for release applications the `ROM` startup type would be used, with the application programmed into the on-chip flash for execution when the board boots. It is still possible to use the hardware debugging support to debug such flash-based ROM applications, and this may be the desired approach if the application is too large for execution from SRAM, or where all of the SRAM and SDRAM is required for application run-time use.

Since the stand-alone STM32F429I-DISCO motherboard has limited I/O there is no support for either RedBoot or GDB stubs by default.

Nevertheless, it is still possible to program a GDB stub or RedBoot `ROM` image into on-chip Flash and download and debug via a serial UART, if pins for the UART are available. In that case, eCos applications are configured for `RAM` startup and then downloaded and run on the board via the debugger **arm-eabi-gdb**, or via the Eclipse IDE. By default for serial communications, all versions run with 8 bits, no parity, and 1 stop bit at 115200 baud. This rate can be changed in the eCos configuration used for building the GDB stub `ROM` image.

Preparing ST-LINK/V2 interface

The support for using the on-chip ITM stimulus ports for diagnostic and instrumentation output requires that the ST-LINK/V2 firmware is at least version `V2.J17.S0`. The firmware for the ST-LINK/V2 interface can be checked, and updated if needed, using a tool available from STMicroelectronics. The firmware version is also reported when the **openocd** command is executed (using a suitable configuration file). For example, the following OpenOCD output reports `JTAG v17`:

```
Info : STLINK v2 JTAG v17 API v2 SWIM v0 VID 0x0483 PID 0x3748
```

Unfortunately the official firmware updater is only available for the Windows platform at the moment. From a Windows machine:

1. Ensure that the Windows PC and STM32F429I-DISCO board are disconnected.

2. Download the STM32 ST-LINK Utility from ST's website.

   The page titled “STSW-LINK004 STM32 ST-LINK utility” provides a free download of the utility http://www.st.com/web/en/catalog/tools/PF258168

3. Install the ST-LINK Utility software on your Windows PC.

   Simply unzip the downloaded file `stsw-link004.zip` and run the `STM32 ST-LINK Utility_v3.0.0.exe` that was contained within it. Follow the on-screen instructions. This will install both the utility application and the ST-LINK/V2 USB driver.

4. Connect the STM32F429I-DISCO board to the PC.

   Connect the STM32F429I-DISCO board to the PC using the ST supplied mini-B USB cable. Windows should correctly identify the USB device and load the device driver. Windows Device Manager should now show “STMicroelectronics STLink dongle” under “Universal Serial Bus controllers”.

5. Run the ST-LINK Utility and ensure the ST-LINK firmware is up to date.

From the Windows “Start” menu run the “STM32 ST-LINK Utility”. Click on the `connect` icon, or select `Target->Connect` from the menu. This should confirm that a successful connection can be made to the board. To update the on-board ST-LINK/V2 firmware select `ST-LINK->Firmware Update` from the menu. In the ST-LINK dialog box that then appears click on the `Device Connect` button. This will likely result in a message “ST-Link is not in DFU mode. Please restart it.”. In this case simply disconnect the board from the PC and then reconnect it after a couple of seconds, then click the `OK` button on the message. In the original ST-Link dialog box click `Device Connect` again. The dialog box should now report the current on-board and available firmware versions, and enable you to upgrade the board by pressing the `Yes >>>>` button. We have tested the system with firmware version `V2.J17.SO` and would recommend this version as a minimum. Clicking `Yes >>>>` will cause a progress bar in the dialog to be animated and should eventually result in a “Update Successful” message. You can then close the various dialogs and exit the ST-LINK Utility. Disconnect and reconnect the board and it is now ready for use with OpenOCD.

Programming ROM images

Since the STM32F429I-DISCO board has a built-in ST-LINK/V2 SWD interface, if the `CN4` jumpers are closed then the micro USB host connection (CN1) and suitable host software (e.g. The OpenOCD package **openocd** tool) can be used to program the flash. Normally a default **openocd** session provides a comand-line via port `4444`. Consult the OpenOCD documentation for more details if a non-default **openocd** configuration is being used.

With a **telnet** connection established to the **openocd** any binary data can easily be written to the on-chip flash. e.g.

```
$ telnet localhost 4444
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Open On-Chip Debugger
> flash write_image test.bin 0x08000000
wrote 32518 bytes from file test.bin in 1.073942s (29.569 KiB/s)
```

To create a binary for flash programming the **arm-eabi-objcopy** command is used. This converts the, ELF format, linked application into a raw binary. For example:

```
$ arm-eabi-objcopy -O binary programname programname.bin
```

Name

Configuration — Platform-specific Configuration Options

Overview

The STM32F429I-DISCO board platform HAL package is loaded automatically when eCos is configured for the `stm32f429i_disco` target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The STM32F429I-DISCO board platform HAL package supports five separate startup types:

ROM

This startup type can be used for finished (stand-alone) applications which will be programmed into internal flash at location 0x08000000. Data and BSS will be put into external SDRAM starting from 0x90000000. The application will be self-contained with no dependencies on services provided by other software. The program expects to boot from reset with ROM mapped at location zero. It will then transfer control to the 0x08000000 region. eCos startup code will perform all necessary hardware initialization.

ROMINT

This startup type can be used for finished applications which will be programmed into internal flash at location 0x08000000. Data and BSS will be put into internal SRAM starting from 0x20000288. Internal SRAM below this address is reserved for vector tables. The application will be self-contained with no dependencies on services provided by other software. The program expects to boot from reset with ROM mapped at location zero. It will then transfer control to the 0x08000000 region. eCos startup code will perform all necessary hardware initialization.

The off-chip SDRAM memory from 0x90000000 is available, but is not referenced by the eCos run-time so is available for application use if required.

JTAG

This is the startup type used to build applications that are loaded via the hardware debugger interface. The application will be self-contained with no dependencies on services provided by other software. The program expects to be loaded from 0x90000000 and entered at that address. eCos startup code will perform all necessary hardware initialization, though since the application is loaded via the hardware debugger interface the host debug environment is responsible for configuring the necessary I/O state to initialise the off-chip SDRAM.

This is the startup type normally used during application development, since the large SDRAM memory space allows for larger debug applications where compiler optimisation may be disable, and run-time assert checking enabled.

Note

Executing code from the SDRAM memory has a performance downside. It is significantly slower than execution from on-chip SRAM or flash. If performance is an issue then hardware debugging can be used for any of the startup types if required.

SRAM

This is a variation of the `JTAG` type that only uses internal memory. The application will be self-contained with no dependencies on services provided by other software. The program expects to be loaded from 0x20000288 and entered at that address. eCos startup code will perform all necessary hardware initialization. Unlike the `JTAG` startup no explicit hardware debugger configuration is needed, since the application (like the ROM and ROMINT startup types) will initialise the off-chip SDRAM memory.

RAM

For the ST-LINK/V2 enabled STM32F429I-DISCO platform this startup type is unlikely to be used. It is provided for completeness.

When the board has RedBoot (or a GDB stub ROM) programmed into internal Flash at location 0x08000000 then the arm-eabi-gdb debugger can communicate with the relevant UART or Ethernet connection to load and debug applications. An application is loaded into memory from 0x90008000. It is assumed that the hardware has already been initialized by RedBoot. By default the application will not be stand-alone, and will use the eCos virtual vectors mechanism to obtain services from RedBoot, including diagnostic output.

RedBoot and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option CYGSEM_HAL_ROM_MONITOR should be set. Typically this option is set only when building RedBoot.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option CYGSEM_HAL_USE_ROM_MONITOR should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, as a testing step before switching to ROM startup.

Note

Though, as previously discussed, since the option of hardware debugging is available as standard on the STM32F429I-DISCO platform it is unlikely that the RAM startup type would be used for development.

SPI Driver

An SPI bus driver is available for the STM32 in the package “ST STM32 SPI driver” (CYGPKG_DEVS_SPI_CORTEXM_STM32).

The only SPI device instantiated by default is for an L3GD20 MEMS device connected to SPI bus 5 with pin PC1 selected as the chip select. To disable the device support, the platform HAL contains an option “SPI devices” (CYGPKG_HAL_CORTEXM_STM32_STM32F429I_DISCO_SPI) which can be disabled. No other SPI devices are instantiated.

Consult the generic SPI driver API documentation in the eCosPro Reference Manual for further details on SPI support in eCosPro, along with the configuration options in the STM32 SPI device driver.

I²C Driver

The STM32 variant HAL provides the main I²C hardware driver itself, configured at CYGPKG_HAL_STM32_I2C. But the platform I²C support can also be configured separately at CYGPKG_HAL_CORTEXM_STM32_STM32F429I_DISCO_I2C. This ensures that the STMPE811 touch-panel device is instantiated and becomes available for applications from <cyg/io/i2c.h>, and also ensures the STM32's I²C bus 5 is enabled for it.

ADC Driver

The STM32 processor variant HAL provides an ADC driver. The STM32F429I-DISCO platform HAL enables the support for the devices ADC1, ADC2 and ADC3 and for configuration of the respective ADC device input channels.

Consult the generic ADC driver API documentation in the eCosPro Reference Manual for further details on ADC support in eCosPro, along with the configuration options in the STM32 ADC device driver.

Flash Driver

The STM32's on-chip Flash may be programmed and managed using the Flash driver located in the “STM32 Flash memory support” (`CYGPKG_DEVS_FLASH_STM32`) package. This driver is enabled automatically if the generic “Flash device drivers” (`CYGPKG_IO_FLASH`) package is included in the eCos configuration. The driver will configure itself automatically for the size and parameters of the specific STM32 variant present on the STM32F429I-DISCO board.

A number of aspects of Flash driver behaviour can be configured within that driver, such as program/erase parallelism and program burst size. Consult the driver for more details.

Name

SWD support — Usage

Use of JTAG/SWD for debugging

JTAG/SWD can be used to single-step and debug loaded applications, or even applications resident in ROM.

Debugging of ROM applications is only possible if using hardware breakpoints. The Cortex-M4 core of the STM32F429ZI only supports six such hardware breakpoints, so they may need to be used sparingly. If using a GDB front-end such as Eclipse, check it has not set unnecessary extra breakpoints such as at `main()`. Some JTAG/SWD devices give the option of whether to set hardware or software breakpoints by default. Be sure to configure your device appropriately.

When debugging via JTAG, you are likely to need to disable the default HAL idle thread action, otherwise there may be issues where the target fails to halt and the debugging session is unreliable. More details can be found in the Cortex-M architectural HAL. This should *not* be necessary when using a SWD-based hardware debugger such as the on-board ST-LINK/V2 interface.

The default eCos configuration does not enable the use of ITM stimulus ports for the output of HAL diagnostics or Kernel instrumentation. The architecture HAL package `CYGPKG_HAL_CORTEXM` provides options to enable such use.

For HAL diagnostic (e.g. `diag_printf()`) output the architecture CDL option `CYGHWR_HAL_CORTEXM_DIAGNOSTICS_INTERFACE` should be updated to select `ITM` as the output destination. Once the `ITM` option has been configured the option `CYGHWR_HAL_CORTEXM_DIAGNOSTICS_ITM_PORT` allows the actual stimulus port used for the diagnostics to be selected.

When the Kernel instrumentation option `CYGPKG_KERNEL_INSTRUMENT` is enabled then the `CYGHWR_HAL_CORTEXM_ITM_INSTRUMENTATION` option can be enabled to direct instrumentation record output via an ITM stimulus port, rather than into a local memory buffer. The stimulus port used can be configured via the `CYGHWR_HAL_CORTEXM_ITM_INSTRUMENTATION_PORT_BASE` option.

However, when using the STM32F429I-DISCO board via the ST-LINK/V2 interface then it is recommended that the `gdb_hwdebug_fileio` approach is used to provide access to diagnostics via the GDB debug connection. When ITM support is used it has been observed that the ST-LINK/V2 firmware can drop data, leading to the possibility of confusing output. However, with care the ITM system can be tuned to provide diagnostic and instrumentation via the host SWD debugger.

Using the ST-LINK/V2 connection allows for a single cable to provide board power, hardware debug support and diagnostic output.

OpenOCD notes

The OpenOCD debugger can be configured to support the on-board ST-LINK/V2 interface available via the USB CN1 connection, with the CN4 links closed to directly connect to the target STM32F429 CPU. When configuring the **openocd** tool build, the **configure** script can be given the option `--enable-stlink` to provide for ST-LINK support.

An example OpenOCD configuration file `openocd.stm32f429i_disco.cfg` is provided within the eCos platform HAL package in the source repository. This will be in the directory `packages/hal/cortexm/stm32/stm32f429i_disco/current/misc` relative to the root of your eCos installation.

This configuration file can be used with OpenOCD on the host as follows:

```
$ openocd -f openocd.stm32f429i_disco.cfg
Open On-Chip Debugger 0.9.0 (2015-09-18-16:19)
Licensed under GNU GPL v2
For bug reports, read
    http://openocd.org/doc/doxygen/bugs.html
Info : The selected transport took over low-level target control. The results might differ compared to plain JTAG/SWD
adapter speed: 2000 kHz
adapter_nsrst_delay: 100
none separate
Info : Unable to match requested speed 2000 kHz, using 1800 kHz
Info : Unable to match requested speed 2000 kHz, using 1800 kHz
Info : clock speed 1800 kHz
Info : STLINK v2 JTAG v17 API v2 SWIM v0 VID 0x0483 PID 0x3748
Info : using stlink api v2
Info : Target voltage: 2.886506
Info : stm32f4x.cpu: hardware has 6 breakpoints, 4 watchpoints
```

By default **openocd** provides a console on port `4444`, and this can be used to interact with the target system. This console interface can be used to perform debugging, program the flash, etc.

Normally **arm-eabi-gdb** is used to connect to the default GDB server port `3333` for debugging. For example:

```
(gdb) target remote localhost:3333
Remote debugging using localhost:3333
0x00000000 in ?? ()
(gdb) monitor reset halt
target state: halted
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x0800422c msp: 0x20000c80
(gdb)
```

The application can then be loaded and executed under GDB as normal. If you are using Eclipse then you will need to define a “preload” gdb macro to emit the **monitor reset halt** command to OpenOCD. See the “Hardware Assisted Debugging” section of the “Eclipse/CDT for eCos application development” document's “Debugging eCos applications” chapter.

If the HAL diagnostics are configured to use ITM, and stimulus port 31 is configured as the HAL diagnostic destination, then the configuration example above will direct OpenOCD to direct ITM output (and also DWT and ETM) to a file named `tpiu.out` in the current directory of the shell which was used to run the **openocd** command. A more specific filename can be used by adjusting the OpenOCD configuration file.

To extract the ITM output, the Cortex-M architecture HAL package provides a helper program **parseitm** in the directory `packages/hal/cortexm/arch/current/host` relative to the root of your eCos installation. It can be compiled simply with:

```
$ gcc -o parseitm parseitm.c
```

You simply run it with the desired ITM stimulus port and name of the file containing the ITM output, for example:

```
$ parseitm -p 31 -f itm.out
```

It will echo all ITM stimulus for that port, continuing to read from the file until interrupted with Ctrl-C. Note that limited buffer space in debug hardware such as the ST-LINK can result in occasionally missed ITM data. eCosPro provides a workaround of throttling data within the `CYGHWR_HAL_CORTEXM_ITM_DIAGNOSTICS_THROTTLE` CDL configuration component in order to reduce or avoid lost ITM data. For further details, see the note in OpenOCD ITM support.

Similarly, if the eCos application is built with Kernel instrumentation enabled and configured for ITM output, then the default stimulus port 24 output can be captured. For example, assuming the application **cminfo** is the ELF file built from an eCos configuration with ITM instrumentation enabled, and is loaded and run via **openocd**, then we could run **parseitm** to capture instrumentation whilst the program executes, and then view the gathered data using the example **instdump** tool provided in the Kernel package.

```
$ parseitm -p 24 -f tpiu.out > inst.bin
^C
$ instdump -r inst.bin cminfo
Threads:
 threadid 1 threadobj 200045D0 "idle_thread"

     0:[THREAD:CREATE][THREAD 4095][TSHAL 4][TSTICK 0][ARG1:200045D0] { ts 4 microseconds }
     1:[SCHED:LOCK][THREAD 4095][TSHAL 45][TSTICK 0][ARG1:00000002] { ts 45 microseconds }
     2:[SCHED:UNLOCK][THREAD 4095][TSHAL 195][TSTICK 0][ARG1:00000002] { ts 195 microseconds }
     3:[SCHED:LOCK][THREAD 4095][TSHAL 346][TSTICK 0][ARG1:00000002] { ts 346 microseconds }
     4:[SCHED:UNLOCK][THREAD 4095][TSHAL 495][TSTICK 0][ARG1:00000002] { ts 495 microseconds }
     5:[THREAD:RESUME][THREAD 1][TSHAL 647][TSTICK 0][ARG1:200045D0][ARG2:200045D0] { ts 647 microseconds }
     6:[SCHED:LOCK][THREAD 1][TSHAL 795][TSTICK 0][ARG1:00000002] { ts 795 microseconds }
     7:[MLQ:ADD][THREAD 1][TSHAL 945][TSTICK 0][ARG1:200045D0][ARG2:0000001F] { ts 945 microseconds }
     8:[SCHED:UNLOCK][THREAD 1][TSHAL 1096][TSTICK 0][ARG1:00000002] { ts 1096 microseconds }
     9:[INTR:ATTACH][THREAD 1][TSHAL 0][TSTICK 0][ARG1:00000000] { ts 10000 microseconds }
    10:[INTR:UNMASK][THREAD 1][TSHAL 149][TSTICK 0][ARG1:00000000] { ts 10149 microseconds }
    11:[INTR:ATTACH][THREAD 1][TSHAL 305][TSTICK 0][ARG1:00000054] { ts 10305 microseconds }
    12:[INTR:UNMASK][THREAD 1][TSHAL 449][TSTICK 0][ARG1:00000054] { ts 10449 microseconds }
```

Configuration of JTAG/SWD applications

JTAG/SWD applications can be loaded directly into SRAM or SDRAM without requiring a ROM monitor. Loading can be done directly through the JTAG/SWD device, or through GDB where supported by the JTAG/SWD device.

In order to configure the application to support this mode, it is recommended to use the `JTAG` startup type which will implicitly cause two important settings to change. Firstly, `CYGSEM_HAL_USE_ROM_MONITOR` must be disabled. Secondly the `CYGDBG_HAL_DIAG_TO_DEBUG_CHAN` option should be enabled in order to prevent HAL diagnostic output being encoded into GDB ($O) packets. These configuration changes could be made by hand, but use of the `JTAG` startup type will just work.

With these changes, any diagnostic output will appear out of the configured diagnostic channel. An eCosCentric extension allows diagnostic output to appear in GDB, which is normally required for the STM32F429I-DISCO platform since it has no serial ports available. For this feature to work, you must enable the configuration option `CYGSEM_HAL_DIAG_TO_GDBFILEIO_CHAN` in the common HAL package. Then, after you load your application but before running it, you must give GDB the command:

```
(gdb) set hwdebug on
```

Eclipse users can do this by creating a GDB command file with the contents:

```
define postload
  set hwdebug on
end
```

They may then reference it from their Eclipse debug launch configuration. Using GDB command files is described in more detail in the "Eclipse/CDT for eCos application development" manual.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the STM32F429I-DISCO board hardware, and should be read in conjunction with that specification. The STM32F429I-DISCO platform HAL package complements the Cortex-M architectural HAL and the STM32 variant HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset the HAL will initialize or reinitialize many of the on-chip peripherals. There is an exception for `RAM` startup applications which depend on a ROM monitor for certain services.

For `ROM`, `ROMINT`, `SRAM` and `JTAG` startup types the HAL will perform additional initialization, programming the various internal registers including the PLL, peripheral clocks and GPIO pins. The details of the early hardware startup may be found in the `src/stm32f429i_disco_misc.c` in both `hal_system_init` and `hal_platform_init`.

Memory Map

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

Internal RAM

> This is located at address `0x20000000` of the memory space, and is 192KiB in size. The eCos VSR table occupies the bottom 392 bytes of memory, with the virtual vector table starting at `0x200001AC` and extending to `0x200002AC`. For `ROM`, `ROMINT`, `SRAM` and `JTAG` startups, the top `CYGNUM_HAL_COMMON_INTERRUPTS_STACK_SIZE` bytes are reserved for the interrupt stack. The remainder of internal RAM is available for use by applications.
>
> For all configurations there is also a block of (close-coupled) SRAM located at address `0x10000000` of the memory space, and 64KiB in size.

External RAM

> This is located at address `0x90000000` of the memory space, and is 8MiB long. For `ROM` applications, all of the SDRAM is available for use. For `JTAG` applications the application is loaded from `0x90000000` with the remaining SDRAM after the code+data available for application use.
>
> For `RAM` startup applications, SDRAM below `0x90008000` is reserved for RedBoot and the top `CYGNUM_HAL_COMMON_INTERRUPTS_STACK_SIZE` bytes are reserved for the interrupt stack, the remainder is available for the application.

Internal FLASH

> This is located at address `0x08000000` of the memory space and will be mapped to `0x00000000` at reset. This region is 2048KiB in size. `ROM` and `ROMINT` applications are by default configured to run from this memory.

On-Chip Peripherals

> These are accessible at locations `0x40000000` and `0xE0000000` upwards. Descriptions of the contents can be found in the STM32 User Manual.

Linker Scripts

The platform linker scripts define the following symbols:

hal_vsr_table
: This defines the location of the VSR table. This is set to `0x20000000` for all startup types, and space for 98 entries is reserved.

hal_virtual_vector_table
: This defines the location of the virtual vector table used to communicate between an ROM monitor and an eCos application. This is allocated right after the VSR table, at `0x200001AC`.

hal_interrupt_stack
: This defines the location of the interrupt stack. For `ROM`, `ROMINT`, `SRAM` and `JTAG` startups, this is allocated to the top of internal SRAM, `0x20030000`. For `RAM` startups, it is allocated to the top of external SDRAM, `0x90800000`.

hal_startup_stack
: This defines the location of the startup stack. For all startup types it is initially allocated at the half-way point of the interrupt stack.

Diagnostic LEDs

Two LEDs are fitted on the board for diagnostic purposes: LED0 (green), marked as LD3; and LED1 (red) marked as LD4.

The platform HAL header file at `<cyg/hal/plf_io.h>` defines the following convenience function to allow the LEDs to be set:

```
extern void hal_stm32f429i_disco_led(char c);
```

The lowest 2-bits of the argument `c` correspond to each of the 2 LEDs (with LED0 as the least significant bit).

The platform HAL will automatically light LED0 when the platform initialisation is complete, however the LEDs are free for application use.

Flash wait states

The STM32F429I-DISCO platform HAL provides a configuration option to set the number of Flash read wait states to use: `CYGNUM_HAL_CORTEXM_STM32_FLASH_WAIT_STATES`. It is important to verify and if necessary update this value if changing the CPU clock (HCLK) frequency or CPU voltage. Consult the STM32 Flash programming manual (PM0081) for appropriate values for different clock speeds or voltages. The default of 5 reflects a supply voltage of 3.3V and HCLK of 168MHz.

Real-time characterization

The **tm_basic** kernel test gives statistics gathered about the real-time characterization and performance of the kernel. The sample output is shown here for information. The test was built for `SRAM` startup with optimization flag -O2, since it provides the best performance as both code and data could remain on-chip.

Example 300.1. stm32f429i_disco Real-time characterization

```
eCos Kernel Timings
Notes: all times are in microseconds (.000001) unless otherwise stated

Reading the hardware clock takes 0 'ticks' overhead
... this value will be factored out of all other measurements
Clock interrupt took    4.00 microseconds (4 raw clock ticks)

Testing parameters:
   Clock samples:            32
   Threads:                   8
   Thread switches:         128
   Mutexes:                  16
   Mailboxes:                16
   Semaphores:               16
   Scheduler operations:    128
   Counters:                 16
   Flags:                    16
   Alarms:                   16
```

```
 Stack Size:             1088


                               Confidence
   Ave     Min     Max     Var  Ave  Min  Function
======  ======  ======  ====== ========== ========
  4.63    4.00    5.00    0.47   62%  37% Create thread
  1.00    1.00    1.00    0.00  100% 100% Yield thread [all suspended]
  1.00    1.00    1.00    0.00  100% 100% Suspend [suspended] thread
  1.00    1.00    1.00    0.00  100% 100% Resume thread
  1.38    1.00    2.00    0.47   62%  62% Set priority
  0.38    0.00    1.00    0.47   62%  62% Get priority
  2.75    2.00    3.00    0.38   75%  25% Kill [suspended] thread
  1.00    1.00    1.00    0.00  100% 100% Yield [no other] thread
  1.38    1.00    2.00    0.47   62%  62% Resume [suspended low prio] thread
  0.88    0.00    1.00    0.22   87%  12% Resume [runnable low prio] thread
  1.13    1.00    2.00    0.22   87%  87% Suspend [runnable] thread
  1.00    1.00    1.00    0.00  100% 100% Yield [only low prio] thread
  1.00    1.00    1.00    0.00  100% 100% Suspend [runnable->not runnable]
  2.63    2.00    3.00    0.47   62%  37% Kill [runnable] thread
  2.25    2.00    3.00    0.38   75%  75% Destroy [dead] thread
  4.25    4.00    5.00    0.38   75%  75% Destroy [runnable] thread
  4.75    4.00    5.00    0.38   75%  25% Resume [high priority] thread
  1.61    1.00    3.00    0.49   59%  39% Thread switch

  0.21    0.00    1.00    0.33   78%  78% Scheduler lock
  0.78    0.00    1.00    0.34   78%  21% Scheduler unlock [0 threads]
  0.79    0.00    1.00    0.33   78%  21% Scheduler unlock [1 suspended]
  0.76    0.00    1.00    0.37   75%  24% Scheduler unlock [many suspended]
  0.77    0.00    1.00    0.36   76%  23% Scheduler unlock [many low prio]

  0.31    0.00    1.00    0.43   68%  68% Init mutex
  1.13    1.00    2.00    0.22   87%  87% Lock [unlocked] mutex
  1.13    1.00    2.00    0.22   87%  87% Unlock [locked] mutex
  1.00    1.00    1.00    0.00  100% 100% Trylock [unlocked] mutex
  0.94    0.00    1.00    0.12   93%   6% Trylock [locked] mutex
  0.25    0.00    1.00    0.38   75%  75% Destroy mutex
  5.00    5.00    5.00    0.00  100% 100% Unlock/Lock mutex

  0.44    0.00    1.00    0.49   56%  56% Create mbox
  0.13    0.00    1.00    0.22   87%  87% Peek [empty] mbox
  1.06    1.00    2.00    0.12   93%  93% Put [first] mbox
  0.00    0.00    0.00    0.00  100% 100% Peek [1 msg] mbox
  1.06    1.00    2.00    0.12   93%  93% Put [second] mbox
  0.06    0.00    1.00    0.12   93%  93% Peek [2 msgs] mbox
  1.25    1.00    2.00    0.38   75%  75% Get [first] mbox
  1.00    1.00    1.00    0.00  100% 100% Get [second] mbox
  1.00    1.00    1.00    0.00  100% 100% Tryput [first] mbox
  0.88    0.00    1.00    0.22   87%  12% Peek item [non-empty] mbox
  1.00    1.00    1.00    0.00  100% 100% Tryget [non-empty] mbox
  0.94    0.00    1.00    0.12   93%   6% Peek item [empty] mbox
  0.94    0.00    1.00    0.12   93%   6% Tryget [empty] mbox
  0.19    0.00    1.00    0.30   81%  81% Waiting to get mbox
  0.31    0.00    1.00    0.43   68%  68% Waiting to put mbox
  0.38    0.00    1.00    0.47   62%  62% Delete mbox
  3.00    3.00    3.00    0.00  100% 100% Put/Get mbox

  0.19    0.00    1.00    0.30   81%  81% Init semaphore
  0.94    0.00    1.00    0.12   93%   6% Post [0] semaphore
  0.94    0.00    1.00    0.12   93%   6% Wait [1] semaphore
  1.00    1.00    1.00    0.00  100% 100% Trywait [0] semaphore
  0.88    0.00    1.00    0.22   87%  12% Trywait [1] semaphore
  0.25    0.00    1.00    0.38   75%  75% Peek semaphore
  0.25    0.00    1.00    0.38   75%  75% Destroy semaphore
  3.00    3.00    3.00    0.00  100% 100% Post/Wait semaphore

  0.44    0.00    1.00    0.49   56%  56% Create counter
```

```
    0.50    0.00    1.00    0.50  100%  50% Get counter value
    0.19    0.00    1.00    0.30   81%  81% Set counter value
    1.13    1.00    2.00    0.22   87%  87% Tick counter
    0.25    0.00    1.00    0.38   75%  75% Delete counter

    0.25    0.00    1.00    0.38   75%  75% Init flag
    1.00    1.00    1.00    0.00  100% 100% Destroy flag
    1.00    1.00    1.00    0.00  100% 100% Mask bits in flag
    1.00    1.00    1.00    0.00  100% 100% Set bits in flag [no waiters]
    1.44    1.00    2.00    0.49   56%  56% Wait for flag [AND]
    1.38    1.00    2.00    0.47   62%  62% Wait for flag [OR]
    1.38    1.00    2.00    0.47   62%  62% Wait for flag [AND/CLR]
    1.31    1.00    2.00    0.43   68%  68% Wait for flag [OR/CLR]
    0.19    0.00    1.00    0.30   81%  81% Peek on flag

    0.63    0.00    1.00    0.47   62%  37% Create alarm
    1.44    1.00    2.00    0.49   56%  56% Initialize alarm
    1.00    1.00    1.00    0.00  100% 100% Disable alarm
    1.69    1.00    2.00    0.43   68%  31% Enable alarm
    1.00    1.00    1.00    0.00  100% 100% Delete alarm
    1.25    1.00    2.00    0.38   75%  75% Tick counter [1 alarm]
    3.63    3.00    4.00    0.47   62%  37% Tick counter [many alarms]
    2.06    2.00    3.00    0.12   93%  93% Tick & fire counter [1 alarm]
   18.38   18.00   19.00    0.47   62%  62% Tick & fire counters [>1 together]
    4.56    4.00    5.00    0.49   56%  43% Tick & fire counters [>1 separately]
    3.00    3.00    3.00    0.00  100% 100% Alarm latency [0 threads]
    3.00    3.00    3.00    0.00  100% 100% Alarm latency [2 threads]
    3.00    3.00    3.00    0.00  100% 100% Alarm latency [many threads]
    6.01    6.00    7.00    0.01   99%  99% Alarm -> thread resume latency

     196     172     204                    Worker thread stack used (stack size 1088)
            All done, main thrd : stack used   796 size  1536
            All done : Idlethread stack used   164 size  1280

Timing complete - 27280 ms total

PASS:<Basic timing OK>
EXIT:<done>
```

Name

Test Programs — Details

Test Programs

The STM32F429I-DISCO platform HAL contains some test programs which allow various aspects of the board to be tested.

Manual Test

By default the **manual** test is not built by default. The configuration option `CYGPKG_HAL_CORTEXM_STM32_STM32F429I_DISCO_TESTS_MANUAL` should be enabled to allow the test to be built.

This program tests various aspects of the basic platform port, e.g. flashing LEDs, checking I²C and SPI device access and that the push-button GPIO operates.

Chapter 301. STM32F746G-DISCO Platform HAL

Name

`CYGPKG_HAL_CORTEXM_STM32_STM32F746G_DISCO` — eCos Support for the STM32F746G-DISCO Board

Description

This documentation describes the platform-specific elements of the STM32F746G-DISCO board support within eCos. It should be read in conjunction with the STM32 variant HAL section, which covers the common functionality shared by all STM32 variants, including eCos HAL features and on-chip device support. In addition ST's "Discovery kit for STM32F7 Series with STM32F746NG MCU" (ST User Manual id: UM1907) should be consulted for hardware setup and settings.

The board is equipped with an on-board ST-LINK/V2-1 hardware debugger interface (via the CN14 "USB ST-LINK" connector), which is typically used for eCos application development.

Supported Hardware

The STM32F746NG has two main on-chip memory regions. The device has a SRAM region of 320KiB present at 0x20000000, and a 1MiB FLASH region present at 0x08000000 (which is aliased to 0x00000000 during normal execution). Also, the STM32F746G-DISCO motherboard has 8MiB of SDRAM memory mapped to address 0x60000000.

The STM32 variant HAL includes support for the eight on-chip serial devices which are documented in the variant HAL. However, the STM32F746G-DISCO motherboard only provides access to a single UART (no flow control signals, no RS-232 transceiver) via the CN4 connector.

The STM32 variant HAL also includes support for the I^{2}C buses. Two I^{2}C devices are instantiated as part of the platform port, one for the RK043FN48H touch-panel sensor and another for the WM8994 audio codec. Both are connected via bus I^{2}C3. The descriptors are exported in the normal way via `<cyg/io/i2c.h>`, with the names `hal_stm32f746g_disco_touchpanel` and `hal_stm32f746g_disco_audiocodec` respectively.

Similarly the STM32 variant HAL includes support for the SPI buses. Though the discovery board does not provide any SPI devices as standard.

USB host and peripheral modes are supported on both the FS OTG (connector CN13) and HS OTG (connector CN12) controllers available on the evaluation board. Consult the STM32 variant HAL documentation for USB driver details.

Device drivers are also provided for the STM32 on-chip Ethernet MAC, ADC and SDIO interfaces. Additionally, support is provided for the on-chip watchdog, RTC (wallclock) and a Flash driver exists to permit management of the STM32's on-chip Flash.

Note

The STM32 variant HAL support for the SDIO interface is currently limited to supporting MMC/SD cards. If the multi-bit MMC/SD support is used it is recommended that on-chip SRAM transfer buffers are used to avoid RX overrun or TX underrun due to the slow external SDRAM access speed.

The STM32F7 processor and the STM32F746G-DISCO board provide a wide variety of peripherals, but unless support is specifically indicated, it should be assumed that it is not included.

Tools

The board port is intended to work with GNU tools configured for an **arm-eabi** target. The original port was done using **arm-eabi-gcc** version 4.7.3j, **arm-eabi-gdb** version 7.8.2, and **binutils** version 2.23.2.

Name

Setup — Preparing the STM32F746G-DISCO Board for eCos Development

Overview

Typically, since the STM32F746G-DISCO motherboard has a built-in ST-LINK/V2-1 interface providing hardware debug support, eCos applications are loaded and run via the debugger **arm-eabi-gdb** or via the Eclipse IDE. The debugger then communicates with the "GDB server" provided by the relevant host ST-LINK/V2-1 support tool being used (e.g. OpenOCD).

Normally for release applications the `ROM` startup type would be used, with the application programmed into the on-chip flash for execution when the board boots. It is still possible to use the hardware debugging support to debug such flash-based ROM applications, and this may be the desired approach if the application is too large for execution from on-chip SRAM, or where all of the SRAM and SDRAM is required for application run-time use.

If off-chip non-volatile memory (NVM) is used to hold the main application then the board can boot from the internal flash using a suitable boot loader. For example, the eCosPro BootUp ROM loader, where the BootUp code can start the main application (after an optional update sequence).

If required, it is still possible to program a GDB stub or RedBoot `ROM` image into on-chip Flash and download and debug via a serial connection (using the relevant CN4 pins). In that case, eCos applications are configured for `RAM` startup and then downloaded and run on the board via the debugger **arm-eabi-gdb**, or via the Eclipse IDE as appropriate.

Preparing ST-LINK/V2-1 interface

The support for using the on-chip ITM stimulus ports for diagnostic and instrumentation output requires that the ST-LINK/V2-1 firmware is at least version `V2.J24.S11`. The firmware for the ST-LINK/V2-1 interface can be checked, and updated if needed, using a tool available from STMicroelectronics. The firmware version is also reported when the **openocd** command is executed (using a suitable configuration file). For example, the following OpenOCD output reports `JTAG v24`:

```
Info : STLINK v2 JTAG v24 API v2 SWIM v11 VID 0x0483 PID 0x374B
```

The user should refer to the ST "ST-LINK/V2-1 firmware upgrade" (RN0093) Release Note, which provides detail on the host requirements for upgrading the ST-Link firmware on Linux, Mac OS X and Windows hosts.

Programming ROM images

Since the STM32F746G-DISCO board has a built-in ST-LINK/V2-1 SWD interface, the USB host connection (CN14) and suitable host software (e.g. The OpenOCD package **openocd** tool) can be used to program the flash.

The **openocd** GDB server can directly program flash based applications from the GDB **load** command.

Note

The **openocd** command being used should have been configured and built to support the ST-LINK/V2-1 interface. This is achieved by specifying the **--enable-stlink** when configuring the OpenOCD build. Additional information on running **openocd** may be found in the OpenOCD notes.

For example, assuming that **openocd** is running on the same host as GDB, and is connected to the target board the following will program the "bootup.elf" application into the on-chip flash:

```
$ arm-eabi-gdb install/bin/bootup.elf
GNU gdb (eCosCentric GNU tools 4.7.3j) 7.8.2
[ … GDB output elided … ]
(gdb) target remote localhost:3333
hal_reset_vsr () at path/hal_misc.c:171
```

```
(gdb) load
Loading section .rom_vectors, size 0x14 lma 0x8000000
Loading section .text, size 0x3adc lma 0x8000018
Loading section .rodata, size 0x6c0 lma 0x8003af8
Loading section .data, size 0x6dc lma 0x80041b8
Start address 0x8000018, load size 18572
Transfer rate: 14 KB/sec, 4643 bytes/write.
(gdb)
```

Alternatively, the **openocd** telnet interface can be used to manually program the flash. By default the **openocd** session provides a comand-line via port `4444`. Consult the OpenOCD documentation for more details if a non-default **openocd** configuration is being used.

With a **telnet** connection established to the **openocd** any binary data can easily be written to the on-chip flash. e.g.

```
$ telnet localhost 4444
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Open On-Chip Debugger
> flash write_image test.bin 0x08000000
wrote 32518 bytes from file test.bin in 1.073942s (29.569 KiB/s)
```

To create a binary for flash programming the **arm-eabi-objcopy** command is used. This converts the, ELF format, linked application into a raw binary. For example:

```
$ arm-eabi-objcopy -O binary programname programname.bin
```

Name

Configuration — Platform-specific Configuration Options

Overview

The STM32F746G-DISCO board platform HAL package `CYGPKG_HAL_CORTEXM_STM32_STM32F746G_DISCO` is loaded automatically when eCos is configured for the `stm32f746g_disco` target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The STM32F746G-DISCO board platform HAL package supports five separate startup types:

ROM
: This startup type can be used for finished (stand-alone) applications which will be programmed into internal flash at location 0x08000000. Data and BSS will be put into external SDRAM starting from 0x60000000. The application will be self-contained with no dependencies on services provided by other software. The program expects to boot from reset with ROM mapped at location zero. It will then transfer control to the 0x08000000 region. eCos startup code will perform all necessary hardware initialization.

ROMINT
: This startup type can be used for finished applications which will be programmed into internal flash at location 0x08000000. Data and BSS will be put into internal SRAM starting from 0x200002C8. Internal SRAM below this address is reserved for vector tables. The application will be self-contained with no dependencies on services provided by other software. The program expects to boot from reset with ROM mapped at location zero. It will then transfer control to the 0x08000000 region. eCos startup code will perform all necessary hardware initialization.

The off-chip SDRAM memory from 0x60000000 is available, but is not referenced by the eCos run-time so is available for application use if required.

JTAG
: This is the startup type used to build applications that are loaded via the hardware debugger interface. The application will be self-contained with no dependencies on services provided by other software. The program expects to be loaded into off-chip SDRAM from 0x60000000 and entered at that address. eCos startup code will perform all necessary hardware initialization, though since the application is loaded via the hardware debugger interface the host debug environment is responsible for configuring the necessary I/O state to initialise the off-chip SDRAM.

This is the startup type normally used during application development, since the large SDRAM memory space allows for larger debug applications where compiler optimisation may be disabled, and run-time assert checking enabled.

Note

Executing code from the SDRAM memory has a performance downside. It is significantly slower than execution from on-chip SRAM or flash. If performance is an issue then hardware debugging can be used for any of the startup types if required.

SRAM
: This is a variation of the `JTAG` type that only uses internal memory. The application will be self-contained with no dependencies on services provided by other software. The program expects to be loaded from 0x200002C8 and entered at that address. eCos startup code will perform all necessary hardware initialization. Unlike the `JTAG` startup no explicit hardware debugger configuration is needed, since the application (like the `ROM` and `ROMINT` startup types) will initialise the off-chip SDRAM memory.

RAM
: For the ST-LINK/V2-1 enabled STM32F746G-DISCO platform this startup type is unlikely to be used. It is provided for completeness.

When the board has RedBoot (or a GDB stub ROM) programmed into internal Flash at location 0x08000000 then the arm-eabi-gdb debugger can communicate with a suitably configured UART connection to load and debug applications.

An application is loaded into memory from 0x60008000. It is assumed that the hardware has already been initialized by RedBoot. By default the application will *not* be stand-alone, and will use the eCos virtual vectors mechanism to obtain services from RedBoot, including diagnostic output.

Warning

RedBoot can have an adverse affect on the real-time performance of applications.

RedBoot and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building RedBoot.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a `RAM` startup, disabled otherwise. It can be manually disabled for a `RAM` startup, making the application self-contained, as a testing step before switching to `ROM` startup.

Note

Though, as previously discussed, since the option of hardware debugging is available as standard on the STM32F746G-DISCO platform it is unlikely that the `RAM` startup type would be used for development.

SPI Driver

An SPI bus driver is available for the STM32 in the package “ST STM32 SPI driver” (`CYGPKG_DEVS_SPI_CORTEXM_STM32`).

No SPI devices are instantiated for this platform by default.

Note

An example SPI M25PXX configuration can be enabled for boards suitably modified with an attached compatible flash device. The CDL option `CYGPKG_HAL_CORTEXM_STM32_STM32F746G_DISCO_SPI` can be enabled, and uses SPI bus 2 with the chip-select on PI0.

When configured the `m25pxx_flash_device` device is exported and can be accessed via the standard flash API. The device is given a logical base address of `0x00000000` but is *not* memory-mapped.

Consult the generic SPI driver API documentation in the eCosPro Reference Manual for further details on SPI support in eCosPro, along with the configuration options in the STM32 SPI device driver.

I²C Driver

The STM32 variant HAL provides the main I²C hardware driver itself, configured at `CYGPKG_HAL_STM32_I2C`. However, the platform I²C support can also be configured separately at `CYGPKG_HAL_CORTEXM_STM32_STM32F746G_DISCO_I2C`. This enables I²C buses 1 and 3. The instantiated devices become available for applications via `<cyg/io/i2c.h>`.

Flash Driver

The STM32's on-chip Flash may be programmed and managed using the Flash driver located in the “STM32 Flash memory support” (`CYGPKG_DEVS_FLASH_STM32`) package. This driver is enabled automatically if the generic “Flash device drivers” (`CYGPKG_IO_FLASH`) package is included in the eCos configuration. The driver will configure itself automatically for the size and parameters of the specific STM32 variant present on the STM32F746G-DISCO board.

A number of aspects of Flash driver behaviour can be configured within that driver, such as program/erase parallelism and program burst size. Consult the driver for more details.

Ethernet Driver

The Ethernet MAC is connected to a LAN8742A PHY via the RMII interface and thence to a RJ45 connector at CN9. The external 25MHz crystal is used to supply the clock.

Name

SWD support — Usage

Use of JTAG/SWD for debugging

JTAG/SWD can be used to single-step and debug loaded applications, or even applications resident in ROM.

Debugging of ROM applications is only possible if using hardware breakpoints. The Cortex-M7 core of the STM32F746NG only supports eight such hardware breakpoints, so they may need to be used sparingly. If using a GDB front-end such as Eclipse, check it has not set unnecessary extra breakpoints such as at `main()`. Some JTAG/SWD devices give the option of whether to set hardware or software breakpoints by default. Be sure to configure your device appropriately.

When debugging via JTAG, you are likely to need to disable the default HAL idle thread action, otherwise there may be issues where the target fails to halt and the debugging session is unreliable. More details can be found in the Cortex-M architectural HAL. This should *not* be necessary when using a SWD-based hardware debugger such as the on-board ST-LINK/V2-1 interface.

The default eCos configuration does not enable the use of ITM stimulus ports for the output of HAL diagnostics or Kernel instrumentation. The architecture HAL package `CYGPKG_HAL_CORTEXM` provides options to enable such use.

For HAL diagnostic (e.g. `diag_printf()`) output the architecture CDL option `CYGHWR_HAL_CORTEXM_DIAGNOSTICS_INTERFACE` should be updated to select `ITM` as the output destination. Once the `ITM` option has been configured the option `CYGHWR_HAL_CORTEXM_DIAGNOSTICS_ITM_PORT` allows the actual stimulus port used for the diagnostics to be selected.

When the Kernel instrumentation option `CYGPKG_KERNEL_INSTRUMENT` is enabled then the `CYGHWR_HAL_CORTEXM_ITM_INSTRUMENTATION` option can be enabled to direct instrumentation record output via an ITM stimulus port, rather than into a local memory buffer. The stimulus port used can be configured via the `CYGHWR_HAL_CORTEXM_ITM_INSTRUMENTATION_PORT_BASE` option.

However, when using the STM32F746G-DISCO board via the ST-LINK/V2-1 interface then it is recommended that the `gdb_hwdebug_fileio` approach is used to provide access to diagnostics via the GDB debug connection. When ITM support is used it has been observed that the ST-LINK/V2-1 firmware can drop data, leading to the possibility of confusing output. However, with care the ITM system can be tuned to provide diagnostic and instrumentation via the host SWD debugger.

Using the ST-LINK/V2-1 connection allows for a single cable to provide power, hardware debug support and diagnostic output.

OpenOCD notes

The OpenOCD debugger can be configured to support the on-board ST-LINK/V2-1 interface available via the USB CN14 connection. When configuring the **openocd** tool build, the **configure** script can be given the option `--enable-stlink` to provide for ST-LINK support.

An example OpenOCD configuration file `openocd.stm32f746g_disco.cfg` is provided within the eCos platform HAL package in the source repository. This will be in the directory `packages/hal/cortexm/stm32/stm32f746g_disco/current/misc` relative to the root of your eCos installation.

This configuration file can be used with OpenOCD on the host as follows:

```
$ openocd -f openocd.stm32f746g_disco.cfg
Open On-Chip Debugger 0.9.0 (2015-08-26-09:13)
Licensed under GNU GPL v2
For bug reports, read
    http://openocd.org/doc/doxygen/bugs.html
Info : The selected transport took over low-level target control. The results might differ compared to plain JTAG/SWD
adapter speed: 2000 kHz
adapter_nsrst_delay: 100
srst_only separate srst_nogate srst_open_drain connect_deassert_srst
Info : Unable to match requested speed 2000 kHz, using 1800 kHz
Info : Unable to match requested speed 2000 kHz, using 1800 kHz
Info : clock speed 1800 kHz
Info : STLINK v2 JTAG v24 API v2 SWIM v11 VID 0x0483 PID 0x374B
Info : using stlink api v2
Info : Target voltage: 3.220472
Info : stm32f7x.cpu: hardware has 8 breakpoints, 4 watchpoints
```

By default **openocd** provides a console on port `4444`, and this can be used to interact with the target system. This console interface can be used to perform debugging, program the flash, etc.

Normally **arm-eabi-gdb** is used to connect to the default GDB server port `3333` for debugging. For example:

```
(gdb) target remote localhost:3333
Remote debugging using localhost:3333
0x00000000 in ?? ()
(gdb)
```

The application can then be loaded and executed under GDB as normal. If you are using Eclipse then, if required, you can define a “preload” gdb macro to emit any necessary commands to OpenOCD. See the “Hardware Assisted Debugging” section of the “Eclipse/CDT for eCos application development” document's “Debugging eCos applications” chapter.

If the HAL diagnostics are configured to use ITM, and stimulus port 31 is configured as the HAL diagnostic destination, then the configuration example above will direct OpenOCD to direct ITM output (and also DWT and ETM) to a file named `tpiu.out` in the current directory of the shell which was used to run the **openocd** command. A more specific filename can be used by adjusting the OpenOCD configuration file.

To extract the ITM output, the Cortex-M architecture HAL package provides a helper program **parseitm** in the directory `packages/hal/cortexm/arch/current/host` relative to the root of your eCos installation. It can be compiled simply with:

```
$ gcc -o parseitm parseitm.c
```

You simply run it with the desired ITM stimulus port and name of the file containing the ITM output, for example:

```
$ parseitm -p 31 -f itm.out
```

and it will echo all ITM stimulus for that port, continuing to read from the file until interrupted with Ctrl-C. Note that limited buffer space in debug hardware such as the ST-LINK can result in occasionally missed ITM data. eCosPro provides a workaround of throttling data within the `CYGHWR_HAL_CORTEXM_ITM_DIAGNOSTICS_THROTTLE` CDL configuration component in order to reduce or avoid lost ITM data. For further details, see the note in OpenOCD ITM support.

Similarly, if the eCos application is built with Kernel instrumentation enabled and configured for ITM output, then the default stimulus port 24 output can be captured. For example, assuming the application **cminfo** is the ELF file built from an eCos configuration with ITM instrumentation enabled, and is loaded and run via **openocd**, then we could run **parseitm** to capture instrumentation whilst the program executes, and then view the gathered data using the example **instdump** tool provided in the Kernel package.

```
$ parseitm -p 24 -f tpiu.out > inst.bin
^C
$ instdump -r inst.bin cminfo
Threads:
 threadid 1 threadobj 200045D0 "idle_thread"

     0:[THREAD:CREATE][THREAD 4095][TSHAL 4][TSTICK 0][ARG1:200045D0] { ts 4 microseconds }
     1:[SCHED:LOCK][THREAD 4095][TSHAL 45][TSTICK 0][ARG1:00000002] { ts 45 microseconds }
     2:[SCHED:UNLOCK][THREAD 4095][TSHAL 195][TSTICK 0][ARG1:00000002] { ts 195 microseconds }
     3:[SCHED:LOCK][THREAD 4095][TSHAL 346][TSTICK 0][ARG1:00000002] { ts 346 microseconds }
     4:[SCHED:UNLOCK][THREAD 4095][TSHAL 495][TSTICK 0][ARG1:00000002] { ts 495 microseconds }
     5:[THREAD:RESUME][THREAD 1][TSHAL 647][TSTICK 0][ARG1:200045D0][ARG2:200045D0] { ts 647 microseconds }
     6:[SCHED:LOCK][THREAD 1][TSHAL 795][TSTICK 0][ARG1:00000002] { ts 795 microseconds }
     7:[MLQ:ADD][THREAD 1][TSHAL 945][TSTICK 0][ARG1:200045D0][ARG2:0000001F] { ts 945 microseconds }
     8:[SCHED:UNLOCK][THREAD 1][TSHAL 1096][TSTICK 0][ARG1:00000002] { ts 1096 microseconds }
     9:[INTR:ATTACH][THREAD 1][TSHAL 0][TSTICK 0][ARG1:00000000] { ts 10000 microseconds }
    10:[INTR:UNMASK][THREAD 1][TSHAL 149][TSTICK 0][ARG1:00000000] { ts 10149 microseconds }
    11:[INTR:ATTACH][THREAD 1][TSHAL 305][TSTICK 0][ARG1:00000054] { ts 10305 microseconds }
    12:[INTR:UNMASK][THREAD 1][TSHAL 449][TSTICK 0][ARG1:00000054] { ts 10449 microseconds }
```

Configuration of JTAG/SWD applications

JTAG/SWD applications can be loaded directly into SRAM or SDRAM without requiring a ROM monitor. Loading can be done directly through the JTAG/SWD device, or through GDB where supported by the JTAG/SWD device.

In order to configure the application to support this mode, it is recommended to use the JTAG startup type which will implicitly cause two important settings to change. Firstly, CYGSEM_HAL_USE_ROM_MONITOR must be disabled. Secondly the CYGDBG_HAL_DIAG_TO_DEBUG_CHAN option should be enabled in order to prevent HAL diagnostic output being encoded into GDB ($O) packets. These configuration changes could be made by hand, but use of the JTAG startup type will just work.

With these changes, any diagnostic output will appear out of the configured diagnostic channel. An eCosCentric extension allows diagnostic output to appear in GDB. For this feature to work, you must enable the configuration option CYGSEM_HAL_DIAG_TO_GDBFILEIO_CHAN in the common HAL package. If you are using the graphical configuration tool then you should then accept any suggested solutions to the subsequent configuration conflicts. Older eCos releases also required the gdb "set hwdebug on" command to be used to enable GDB or Eclipse console output, but this is no longer required with the latest tools.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the STM32F746G-DISCO board hardware, and should be read in conjunction with that specification. The STM32F746G-DISCO platform HAL package complements the Cortex-M architectural HAL and the STM32 variant HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset the HAL will initialize or reinitialize many of the on-chip peripherals. There is an exception for `RAM` startup applications which depend on a ROM monitor for certain services.

For `ROM`, `ROMINT`, `SRAM` and `JTAG` startup types the HAL will perform additional initialization, programming the various internal registers including the PLL, peripheral clocks and GPIO pins. The details of the early hardware startup may be found in the `src/stm32f746g_disco_misc.c` in both `hal_system_init` and `hal_platform_init`.

Memory Map

The platform HAL package provides the memory layout information needed to generate the linker script. For all the STARTUP variations the top `CYGNUM_HAL_COMMON_INTERRUPTS_STACK_SIZE` bytes of the on-chip SRAM are reserved for the interrupt stack. The remainder of the internal SRAM is available for use by applications. The key memory locations are as follows:

Internal RAM
: This is located at address `0x20000000` of the memory space, and is 320KiB in size. The eCos VSR table occupies the bottom 456-bytes of memory, with the virtual vector table starting at `0x200001C8` and extending to `0x200002C8`.

  This memory region comprises three contiguous memory blocks, the DTCM (Data Tightly Coupled Memory), SRAM region 1 and SRAM region 2.

External SDRAM
: This is located at address `0x60000000` of the memory space, and is 8MiB long. For `ROM` applications, all of the SDRAM is available for use. For `JTAG` applications the application is loaded from `0x60000000` with the remaining SDRAM after the code+data available for application use.

  For `RAM` startup applications, SDRAM below `0x60008000` is reserved for the debug monitor (e.g. RedBoot).

Internal FLASH
: This is located at address `0x08000000` of the memory space and will be mapped to `0x00000000` at reset. This region is 1024KiB in size. `ROM` and `ROMINT` applications are by default configured to run from this memory.

On-Chip Peripherals
: These are accessible at locations `0x40000000` and `0xE0000000` upwards. Descriptions of the contents can be found in the STM32 User Manual.

Linker Scripts

The platform linker scripts define the following symbols:

hal_vsr_table
: This defines the location of the VSR table. This is set to `0x20000000` for all startup types, and space for 114 entries is reserved.

hal_virtual_vector_table
: This defines the location of the virtual vector table used to communicate between an ROM monitor and an eCos application. This is allocated right after the VSR table, at `0x200001C8`.

hal_interrupt_stack — This defines the location of the interrupt stack. This is allocated to the top of internal SRAM, `0x20050000`.

hal_startup_stack — This defines the location of the startup stack. For all startup types it is initially allocated at the half-way point of the interrupt stack.

Flash wait states

The STM32F746G-DISCO platform HAL provides a configuration option to set the number of Flash read wait states to use: `CYGNUM_HAL_CORTEXM_STM32_FLASH_WAIT_STATES`. It is important to verify and if necessary update this value if changing the CPU clock (HCLK) frequency or CPU voltage. Consult the relevant STM32 datasheets and programming manuals for the STM32F746G parts for appropriate values for different clock speeds or voltages. The default of 5 reflects a supply voltage of 3.3V and HCLK of 180MHz.

Real-time characterization

The **tm_basic** kernel test gives statistics gathered about the real-time characterization and performance of the kernel. The sample output is shown here for information. The test was built for `SRAM` startup with optimization flag -O2, since it provides the best performance as both code and data could remain on-chip.

Example 301.1. stm32f746g_disco Real-time characterization

```
             Startup, main thrd : stack used   360 size  1536
             Startup : Idlethread stack used    84 size  1280

eCos Kernel Timings
Notes: all times are in microseconds (.000001) unless otherwise stated

Reading the hardware clock takes 0 'ticks' overhead
... this value will be factored out of all other measurements
Clock interrupt took    3.03 microseconds (3 raw clock ticks)

Testing parameters:
   Clock samples:            32
   Threads:                  16
   Thread switches:         128
   Mutexes:                  32
   Mailboxes:                32
   Semaphores:               32
   Scheduler operations:    128
   Counters:                 32
   Flags:                    32
   Alarms:                   32
   Stack Size:             1088


                                 Confidence
     Ave     Min     Max     Var  Ave  Min  Function
  ======  ======  ======  ====== ========== ========
INFO:<Ctrl-C disabled until test completion>
    1.94    1.00    3.00    0.23   81%  12% Create thread
    0.38    0.00    1.00    0.47   62%  62% Yield thread [all suspended]
    0.44    0.00    1.00    0.49   56%  56% Suspend [suspended] thread
    0.31    0.00    1.00    0.43   68%  68% Resume thread
    0.56    0.00    1.00    0.49   56%  43% Set priority
    0.13    0.00    1.00    0.22   87%  87% Get priority
    1.13    1.00    2.00    0.22   87%  87% Kill [suspended] thread
    0.38    0.00    1.00    0.47   62%  62% Yield [no other] thread
    0.69    0.00    2.00    0.52   56%  37% Resume [suspended low prio] thread
    0.44    0.00    1.00    0.49   56%  56% Resume [runnable low prio] thread
    0.56    0.00    1.00    0.49   56%  43% Suspend [runnable] thread
    0.44    0.00    1.00    0.49   56%  56% Yield [only low prio] thread
    0.50    0.00    1.00    0.50  100%  50% Suspend [runnable->not runnable]
    1.19    1.00    2.00    0.30   81%  81% Kill [runnable] thread
    1.06    1.00    2.00    0.12   93%  93% Destroy [dead] thread
    2.06    2.00    3.00    0.12   93%  93% Destroy [runnable] thread
    2.63    2.00    3.00    0.47   62%  37% Resume [high priority] thread
    0.78    0.00    2.00    0.35   76%  22% Thread switch

    0.13    0.00    1.00    0.22   87%  87% Scheduler lock
    0.34    0.00    1.00    0.45   66%  66% Scheduler unlock [0 threads]
    0.30    0.00    1.00    0.42   70%  70% Scheduler unlock [1 suspended]
    0.26    0.00    1.00    0.38   74%  74% Scheduler unlock [many suspended]
    0.29    0.00    1.00    0.41   71%  71% Scheduler unlock [many low prio]

    0.22    0.00    1.00    0.34   78%  78% Init mutex
    0.50    0.00    1.00    0.50  100%  50% Lock [unlocked] mutex
    0.47    0.00    1.00    0.50   53%  53% Unlock [locked] mutex
    0.47    0.00    1.00    0.50   53%  53% Trylock [unlocked] mutex
    0.44    0.00    1.00    0.49   56%  56% Trylock [locked] mutex
    0.25    0.00    1.00    0.38   75%  75% Destroy mutex
    4.00    4.00    4.00    0.00  100% 100% Unlock/Lock mutex

    0.22    0.00    1.00    0.34   78%  78% Create mbox
    0.13    0.00    1.00    0.22   87%  87% Peek [empty] mbox
    0.50    0.00    1.00    0.50  100%  50% Put [first] mbox
    0.09    0.00    1.00    0.17   90%  90% Peek [1 msg] mbox
    0.47    0.00    1.00    0.50   53%  53% Put [second] mbox
    0.16    0.00    1.00    0.26   84%  84% Peek [2 msgs] mbox
    0.56    0.00    1.00    0.49   56%  43% Get [first] mbox
    0.56    0.00    1.00    0.49   56%  43% Get [second] mbox
    0.50    0.00    1.00    0.50  100%  50% Tryput [first] mbox
    0.44    0.00    1.00    0.49   56%  56% Peek item [non-empty] mbox
    0.47    0.00    1.00    0.50   53%  53% Tryget [non-empty] mbox
    0.44    0.00    1.00    0.49   56%  56% Peek item [empty] mbox
    0.41    0.00    1.00    0.48   59%  59% Tryget [empty] mbox
    0.13    0.00    1.00    0.22   87%  87% Waiting to get mbox
    0.13    0.00    1.00    0.22   87%  87% Waiting to put mbox
    0.25    0.00    1.00    0.38   75%  75% Delete mbox
    2.59    2.00    3.00    0.48   59%  40% Put/Get mbox

    0.16    0.00    1.00    0.26   84%  84% Init semaphore
    0.34    0.00    1.00    0.45   65%  65% Post [0] semaphore
    0.44    0.00    1.00    0.49   56%  56% Wait [1] semaphore
    0.31    0.00    1.00    0.43   68%  68% Trywait [0] semaphore
    0.41    0.00    1.00    0.48   59%  59% Trywait [1] semaphore
    0.16    0.00    1.00    0.26   84%  84% Peek semaphore
    0.19    0.00    1.00    0.30   81%  81% Destroy semaphore
    2.31    2.00    3.00    0.43   68%  68% Post/Wait semaphore

    0.25    0.00    1.00    0.38   75%  75% Create counter
    0.19    0.00    1.00    0.30   81%  81% Get counter value
    0.16    0.00    1.00    0.26   84%  84% Set counter value
    0.59    0.00    1.00    0.48   59%  40% Tick counter
    0.13    0.00    1.00    0.22   87%  87% Delete counter

    0.13    0.00    1.00    0.22   87%  87% Init flag
    0.50    0.00    1.00    0.50  100%  50% Destroy flag
    0.34    0.00    1.00    0.45   65%  65% Mask bits in flag
    0.50    0.00    1.00    0.50  100%  50% Set bits in flag [no waiters]
    0.66    0.00    1.00    0.45   65%  34% Wait for flag [AND]
    0.59    0.00    1.00    0.48   59%  40% Wait for flag [OR]
    0.72    0.00    1.00    0.40   71%  28% Wait for flag [AND/CLR]
    0.66    0.00    1.00    0.45   65%  34% Wait for flag [OR/CLR]
    0.13    0.00    1.00    0.22   87%  87% Peek on flag

    0.25    0.00    1.00    0.38   75%  75% Create alarm
    0.69    0.00    1.00    0.43   68%  31% Initialize alarm
    0.38    0.00    1.00    0.47   62%  62% Disable alarm
    0.69    0.00    1.00    0.43   68%  31% Enable alarm
    0.50    0.00    1.00    0.50  100%  50% Delete alarm
    0.53    0.00    1.00    0.50   53%  46% Tick counter [1 alarm]
    2.19    2.00    3.00    0.30   81%  81% Tick counter [many alarms]
    1.00    1.00    1.00    0.00  100% 100% Tick & fire counter [1 alarm]
   13.06   13.00   14.00    0.12   93%  93% Tick & fire counters [>1 together]
    2.56    2.00    3.00    0.49   56%  43% Tick & fire counters [>1 separately]
    3.00    3.00    3.00    0.00  100% 100% Alarm latency [0 threads]
    2.05    2.00    3.00    0.10   94%  94% Alarm latency [2 threads]
    2.62    2.00    3.00    0.47   61%  38% Alarm latency [many threads]
    4.01    4.00    5.00    0.01   99%  99% Alarm -> thread resume latency

    0.00    0.00    0.00    0.00            Clock/interrupt latency

    1.83    1.00    2.00    0.00            Clock DSR latency

     200     180     212                    Worker thread stack used (stack size 1088)
            All done, main thrd : stack used   804 size  1536
            All done : Idlethread stack used   172 size  1280

Timing complete - 29740 ms total

PASS:<Basic timing OK>
EXIT:<done>
```

Name

Test Programs — Details

Test Programs

The STM32F746G-DISCO platform HAL contains some test programs which allow various aspects of the board to be tested.

Manual Test

The **manual** test is not built by default. The configuration option `CYGBLD_HAL_CORTEXM_STM32F746G_DISCO_TESTS_MANUAL` should be enabled to allow the test to be built.

This program tests various aspects of the basic platform port, e.g. checking I²C device access and that the push-button GPIO operates.

Chapter 302. STM32H735-DISCO Platform HAL

Name

`CYGPKG_HAL_CORTEXM_STM32_STM32H735_DISCO` — eCos Support for the STM32H735-DISCO Board

Description

This documentation describes the platform-specific elements of the STM32H735-DISCO board support within eCos. It should be read in conjunction with the STM32 variant HAL section, which covers the common functionality shared by all STM32 variants, including eCos HAL features and on-chip device support. In addition ST's “Discovery kit with STM32H735IG MCU” (ST User Manual id: UM2679) should be consulted for hardware setup and settings.

The board is equipped with an on-board ST-LINK/V2-1 hardware debugger interface (via the CN15 “USB ST-LINK” connector), which is typically used for eCos application development.

Supported Hardware

The STM32H735IG has two main on-chip memory regions. The device has a SRAM region of 320KiB present at 0x24000000, and a 1MiB FLASH region present at 0x08000000 (which is aliased to 0x00000000 during normal execution). A 512Mbit MX25LM51245G Octo SPI flash device is available through the OCTOSPI controller.

The STM32 variant HAL includes support for the eleven on-chip serial devices which are documented in the variant HAL. However, the STM32H735-DISCO motherboard only provides direct access to a single UART (no flow control signals, no RS-232 transceiver) via the CN8 connector. Indirect access to another UART is available via the ST-LINK hardware debugger.

The STM32 variant HAL also includes support for the I²C buses. There are no I²C devices on the board that eCos supports. I²C bus 4 is available on CN4.

Similarly the STM32 variant HAL includes support for the SPI buses. The discovery board does not provide any SPI devices as standard, but SPI bus 5 is available on CN4.

Device drivers are also provided for the STM32 on-chip Ethernet MAC and ADC controllers. Additionally, support is provided for the on-chip watchdog, and a Flash driver exists to permit management of the STM32's on-chip Flash.

The STM32H7 processor and the STM32H735-DISCO board provide a wide variety of peripherals, but unless support is specifically indicated, it should be assumed that it is not included.

Tools

The board port is intended to work with GNU tools configured for an **arm-eabi** target. The original port was done using **arm-eabi-gcc** version 7.3.0d, **arm-eabi-gdb** version 8.1, and **binutils** version 2.30.

Name

Setup — Preparing the STM32H735-DISCO Board for eCos Development

Overview

Typically, since the STM32H735-DISCO motherboard has a built-in ST-LINK/V2-1 interface providing hardware debug support, eCos applications are loaded and run via the debugger **arm-eabi-gdb** or via the Eclipse IDE. The debugger then communicates with the "GDB server" provided by the relevant host ST-LINK/V2-1 support tool being used (e.g. OpenOCD).

Normally for release applications the `ROM` startup type would be used, with the application programmed into the on-chip flash for execution when the board boots. It is still possible to use the hardware debugging support to debug such flash-based ROM applications, and this may be the desired approach if the application is too large for execution from on-chip SRAM, or where all of the SRAM is required for application run-time use.

If off-chip non-volatile memory (NVM) is used to hold the main application then the board can boot from the internal flash using a suitable boot loader. For example, the eCosPro BootUp ROM loader, where the BootUp code can start the main application (after an optional update sequence).

Preparing Board for Use

The STM32H735-DISCO board is distributed with some example firmware already loaded into the flash. This is useful for checking that the board is functional after unpacking. However, it is recommended that it be replaced before loading eCos applications for development since it can interfere with the board setup that eCos applications expect.

An executable, `stminfo.elf`, is provided as part of the release within the `prebuilt` subdirectory of the eCosPro release installation and this should be programmed into the ROM before use. Details on how to do this are described in the Programming ROM images section below.

Preparing ST-LINK/V3E interface

The ST-LINK/V3E firmware delivered with the board should be sufficiently up to date to work with debug servers like OpenOCD. The firmware for the ST-LINK/V3E interface can be checked, and updated if needed, using a tool available from STMicroelectronics. The firmware version is also reported when the **openocd** command is executed (using a suitable configuration file):

```
Info : STLINK V3J7M2 (API v3) VID:PID 0483:374E
```

The user should refer to the ST "ST-LINK/V3E firmware upgrade" Release Note, which provides detail on the host requirements for upgrading the ST-Link firmware on Linux, Mac OS X and Windows hosts.

Programming ROM images

Since the STM32H735-DISCO board has a built-in ST-LINK/V3E SWD interface, the USB host connection (CN15) and suitable host software (e.g. The OpenOCD package **openocd** tool) can be used to program the flash.

The **openocd** GDB server can directly program flash based applications from the GDB **load** command.

Note

The **openocd** command provided with the eCosPro Host Tools has been configured and built to support the ST-LINK/V3E interface. Should you wish to rebuild **openocd** yourself, you must specify the **--enable-stlink** option when configuring the OpenOCD build. Additional information on running **openocd** may be found in the OpenOCD notes.

For example, assuming that **openocd** is running on the same host as GDB, and is connected to the target board the following will program the `stminfo.elf` application into the on-chip flash:

```
$ arm-eabi-gdb stminfo.elf
GNU gdb (eCosCentric GNU tools 7.3.0d) 8.1
[ … GDB output elided … ]
(gdb) target extended-remote localhost:3333
Remote debugging using localhost:3333
=> 0x8000d40: push {r3, r4, r5, r6, r7, lr}
0x08000d40 in ?? ()
(gdb) load
Loading section .rom_vectors, size 0x8 lma 0x8000000
Loading section .text, size 0x3be8 lma 0x8000008
Loading section .rodata, size 0x6bc lma 0x8003bf0
Loading section .data, size 0x1c8 lma 0x80042b0
Start address 0x8000008, load size 17524
Transfer rate: 12 KB/sec, 4381 bytes/write.
(gdb) cont
```

Following the **cont** command, the following output should appear on the virtual UART:

```
INFO:<code from 0x08000008 -> 0x08003bf0, CRC 9425>
INFO:<STM32 CPU information>
INFO:<CDL Cortex-M7>
INFO:<MCU ID 10016483 DEV H72x/H73x REV Z>
INFO:<CPU reports flash size 1024K>
INFO:<Unique-ID: 0030001F 31395119 38323331>
INFO:<Variant Unique-ID maximum length 12>
INFO:<CYGARC_HAL_CORTEXM_STM32_INPUT_CLOCK 25000000>
INFO:<SYSCLK 550000000>
INFO:<HCLK   275000000>
INFO:<PCLK1  137500000>
INFO:<PCLK2  137500000>
INFO:<PCLK3  137500000>
INFO:<PCLK4  137500000>
INFO:<QCLK   68750000>
INFO:<Cortex-M systick 68750000>
PASS:<Done>
EXIT:<done>
```

Alternatively, the **openocd** telnet interface can be used to manually program the flash. By default the **openocd** session provides a comand-line via port `4444`. Consult the OpenOCD documentation for more details if a non-default **openocd** configuration is being used.

With a **telnet** connection established to the **openocd** any binary data can easily be written to the on-chip flash. e.g.

```
$ telnet localhost 4444
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Open On-Chip Debugger
> flash write_image stminfo.bin 0x08000000
Device: STM32H72x/73x
flash size probed value 1024
STM32H7 flash has a single bank
Bank (0) size is 1024 kb, base address is 0x08000000
Padding image section 0 at 0x08004318 with 8 bytes (bank write end alignment)
wrote 17184 bytes from file app.bin in 0.270935s (61.938 KiB/s)
```

To create a binary for flash programming the **arm-eabi-objcopy** command is used. This converts the, ELF format, linked application into a raw binary. For example:

```
$ arm-eabi-objcopy -O binary programname programname.bin
```

Name

Configuration — Platform-specific Configuration Options

Overview

The STM32H735-DISCO board platform HAL package CYGPKG_HAL_CORTEXM_STM32_STM32H735_DISCO is loaded automatically when eCos is configured for the stm32h735_disco target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The STM32H735-DISCO board platform HAL package supports five separate startup types:

ROM
: This startup type can be used for finished applications which will be programmed into internal flash at location 0x08000000. Data and BSS will be put into internal SRAM starting from 0x240003CC. Internal SRAM below this address is reserved for vector tables. The application will be self-contained with no dependencies on services provided by other software. The program expects to boot from reset with ROM mapped at location zero. It will then transfer control to the 0x08000000 region. eCos startup code will perform all necessary hardware initialization.

  This startup type will normally be used for production applications. It may also be used for development but over-use of flash during debugging may result in flash wear. It is advised to use the JTAG startup type during development if possible.

JTAG
: This is the startup type used to build applications that are loaded via the hardware debugger interface. The application will be self-contained with no dependencies on services provided by other software. The program expects to be loaded into on-chip SRAM from 0x24000000 and entered at that address. eCos startup code will perform all necessary hardware initialization, though since the application is loaded via the hardware debugger interface the host debug environment may perform some initialization.

  This is the startup type normally used during application development, since it avoids wear on the flash memory. However, SRAM is only 320kiB and not all applications will fit solely into SRAM. In such cases, a ROM startup application should be used.

SPI Driver

An SPI bus driver is available for the STM32 in the package “ST STM32 SPI driver” (CYGPKG_DEVS_SPI_CORTEXM_STM32).

SPI bus 5 is instantiated at CYGPKG_HAL_CORTEXM_STM32_STM32H735_DISCO_SPI and is available on Arduino header CN4. No SPI devices are instantiated for this platform by default.

Consult the generic SPI driver API documentation in the eCosPro Reference Manual for further details on SPI support in eCosPro, along with the configuration options in the STM32 SPI device driver.

I²C Driver

The STM32 variant HAL provides the main I²C hardware driver itself, configured at CYGPKG_HAL_STM32_I2C. However, the platform I²C support can also be configured separately at CYGPKG_HAL_CORTEXM_STM32_STM32H735_DISCO_I2C. This enables I²C bus 4 which is available on Arduino header CN4.

Flash Driver

The STM32's on-chip Flash may be programmed and managed using the Flash driver located in the “STM32 Flash memory support” (CYGPKG_DEVS_FLASH_STM32) package. This driver is enabled automatically if the generic “Flash device drivers” (CYG-

PKG_IO_FLASH) package is included in the eCos configuration. The driver will configure itself automatically for the size and parameters of the specific STM32 variant present on the STM32H735-DISCO board.

A number of aspects of Flash driver behaviour can be configured within that driver, such as program/erase parallelism and program burst size. Consult the driver for more details.

OCTOSPI Flash Driver

When OCTOSPI NOR flash support is enabled in the configuration with CYGHWR_HAL_CORTEXM_STM32_FLASH_OCTOSPI, then the cyg_stm32_octospi1_device device is exported and can be accessed via the standard flash API. The device is given a logical base address to match its physical base address of 0x90000000 (corresponding to FMC bank 4) when it is memory mapped (if CYGFUN_DEVS_FLASH_OCTOSPI1_CORTEXM_STM32_MEMMAPPED is enabled in the OCTOSPI driver, which is not the default). When memory mapping is disabled, using the eCos Flash API will still allow the device to be read/written at that logical base address.

Ethernet Driver

The Ethernet MAC is connected to a LAN8742A PHY via the RMII interface and thence to a RJ45 connector at CN3. The external 25MHz crystal is used to supply the clock.

Note

It is **highly** recommended that the configuration option CYGHWR_HAL_CORTEXM_STM32_SRAM_ALTERNATE is **ENABLED**. Enabling that feature configures the Ethernet driver RX memory buffers to the SRAM2 space in the D2 domain. This is required to avoid an undocumented STM32H735 revZ errata where the Ethernet MAC would occasionally (rare) corrupt memory if the AXI SRAM was used for the RX buffers. The downside of the option is that it will mean a smaller number of RX buffers being available than is possible with the larger (main) AXI SRAM space.

ADC Driver

The STM32 processor variant HAL provides an ADC driver. The STM32H735-DISCO platform HAL enables the support for all three devices and for configuration of the respective ADC device input channels.

Consult the generic ADC driver API documentation in the eCosPro Reference Manual for further details on ADC support in eCosPro, along with the configuration options in the STM32 ADC device driver.

Name

SWD support — Usage

Use of JTAG/SWD for debugging

JTAG/SWD can be used to single-step and debug loaded applications, or even applications resident in ROM.

Debugging of ROM applications is only possible if using hardware breakpoints. The Cortex-M7 core of the STM32H735IG only supports eight such hardware breakpoints, so they may need to be used sparingly. If using a GDB front-end such as Eclipse, check it has not set unnecessary extra breakpoints such as at `main()`. Some JTAG/SWD devices give the option of whether to set hardware or software breakpoints by default. Be sure to configure your device appropriately.

When debugging via JTAG, you are likely to need to disable the default HAL idle thread action, otherwise there may be issues where the target fails to halt and the debugging session is unreliable. More details can be found in the Cortex-M architectural HAL. This should *not* be necessary when using a SWD-based hardware debugger such as the on-board ST-LINK/V3E interface.

Using the ST-LINK/V3E USB connection allows for a single cable to provide power, hardware debug support and diagnostic output. The latter is provided via a virtual UART which instantiates an ACM compatible serial channel in the host which is connected to USART2 on the STM32H735IG. A separate application may be run alongside the debugger to capture the output from this UART, such as **minicom** under Linux or **PuTTY** under Windows.

OpenOCD notes

OpenOCD version `0.11.0` and above is required to support the STM32H735IG MCU. A prebuilt host **openocd** executable which also supports the on-board ST-LINK/V3E interface available via the USB CN15 connection has been provided with the eCosPro Host Tools version `5.0.0` and above.

Should you wish to rebuild **openocd** yourself, you must specify the `--enable-stlink` option when configuring the OpenOCD build to provide for ST-LINK support.

An example OpenOCD configuration file `openocd.stm32h735_disco.cfg` is provided within the eCos platform HAL package in the source repository. This will be in the directory `packages/hal/cortexm/stm32/stm32h735_disco/current/misc` relative to the root of your eCosPro installation, but for convenience it is copied into the `install/etc` subdirectory as `openocd.cfg` during the **make etc** build process of the eCosPro library.

This configuration file can be used with OpenOCD on the host as follows:

```
$ openocd -f openocd.stm32h735_disco.cfg
Open On-Chip Debugger 0.11.0 (eCosCentric)
Licensed under GNU GPL v2
For bug reports, read
 http://openocd.org/doc/doxygen/bugs.html
Info : The selected transport took over low-level target control. The results might differ compared to plain JTAG/SWD
Info : Listening on port 6666 for tcl connections
Info : Listening on port 4444 for telnet connections
Info : clock speed 1800 kHz
Info : STLINK V3J7M2 (API v3) VID:PID 0483:374E
Info : Target voltage: 3.287824
Info : stm32h7x.cpu0: hardware has 8 breakpoints, 4 watchpoints
Info : starting gdb server for stm32h7x.cpu0 on 3333
Info : Listening on port 3333 for gdb connections
```

By default **openocd** provides a telnet console on port `4444`, and this can be used to interact with the target system. This console interface can be used to perform debugging, program the flash, etc.

Normally **arm-eabi-gdb** is used to connect to the default GDB server port `3333` for debugging. For example:

```
(gdb) target extended-remote localhost:3333
Remote debugging using localhost:3333
0x00000000 in ?? ()
```

```
(gdb)
```

OpenOCD should report the following on its terminal when a GDB connection is made:

```
Info : accepting 'gdb' connection on tcp/3333

Initialising CPU...
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x08000d40 msp: 0x2404fc00
force hard breakpoints
Note: Breakpoints limited to 8 hardware breakpoints
Invalidate ICACHE
Invalidate DCACHE
Disable Caches

Info : Device: STM32H72x/73x
Info : flash size probed value 1024
Info : STM32H7 flash has a single bank
Info : Bank (0) size is 1024 kb, base address is 0x08000000
```

The application can then be loaded and executed under GDB as normal. If you are using Eclipse then, if required, you can define a “preload” gdb macro to emit any necessary commands to OpenOCD. See the “Hardware Assisted Debugging” section of the “Eclipse/CDT for eCos application development” document's “Debugging eCos applications” chapter.

Note

Incompatibilities between OpenOCD and the STM32H7 cache support mean that at present only hardware breakpoints should be used for debugging. The OpenOCD configuration file provided within the eCosPro Developer's kit enforces this.

Configuration of JTAG/SWD applications

JTAG/SWD applications can be loaded directly into SRAM without requiring a ROM monitor. Loading can be done directly through the JTAG/SWD device, or through GDB where supported by the JTAG/SWD device.

In order to configure the application to support this mode, it is recommended to use the `JTAG` startup type which will implicitly cause two important settings to change. Firstly, `CYGSEM_HAL_USE_ROM_MONITOR` must be disabled. Secondly the `CYGDBG_HAL_DIAG_TO_DEBUG_CHAN` option should be enabled in order to prevent HAL diagnostic output being encoded into GDB ($O) packets. These configuration changes could be made by hand, but use of the `JTAG` startup type will just work.

With these changes, any diagnostic output will appear out of the configured diagnostic channel. An eCosCentric extension allows diagnostic output to appear in GDB. For this feature to work, you must enable the configuration option `CYGSEM_HAL_DIAG_TO_GDBFILEIO_CHAN` in the common HAL package. If you are using the graphical configuration tool then you should then accept any suggested solutions to the subsequent configuration conflicts. Older eCos releases also required the gdb "set hwdebug on" command to be used to enable GDB or Eclipse console output, but this is no longer required with the latest tools.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the STM32H735-DISCO board hardware, and should be read in conjunction with that specification. The STM32H735-DISCO platform HAL package complements the Cortex-M architectural HAL and the STM32 variant HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset the HAL will initialize or reinitialize many of the on-chip peripherals. For both `ROM` and `JTAG` startup types the HAL will perform all initialization, programming the various internal registers including the PLLs, peripheral clocks and GPIO pins. The details of the early hardware startup may be found in the `src/stm32h735_disco_misc.c` in both `hal_system_init` and `hal_platform_init`.

Memory Map

The platform HAL package provides the memory layout information needed to generate the linker script. For all the STARTUP variations the top `CYGNUM_HAL_COMMON_INTERRUPTS_STACK_SIZE` bytes of the on-chip SRAM are reserved for the interrupt stack. The remainder of the internal SRAM is available for use by applications. The key memory locations are as follows:

Internal RAM
: This is located at address `0x24000000` of the memory space, and is 320KiB in size. The eCos VSR table occupies the bottom 716-bytes of memory, with the virtual vector table starting at `0x240002CC` and extending to `0x240003CC`.

This memory region comprises two contiguous memory blocks, the 128kiB AXI-SRAM plus 192kiB of shared SRAM in the default configuration where it is assigned to the AXI-SRAM.

SRAM1
: This is located at address `0x30000000` of the memory space, and is 16KiB in size. This is used to contain the Ethernet transmit and receive descriptor rings. It is mapped by an MPU region that disables caching for accesses.

Internal FLASH
: This is located at address `0x08000000` of the memory space and will be mapped to `0x00000000` at reset. This region is 1024KiB in size. `ROM` applications are by default configured to run from this memory. The 256kiB from offset 0xC0000 is used for flash testing. The test space is defined in `__STM32H735_DISCO_FLASHTEST_ONCHIP` in `plf_io.h`.

OCTOSPI NOR Flash
: The OCTOSPI NOR flash is accessible through the flash API. The 256kiB from offset 0x100000 is used for flash testing. The test space is defined in `__STM32H735_DISCO_FLASHTEST_OCTOSPI` in `plf_io.h`.

On-Chip Peripherals
: These are accessible at locations `0x40000000` and `0xE0000000` upwards. Descriptions of the contents can be found in the STM32 User Manual.

Linker Scripts

The platform linker scripts define the following symbols:

hal_vsr_table
: This defines the location of the VSR table. This is set to `0x20000000` for all startup types, and space for 114 entries is reserved.

hal_virtual_vector_table
: This defines the location of the virtual vector table used to communicate between an ROM monitor and an eCos application. This is allocated right after the VSR table, at `0x200001C8`.

hal_interrupt_stack
This defines the location of the interrupt stack. This is allocated to the top of internal SRAM, `0x20050000`.

hal_startup_stack
This defines the location of the startup stack. For all startup types it is initially allocated at the half-way point of the interrupt stack.

Flash wait states

The STM32H735-DISCO platform HAL provides a configuration option to set the number of Flash read wait states to use: `CYGNUM_HAL_CORTEXM_STM32_FLASH_WAIT_STATES`. It is important to verify and if necessary update this value if changing the CPU clock (HCLK) frequency or CPU voltage. Consult the relevant STM32 datasheets and programming manuals for the STM32H735 parts for appropriate values for different clock speeds or voltages. The default of 5 reflects the default HCLK frequency.

Real-time characterization

The **tm_basic** kernel test gives statistics gathered about the real-time characterization and performance of the kernel. The sample output is shown here for information. The test was built for `ROM` startup with optimization flag -O2, since it provides the best performance as both code and data could remain on-chip.

Example 302.1. stm32h735_disco Real-time characterization

```
             Startup, main thrd : stack used   356 size  2048
             Startup : Idlethread stack used    76 size  1280

eCos Kernel Timings
Notes: all times are in microseconds (.000001) unless otherwise stated

Reading the hardware clock takes 0 'ticks' overhead
... this value will be factored out of all other measurements
Clock interrupt took    1.00 microseconds (1 raw clock ticks)

Testing parameters:
   Clock samples:            32
   Threads:                  15
   Thread switches:         128
   Mutexes:                  32
   Mailboxes:                32
   Semaphores:               32
   Scheduler operations:    128
   Counters:                 32
   Flags:                    32
   Alarms:                   32
   Stack Size:             1088


                                 Confidence
     Ave     Min     Max     Var  Ave  Min  Function
  ======  ======  ======  ====== ========== ========
INFO:<Ctrl-C disabled until test completion>
    0.80    0.00    2.00    0.43   66%  26% Create thread
    0.27    0.00    1.00    0.39   73%  73% Yield thread [all suspended]
    0.27    0.00    1.00    0.39   73%  73% Suspend [suspended] thread
    0.27    0.00    1.00    0.39   73%  73% Resume thread
    0.27    0.00    1.00    0.39   73%  73% Set priority
    0.07    0.00    1.00    0.12   93%  93% Get priority
    0.53    0.00    1.00    0.50   53%  46% Kill [suspended] thread
    0.27    0.00    1.00    0.39   73%  73% Yield [no other] thread
    0.27    0.00    1.00    0.39   73%  73% Resume [suspended low prio] thread
    0.27    0.00    1.00    0.39   73%  73% Resume [runnable low prio] thread
    0.33    0.00    1.00    0.44   66%  66% Suspend [runnable] thread
    0.20    0.00    1.00    0.32   80%  80% Yield [only low prio] thread
    0.13     0.00     1.00     0.23    86%  86% Suspend [runnable->not runnable]
    0.47     0.00     1.00     0.50    53%  53% Kill [runnable] thread
    0.47     0.00     1.00     0.50    53%  53% Destroy [dead] thread
    1.07     1.00     2.00     0.12    93%  93% Destroy [runnable] thread
    1.20     1.00     2.00     0.32    80%  80% Resume [high priority] thread
    0.40     0.00     1.00     0.48    60%  60% Thread switch

    0.07     0.00     1.00     0.13    92%  92% Scheduler lock
    0.20     0.00     1.00     0.31    80%  80% Scheduler unlock [0 threads]
    0.19     0.00     1.00     0.30    81%  81% Scheduler unlock [1 suspended]
    0.17     0.00     1.00     0.28    82%  82% Scheduler unlock [many suspended]
    0.05     0.00     1.00     0.10    94%  94% Scheduler unlock [many low prio]

    0.09     0.00     1.00     0.17    90%  90% Init mutex
    0.25     0.00     1.00     0.38    75%  75% Lock [unlocked] mutex
    0.25     0.00     1.00     0.38    75%  75% Unlock [locked] mutex
    0.25     0.00     1.00     0.38    75%  75% Trylock [unlocked] mutex
    0.22     0.00     1.00     0.34    78%  78% Trylock [locked] mutex
    0.09     0.00     1.00     0.17    90%  90% Destroy mutex
    2.00     2.00     2.00     0.00   100% 100% Unlock/Lock mutex

    0.13     0.00     1.00     0.22    87%  87% Create mbox
    0.09     0.00     1.00     0.17    90%  90% Peek [empty] mbox
    0.25     0.00     1.00     0.38    75%  75% Put [first] mbox
    0.09     0.00     1.00     0.17    90%  90% Peek [1 msg] mbox
    0.25     0.00     1.00     0.38    75%  75% Put [second] mbox
    0.06     0.00     1.00     0.12    93%  93% Peek [2 msgs] mbox
    0.28     0.00     1.00     0.40    71%  71% Get [first] mbox
    0.19     0.00     1.00     0.30    81%  81% Get [second] mbox
    0.25     0.00     1.00     0.38    75%  75% Tryput [first] mbox
    0.19     0.00     1.00     0.30    81%  81% Peek item [non-empty] mbox
    0.22     0.00     1.00     0.34    78%  78% Tryget [non-empty] mbox
    0.22     0.00     1.00     0.34    78%  78% Peek item [empty] mbox
    0.22     0.00     1.00     0.34    78%  78% Tryget [empty] mbox
    0.06     0.00     1.00     0.12    93%  93% Waiting to get mbox
    0.06     0.00     1.00     0.12    93%  93% Waiting to put mbox
    0.13     0.00     1.00     0.22    87%  87% Delete mbox
    1.16     1.00     2.00     0.26    84%  84% Put/Get mbox

    0.03     0.00     1.00     0.06    96%  96% Init semaphore
    0.16     0.00     1.00     0.26    84%  84% Post [0] semaphore
    0.25     0.00     1.00     0.38    75%  75% Wait [1] semaphore
    0.19     0.00     1.00     0.30    81%  81% Trywait [0] semaphore
    0.25     0.00     1.00     0.38    75%  75% Trywait [1] semaphore
    0.03     0.00     1.00     0.06    96%  96% Peek semaphore
    0.03     0.00     1.00     0.06    96%  96% Destroy semaphore
    1.00     1.00     1.00     0.00   100% 100% Post/Wait semaphore

    0.06     0.00     1.00     0.12    93%  93% Create counter
    0.09     0.00     1.00     0.17    90%  90% Get counter value
    0.09     0.00     1.00     0.17    90%  90% Set counter value
    0.22     0.00     1.00     0.34    78%  78% Tick counter
    0.06     0.00     1.00     0.12    93%  93% Delete counter

    0.09     0.00     1.00     0.17    90%  90% Init flag
    0.22     0.00     1.00     0.34    78%  78% Destroy flag
    0.19     0.00     1.00     0.30    81%  81% Mask bits in flag
    0.22     0.00     1.00     0.34    78%  78% Set bits in flag [no waiters]
    0.25     0.00     1.00     0.38    75%  75% Wait for flag [AND]
    0.22     0.00     1.00     0.34    78%  78% Wait for flag [OR]
    0.31     0.00     1.00     0.43    68%  68% Wait for flag [AND/CLR]
    0.28     0.00     1.00     0.40    71%  71% Wait for flag [OR/CLR]
    0.06     0.00     1.00     0.12    93%  93% Peek on flag

    0.13     0.00     1.00     0.22    87%  87% Create alarm
    0.31     0.00     1.00     0.43    68%  68% Initialize alarm
    0.16     0.00     1.00     0.26    84%  84% Disable alarm
    0.34    0.00    1.00    0.45   65%  65% Enable alarm
    0.13    0.00    1.00    0.22   87%  87% Delete alarm
    0.28    0.00    1.00    0.40   71%  71% Tick counter [1 alarm]
    0.91    0.00    1.00    0.17   90%   9% Tick counter [many alarms]
    0.41    0.00    1.00    0.48   59%  59% Tick & fire counter [1 alarm]
    5.81    5.00    6.00    0.31   81%  18% Tick & fire counters [>1 together]
    1.06    1.00    2.00    0.12   93%  93% Tick & fire counters [>1 separately]
    1.00    1.00    1.00    0.00  100% 100% Alarm latency [0 threads]
    1.00    1.00    1.00    0.00  100% 100% Alarm latency [2 threads]
    1.00    1.00    1.00    0.00  100% 100% Alarm latency [many threads]
    2.00    2.00    2.00    0.00  100% 100% Alarm -> thread resume latency

    0.00    0.00    0.00    0.00            Clock/interrupt latency

    1.00    1.00    1.00    0.00            Clock DSR latency

     204     180     220                    Worker thread stack used (stack size 1088)
            All done, main thrd : stack used   696 size  2048
            All done : Idlethread stack used   172 size  1280

Timing complete - 29810 ms total

PASS:<Basic timing OK>
EXIT:<done>
```

Name

Test Programs — Details

Test Programs

The STM32H735-DISCO platform HAL contains some test programs which allow various aspects of the board to be tested.

ADC Test

There are two tests for the ADC, which may be built by enabling `CYGBLD_HAL_CORTEXM_STM32H735_DISCO_TESTS_ADC`.

The **adc1** test reads the `Vrefint` ADC reference voltage, `Vsense` internal temperature, and `Vbat` battery voltage from channels 16, 17 and 18 of ADC3. These are printed out whenever any changes by a significant amount.

The **adc2** test reads the same sensors as **adc1** and in addition reads channel 10 of ADC1, which is connected to the A0 pin of Arduino connector CN9. A potentiometer attached to this pin allows various tests of the ADC system to be performed as prompted by the test itself.

Chapter 303. STM32H7 Nucleo-144 Platform HAL

Name

`CYGPKG_HAL_CORTEXM_STM32_NUCLEO144` — eCos Support for the STM32H7 Nucleo-144 Board

Description

This documentation describes the platform-specific elements of the STM32H7 Nucleo-144 (MB1364) board support within eCos. It should be read in conjunction with the STM32 variant HAL section, which covers the common functionality shared by all STM32 variants, including eCos HAL features and on-chip device support. In addition ST's “STM32H7 Nucleo-144 boards” (ST User Manual id: UM2407) should be consulted for hardware setup and settings.

The board is equipped with an on-board ST-LINK-V3 hardware debugger interface (via the CN1 “USB ST-LINK” connector), which is typically used for eCos application development.

Note

The STM32H7 Nucleo-144 design has multiple motherboard variants and CPU combinations. Currently only the STM32H723ZG variant is supported by eCos using the `nucleo144_stm32h723` platform.

Supported Hardware

The STM32H723ZG has three main on-chip memory regions. The device has a SRAM region of 320KiB present at 0x24000000, a 16K SRAM region at 0x30000000 and a 1MiB FLASH region present at 0x08000000 (which is aliased to 0x00000000 during normal execution).

Optionally a QSPI flash device can be connected to the relevant CN10 pins. When eCos is suitably configured access will be provided through the OCTOSPI1 controller.

The STM32 variant HAL includes support for the eleven on-chip serial devices which are documented in the variant HAL. However, the STM32H7 Nucleo-144 (MB1364) motherboard only provides direct access to a single UART (no RS-232 transceiver) via the CN9 connector. Indirect access to another UART is available via the ST-LINK hardware debugger.

The STM32 variant HAL also includes support for the I²C buses. There are no I²C devices on the board that eCos supports. I²C bus 1 or 4 is available on CN7. I²C bus 2 or 5 is available on CN9.

Similarly the STM32 variant HAL includes support for the SPI buses. The MB1364 board does not provide any SPI devices as standard, but SPI bus 1 is available on CN7 (MB1364 label SPI_A).

Device drivers are also provided for the STM32 on-chip Ethernet MAC and ADC controllers. Additionally, support is provided for the on-chip watchdog, and a Flash driver exists to permit management of the STM32's on-chip Flash.

The STM32H7 processor and the STM32H7 Nucleo-144 board provides for a wide variety of connected peripherals, but unless support is specifically indicated, it should be assumed that it is not included.

Tools

The board port is intended to work with GNU tools configured for an **arm-eabi** target. The original port was done using **arm-eabi-gcc** version 7.3.0d, **arm-eabi-gdb** version 8.1, and **binutils** version 2.30.

Name

Setup — Preparing the STM32H7 Nucleo-144 Board for eCos Development

Overview

Typically, since the STM32H7 Nucleo-144 motherboard has a built-in STLINK-V3 interface providing hardware debug support, eCos applications are loaded and run via the debugger **arm-eabi-gdb** or via the Eclipse IDE. The debugger then communicates with the “GDB server” provided by the relevant host STLINK-V3 support tool being used (e.g. OpenOCD).

Normally for release applications the `ROM` startup type would be used, with the application programmed into the on-chip flash for execution when the board boots. It is still possible to use the hardware debugging support to debug such flash-based ROM applications, and this may be the desired approach if the application is too large for execution from on-chip SRAM, or where all of the SRAM is required for application run-time use.

If off-chip non-volatile memory (NVM) is used to hold the main application then the board can boot from the internal flash using a suitable boot loader. For example, the eCosPro BootUp ROM loader, where the BootUp code can start the main application (after an optional update sequence).

Preparing Board for Use

The STM32H7 Nucleo-144 board is distributed with some example firmware already loaded into the flash. This is useful for checking that the board is functional after unpacking. However, it is recommended that it be replaced before loading eCos applications for development since it can interfere with the board setup that eCos applications expect.

An executable, `stminfo.elf`, is provided as part of the release within the `prebuilt` subdirectory of the eCosPro release installation and this should be programmed into the ROM before use. Details on how to do this are described in the Programming ROM images section below.

Preparing the ST-LINK/V3E interface

The ST-LINK/V3E firmware delivered with the board should be sufficiently up to date to work with debug servers like OpenOCD. The firmware for the ST-LINK/V3E interface can be checked, and updated if needed, using a tool available from STMicroelectronics. The firmware version is also reported when the **openocd** command is executed (using a suitable configuration file):

```
Info : STLINK V3J8M3 (API v3) VID:PID 0483:374E
```

The user should refer to the ST “ST-LINK/V3E firmware upgrade” Release Note, which provides detail on the host requirements for upgrading the ST-Link firmware on Linux, Mac OS X and Windows hosts.

Programming ROM images

Since the STM32H7 Nucleo-144 board has a built-in ST-LINK/V3E SWD interface, the USB host connection (CN1) and suitable host software (e.g. The OpenOCD package **openocd** tool) can be used to program the flash.

The **openocd** GDB server can directly program flash based applications from the GDB **load** command.

Note

The **openocd** command provided with the eCosPro Host Tools has been configured and built to support the ST-LINK/V3E interface. Should you wish to rebuild **openocd** yourself, you must specify the **--enable-stlink** option when configuring the OpenOCD build. Additional information on running **openocd** may be found in the OpenOCD notes.

For example, assuming that **openocd** is running on the same host as GDB, and is connected to the target board the following will program the `stminfo.elf` application into the on-chip flash:

```
$ arm-eabi-gdb stminfo.elf
GNU gdb (eCosCentric GNU tools 7.3.0d) 8.1
[ … GDB output elided … ]
(gdb) target extended-remote localhost:3333
Remote debugging using localhost:3333
=> 0x8000d40: push {r3, r4, r5, r6, r7, lr}
0x08000d40 in ?? ()
(gdb) load
Loading section .rom_vectors, size 0x8 lma 0x8000000
Loading section .text, size 0x3be8 lma 0x8000008
Loading section .rodata, size 0x6bc lma 0x8003bf0
Loading section .data, size 0x1c8 lma 0x80042b0
Start address 0x8000008, load size 17524
Transfer rate: 12 KB/sec, 4381 bytes/write.
(gdb) cont
```

Following the **cont** command, the following output should appear on the virtual UART:

```
INFO:<code from 0x08000008 -> 0x08003f00, CRC c29f>
INFO:<STM32 CPU information>
INFO:<CDL Cortex-M7>
INFO:<MCU ID 10016483 DEV H72x/H73x REV Z>
INFO:<CPU reports flash size 1024K>
INFO:<Unique-ID: 000E000F 31395118 38323331>
INFO:<Variant Unique-ID maximum length 12>
INFO:<CYGARC_HAL_CORTEXM_STM32_INPUT_CLOCK 8000000>
INFO:<SYSCLK 400000000>
INFO:<HCLK   200000000>
INFO:<PCLK1  100000000>
INFO:<PCLK2  100000000>
INFO:<PCLK3  100000000>
INFO:<PCLK4  100000000>
INFO:<QCLK    50000000>
INFO:<Cortex-M systick 50000000>
INFO:<Dcache enabled>
INFO:<Icache enabled>
PASS:<Done>
EXIT:<done>
```

Alternatively, the **openocd** telnet interface can be used to manually program the flash. By default the **openocd** session provides a comand-line via port `4444`. Consult the OpenOCD documentation for more details if a non-default **openocd** configuration is being used.

With a **telnet** connection established to the **openocd** any binary data can easily be written to the on-chip flash. e.g.

```
$ telnet localhost 4444
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Open On-Chip Debugger
> flash write_image stminfo.bin 0x08000000
Padding image section 0 at 0x080047c4 with 28 bytes (bank write end alignment)
wrote 18400 bytes from file stminfo.bin in 0.073288s (245.180 KiB/s)
```

To create a binary for flash programming the **arm-eabi-objcopy** command is used. This converts the, ELF format, linked application into a raw binary. For example:

```
$ arm-eabi-objcopy -O binary programname programname.bin
```

Name

Configuration — Platform-specific Configuration Options

Overview

The STM32H7 Nucleo-144 board platform HAL package `CYGPKG_HAL_CORTEXM_STM32_NUCLEO144` is loaded automatically when eCos is configured for the `nucleo144_stm32h723` target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The STM32H7 Nucleo-144 board platform HAL package supports two startup types:

ROM
: This startup type can be used for finished applications which will be programmed into internal flash at location 0x08000000. Data and BSS will be put into internal SRAM starting from 0x240003CC. Internal SRAM below this address is reserved for vector tables. The application will be self-contained with no dependencies on services provided by other software. The program expects to boot from reset with ROM mapped at location zero. It will then transfer control to the 0x08000000 region. eCos startup code will perform all necessary hardware initialization.

  This startup type will normally be used for production applications. It may also be used for development but over-use of flash during debugging may result in flash wear. It is advised to use the JTAG startup type during development if possible.

JTAG
: This is the startup type used to build applications that are loaded via the hardware debugger interface. The application will be self-contained with no dependencies on services provided by other software. The program expects to be loaded into on-chip SRAM from 0x24000000 and entered at that address. eCos startup code will perform all necessary hardware initialization, though since the application is loaded via the hardware debugger interface the host debug environment may perform some initialization.

  This is the startup type normally used during application development, since it avoids wear on the flash memory. However, SRAM is only 320kiB and not all applications will fit solely into SRAM. In such cases, a ROM startup application should be used.

SPI Driver

An SPI bus driver is available for the STM32 in the package “ST STM32 SPI driver” (`CYGPKG_DEVS_SPI_CORTEXM_STM32`).

SPI bus 1 is instantiated at `CYGPKG_HAL_CORTEXM_STM32_NUCLEO144_SPI` and is available on Arduino header CN7. No SPI devices are instantiated for this platform by default.

Consult the generic SPI driver API documentation in the eCosPro Reference Manual for further details on SPI support in eCosPro, along with the configuration options in the STM32 SPI device driver.

I²C Driver

The STM32 variant HAL provides the main I²C hardware driver itself, configured at `CYGPKG_HAL_STM32_I2C`. However, the platform I²C support can also be configured separately at `CYGPKG_HAL_CORTEXM_STM32_NUCLEO144_I2C`. This enables I²C bus 4 which is available on Arduino header CN7.

Flash Driver

The STM32's on-chip Flash may be programmed and managed using the Flash driver located in the “STM32 Flash memory support” (`CYGPKG_DEVS_FLASH_STM32`) package. This driver is enabled automatically if the generic “Flash device drivers” (`CYG-`

PKG_IO_FLASH) package is included in the eCos configuration. The driver will configure itself automatically for the size and parameters of the specific STM32 variant present on the STM32H7 Nucleo-144 board.

A number of aspects of Flash driver behaviour can be configured within that driver, such as program/erase parallelism and program burst size. Consult the driver for more details.

OCTOSPI Flash Driver

When OCTOSPI NOR flash support is enabled in the configuration with CYGHWR_HAL_CORTEXM_STM32_FLASH_OCTOSPI, then the cyg_stm32_octospi1_device device is exported and can be accessed via the standard flash API. The device is given a logical base address to match its physical base address of 0x90000000 (corresponding to FMC bank 4) when it is memory mapped (if CYGFUN_DEVS_FLASH_OCTOSPI1_CORTEXM_STM32_MEMMAPPED is enabled in the OCTOSPI driver, which is not the default). When memory mapping is disabled, using the eCos Flash API will still allow the device to be read/written at that logical base address.

Ethernet Driver

The Ethernet MAC is connected to a LAN8742A PHY via the RMII interface and thence to a RJ45 connector at CN143. The external 25MHz crystal X4 is used to supply the clock.

Note

It is **highly** recommended that the configuration option CYGHWR_HAL_CORTEXM_STM32_SRAM_ALTERNATE is **ENABLED**. Enabling that feature configures the Ethernet driver RX memory buffers to the SRAM2 space in the D2 domain. This is required to avoid an undocumented STM32H723 revZ errata where the Ethernet MAC would occasionally (rare) corrupt memory if the AXI SRAM was used for the RX buffers. The downside of the option is that it will mean a smaller number of RX buffers being available than is possible with the larger (main) AXI SRAM space.

ADC Driver

The STM32 processor variant HAL provides an ADC driver. The STM32H7 Nucleo-144 platform HAL enables the support for all three devices and for configuration of the respective ADC device input channels.

Consult the generic ADC driver API documentation in the eCosPro Reference Manual for further details on ADC support in eCosPro, along with the configuration options in the STM32 ADC device driver.

Name

SWD support — Usage

Use of JTAG/SWD for debugging

JTAG/SWD can be used to single-step and debug loaded applications, or even applications resident in ROM.

Debugging of ROM applications is only possible if using hardware breakpoints. The Cortex-M7 core of the STM32H723ZG only supports eight such hardware breakpoints, so they may need to be used sparingly. If using a GDB front-end such as Eclipse, check it has not set unnecessary extra breakpoints such as at `main()`. Some JTAG/SWD devices give the option of whether to set hardware or software breakpoints by default. Be sure to configure your device appropriately.

When debugging via JTAG, you are likely to need to disable the default HAL idle thread action, otherwise there may be issues where the target fails to halt and the debugging session is unreliable. More details can be found in the Cortex-M architectural HAL. This should *not* be necessary when using a SWD-based hardware debugger such as the on-board ST-LINK/V3E interface.

Using the ST-LINK/V3E USB connection allows for a single cable to provide power, hardware debug support and diagnostic output. The latter is provided via a virtual UART which instantiates an ACM compatible serial channel in the host which is connected to USART3 on the STM32H723ZG. A separate application may be run alongside the debugger to capture the output from this UART, such as **minicom** under Linux or **PuTTY** under Windows.

OpenOCD notes

OpenOCD version `0.11.0` and above is required to support the STM32H723ZG MCU. A prebuilt host **openocd** executable which also supports the on-board ST-LINK/V3E interface available via the USB CN1 connection has been provided with the eCosPro Host Tools version `5.0.0` and above.

Should you wish to rebuild **openocd** yourself, you must specify the `--enable-stlink` option when configuring the OpenOCD build to provide for ST-LINK support.

An example OpenOCD configuration file `openocd.nucleo144.cfg` is provided within the eCos platform HAL package in the source repository. This will be in the directory `packages/hal/cortexm/stm32/nucleo144/current/misc` relative to the root of your eCosPro installation, but for convenience it is copied into the `install/etc` subdirectory as `openocd.cfg` during the **make etc** build process of the eCosPro library.

This configuration file can be used with OpenOCD on the host as follows:

```
$ openocd -f openocd.nucleo144_disco.cfg
Open On-Chip Debugger 0.11.0 (eCosCentric)
Licensed under GNU GPL v2
For bug reports, read
 http://openocd.org/doc/doxygen/bugs.html
Info : The selected transport took over low-level target control. The results might differ compared to plain JTAG/SWD
Info : Listening on port 6666 for tcl connections
Info : Listening on port 4444 for telnet connections
Info : clock speed 1800 kHz
Info : STLINK V3J8M3 (API v3) VID:PID 0483:374E
Info : Target voltage: 3.287824
Info : stm32h7x.cpu0: hardware has 8 breakpoints, 4 watchpoints
Info : starting gdb server for stm32h7x.cpu0 on 3333
Info : Listening on port 3333 for gdb connections
```

By default **openocd** provides a telnet console on port `4444`, and this can be used to interact with the target system. This console interface can be used to perform debugging, program the flash, etc.

Normally **arm-eabi-gdb** is used to connect to the default GDB server port `3333` for debugging. For example:

```
(gdb) target extended-remote localhost:3333
Remote debugging using localhost:3333
0x00000000 in ?? ()
```

```
(gdb)
```

OpenOCD should report the following on its terminal when a GDB connection is made:

```
Info : accepting 'gdb' connection on tcp/3333

Initialising CPU...
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x08000d40 msp: 0x2404fc00
force hard breakpoints
Note: Breakpoints limited to 8 hardware breakpoints
Invalidate ICACHE
Invalidate DCACHE
Disable Caches

Info : Device: STM32H72x/73x
Info : flash size probed value 1024
Info : STM32H7 flash has a single bank
Info : Bank (0) size is 1024 kb, base address is 0x08000000
```

The application can then be loaded and executed under GDB as normal. If you are using Eclipse then, if required, you can define a “preload” gdb macro to emit any necessary commands to OpenOCD. See the “Hardware Assisted Debugging” section of the “Eclipse/CDT for eCos application development” document's “Debugging eCos applications” chapter.

Note

Incompatibilities between OpenOCD and the STM32H7 cache support mean that at present only hardware breakpoints should be used for debugging. The OpenOCD configuration file provided within the eCosPro Developer's kit enforces this.

Configuration of JTAG/SWD applications

JTAG/SWD applications can be loaded directly into SRAM without requiring a ROM monitor. Loading can be done directly through the JTAG/SWD device, or through GDB where supported by the JTAG/SWD device.

In order to configure the application to support this mode, it is recommended to use the `JTAG` startup type which will implicitly cause two important settings to change. Firstly, `CYGSEM_HAL_USE_ROM_MONITOR` must be disabled. Secondly the `CYGDBG_HAL_DIAG_TO_DEBUG_CHAN` option should be enabled in order to prevent HAL diagnostic output being encoded into GDB ($O) packets. These configuration changes could be made by hand, but use of the `JTAG` startup type will just work.

With these changes, any diagnostic output will appear out of the configured diagnostic channel. An eCosCentric extension allows diagnostic output to appear in GDB. For this feature to work, you must enable the configuration option `CYGSEM_HAL_DIAG_TO_GDBFILEIO_CHAN` in the common HAL package. If you are using the graphical configuration tool then you should then accept any suggested solutions to the subsequent configuration conflicts. Older eCos releases also required the gdb "set hwdebug on" command to be used to enable GDB or Eclipse console output, but this is no longer required with the latest tools.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the STM32H7 Nucleo-144 board hardware, and should be read in conjunction with that specification. The NUCLEO-144 Board HAL package complements the Cortex-M architectural HAL and the STM32 variant HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset the HAL will initialize or reinitialize many of the on-chip peripherals. For both `ROM` and `JTAG` startup types the HAL will perform all initialization, programming the various internal registers including the PLLs, peripheral clocks and GPIO pins. The details of the early hardware startup may be found in the `src/nucleo144_misc.c` in both `hal_system_init` and `hal_platform_init`.

Memory Map

The platform HAL package provides the memory layout information needed to generate the linker script. For all the STARTUP variations the top `CYGNUM_HAL_COMMON_INTERRUPTS_STACK_SIZE` bytes of the on-chip SRAM are reserved for the interrupt stack. The remainder of the internal SRAM is available for use by applications. The key memory locations are as follows:

Internal RAM
: This is located at address `0x24000000` of the memory space, and is 320KiB in size. The eCos VSR table occupies the bottom 716-bytes of memory, with the virtual vector table starting at `0x240002CC` and extending to `0x240003CC`.

  This memory region comprises two contiguous memory blocks, the 128kiB AXI-SRAM plus 192kiB of shared SRAM in the default configuration where it is assigned to the AXI-SRAM.

SRAM1
: This is located at address `0x30000000` of the memory space, and is 16KiB in size. This is used to contain the Ethernet transmit and receive descriptor rings. It is mapped by an MPU region that disables caching for accesses.

Internal FLASH
: This is located at address `0x08000000` of the memory space and will be mapped to `0x00000000` at reset. This region is 1024KiB in size. `ROM` applications are by default configured to run from this memory. The 256kiB from offset 0xC0000 is used for flash testing. The test space is defined in `__NUCLEO144_FLASHTEST_ONCHIP` in `plf_io.h`.

QSPI NOR Flash
: The QSPI NOR flash is accessible through the flash API. The 256kiB from offset 0x100000 is used for flash testing. The test space is defined in `__NUCLEO144_FLASHTEST_OCTOSPI` in `plf_io.h`.

On-Chip Peripherals
: These are accessible at locations `0x40000000` and `0xE0000000` upwards. Descriptions of the contents can be found in the STM32 User Manual.

Linker Scripts

The platform linker scripts define the following symbols:

hal_vsr_table
: This defines the location of the VSR table. This is set to `0x20000000` for all startup types, and space for 114 entries is reserved.

hal_virtual_vector_table
: This defines the location of the virtual vector table used to communicate between an ROM monitor and an eCos application. This is allocated right after the VSR table, at `0x240002cc`.

hal_interrupt_stack
: This defines the location of the interrupt stack. This is allocated to the top of internal SRAM, 0x24050000.

hal_startup_stack
: This defines the location of the startup stack. For all startup types it is initially allocated at the half-way point of the interrupt stack.

Flash wait states

The STM32H7 Nucleo-144 platform HAL provides a configuration option to set the number of Flash read wait states to use: CYGNUM_HAL_CORTEXM_STM32_FLASH_WAIT_STATES. It is important to verify and if necessary update this value if changing the CPU clock (HCLK) frequency or CPU voltage. Consult the relevant STM32 datasheets and programming manuals for the STM32H723 parts for appropriate values for different clock speeds or voltages. The default of 5 reflects the default HCLK frequency.

Real-time characterization

The **tm_basic** kernel test gives statistics gathered about the real-time characterization and performance of the kernel. The sample output is shown here for information. The test was built for ROM startup with optimization flag -O2, since it provides the best performance as both code and data could remain on-chip.

Example 303.1. nucleo144_stm32h723 Real-time characterization

```
             Startup, main thrd : stack used   356 size  2048
             Startup : Idlethread stack used    76 size  1280

eCos Kernel Timings
Notes: all times are in microseconds (.000001) unless otherwise stated

Reading the hardware clock takes 0 'ticks' overhead
... this value will be factored out of all other measurements
Clock interrupt took    1.00 microseconds (1 raw clock ticks)

Testing parameters:
   Clock samples:            32
   Threads:                  15
   Thread switches:         128
   Mutexes:                  32
   Mailboxes:                32
   Semaphores:               32
   Scheduler operations:    128
   Counters:                 32
   Flags:                    32
   Alarms:                   32
   Stack Size:             1088


                                 Confidence
     Ave     Min     Max     Var  Ave  Min  Function
  ======  ======  ======  ====== ========== ========
INFO:<Ctrl-C disabled until test completion>
    0.80    0.00    2.00    0.43   66%  26% Create thread
    0.27    0.00    1.00    0.39   73%  73% Yield thread [all suspended]
    0.27    0.00    1.00    0.39   73%  73% Suspend [suspended] thread
    0.27    0.00    1.00    0.39   73%  73% Resume thread
    0.27    0.00    1.00    0.39   73%  73% Set priority
    0.07    0.00    1.00    0.12   93%  93% Get priority
    0.53    0.00    1.00    0.50   53%  46% Kill [suspended] thread
    0.27    0.00    1.00    0.39   73%  73% Yield [no other] thread
    0.27    0.00    1.00    0.39   73%  73% Resume [suspended low prio] thread
    0.27    0.00    1.00    0.39   73%  73% Resume [runnable low prio] thread
    0.33    0.00    1.00    0.44   66%  66% Suspend [runnable] thread
    0.20    0.00    1.00    0.32   80%  80% Yield [only low prio] thread
    0.13    0.00    1.00    0.23   86%  86% Suspend [runnable->not runnable]
    0.47    0.00    1.00    0.50   53%  53% Kill [runnable] thread
    0.47    0.00    1.00    0.50   53%  53% Destroy [dead] thread
    1.07    1.00    2.00    0.12   93%  93% Destroy [runnable] thread
    1.20    1.00    2.00    0.32   80%  80% Resume [high priority] thread
    0.40    0.00    1.00    0.48   60%  60% Thread switch

    0.07    0.00    1.00    0.13   92%  92% Scheduler lock
    0.20    0.00    1.00    0.31   80%  80% Scheduler unlock [0 threads]
    0.19    0.00    1.00    0.30   81%  81% Scheduler unlock [1 suspended]
    0.17    0.00    1.00    0.28   82%  82% Scheduler unlock [many suspended]
    0.05    0.00    1.00    0.10   94%  94% Scheduler unlock [many low prio]

    0.09    0.00    1.00    0.17   90%  90% Init mutex
    0.25    0.00    1.00    0.38   75%  75% Lock [unlocked] mutex
    0.25    0.00    1.00    0.38   75%  75% Unlock [locked] mutex
    0.25    0.00    1.00    0.38   75%  75% Trylock [unlocked] mutex
    0.22    0.00    1.00    0.34   78%  78% Trylock [locked] mutex
    0.09    0.00    1.00    0.17   90%  90% Destroy mutex
    2.00    2.00    2.00    0.00  100% 100% Unlock/Lock mutex

    0.13    0.00    1.00    0.22   87%  87% Create mbox
    0.09    0.00    1.00    0.17   90%  90% Peek [empty] mbox
    0.25    0.00    1.00    0.38   75%  75% Put [first] mbox
    0.09    0.00    1.00    0.17   90%  90% Peek [1 msg] mbox
    0.25    0.00    1.00    0.38   75%  75% Put [second] mbox
    0.06    0.00    1.00    0.12   93%  93% Peek [2 msgs] mbox
    0.28    0.00    1.00    0.40   71%  71% Get [first] mbox
    0.19    0.00    1.00    0.30   81%  81% Get [second] mbox
    0.25    0.00    1.00    0.38   75%  75% Tryput [first] mbox
    0.19    0.00    1.00    0.30   81%  81% Peek item [non-empty] mbox
    0.22    0.00    1.00    0.34   78%  78% Tryget [non-empty] mbox
    0.22    0.00    1.00    0.34   78%  78% Peek item [empty] mbox
    0.22    0.00    1.00    0.34   78%  78% Tryget [empty] mbox
    0.06    0.00    1.00    0.12   93%  93% Waiting to get mbox
    0.06    0.00    1.00    0.12   93%  93% Waiting to put mbox
    0.13    0.00    1.00    0.22   87%  87% Delete mbox
    1.16    1.00    2.00    0.26   84%  84% Put/Get mbox

    0.03    0.00    1.00    0.06   96%  96% Init semaphore
    0.16    0.00    1.00    0.26   84%  84% Post [0] semaphore
    0.25    0.00    1.00    0.38   75%  75% Wait [1] semaphore
    0.19    0.00    1.00    0.30   81%  81% Trywait [0] semaphore
    0.25    0.00    1.00    0.38   75%  75% Trywait [1] semaphore
    0.03    0.00    1.00    0.06   96%  96% Peek semaphore
    0.03    0.00    1.00    0.06   96%  96% Destroy semaphore
    1.00    1.00    1.00    0.00  100% 100% Post/Wait semaphore

    0.06    0.00    1.00    0.12   93%  93% Create counter
    0.09    0.00    1.00    0.17   90%  90% Get counter value
    0.09    0.00    1.00    0.17   90%  90% Set counter value
    0.22    0.00    1.00    0.34   78%  78% Tick counter
    0.06    0.00    1.00    0.12   93%  93% Delete counter

    0.09    0.00    1.00    0.17   90%  90% Init flag
    0.22    0.00    1.00    0.34   78%  78% Destroy flag
    0.19    0.00    1.00    0.30   81%  81% Mask bits in flag
    0.22    0.00    1.00    0.34   78%  78% Set bits in flag [no waiters]
    0.25    0.00    1.00    0.38   75%  75% Wait for flag [AND]
    0.22    0.00    1.00    0.34   78%  78% Wait for flag [OR]
    0.31    0.00    1.00    0.43   68%  68% Wait for flag [AND/CLR]
    0.28    0.00    1.00    0.40   71%  71% Wait for flag [OR/CLR]
    0.06    0.00    1.00    0.12   93%  93% Peek on flag

    0.13    0.00    1.00    0.22   87%  87% Create alarm
    0.31    0.00    1.00    0.43   68%  68% Initialize alarm
    0.16    0.00    1.00    0.26   84%  84% Disable alarm
    0.34    0.00    1.00    0.45   65%  65% Enable alarm
    0.13    0.00    1.00    0.22   87%  87% Delete alarm
    0.28    0.00    1.00    0.40   71%  71% Tick counter [1 alarm]
    0.91    0.00    1.00    0.17   90%   9% Tick counter [many alarms]
    0.41    0.00    1.00    0.48   59%  59% Tick & fire counter [1 alarm]
    5.81    5.00    6.00    0.31   81%  18% Tick & fire counters [>1 together]
    1.06    1.00    2.00    0.12   93%  93% Tick & fire counters [>1 separately]
    1.00    1.00    1.00    0.00  100% 100% Alarm latency [0 threads]
    1.00    1.00    1.00    0.00  100% 100% Alarm latency [2 threads]
    1.00    1.00    1.00    0.00  100% 100% Alarm latency [many threads]
    2.00    2.00    2.00    0.00  100% 100% Alarm -> thread resume latency

    0.00    0.00    0.00    0.00            Clock/interrupt latency

    1.00    1.00    1.00    0.00            Clock DSR latency

     204     180     220                    Worker thread stack used (stack size 1088)
            All done, main thrd : stack used   696 size  2048
            All done : Idlethread stack used   172 size  1280

Timing complete - 29810 ms total

PASS:<Basic timing OK>
EXIT:<done>
```

Name

Test Programs — Details

Test Programs

The eCos NUCLEO-144 Board HAL contains a test program which allow various aspects of the board to be tested.

Manual Test

The simple manual test may be built by enabling `CYGPKG_HAL_CORTEXM_STM32_NUCLEO144_TESTS_MANUAL`.

The **manual** test will flash the LEDs and allow track use of the user (normally blue) button B1.

Chapter 304. STM32F4DISCOVERY Platform HAL

Name

`CYGPKG_HAL_CORTEXM_STM32_STM32F4DIS` — eCos Support for the STM32F4DISCOVERY Board

Description

The STMicroelectronics STM32F4DISCOVERY board has a STM32F407VG microcontroller which incorporates 1MiB of internal Flash ROM and 128KiB of internal SRAM. The microcontroller also has 64KiB of internal “core coupled” SRAM for CPU use.

The STM32F4DISCOVERY board has limited I/O interfaces, with most of the I/O signals being propogated via multi-pin connectors. The STM32F4DIS-BB daughterboard has a connector for Ethernet, a standard DB9 serial port, and a MicroSD interface. On both the main STMF4DISCOVERY motherboard and the STM32F4DIS-BB daughterboard it is possible to access signals for UARTs, I²C, and SPI, as well as various other devices.

For the STM32F4DISCOVERY board the expected eCos development model is that programs may be downloaded and debugged via a SWD debugger, normally attached via the on-board USB ST-LINK/V2 connector CN1. This differs from the traditional eCos development model, where RedBoot or a GDB stub image is programmed into internal FLASH and the CPU boots directly into that. However, due to the small amount of RAM available, the RAM requirements of a ROM monitor would further limit the size of applications that could be loaded via such a monitor. Nevertheless it is still possible to program a GDB stub image into Flash and download and debug eCos applications with the GDB debugger via available UART pins.

This documentation describes platform-specific elements of the STM32F4DISCOVERY board support within eCos. The STM32 variant HAL documentation covers various topics including HAL support common to STM32 variants, and on-chip device support. This document complements the STM32 documentation.

Supported Hardware

The STM32F407VG has three on-chip memory regions. A RAM region of 128KiB is present at 0x20000000, a 64KiB core coupled RAM region present at 0x1000000, and a 1MiB FLASH region is present at 0x08000000 which is aliased to 0x00000000 during normal execution.

The STM32 variant HAL includes support for the six on-chip serial devices which are documented in the variant HAL, however it is assumed that only USART6 is available when the STM32F4DIS-BB daughterboard is present. There is no connection for hardware flow control (RTS/CTS) lines.

The STM32 variant HAL also includes support for the I²C bus. However the STM32F4DISCOVRY mainboard I²C devices are not supported due to I/O pin clashes with the STM32F4DIS-BB daughterboard. Instead, testing has been performed using an external Total Phase Aardvark activity board with on-board I²C EEPROM.

Device drivers are also provided for the STM32 on-chip SPI and MMCSD interfaces, plus the Ethernet MAC. Additionally, support is provided for the on-chip watchdog, RTC (wallclock), and a Flash driver exists to permit management of the STM32's on-chip Flash.

Tools

The board port is intended to work with GNU tools configured for an **arm-eabi** target. The original port was done using **arm-eabi-gcc** version 4.4.5, **arm-eabi-gdb** version 7.2, and **binutils** version 2.20.1.

Name

Setup — Preparing the STM32F4DISCOVERY Board for eCos Development

Overview

Given the limited available RAM memory, it is expected that the most common development method is to use JTAG/SWD for development, either by loading smaller applications into RAM, or by programming larger applications directly into on-chip Flash. In the first case, eCos applications should be configured for the `JTAG` startup type, and in the second case for the `ROM` startup type.

Nevertheless, it is still possible to program a GDB stub ROM image into on-chip Flash and download and debug via a serial UART, if pins for the UART are available. In that case, eCos applications are configured for RAM startup and then downloaded and run on the board via the debugger **arm-eabi-gdb**, or via the Eclipse IDE. For serial communications, all versions run with 8 bits, no parity, and 1 stop bit at 115200 baud. This rate can be changed in the eCos configuration used for building the GDB stub ROM image.

HAL startup types

The following startup types may be selected for applications:

Configuration	Description
ROM	Programs running from internal FLASH
JTAG	Programs running from RAM, loaded via JTAG or SWD debug hardware
RAM	Programs loading via a GDB stub ROM and serial connection into RAM

Further details are available later in this manual.

Preparing ST-LINK/V2 interface

The support for using the on-chip ITM stimulus ports for diagnostic and instrumentation output requires that the ST-LINK/V2 firmware is at least version `V2.J17.S0`. The firmware for the ST-LINK/V2 interface can be checked, and updated if needed, using a tool available from STMicroelectronics. The firmware version is also reported when the **openocd** command is executed (using a suitable configuration file). For example, the following OpenOCD output reports `JTAG v17`:

```
Info : STLINK v2 JTAG v17 API v2 SWIM v0 VID 0x0483 PID 0x3748
```

Unfortunately the official firmware updater is only available for the Windows platform at the moment. From a Windows machine:

1. Ensure that the Windows PC and STM32F4DISCOVERY board are disconnected.

2. Download the STM32 ST-LINK Utility from ST's website.

   The page titled “STSW-LINK004 STM32 ST-LINK utility” provides a free download of the utility http://www.st.com/web/en/catalog/tools/PF258168

3. Install the ST-LINK Utility software on your Windows PC.

   Simply unzip the downloaded file `stsw-link004.zip` and run the `STM32 ST-LINK Utility_v3.0.0.exe` that was contained within it. Follow the on-screen instructions. This will install both the utility application and the ST-LINK/V2 USB driver.

4. Connect the STM32F4DISCOVERY board to the PC.

   Connect the STM32F4DISCOVERY board to the PC using the ST supplied mini-B USB cable. Windows should correctly identify the USB device and load the device driver. Windows Device Manager should now show “STMicroelectronics STLink dongle” under “Universal Serial Bus controllers”.

5. Run the ST-LINK Utility and ensure the ST-LINK firmware is up to date.

   From the Windows “Start” menu run the “STM32 ST-LINK Utility”. Click on the `connect` icon, or select `Target->Connect` from the menu. This should confirm that a successful connection can be made to the board. To update the on-board ST-LINK/V2 firmware select `ST-LINK->Firmware Update` from the menu. In the ST-LINK dialog box that then appears click on the `Device Connect` button. This will likely result in a message “ST-Link is not in DFU mode. Please restart it.”. In this case simply disconnect the board from the PC and then reconnect it after a couple of seconds, then click the `OK` button on the message. In the original ST-Link dialog box click `Device Connect` again. The dialog box should now report the current on-board and available firmware versions, and enable you to upgrade the board by pressing the `Yes >>>>` button. We have tested the system with firmware version `V2.J17.S0` and would recommend this version as a minimum. Clicking `Yes >>>>` will cause a progress bar in the dialog to be animated and should eventually result in a “Update Successful” message. You can then close the various dialogs and exit the ST-LINK Utility. Disconnect and reconnect the board and it is now ready for use with OpenOCD.

Programming ROM images

To program ROM startup applications into Flash, including the GDB stub ROM, a JTAG/SWD debugger that understands the STM32 flash may be used, such as a Ronetix PEEDI.

However, since the STM32F4DISCOVERY board has a built-in ST-LINK/V2 SWD interface, if the `CN3` jumpers are closed then the micro USB host connection and suitable host software (e.g. The OpenOCD package **openocd** tool) can be used to program the flash. Normally a default **openocd** session provides a comand-line via port `4444`. Consult the OpenOCD documentation for more details if a non-default **openocd** configuration is being used.

With a **telnet** connection established to the **openocd** any binary data can easily be written to the on-chip flash. e.g.

```
$ telnet localhost 4444
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Open On-Chip Debugger
> flash write_image test.bin 0x08000000
wrote 32518 bytes from file test.bin in 1.073942s (29.569 KiB/s)
```

Programming ROM images with a Ronetix PEEDI

This section describes how to program ROM images using a Ronetix PEEDI debugger.

The PEEDI must be configured to allow communication with your local network, and configured with the parameters for interfacing with the target board. It must then be used to download and program the ROM image into the internal flash. The following steps give a typical outline for doing this. Consult the PEEDI documentation for alternative approaches, such as using FTP or HTTP instead of TFTP.

Preparing the Ronetix PEEDI JTAG debugger

1. Prepare a PC to act as a host and start a TFTP server on it.

2. Connect the PEEDI JTAG debugger via both serial and ethernet to the host PC and power it on. Use the serial cable supplied with the PEEDI (straight through, not null modem).

3. Verify the PEEDI is using up-to-date firmware, of version 11.10.1 or later. Older PEEDI firmware does not support the STM32 F4 family correctly, particularly if wishing to use the PEEDI's own '**flash**' commands to modify the on-chip Flash. If the firmware is not recent enough, follow the PEEDI User Manual's instructions which describe how to update the PEEDI firmware.

4. Locate the PEEDI configuration file `peedi.stm32f4dis.cfg` within the eCos platform HAL package in the source repository. This will be in the directory `packages/hal/cortexm/stm32/stm32f4dis/current/misc` relative to the root of your eCos installation.

5. Place the PEEDI configuration file in a location on the PC accessible to the TFTP server. Later you will configure the PEEDI to load this file via TFTP as its configuration file.

6. Open `peedi.stm32f4dis.cfg` in an editor such as emacs or notepad and insert your own license information in the `[LICENSE]` section.

7. Install and configure the PEEDI in line with the PEEDI Quick Start Guide or User's Manual, especially configuring PEEDI's RedBoot with the network information. Configure it to use the `peedi.stm32f4dis.cfg` target configuration file on the TFTP server at the appropriate point of the **config** process, for example with a path such as: `tftp://192.168.7.9/peedi.stm32f4dis.cfg`

8. Reset the PEEDI.

9. Connect to the PEEDI's CLI interface via TCP/IP on the standard telnet port 23. The **telnet** application is suitable for this. You should see output similar to the following:

```
$ telnet 192.168.7.225
Trying 192.168.7.225...
Connected to 192.168.7.225.
Escape character is '^]'.

PEEDI - Powerful Embedded Ethernet Debug Interface
Copyright (c) 2005-2011 www.ronetix.at - All rights reserved
Hw:1.2, L:JTAG v1.6 Fw:12.6.1, SN: PD-XXXX-XXXX-XXXX
------------------------------------------------------------

stm32f4discovery>
```

Preparing the STM32F4DISCOVERY board for programming with PEEDI

Follow the steps in this section in order to allow communication between the board and the host PC, and between the board and the JTAG device.

If programming a GDB stub ROM or an application which uses serial output, you should first:

1. Connect an adaptor from the serial pins on the board to an RS232 DB9 serial connector or cable, then connect from there to a serial port on the host computer with a null modem DB9 RS232 serial cable.

2. Start a suitable terminal emulator on the host computer such as **minicom** on Linux or PuTTY on Windows. Set the communication parameters to 115200 baud, 8 data bits, no parity bit and 1 stop bit with no flow control.

For all applications, you must:

1. Connect the board to the PEEDI. Since the STM32F4DISCOVERY does not have a standard ARM Cortex-M connector a custom connection from the Target port on the PEEDI to the STM32F4DISCOVERY needs to be wired.

Note

This is a direct connection from the PEEDI *without* any PEEDI adapter installed. Also the STM32F4DISCOVERY CN3 jumpers should be removed to disconnect the ST-Link from driving the target STM32F4 SWD interface.

The following table maps the PEEDI TARGET connector pin numbers to the STM32F4DISCOVERY P1 and P2 connector pins, and assumes a standard PEEDI `PLATFORM = CortexM3_SWD` configuration.

	PEEDI TARGET	**STM32F4DISCOVERY**
Vdd	1	P1#3 (VDD)
SWCLK	3	P2#39 (PA14)

	PEEDI TARGET	STM32F4DISCOVERY
SWDIO	7	P2#42 (PA13)
SWO	11	P2#28 (PB3)
nRST	20	P1#6 (NRST)
GND	16	P1#49 (GND)

2. Power up the STMF4DISCOVERY board.

3. Connect to the PEEDI's telnet CLI on port 23 as before.

4. Confirm correct connection with the PEEDI with the **reset reset** command as follows:

   ```
   stm32f4discovery> reset reset
   ++ info: RESET and TRST asserted
   ++ info: TRST released
   ++ info: TAP : IDCODE = 0x2BA01477, Cortex M3 SWD
   ++ info: RESET released
   ++ info: core connected
   ++ info: core 0: initialized
   stm32f4discovery>
   ```

Installation into Flash

The following describes the procedure for installing a ROM application into on-chip Flash, using the GDB stub ROM image as an example of such an application.

1. Use **arm-eabi-objcopy** to convert the linked application, in ELF format, into binary format. For example:

   ```
   $ arm-eabi-objcopy -O binary programname programname.bin
   ```

   In the case of the GDB stub ROM image, a prebuilt image is available with the name `gdb_module.bin` within the `loaders` subdirectory of the base of the eCos installation.

2. Copy the binary file (.bin file) into a location on the host computer accessible to its TFTP server.

3. Connect to the PEEDI's telnet interface, and program the image into Flash with the following command, replacing *`TFTP_SERVER`* with the address of the TFTP server and *`/BINPATH`* with the location of the .bin file relative to the TFTP server root directory. For example for the GDB stub ROM:

   ```
   stm32f4discovery> flash program tftp://TFTP_SERVER/BINPATH/gdb_module.bin bin 0x08000000 erase
   ++ info: Programming image file: tftp://TFTP_SERVER/BINPATH/gdb_module.bin
   ++ info: Programming using agent, buffer = 4096 bytes
   ++ info: At absolute address:    0x08000000
   erasing     at 0x08000000 (sector #0)
   programming at 0x08000000
   programming at 0x08001000
   programming at 0x08002000
   programming at 0x08003000
   erasing     at 0x08004000 (sector #1)
   programming at 0x08004000

   ++ info: successfully programmed 20.00 KB in 1.11 sec

   stm32f4discovery>
   ```

The installation into Flash is now complete. For applications which print output on startup to the USART3 RS232 serial port, such as the GDB stub ROM application, this can easily by tested by powering off the board, disconnecting the JTAG, and then powering on the board again. In the case of the GDB stub ROM image, output similar to the following should be visible (although specific numbers may differ):

```
$T050f:72250008;0d:f0ff0120;#8a
```

Rebuilding the GDB stub

Should it prove necessary to rebuild the GDB stub ROM binary, this is done most conveniently at the command line. The steps needed are:

```
$ mkdir gdbstub_stm32f4dis
$ cd gdbstub_stm32f4dis
$ ecosconfig new stm32f4dis stubs
[ … ecosconfig output elided … ]
$ ecosconfig tree
$ make
```

At the end of the build, the `install/bin` subdirectory should contain the file `gdb_module.bin`. This may be programmed to the board using the above procedure.

Name

Configuration — Platform-specific Configuration Options

Overview

The STM32F4DISCOVERY board platform HAL package is loaded automatically when eCos is configured for the `stm32f4dis` or `stm32f4disbb` targets. The `stm32f4dis` target enables the necessary hardware support for the bare STM32F4DISCOVERY board, whereas the `stm32f4disbb` target is intended to support the STM32F4DISCOVERY board stacked with a “STM32F4DISCOVERY Base Board” (STM32F4DIS-BB). The essential difference between the two targets being that the Base Board target adds Ethernet support. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The STM32F4DISCOVERY board platform HAL package supports three separate startup types:

ROM

> This startup type can be used for finished applications which will be programmed into internal Flash ROM at location `0x08000000`. Data and BSS will be put into internal SRAM starting from `0x20000288`. The application will be self-contained with no dependencies on services provided by other software. The program expects to boot from reset with ROM mapped at location zero. It will then transfer control to the `0x08000000` region. eCos startup code will perform all necessary hardware initialization.

RAM

> This is the startup type which is used if relying on a GDB stub ROM image programmed into internal Flash to download and run applications into RAM via **arm-eabi-gdb** and a serial UART. RAM from `0x20000000` to `0x20001000` is reserved for the GDB stub, but then the RAM startup application may be loaded into memory from `0x20001000` and debugged using GDB. It is assumed that the hardware has already been initialized by the GDB stub ROM. By default the application will use the eCos virtual vectors mechanism to obtain services from the GDB stub ROM, including diagnostic output.

JTAG

> This is the startup type used to build applications that are loaded via a JTAG/SWD interface. The application will be self-contained with no dependencies on services provided by other software. The program expects to be loaded from `0x20000288` and entered at that address. Memory from `0x20000000` to `0x20000288` is set aside for vector tables. eCos startup code will perform all necessary hardware initialization.

Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building the GDB stub ROM (or RedBoot).

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then the serial port will be claimed for HAL diagnostic output.

UART Serial Driver

The STM32F4DISCOVERY board uses the STM32's internal UART serial support. The HAL diagnostic interface, used for both polled diagnostic output and GDB stub communication, is only expected to be available to be used on the UART6 port (counting the first UART as UART1). The bare STM32F4DISCOVERY board only exports the UART6 connection via connector P2, but the STM32F4DIS-BB daughterboard provides UART6 on the COM1 (DB9) connector.

Note

In reality when using a hardware SWD debugger (e.g. the on-board ST-LINK/V2 interface) it is preferable to use the on-chip ITM support for HAL diagnostic output. The ITM stimulus port HAL diagnostic interface is significantly faster than using a UART, and provides for a simpler, single cable, debug and diagnostic connection to the target board. The use of ITM for HAL diagnostics is configurable in the architecture HAL.

As well as the polled HAL diagnostic interface, there is also a CYGPKG_IO_SERIAL_CORTEXM_STM32 package which contains all the code necessary to support interrupt-driven operation with greater functionality. All six UARTs can be supported by this driver. Note that it is not recommended to use this driver with a port at the same time as using that port for HAL diagnostic I/O.

This driver is not active until the CYGPKG_IO_SERIAL_DEVICES configuration option (within the generic serial driver support package CYGPKG_IO_SERIAL) is enabled in the configuration. By default this will only enable support in the driver for the USART6 port (the same as the HAL diagnostic interface), but the default configuration can be modified to enable support for other serial ports. Note that in this package, serial port numbering starts at 0, rather than 1. So, for example, to enable the serial driver for ports USART1 and USART2, enable the configuration options "ST STM32 serial port 0 driver" (CYGPKG_IO_SERIAL_CORTEXM_STM32_SERIAL0) and "ST STM32 serial port 1 driver" (CYGPKG_IO_SERIAL_CORTEXM_STM32_SERIAL1).

Ethernet Driver

The STM32F4DIS-BB daughterboard is fitted with an Ethernet port connected via a SMSC LAN8720 PHY to the STM32's on-chip Ethernet MAC. This is supported in eCosPro with a driver for the lwIP networking stack, contained in the package CYGPKG_DEVS_ETH_CORTEXM_STM32. At the present time it only supports the lwIP networking stack, and cannot be used for either the BSD networking stack or RedBoot.

When using a default eCos configuration the driver will be inactive (not built and greyed out in the eCos Configuration Tool) since that configuration does not include the networking packages. The driver is enabled when the platform HAL option "STM32 Ethernet Support" (CYGPKG_HAL_CORTEXM_STM32_STM32F4DIS_ETH0) is enabled. This option in turn is only active if the "Common Ethernet support" (CYGPKG_IO_ETH_DRIVERS) package is included in your configuration. As the STM32 ethernet driver is an lwIP-only driver, it is most appropriate to choose the lwip_eth template as a starting point when choosing an eCos configuration, which will cause the necessary packages to be automatically included.

SPI Driver

An SPI bus driver is available for the STM32 in the package "ST STM32 SPI driver" (CYGPKG_DEVS_SPI_CORTEXM_STM32).

Two SPI device entries have been created for use, and the SPI devices structures are defined in the file stm32f4dis_spi.c.

The first device is configured for accessing the MEMS (LIS302DL motion Sensor) on SPI bus 1, using pin PE3 as the chip select.

The second SPI device is for use with the Aardvark SPI test board, which has an SPI EEPROM available. This is only intended for testing. If used, it is present on SPI bus 2 and will use pin PB14 as the SPI chip select pin.

To disable support for both the above SPI devices, the platform HAL contains an option "SPI devices" (CYGPKG_HAL_CORTEXM_STM32_STM32F4DIS_SPI) which can be disabled. No other SPI devices are instantiated.

Consult the generic SPI driver API documentation in the eCosPro Reference Manual for further details on SPI support in eCosPro, along with the configuration options in the STM32 SPI device driver.

I²C Driver

The STM32 variant HAL provides the main I²C hardware driver itself, configured at `CYGPKG_HAL_STM32_I2C`. However, the platform I²C support can also be configured separately at `CYGPKG_HAL_CORTEXM_STM32_STM32F4DIS_I2C`. This option is present to allow use of the external Aardvark test board which has an I²C EEPROM fitted that is used for testing purposes. The option also ensures the STM32's I²C bus 1 is enabled as required.

Flash Driver

The STM32's on-chip Flash may be programmed and managed using the Flash driver located in the "STM32 Flash memory support" (`CYGPKG_DEVS_FLASH_STM32`) package. This driver is enabled automatically if the generic "Flash device drivers" (`CYGPKG_IO_FLASH`) package is included in the eCos configuration.

The driver will configure itself automatically for the size and parameters of the specific STM32 variant present on the STM32F4DISCOVERY board. However, if necessary the driver contains a configuration option "Flash size override (kb)" (`CYGNUM_DEVS_FLASH_STM32_FLASH_SIZE_OVERRIDE`) which allows the detected Flash size to be manually overriden, but this should not normally need to be changed.

A number of other aspects of Flash driver behaviour can be configured within that driver, such as program/erase parallelism and programburst size. Consult the driver for more details.

Name

JTAG/SWD support — Usage

Use of JTAG/SWD for debugging

JTAG/SWD can be used to single-step and debug loaded applications, or even applications resident in ROM, including the GDB stub ROM.

Debugging of ROM applications is only possible if using hardware breakpoints. The Cortex-M4 core of the STM32F407VG only supports six such hardware breakpoints, so they may need to be used sparingly. If using a GDB front-end such as Eclipse, check it has not set unnecessary extra breakpoints such as at `main()`. Some JTAG/SWD devices give the option of whether to set hardware or software breakpoints by default. Be sure to configure your device appropriately.

When debugging via JTAG, you are likely to need to disable the default HAL idle thread action, otherwise there may be issues where the target fails to halt and the debugging session is unreliable. More details can be found in the Cortex-M architectural HAL. This should *not* be necessary when using a SWD-based hardware debugger such as the on-board ST-LINK/V2 interface.

The default eCos configuration does not enable the use of ITM stimulus ports for the output of HAL diagnostics or Kernel instrumentation. The architecture HAL package `CYGPKG_HAL_CORTEXM` provides options to enable such use.

For HAL diagnostic (e.g. `diag_printf()`) output the architecture CDL option `CYGHWR_HAL_CORTEXM_DIAGNOSTICS_INTERFACE` should be updated to select `ITM` as the output destination. Once the `ITM` option has been configured the option `CYGHWR_HAL_CORTEXM_DIAGNOSTICS_ITM_PORT` allows the actual stimulus port used for the diagnostics to be selected.

When the Kernel instrumentation option `CYGPKG_KERNEL_INSTRUMENT` is enabled then the `CYGHWR_HAL_CORTEXM_ITM_INSTRUMENTATION` option can be enabled to direct instrumentation record output via an ITM stimulus port, rather than into a local memory buffer. The stimulus port used can be configured via the `CYGHWR_HAL_CORTEXM_ITM_INSTRUMENTATION_PORT_BASE` option.

When using the STM32F4DISCOVERY board via the ST-LINK/V2 interface then it is recommended that the ITM stimulus port approach is used to provide access to diagnostics and instrumentation over the single USB host connection. This avoids the need for a seperate, and slower, UART connection for provision of the HAL diagnostic output, plus continuous streaming instrumentation can be output via ITM where only very small local RAM buffers would normally be available for the default “memory buffer” instrumentation support. This allows a single cable ST-LINK/V2 connection to provide board power, hardware debug support, diagnostic output and instrumentation capture features.

Normally a notable disadvantage with JTAG/SWD debugging is that it does not allow thread-aware debugging, such as the ability to inspect different eCos threads or their stack backtraces, set thread-specific breakpoints, and so on. Fortunately the Ronetix PEEDI JTAG unit does support thread-aware debugging of eCos applications, however extra configuration steps are required. Consult the PEEDI documentation for more details as usage is beyond the scope of this document.

OpenOCD notes

The OpenOCD debugger can be configured to support the on-board ST-LINK/V2 interface available via the USB CN1 connection, with the CN3 links closed to directly connect to the target STM32F407 CPU. When configuring the **openocd** tool build, the **configure** script can be given the option `--enable-stlink` to provide for ST-LINK support.

An example OpenOCD configuration file `openocd.stm32f4dis.cfg` is provided within the eCos platform HAL package in the source repository. This will be in the directory `packages/hal/cortexm/stm32/stm32f4dis/current/misc` relative to the root of your eCos installation.

This configuration file can be used with OpenOCD on the host as follows:

```
$ openocd -f openocd.stm32f4dis.cfg
```

```
Open On-Chip Debugger 0.9.0 (2015-09-18-16:19)
Licensed under GNU GPL v2
For bug reports, read
    http://openocd.org/doc/doxygen/bugs.html
Info : The selected transport took over low-level target control. The results might differ compared to plain JTAG/SWD
adapter speed: 2000 kHz
adapter_nsrst_delay: 100
none separate
Info : Unable to match requested speed 2000 kHz, using 1800 kHz
Info : Unable to match requested speed 2000 kHz, using 1800 kHz
Info : clock speed 1800 kHz
Info : STLINK v2 JTAG v17 API v2 SWIM v0 VID 0x0483 PID 0x3748
Info : using stlink api v2
Info : Target voltage: 2.886506
Info : stm32f4x.cpu: hardware has 6 breakpoints, 4 watchpoints
```

By default **openocd** provides a console on port 4444, and this can be used to interact with the target system. This console interface can be used to perform debugging, program the flash, etc.

Normally **arm-eabi-gdb** is used to connect to the default GDB server port 3333 for debugging. For example:

```
(gdb) target remote localhost:3333
Remote debugging using localhost:3333
0x00000000 in ?? ()
(gdb) monitor reset halt
target state: halted
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x0800422c msp: 0x20000c80
(gdb)
```

The application can then be loaded and executed under GDB as normal. If you are using Eclipse then you will need to define a “preload” gdb macro to emit the **monitor reset halt** command to OpenOCD. See the “Hardware Assisted Debugging” section of the “Eclipse/CDT for eCos application development” document's “Debugging eCos applications” chapter.

If the HAL diagnostics are configured to use ITM, and stimulus port 31 is configured as the HAL diagnostic destination, then the configuration example above will direct OpenOCD to direct ITM output (and also DWT and ETM) to a file named tpiu.out in the current directory of the shell which was used to run the **openocd** command. A more specific filename can be used by adjusting the OpenOCD configuration file.

To extract the ITM output, the Cortex-M architecture HAL package provides a helper program **parseitm** in the directory packages/hal/cortexm/arch/current/host relative to the root of your eCos installation. It can be compiled simply with:

```
$ gcc -o parseitm parseitm.c
```

You simply run it with the desired ITM stimulus port and name of the file containing the ITM output, for example:

```
$ parseitm -p 31 -f itm.out
```

It will echo all ITM stimulus for that port, continuing to read from the file until interrupted with Ctrl-C. Note that limited buffer space in debug hardware such as the ST-LINK can result in occasionally missed ITM data. eCosPro provides a workaround of throttling data within the CYGHWR_HAL_CORTEXM_ITM_DIAGNOSTICS_THROTTLE CDL configuration component in order to reduce or avoid lost ITM data. For further details, see the note in OpenOCD ITM support.

Similarly, if the eCos application is built with Kernel instrumentation enabled and configured for ITM output, then the default stimulus port 24 output can be captured. For example, assuming the application **cminfo** is the ELF file built from an eCos configuration with ITM instrumentation enabled, and is loaded and run via **openocd**, then we could run **parseitm** to capture instrumentation whilst the program executes, and then view the gathered data using the example **instdump** tool provided in the Kernel package.

```
$ parseitm -p 24 -f tpiu.out > inst.bin
^C
$ instdump -r inst.bin cminfo
Threads:
 threadid 1 threadobj 200045D0 "idle_thread"

     0:[THREAD:CREATE][THREAD 4095][TSHAL 4][TSTICK 0][ARG1:200045D0] { ts 4 microseconds }
     1:[SCHED:LOCK][THREAD 4095][TSHAL 45][TSTICK 0][ARG1:00000002] { ts 45 microseconds }
     2:[SCHED:UNLOCK][THREAD 4095][TSHAL 195][TSTICK 0][ARG1:00000002] { ts 195 microseconds }
     3:[SCHED:LOCK][THREAD 4095][TSHAL 346][TSTICK 0][ARG1:00000002] { ts 346 microseconds }
     4:[SCHED:UNLOCK][THREAD 4095][TSHAL 495][TSTICK 0][ARG1:00000002] { ts 495 microseconds }
     5:[THREAD:RESUME][THREAD 1][TSHAL 647][TSTICK 0][ARG1:200045D0][ARG2:200045D0] { ts 647 microseconds }
     6:[SCHED:LOCK][THREAD 1][TSHAL 795][TSTICK 0][ARG1:00000002] { ts 795 microseconds }
     7:[MLQ:ADD][THREAD 1][TSHAL 945][TSTICK 0][ARG1:200045D0][ARG2:0000001F] { ts 945 microseconds }
     8:[SCHED:UNLOCK][THREAD 1][TSHAL 1096][TSTICK 0][ARG1:00000002] { ts 1096 microseconds }
     9:[INTR:ATTACH][THREAD 1][TSHAL 0][TSTICK 0][ARG1:00000000] { ts 10000 microseconds }
    10:[INTR:UNMASK][THREAD 1][TSHAL 149][TSTICK 0][ARG1:00000000] { ts 10149 microseconds }
    11:[INTR:ATTACH][THREAD 1][TSHAL 305][TSTICK 0][ARG1:00000054] { ts 10305 microseconds }
    12:[INTR:UNMASK][THREAD 1][TSHAL 449][TSTICK 0][ARG1:00000054] { ts 10449 microseconds }
```

Ronetix PEEDI notes

On the Ronetix PEEDI, the `peedi.stm32f4dis.cfg` file supplied in the platform HAL package should be used to setup and configure the hardware to an appropriate state to load programs.

The `peedi.stm32f4dis.cfg` file also contains an option to define whether hardware or software breakpoints are used by default, using the `CORE0_BREAKMODE` directive in the `[TARGET]` section. The supplied version of the file defaults to software breakpoints. With this default, hardware breakpoints can still be set from GDB using the **hbreak** command, or in the eCosPro version of Eclipse by setting the Breakpoint Type - consult the "Eclipse/CDT for eCos application development" manual for details. The default can be changed to hardware breakpoints, and remember to use the **reboot** command on the PEEDI command line interface, or press the reset button to make the changes take effect.

On the PEEDI, debugging can be performed either via the telnet interface or using **arm-eabi-gdb**. In the case of the latter, **arm-eabi-gdb** needs to connect to TCP port 2000 on the PEEDI's IP address. For example:

```
(gdb) target remote 111.222.333.444:2000
```

By default when the PEEDI is powered up, the target will always run the initialization section of the `peedi.stm32f4dis.cfg` file, and halts the target. This behaviour is repeated with the **reset** command.

If the board is reset with the '**reset**' command, and then the '**go**' command is given, the board will boot from ROM as normal. A similar effect can be achieved in GDB by connecting with **target remote** and immediately typing **continue** or **c**.

It is also possible for the target to always run, without initialization, after reset. This mode is selected with the `CORE0_STARTUP_MODE` directive in the `[TARGET]` section of the `peedi.stm32f4dis.cfg` file. This conveniently allows the target to be connected to the PEEDI JTAG debugger, and be able to reset and run the resident Flash program without being required to always type '**go**' every time. Finally, it is also possible to set a temporary default (unless the PEEDI is reset) by giving an argument to the **reset** command, for example **reset run**. Use the command **help reset** at the PEEDI command prompt for more options.

Suitably configured applications can be loaded either via GDB, or directly via the telnet CLI into RAM for execution. For example:

```
stm32f4discovery> memory load tftp://192.168.7.9/test.bin bin 0x20000000
++ info: Loading image file:  tftp://192.168.7.9/test.bin
++ info: At absolute address: 0x20000000
loading at 0x20000000
loading at 0x20004000

Successfully loaded 28KB (29064 bytes) in 0.1s
stm32f4discovery> go 0x20000000
```

Consult the PEEDI documentation for information on other formats and loading mechanisms.

For Eclipse users wishing to debug ROM startup programs resident in Flash, it is worth highlighting that it is possible to use the eCosCentric Eclipse plugin to automatically reprogram Flash as the load sequence. To do so, you will need to install and use a TFTP server so that your application can be accessed from the PEEDI from there. You may then use a GDB command file, as described in more detail in the "Eclipse/CDT for eCos application development" manual. This file can then contain contents similar to the following example:

```
define doload
  shell arm-eabi-objcopy -O binary /path/to/eclipse/workspace/projectname/Debug/myapp /path/to/tftp/server/area/myapp.bin
  monitor flash program tftp://10.1.1.1/myapp.bin bin 0x08000000 erase
  set $pc=0x08000000
```

```
end
```

Obviously you will need to adjust the paths and names for your system and TFTP server requirements.

Configuration of JTAG/SWD applications

JTAG/SWD applications can be loaded directly into RAM without requiring a ROM monitor. Loading can be done directly through the JTAG/SWD device, or through GDB where supported by the JTAG/SWD device.

In order to configure the application to support this mode, it is recommended to use the `JTAG` startup type which will implicitly cause two important settings to change. Firstly, `CYGSEM_HAL_USE_ROM_MONITOR` must be disabled. Secondly the `CYGDBG_HAL_DIAG_TO_DEBUG_CHAN` option should be enabled in order to prevent HAL diagnostic output being encoded into GDB ($O) packets. These configuration changes could be made by hand, but use of the `JTAG` startup type will just work.

With these changes, any diagnostic output will appear out of the configured diagnostic channel, usually a serial port. An eCosCentric extension allows diagnostic output to appear in GDB instead. For this to work, you must enable the configuration option `CYGSEM_HAL_DIAG_TO_GDBFILEIO_CHAN` in the common HAL package. Then, after you load your application but before running it, you must give GDB the command:

```
(gdb) set hwdebug on
```

Eclipse users can do this by creating a GDB command file with the contents:

```
define postload
  set hwdebug on
end
```

Thay may then reference it from their Eclipse debug launch configuration. Using GDB command files is described in more detail in the "Eclipse/CDT for eCos application development" manual.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the STM32F4DISCOVERY board hardware and should be read in conjunction with that specification. The STM32F4DISCOVERY platform HAL package complements the Cortex-M architectural HAL and the STM32 variant HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset the HAL will initialize or reinitialize many of the on-chip peripherals. There is an exception for RAM startup applications which depend on a ROM monitor for certain services.

For `ROM` and `JTAG` startup, the HAL will perform additional initialization, programming the various internal registers including the PLL, peripheral clocks and GPIO pins. The details of the early hardware startup may be found in the `src/stm32f4dis_misc.c` in both `hal_system_init` and `hal_platform_init`.

Memory Map

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

Internal RAM

> This is located at address `0x20000000` of the memory space, and is 128KiB in size. The eCos VSR table occupies the bottom 392-bytes. The virtual vector table starts at `0x20000188` and extends to `0x20000288`. For `ROM`, and `JTAG` startups, the top `CYGNUM_HAL_COMMON_INTERRUPTS_STACK_SIZE` bytes are reserved for the interrupt stack. The remainder of internal RAM is available for use by applications.

Internal FLASH

> This is located at address `0x08000000` of the memory space and will be mapped to `0x00000000` at reset. This region is 1MiB in size. ROM applications are by default configured to run from this memory.

On-Chip Peripherals

> These are accessible at locations `0x40000000` and `0xE0000000` upwards. Descriptions of the contents can be found in the STM32F4 Reference Manual (RM0090).

Linker Scripts

The platform linker scripts define the following symbols:

hal_vsr_table

> This defines the location of the VSR table. This is set to `0x20000000` for all startup types, and space for 98 entries is reserved due to the use of the STM32F4 processor.

hal_virtual_vector_table

> This defines the location of the virtual vector table used to communicate between an ROM monitor and an eCos application. This is allocated right after the VSR table, at `0x20000188`.

hal_interrupt_stack

> This defines the location of the interrupt stack. For all startup types this is allocated to the top of available internal SRAM, which is normally `0x20020000`.

hal_startup_stack

> This defines the location of the startup stack. For all startup types it is initially allocated at the half-way point of the interrupt stack.

Diagnostic LEDs

Four LEDs are fitted on the board for diagnostic purposes:

Platform HAL manifest	Colour	Board Label
CYGHWR_HAL_STM32F4DIS_LED0	Orange	LD3
CYGHWR_HAL_STM32F4DIS_LED1	Green	LD4
CYGHWR_HAL_STM32F4DIS_LED2	Red	LD5
CYGHWR_HAL_STM32F4DIS_LED3	Blue	LD6

The platform HAL header file at <cyg/hal/plf_io.h> defines the following convenience function to allow the LEDs to be set:

```
extern void hal_stm32f4dis_led(char c);
```

The lowest 4-bits of the argument c correspond to each of the 4 LEDs (with LED0 as the least significant bit).

The platform HAL will automatically light LED0 when the platform initialisation is complete, however the LEDs are free for application use.

Flash wait states

The STM32F4DISCOVERY platform HAL provides a configuration option to set the number of Flash read wait states to use: CYGNUM_HAL_CORTEXM_STM32_FLASH_WAIT_STATES. It is important to verify and if necessary update this value if changing the CPU clock (HCLK) frequency or CPU voltage. Consult the STM32F40xxx Flash programming manual (PM0081) for appropriate values for different clock speeds or voltages. The default of 5 reflects a supply voltage of 3.3V and HCLK of 168MHz.

Real-time characterization

The **tm_basic** kernel test gives statistics gathered about the real-time characterization and performance of the kernel. The sample output is shown here for information. The test was built for JTAG startup with optimization flag -O2.

Example 304.1. stm32f4dis Real-time characterization

```
            Startup, main stack : stack used    84 size  1536
             Startup : Idlethread stack used    80 size  1280

eCos Kernel Timings
Notes: all times are in microseconds (.000001) unless otherwise stated

Reading the hardware clock takes 0 'ticks' overhead
... this value will be factored out of all other measurements
Clock interrupt took    4.13 microseconds (4 raw clock ticks)

Testing parameters:
   Clock samples:            32
   Threads:                   6
   Thread switches:         128
   Mutexes:                  32
   Mailboxes:                32
   Semaphores:               32
   Scheduler operations:    128
   Counters:                 32
   Flags:                    32
   Alarms:                   32
   Stack Size:             1096


                                 Confidence
     Ave     Min     Max     Var  Ave  Min  Function
```

```
======  ======  ======  ====== ========== ========
  4.67    4.00    5.00    0.44   66%  33% Create thread
  1.00    1.00    1.00    0.00  100% 100% Yield thread [all suspended]
  1.00    1.00    1.00    0.00  100% 100% Suspend [suspended] thread
  1.00    1.00    1.00    0.00  100% 100% Resume thread
  1.67    1.00    2.00    0.44   66%  33% Set priority
  0.00    0.00    0.00    0.00  100% 100% Get priority
  2.83    2.00    3.00    0.28   83%  16% Kill [suspended] thread
  1.00    1.00    1.00    0.00  100% 100% Yield [no other] thread
  1.50    1.00    2.00    0.50  100%  50% Resume [suspended low prio] thread
  1.00    1.00    1.00    0.00  100% 100% Resume [runnable low prio] thread
  1.17    1.00    2.00    0.28   83%  83% Suspend [runnable] thread
  1.00    1.00    1.00    0.00  100% 100% Yield [only low prio] thread
  1.00    1.00    1.00    0.00  100% 100% Suspend [runnable->not runnable]
  2.67    2.00    3.00    0.44   66%  33% Kill [runnable] thread
  2.50    2.00    3.00    0.50  100%  50% Destroy [dead] thread
  4.33    4.00    5.00    0.44   66%  66% Destroy [runnable] thread
  5.17    5.00    6.00    0.28   83%  83% Resume [high priority] thread
  1.86    1.00    2.00    0.24   85%  14% Thread switch

  0.21    0.00    1.00    0.33   78%  78% Scheduler lock
  0.80    0.00    1.00    0.31   80%  19% Scheduler unlock [0 threads]
  0.85    0.00    1.00    0.25   85%  14% Scheduler unlock [1 suspended]
  0.84    0.00    1.00    0.27   83%  16% Scheduler unlock [many suspended]
  0.88    0.00    1.00    0.22   87%  12% Scheduler unlock [many low prio]

  0.22    0.00    1.00    0.34   78%  78% Init mutex
  1.19    1.00    2.00    0.30   81%  81% Lock [unlocked] mutex
  1.44    1.00    2.00    0.49   56%  56% Unlock [locked] mutex
  1.13    1.00    2.00    0.22   87%  87% Trylock [unlocked] mutex
  0.97    0.00    1.00    0.06   96%   3% Trylock [locked] mutex
  0.28    0.00    1.00    0.40   71%  71% Destroy mutex
  6.00    6.00    6.00    0.00  100% 100% Unlock/Lock mutex

  0.44    0.00    1.00    0.49   56%  56% Create mbox
  0.19    0.00    1.00    0.30   81%  81% Peek [empty] mbox
  1.19    1.00    2.00    0.30   81%  81% Put [first] mbox
  0.16    0.00    1.00    0.26   84%  84% Peek [1 msg] mbox
  1.19    1.00    2.00    0.30   81%  81% Put [second] mbox
  0.22    0.00    1.00    0.34   78%  78% Peek [2 msgs] mbox
  1.28    1.00    2.00    0.40   71%  71% Get [first] mbox
  1.19    1.00    2.00    0.30   81%  81% Get [second] mbox
  1.03    1.00    2.00    0.06   96%  96% Tryput [first] mbox
  1.03    1.00    2.00    0.06   96%  96% Peek item [non-empty] mbox
  1.16    1.00    2.00    0.26   84%  84% Tryget [non-empty] mbox
  1.00    1.00    1.00    0.00  100% 100% Peek item [empty] mbox
  1.00    1.00    1.00    0.00  100% 100% Tryget [empty] mbox
  0.22    0.00    1.00    0.34   78%  78% Waiting to get mbox
  0.22    0.00    1.00    0.34   78%  78% Waiting to put mbox
  0.44    0.00    1.00    0.49   56%  56% Delete mbox
  3.84    3.00    4.00    0.26   84%  15% Put/Get mbox

  0.28    0.00    1.00    0.40   71%  71% Init semaphore
  0.97    0.00    1.00    0.06   96%   3% Post [0] semaphore
  1.00    1.00    1.00    0.00  100% 100% Wait [1] semaphore
  0.97    0.00    1.00    0.06   96%   3% Trywait [0] semaphore
  1.00    1.00    1.00    0.00  100% 100% Trywait [1] semaphore
  0.25    0.00    1.00    0.38   75%  75% Peek semaphore
  0.41    0.00    1.00    0.48   59%  59% Destroy semaphore
  3.78    3.00    4.00    0.34   78%  21% Post/Wait semaphore

  0.47    0.00    1.00    0.50   53%  53% Create counter
  0.34    0.00    1.00    0.45   65%  65% Get counter value
  0.16    0.00    1.00    0.26   84%  84% Set counter value
  1.09    1.00    2.00    0.17   90%  90% Tick counter
  0.34    0.00    1.00    0.45   65%  65% Delete counter
```

```
    0.31    0.00    1.00    0.43   68%  68% Init flag
    1.09    1.00    2.00    0.17   90%  90% Destroy flag
    0.97    0.00    1.00    0.06   96%   3% Mask bits in flag
    1.00    1.00    1.00    0.00  100% 100% Set bits in flag [no waiters]
    1.56    1.00    2.00    0.49   56%  43% Wait for flag [AND]
    1.50    1.00    2.00    0.50  100%  50% Wait for flag [OR]
    1.56    1.00    2.00    0.49   56%  43% Wait for flag [AND/CLR]
    1.50    1.00    2.00    0.50  100%  50% Wait for flag [OR/CLR]
    0.25    0.00    1.00    0.38   75%  75% Peek on flag

    0.63    0.00    1.00    0.47   62%  37% Create alarm
    1.63    1.00    2.00    0.47   62%  37% Initialize alarm
    0.97    0.00    1.00    0.06   96%   3% Disable alarm
    1.50    1.00    2.00    0.50  100%  50% Enable alarm
    1.06    1.00    2.00    0.12   93%  93% Delete alarm
    1.25    1.00    2.00    0.38   75%  75% Tick counter [1 alarm]
    8.66    8.00    9.00    0.45   65%  34% Tick counter [many alarms]
    2.22    2.00    3.00    0.34   78%  78% Tick & fire counter [1 alarm]
   39.91   39.00   40.00    0.17   90%   9% Tick & fire counters [>1 together]
    9.59    9.00   10.00    0.48   59%  40% Tick & fire counters [>1 separately]
    4.00    4.00    4.00    0.00  100% 100% Alarm latency [0 threads]
    3.25    3.00    4.00    0.38   75%  75% Alarm latency [2 threads]
    3.22    3.00    4.00    0.34   78%  78% Alarm latency [many threads]
    7.01    7.00    8.00    0.01   99%  99% Alarm -> thread resume latency

  224    220     228  (main stack:   877)  Thread stack used (1096 total)
           All done, main stack : stack used   877 size  1536
            All done : Idlethread stack used   168 size  1280

Timing complete - 27850 ms total

PASS:<Basic timing OK>
EXIT:<done>
```

Chapter 305. STM324X9I-EVAL Platform HAL

Name

`CYGPKG_HAL_CORTEXM_STM32_STM324X9I_EVAL` — eCos Support for the STM324X9I-EVAL Board

Description

This documentation describes the platform-specific elements of the STM324X9I-EVAL board support within eCos. It should be read in conjunction with the STM32 variant HAL section, which covers the common functionality shared by all STM32 variants, including eCos HAL features and on-chip device support. In addition ST's "STM32429I-EVAL evaluation board for the STM32F429 line" (ST User Manual id: UM1667) should be consulted for hardware setup and settings.

The board is equipped with an on-board ST-LINK/V2 hardware debugger interface (via the CN21 “USB ST-LINK” connector), which is typically used for eCos application development. Alternatively the CN13 trace and CN16 JTAG/SWD connectors are available for connecting off-board hardware debuggers.

Supported Hardware

The STM32F429NI has three main on-chip memory regions. The device has a SRAM region of 192KiB present at 0x20000000, and a 2MiB FLASH region present at 0x08000000 (which is aliased to 0x00000000 during normal execution). There is another on-chip RAM region of 64KiB present at 0x10000000 that is only accessible via the CPU core. Also, the STM324X9I-EVAL motherboard has 32MiB of SDRAM memory mapped to address 0x80000000, 2MiB of SRAM memory mapped to address 0x64000000 and 16MiB of NOR-flash memory mapped to address 0x60000000.

Warning

Prior to silicon Rev3 an errata exists that precludes concurrent use of static and dynamic FMC memories. The eCos `STARTUP` type configures which RAM is used for the main application memory, and eCos does not provide any specific workaround. How the non-eCos memory is used is the domain of the application.

For example, if developing applications for silicon revisions that exhibit the problem then if the memory-mapped NOR flash is required a STARTUP selecting the off-chip PSRAM should be configured, and the off-chip SDRAM *not* accessed.

When targeting this STM324x9I-EVAL platform where an early chip revision is present then the STM32 variant option `CYGHWR_HAL_CORTEXM_STM32_F42_ERRATA_FMC_BANKSWITCH` can be enabled. This will ensure that the external NOR flash definitions are *NOT* provided for STARTUP types where SDRAM is selected for use. Alternatively the application developer is free to leave this `ERRATA` option disabled and use run-time logic to ascertain the chip revision from the relevant I/O registers. The application code can then allow NOR flash access as appropriate.

The STM32 variant HAL includes support for the eight on-chip serial devices which are documented in the variant HAL. However, the STM324X9I-EVAL motherboard only provides a single standard DB9 UART connector CN8.

The STM32 variant HAL also includes support for the I²C buses. A single I²C device is instantiated as part of the platform port, which is for the STMPE811 touch-panel sensor connected via bus I²C1. It is exported to `<cyg/io/i2c.h>` with the name `hal_stm324x9i_eval_touchpanel` in the normal way.

Similarly the STM32 variant HAL includes support for the SPI buses. Though the evaluation board does not provide any SPI devices as standard.

USB host and peripheral modes are supported on both the FS OTG1 (connector CN14) and HS OTG2 (connector CN9) controllers available on the evaluation board. The HS OTG2 controller support is currently limited to use at FS speed only. Consult the STM32 variant HAL documentation for USB driver details.

Note

The evaluation board does not support the use of the FS OTG2 (connector CN15) without a hardware modification detailed in the ST evaluation board user manual.

Device drivers are also provided for the STM32 on-chip Ethernet MAC, ADC, BXCAN and SDIO interfaces. Additionally, support is provided for the on-chip watchdog, RTC (wallclock) and a Flash driver exists to permit management of the STM32's on-chip Flash.

Note

The STM32 variant HAL support for the SDIO interface is currently limited to supporting MMC/SD cards. If the multi-bit MMC/SD support is used it is recommended that on-chip SRAM transfer buffers are used to avoid RX overrun or TX underrun due to the slow external SDRAM access speed.

The STM32F4 processor and the STM324X9I-EVAL board provide a wide variety of peripherals, but unless support is specifically indicated, it should be assumed that it is not included.

Tools

The board port is intended to work with GNU tools configured for an **arm-eabi** target. The original port was done using **arm-eabi-gcc** version 4.7.3e, **arm-eabi-gdb** version 7.6.1, and **binutils** version 2.23.2.

Name

Setup — Preparing the STM324X9I-EVAL Board for eCos Development

Overview

Typically, since the STM324X9I-EVAL motherboard has a built-in ST-LINK/V2 interface providing hardware debug support, eCos applications are loaded and run via the debugger **arm-eabi-gdb** or via the Eclipse IDE. The debugger then communicates with the “GDB server” provided by the relevant host ST-LINK/V2 support tool being used (e.g. OpenOCD).

Normally for release applications the `ROM` startup type would be used, with the application programmed into the on-chip flash for execution when the board boots. It is still possible to use the hardware debugging support to debug such flash-based ROM applications, and this may be the desired approach if the application is too large for execution from on-chip SRAM, or where all of the SRAM and SDRAM is required for application run-time use.

If off-chip non-volatile memory (NVM) is used to hold the main application then the board can boot from the internal flash using the BootUp ROM loader. This BootUp code will then start the main application (after an optional update sequence).

If required, it is still possible to program a GDB stub or RedBoot `ROM` image into on-chip Flash and download and debug via the serial UART (CN8). In that case, eCos applications are configured for `RAM` startup and then downloaded and run on the board via the debugger **arm-eabi-gdb**, or via the Eclipse IDE. By default for serial communications, all versions run with 8 bits, no parity, and 1 stop bit at 115200 baud. This rate can be changed in the eCos configuration used for building the GDB stub `ROM` image.

Preparing ST-LINK/V2 interface

The support for using the on-chip ITM stimulus ports for diagnostic and instrumentation output requires that the ST-LINK/V2 firmware is at least version `V2.J17.S0`. The firmware for the ST-LINK/V2 interface can be checked, and updated if needed, using a tool available from STMicroelectronics. The firmware version is also reported when the **openocd** command is executed (using a suitable configuration file). For example, the following OpenOCD output reports `JTAG v23`:

```
Info : STLINK v2 JTAG v27 API v2 SWIM v0 VID 0x0483 PID 0x3748
```

Unfortunately the official firmware updater is only available for the Windows platform at the moment. From a Windows machine:

1. Ensure that the Windows PC and STM324X9I-EVAL board are disconnected.

2. Download the STM32 ST-Link USB driver from ST's website.

   The page titled “ST-Link, ST-Link/V2, ST-Link/V2-1 USB driver signed for XP, Windows7, Windows8” provides the driver download http://www.st.com/content/st_com/en/products/embedded-software/development-tool-software/stsw-link009.html

3. Install the ST-Link USB driver on your Windows PC, by simply unzipping the downloaded file and running the installer contained within.

4. Download the STM32 ST-LINK Utility from ST's website.

   The page titled “STSW-LINK004 STM32 ST-LINK utility” provides the download of the utility http://www.st.com/web/en/catalog/tools/PF258168

5. Install the ST-LINK Utility software on your Windows PC. This is achieved by simply unzipping the downloaded file `stsw-link004.zip` and running the `STM32 ST-LINK Utility_vX.X.0.exe` that was contained within it. Follow the on-screen instructions.

6. Connect the STM324X9I-EVAL board to the PC.

   Connect the STM324X9I-EVAL board to the PC using the ST supplied USB-B cable. Windows should correctly identify the USB device and load the device driver. Windows Device Manager should now show “STMicroelectronics STLink dongle” under “Universal Serial Bus controllers”.

7. Run the ST-LINK Utility and ensure the ST-LINK firmware is up to date.

   From the Windows “Start” menu run the “STM32 ST-LINK Utility”. Click on the `connect` icon, or select `Target->Connect` from the menu. This should confirm that a successful connection can be made to the board. To update the on-board ST-LINK/V2 firmware select `ST-LINK->Firmware Update` from the menu. In the ST-LINK dialog box that then appears click on the `Device Connect` button. This will likely result in a message “ST-Link is not in DFU mode. Please restart it.”. In this case simply disconnect the board from the PC, power cycle the board and then reconnect it after a couple of seconds. Click the `OK` button on the message. In the original ST-Link dialog box click `Device Connect` again. The dialog box should now report the current on-board and available firmware versions, and enable you to upgrade the board by pressing the `Yes >>>>` button. We have tested the system with firmware version `V2.J17.S0` and would recommend this version as a minimum. Clicking `Yes >>>>` will cause a progress bar in the dialog to be animated and should eventually result in a “Update Successful” message. You can then close the various dialogs and exit the ST-LINK Utility. Disconnect and power-cycle the board. Reconnect the board and it is now ready for use with OpenOCD.

Programming ROM images

Since the STM324X9I-EVAL board has a built-in ST-LINK/V2 SWD interface, the USB host connection (CN21) and suitable host software (e.g. The OpenOCD package **openocd** tool) can be used to program the flash.

The **openocd** GDB server can directly program flash based applications from the GDB **load** command.

Note

The **openocd** command being used should have been configured and built to support the ST-LINK/V2 interface. This is achieved by specifying the **--enable-stlink** when configuring the OpenOCD build. Additional information on running **openocd** may be found in the OpenOCD notes.

For example, assuming that **openocd** is running on the same host as GDB, and is connected to the target board the following will program the “bootup.elf” application into the on-chip flash:

```
$ arm-eabi-gdb install/bin/bootup.elf
GNU gdb (eCosCentric GNU tools 4.7.3c) 7.6.1
[ … GDB output elided … ]
(gdb) target remote localhost:3333
hal_reset_vsr () at path/hal_misc.c:171
(gdb) load
Loading section .rom_vectors, size 0x14 lma 0x8000000
Loading section .text, size 0x3adc lma 0x8000018
Loading section .rodata, size 0x6c0 lma 0x8003af8
Loading section .data, size 0x6dc lma 0x80041b8
Start address 0x8000018, load size 18572
Transfer rate: 14 KB/sec, 4643 bytes/write.
(gdb)
```

Alternatively, the **openocd** telnet interface can be used to manually program the flash. By default the **openocd** session provides a command-line via port `4444`. Consult the OpenOCD documentation for more details if a non-default **openocd** configuration is being used.

With a **telnet** connection established to the **openocd** any binary data can easily be written to the on-chip flash. e.g.

```
$ telnet localhost 4444
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Open On-Chip Debugger
> flash write_image test.bin 0x08000000
wrote 32518 bytes from file test.bin in 1.073942s (29.569 KiB/s)
```

To create a binary for flash programming the **arm-eabi-objcopy** command is used. This converts the, ELF format, linked application into a raw binary. For example:

```
$ arm-eabi-objcopy -O binary programname programname.bin
```

Name

Configuration — Platform-specific Configuration Options

Overview

The STM324X9I-EVAL board platform HAL package is loaded automatically when eCos is configured for the `stm32429i_eval` or `stm32439i_eval` targets. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The STM324X9I-EVAL board platform HAL package supports six separate startup types:

ROM

> This startup type can be used for finished (stand-alone) applications which will be programmed into internal flash at location 0x08000000. Data and BSS will be put into external SDRAM starting from 0x80000000. The application will be self-contained with no dependencies on services provided by other software. The program expects to boot from reset with ROM mapped at location zero. It will then transfer control to the 0x08000000 region. eCos startup code will perform all necessary hardware initialization.

ROMAPP

> This startup type can be used for finished applications which will be programmed into internal (on-chip) flash at the configured offset from location (0x08000000+`CYGIMP_BOOTUP_RESERVED`), and started via a suitably configured BootUp ROM loader. Data and BSS will be put into internal SRAM. The application will be self-contained with no dependencies on services provided by other software.

ROMINT

> This startup type can be used for finished applications which will be programmed into internal flash at location 0x08000000. Data and BSS will be put into internal SRAM starting from 0x20000288. Internal SRAM below this address is reserved for vector tables. The application will be self-contained with no dependencies on services provided by other software. The program expects to boot from reset with ROM mapped at location zero. It will then transfer control to the 0x08000000 region. eCos startup code will perform all necessary hardware initialization.
>
> The off-chip SDRAM memory from 0x80000000 and off-chip SRAM memory from 0x64000000 are available, but are not referenced by the eCos run-time so are available for application use if required.

JTAG

> This is the startup type used to build applications that are loaded via the hardware debugger interface. The application will be self-contained with no dependencies on services provided by other software. The program expects to be loaded from 0x80000000 and entered at that address. eCos startup code will perform all necessary hardware initialization, though since the application is loaded via the hardware debugger interface the host debug environment is responsible for configuring the necessary I/O state to initialise the off-chip SDRAM.
>
> This is the startup type normally used during application development, since the large SDRAM memory space allows for larger debug applications where compiler optimisation may be disabled, and run-time assert checking enabled.
>
> ### Note
>
> Executing code from the SDRAM memory has a performance downside. It is significantly slower than execution from on-chip SRAM or flash. If performance is an issue then hardware debugging can be used for any of the startup types if required.

SRAM

This is a variation of the JTAG type that only uses internal memory. The application will be self-contained with no dependencies on services provided by other software. The program expects to be loaded from 0x20000288 and entered at that address. eCos startup code will perform all necessary hardware initialization. Unlike the JTAG startup no explicit hardware debugger configuration is needed, since the application (like the ROM and ROMINT startup types) will initialise the off-chip SDRAM memory.

SRAMEXT

This is a variation of the JTAG type that uses the external SRAM memory from 0x64000000.

RAM

For the ST-LINK/V2 enabled STM324X9I-EVAL platform this startup type is unlikely to be used. It is provided for completeness.

When the board has RedBoot (or a GDB stub ROM) programmed into internal Flash at location 0x08000000 then the arm-eabi-gdb debugger can communicate with the relevant UART connection to load and debug applications. An application is loaded into memory from 0x80008000. It is assumed that the hardware has already been initialized by RedBoot. By default the application will not be stand-alone, and will use the eCos virtual vectors mechanism to obtain services from RedBoot, including diagnostic output.

RedBoot and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building RedBoot.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a `RAM` startup, disabled otherwise. It can be manually disabled for a `RAM` startup, making the application self-contained, as a testing step before switching to `ROM` startup.

Note

Though, as previously discussed, since the option of hardware debugging is available as standard on the STM324X9I-EVAL platform it is unlikely that the `RAM` startup type would be used for development.

SPI Driver

An SPI bus driver is available for the STM32 in the package “ST STM32 SPI driver” (`CYGPKG_DEVS_SPI_CORTEXM_STM32`).

No SPI devices are instantiated for this platform by default.

Note

An example SPI M25PXX configuration can be enabled for boards suitably modified with an attached compatible flash device. The CDL option `CYGPKG_HAL_CORTEXM_STM32_STM324X9I_EVAL_SPI1_FLASH` can be enabled, and uses SPI bus 1 with the chip-select on PA4.

When configured the `m25pxx_flash_device` device is exported and can be accessed via the standard flash API. The device is given a logical base address of `0x00000000` but is *not* memory-mapped.

Consult the generic SPI driver API documentation in the eCosPro Reference Manual for further details on SPI support in eCosPro, along with the configuration options in the STM32 SPI device driver.

I²C Driver

The STM32 variant HAL provides the main I²C hardware driver itself, configured at `CYGPKG_HAL_STM32_I2C`. Since the platform uses an I²C bus 1 based I/O expander the I²C support is always enabled. The STMPE811 touch-panel device is instantiated and becomes available for applications from `<cyg/io/i2c.h>`.

ADC Driver

The STM32 processor variant HAL provides an ADC driver. The STM324X9I-EVAL platform HAL enables the support for the devices ADC1, ADC2 and ADC3 and for configuration of the respective ADC device input channels.

Consult the generic ADC driver API documentation in the eCosPro Reference Manual for further details on ADC support in eCosPro, along with the configuration options in the STM32 ADC device driver.

CAN Driver

The STM32 has a dual BXCAN device for CAN support. This consists of a master device, BXCAN1, and a slave device, BXCAN2. If BXCAN2 is to be used, BXCAN1 must be powered and clocked, regardless of whether it is to be used for CAN traffic. BXCAN1 is the only device connected to an external D-Sub socket at CN22. It shares an IO pin with the OTG FS controller. JP16 controls connection of CAN1_RX to PA11. By default this jumper is not fitted, so one must be fitted to enable BXCAN1. Additionally, the OTG_FS1 connector at CN14 cannot now be used and must be left unconnected. This means that the OTG_FS USB controller and CAN cannot be used concurrently.

Consult the generic CAN driver API documentation in the eCosPro Reference Manual for further details on CAN support in eCosPro, along with the documentation and configuration options in the BXCAN device driver.

Flash Driver

The STM32's on-chip Flash may be programmed and managed using the Flash driver located in the “STM32 Flash memory support” (`CYGPKG_DEVS_FLASH_STM32`) package. This driver is enabled automatically if the generic “Flash device drivers” (`CYGPKG_IO_FLASH`) package is included in the eCos configuration. The driver will configure itself automatically for the size and parameters of the specific STM32 variant present on the STM324X9I-EVAL board.

A number of aspects of Flash driver behaviour can be configured within that driver, such as program/erase parallelism and program burst size. Consult the driver for more details.

Name

SWD support — Usage

Use of JTAG/SWD for debugging

JTAG/SWD can be used to single-step and debug loaded applications, or even applications resident in ROM.

Debugging of ROM applications is only possible if using hardware breakpoints. The Cortex-M4 core of the STM32F429ZI only supports six such hardware breakpoints, so they may need to be used sparingly. If using a GDB front-end such as Eclipse, check it has not set unnecessary extra breakpoints such as at `main()`. Some JTAG/SWD devices give the option of whether to set hardware or software breakpoints by default. Be sure to configure your device appropriately.

When debugging via JTAG, you are likely to need to disable the default HAL idle thread action, otherwise there may be issues where the target fails to halt and the debugging session is unreliable. More details can be found in the Cortex-M architectural HAL. This should *not* be necessary when using a SWD-based hardware debugger such as the on-board ST-LINK/V2 interface.

The default eCos configuration does not enable the use of ITM stimulus ports for the output of HAL diagnostics or Kernel instrumentation. The architecture HAL package `CYGPKG_HAL_CORTEXM` provides options to enable such use.

For HAL diagnostic (e.g. `diag_printf()`) output the architecture CDL option `CYGHWR_HAL_CORTEXM_DIAGNOSTICS_INTERFACE` should be updated to select `ITM` as the output destination. Once the `ITM` option has been configured the option `CYGHWR_HAL_CORTEXM_DIAGNOSTICS_ITM_PORT` allows the actual stimulus port used for the diagnostics to be selected.

When the Kernel instrumentation option `CYGPKG_KERNEL_INSTRUMENT` is enabled then the `CYGHWR_HAL_CORTEXM_ITM_INSTRUMENTATION` option can be enabled to direct instrumentation record output via an ITM stimulus port, rather than into a local memory buffer. The stimulus port used can be configured via the `CYGHWR_HAL_CORTEXM_ITM_INSTRUMENTATION_PORT_BASE` option.

However, when using the STM324X9I-EVAL board via the ST-LINK/V2 interface then it is recommended that the `gdb_hwdebug_fileio` approach is used to provide access to diagnostics via the GDB debug connection. When ITM support is used it has been observed that the ST-LINK/V2 firmware can drop data, leading to the possibility of confusing output. However, with care the ITM system can be tuned to provide diagnostic and instrumentation via the host SWD debugger.

Using the ST-LINK/V2 connection allows for a single cable to provide power (JP12), hardware debug support and diagnostic output.

OpenOCD notes

The OpenOCD debugger can be configured to support the on-board ST-LINK/V2 interface available via the USB CN21 connection. When configuring the **openocd** tool build, the **configure** script can be given the option `--enable-stlink` to provide for ST-LINK support.

An example OpenOCD configuration file `openocd.stm324x9i_eval.cfg` is provided within the eCos platform HAL package in the source repository. This will be in the directory `packages/hal/cortexm/stm32/stm324x9i_eval/current/misc` relative to the root of your eCos installation.

This configuration file can be used with OpenOCD on the host as follows:

```
$ openocd -f openocd.stm324x9i_eval.cfg
Open On-Chip Debugger 0.9.0 (2015-09-18-16:19)
Licensed under GNU GPL v2
For bug reports, read
    http://openocd.org/doc/doxygen/bugs.html
Info : The selected transport took over low-level target control. The results might differ compared \
       to plain JTAG/SWD
```

```
adapter speed: 2000 kHz
adapter_nsrst_delay: 100
none separate
Info : Unable to match requested speed 2000 kHz, using 1800 kHz
Info : Unable to match requested speed 2000 kHz, using 1800 kHz
Info : clock speed 1800 kHz
Info : STLINK v2 JTAG v17 API v2 SWIM v0 VID 0x0483 PID 0x3748
Info : using stlink api v2
Info : Target voltage: 2.886506
Info : stm32f4x.cpu: hardware has 6 breakpoints, 4 watchpoints
```

By default **openocd** provides a console on port `4444`, and this can be used to interact with the target system. This console interface can be used to perform debugging, program the flash, etc.

Normally **arm-eabi-gdb** is used to connect to the default GDB server port `3333` for debugging. For example:

```
(gdb) target remote localhost:3333
Remote debugging using localhost:3333
0x00000000 in ?? ()
(gdb)
```

The application can then be loaded and executed under GDB as normal. If you are using Eclipse then see the “eCos Hardware Debugging” section of the “eCosPro CDT plug-in user's guide” document's “Debugging eCos applications” chapter.

If the HAL diagnostics are configured to use ITM, and stimulus port 31 is configured as the HAL diagnostic destination, then the configuration example above will direct OpenOCD to direct ITM output (and also DWT and ETM) to a file named `tpiu.out` in the current directory of the shell which was used to run the **openocd** command. A more specific filename can be used by adjusting the OpenOCD configuration file.

To extract the ITM output, the Cortex-M architecture HAL package provides a helper program **parseitm** in the directory `packages/hal/cortexm/arch/current/host` relative to the root of your eCos installation. It can be compiled simply with:

```
$ gcc -o parseitm parseitm.c
```

You simply run it with the desired ITM stimulus port and name of the file containing the ITM output, for example:

```
$ parseitm -p 31 -f itm.out
```

It will then echo all ITM stimulus for that port, continuing to read from the file until interrupted with Ctrl-C. Note that limited buffer space in debug hardware such as the ST-LINK can result in occasionally missed ITM data. eCosPro provides a workaround of throttling data within the `CYGHWR_HAL_CORTEXM_ITM_DIAGNOSTICS_THROTTLE` CDL configuration component in order to reduce or avoid lost ITM data. For further details, see the note in OpenOCD ITM support.

Similarly, if the eCos application is built with Kernel instrumentation enabled and configured for ITM output, then the default stimulus port 24 output can be captured. For example, assuming the application **cminfo** is the ELF file built from an eCos configuration with ITM instrumentation enabled, and is loaded and run via **openocd**, then we could run **parseitm** to capture instrumentation whilst the program executes, and then view the gathered data using the example **instdump** tool provided in the Kernel package.

```
$ parseitm -p 24 -f tpiu.out > inst.bin
^C
$ instdump -r inst.bin cminfo
Threads:
 threadid 1 threadobj 200045D0 "idle_thread"

   0:[THREAD:CREATE][THREAD 4095][TSHAL 4][TSTICK 0][ARG1:200045D0] { ts 4 microseconds }
   1:[SCHED:LOCK][THREAD 4095][TSHAL 45][TSTICK 0][ARG1:00000002] { ts 45 microseconds }
   2:[SCHED:UNLOCK][THREAD 4095][TSHAL 195][TSTICK 0][ARG1:00000002] { ts 195 microseconds }
   3:[SCHED:LOCK][THREAD 4095][TSHAL 346][TSTICK 0][ARG1:00000002] { ts 346 microseconds }
   4:[SCHED:UNLOCK][THREAD 4095][TSHAL 495][TSTICK 0][ARG1:00000002] { ts 495 microseconds }
   5:[THREAD:RESUME][THREAD 1][TSHAL 647][TSTICK 0][ARG1:200045D0][ARG2:200045D0] { ts 647 microseconds }
   6:[SCHED:LOCK][THREAD 1][TSHAL 795][TSTICK 0][ARG1:00000002] { ts 795 microseconds }
   7:[MLQ:ADD][THREAD 1][TSHAL 945][TSTICK 0][ARG1:200045D0][ARG2:0000001F] { ts 945 microseconds }
   8:[SCHED:UNLOCK][THREAD 1][TSHAL 1096][TSTICK 0][ARG1:00000002] { ts 1096 microseconds }
   9:[INTR:ATTACH][THREAD 1][TSHAL 0][TSTICK 0][ARG1:00000000] { ts 10000 microseconds }
  10:[INTR:UNMASK][THREAD 1][TSHAL 149][TSTICK 0][ARG1:00000000] { ts 10149 microseconds }
  11:[INTR:ATTACH][THREAD 1][TSHAL 305][TSTICK 0][ARG1:00000054] { ts 10305 microseconds }
  12:[INTR:UNMASK][THREAD 1][TSHAL 449][TSTICK 0][ARG1:00000054] { ts 10449 microseconds }
```

Configuration of JTAG/SWD applications

JTAG/SWD applications can be loaded directly into SRAM or SDRAM without requiring a ROM monitor. Loading can be done directly through the JTAG/SWD device, or through GDB where supported by the JTAG/SWD device.

In order to configure the application to support this mode, it is recommended to use the JTAG startup type which will implicitly cause two important settings to change. Firstly, CYGSEM_HAL_USE_ROM_MONITOR must be disabled. Secondly the CYGDBG_HAL_DIAG_TO_DEBUG_CHAN option should be enabled in order to prevent HAL diagnostic output being encoded into GDB ($O) packets. These configuration changes could be made by hand, but use of the JTAG startup type will just work.

With these changes, any diagnostic output will appear out of the configured diagnostic channel. An eCosCentric extension allows diagnostic output to appear in GDB. For this feature to work, you must enable the configuration option CYGSEM_HAL_DIAG_TO_GDBFILEIO_CHAN in the common HAL package.

For details of using hardware debug with the Eclipse IDE see the “eCos Hardware Debugging” section of the “eCosPro CDT plug-in user's guide” manual.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the STM324X9I-EVAL board hardware, and should be read in conjunction with that specification. The STM324X9I-EVAL platform HAL package complements the Cortex-M architectural HAL and the STM32 variant HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset the HAL will initialize or reinitialize many of the on-chip peripherals. There is an exception for `RAM` startup applications which depend on a ROM monitor for certain services.

For `ROM`, `ROMINT`, `SRAM`, `JTAG` and `SRAMEXT` startup types the HAL will perform additional initialization, programming the various internal registers including the PLL, peripheral clocks and GPIO pins. The details of the early hardware startup may be found in the `src/stm324x9i_eval_misc.c` in both `hal_system_init` and `hal_platform_init`.

Memory Map

The platform HAL package provides the memory layout information needed to generate the linker script. For all the STARTUP variations the top `CYGNUM_HAL_COMMON_INTERRUPTS_STACK_SIZE` bytes of the on-chip SRAM are reserved for the interrupt stack. The remainder of the internal SRAM is available for use by applications. The key memory locations are as follows:

Internal RAM

> This is located at address `0x20000000` of the memory space, and is 192KiB in size. The eCos VSR table occupies the bottom 392 bytes of memory, with the virtual vector table starting at `0x200001AC` and extending to `0x200002AC`.
>
> For all configurations there is also a block of (close-coupled) SRAM located at address `0x10000000` of the memory space, and 64KiB in size.

External SDRAM

> This is located at address `0x80000000` of the memory space, and is 32MiB long. For `ROM` applications, all of the SDRAM is available for use. For `JTAG` applications the application is loaded from `0x80000000` with the remaining SDRAM after the code+data available for application use.
>
> For `RAM` startup applications, SDRAM below `0x80008000` is reserved for the debug monitor (e.g. RedBoot).

External SRAM

> This is located at address `0x64000000` of the memory space, and is 2MiB long. For `SRAMEXT` applications, all of the external SRAM is available for use.

Internal FLASH

> This is located at address `0x08000000` of the memory space and will be mapped to `0x00000000` at reset. This region is 2048KiB in size. `ROM` and `ROMINT` applications are by default configured to run from this memory.

External FLASH

> This is located at address `0x60000000` of the memory space. This region is 16MiB in size.

On-Chip Peripherals

These are accessible at locations `0x40000000` and `0xE0000000` upwards. Descriptions of the contents can be found in the STM32 User Manual.

Linker Scripts

The platform linker scripts define the following symbols:

hal_vsr_table
This defines the location of the VSR table. This is set to `0x20000000` for all startup types, and space for 98 entries is reserved.

hal_virtual_vector_table
This defines the location of the virtual vector table used to communicate between an ROM monitor and an eCos application. This is allocated right after the VSR table, at `0x200001AC`.

hal_interrupt_stack
This defines the location of the interrupt stack. This is allocated to the top of internal SRAM, `0x20030000`.

hal_startup_stack
This defines the location of the startup stack. For all startup types it is initially allocated at the half-way point of the interrupt stack.

I/O expander

If the `CYGPKG_KERNEL` is configured then a "helper" thread is created to handle forwarding of signals received by the I²C bus 1 based STMPE1600 I/O expander as eCos interrupts. The design approach minimises the overall system ISR latency which would occur if the I²C was directly accessed from the system ISR processing, at the cost of deferring the demultiplexing of the I/O expander interrupt source.

Device drivers and applications can access the signals using the same APIs as for standard STM32 GPIO lines, with respect to attaching interrupt handlers or examining/setting pin state.

Note

The priority of the `ioexp_helper_thread()` should be configured as high as possible, to minimise the latency in forwarding the de-multiplexed "virtual" interrupt sources to the relevant device driver (e.g. Ethernet PHY status change).

Diagnostic LEDs

Four LEDs are fitted on the board for diagnostic purposes and are labelled LD1 (green), LD2 (orange), LD3 (red) and LD4 (blue).

The platform HAL header file at `<cyg/hal/plf_io.h>` defines the following convenience function to allow the LEDs to be set:

```
extern void hal_stm324x9i_eval_led(char c);
```

The lowest 4-bits of the argument `c` correspond to each of the 4 LEDs (with LED0 as the least significant bit).

Table 305.1. LEDs

eCos LED GPIO manifest	STM32F4 GPIO	Bit number	Board label	Colour
`CYGHWR_HAL_STM324X9I_EVAL_LED0`	`PG6`	`0`	`LD1`	Green
`CYGHWR_HAL_STM324X9I_EVAL_LED1`	`PG7`	`1`	`LD2`	Orange
`CYGHWR_HAL_STM324X9I_EVAL_LED2`	`PG10`	`2`	`LD3`	Red

eCos LED GPIO manifest	STM32F4 GPIO	Bit number	Board label	Colour
CYGHWR_HAL_STM324X9I_EVAL_LED3	PG12	3	LD4	Blue

The platform HAL will automatically light LED0 when the platform initialisation is complete, however the LEDs are then free for application use.

Note

If the CDL option CYGPKG_HAL_CORTEXM_STM32_STM324X9I_EVAL_SPI1_FLASH is configured (for example, as is the case for the modified “drb” platform with externally attached SPI flash) then the HAL_PLF_DEVS_DISK_MMC_FEEDBACK macro is defined by the platform plf_io.h to provide MMC/SD card access feedback using the CYGHWR_HAL_STM324X9I_EVAL_LED3 (blue) LED.

Flash wait states

The STM324X9I-EVAL platform HAL provides a configuration option to set the number of Flash read wait states to use: CYGNUM_HAL_CORTEXM_STM32_FLASH_WAIT_STATES. It is important to verify and if necessary update this value if changing the CPU clock (HCLK) frequency or CPU voltage. Consult the STM32 Flash programming manual (PM0081) for appropriate values for different clock speeds or voltages. The default of 5 reflects a supply voltage of 3.3V and HCLK of 168MHz.

Real-time characterization

The **tm_basic** kernel test gives statistics gathered about the real-time characterization and performance of the kernel. The sample output is shown here for information. The test was built for SRAM startup with optimization flag -O2, since it provides the best performance as both code and data could remain on-chip.

Example 305.1. stm324x9i_eval Real-time characterization

```
             Startup, main thrd : stack used   344 size  1536
             Startup : Idlethread stack used    84 size  1280

eCos Kernel Timings
Notes: all times are in microseconds (.000001) unless otherwise stated

Reading the hardware clock takes 0 'ticks' overhead
... this value will be factored out of all other measurements
Clock interrupt took    5.00 microseconds (5 raw clock ticks)

Testing parameters:
   Clock samples:            32
   Threads:                   8
   Thread switches:         128
   Mutexes:                  16
   Mailboxes:                16
   Semaphores:               16
   Scheduler operations:    128
   Counters:                 16
   Flags:                    16
   Alarms:                   16
   Stack Size:             1088


                                 Confidence
     Ave     Min     Max     Var  Ave  Min  Function
  ======  ======  ======  ====== ========== ========
INFO:<Ctrl-C disabled until test completion>
    4.00    4.00    4.00    0.00  100% 100% Create thread
    1.00    1.00    1.00    0.00  100% 100% Yield thread [all suspended]
    0.88    0.00    1.00    0.22   87%  12% Suspend [suspended] thread
    0.88    0.00    1.00    0.22   87%  12% Resume thread
    1.25    1.00    2.00    0.38   75%  75% Set priority
    0.38    0.00    1.00    0.47   62%  62% Get priority
    2.63    2.00    3.00    0.47   62%  37% Kill [suspended] thread
    1.00    1.00    1.00    0.00  100% 100% Yield [no other] thread
    1.38    1.00    2.00    0.47   62%  62% Resume [suspended low prio] thread
    0.88    0.00    1.00    0.22   87%  12% Resume [runnable low prio] thread
    1.13    1.00    2.00    0.22   87%  87% Suspend [runnable] thread
    0.88    0.00    1.00    0.22   87%  12% Yield [only low prio] thread
    0.88    0.00    1.00    0.22   87%  12% Suspend [runnable->not runnable]
    2.75    2.00    3.00    0.38   75%  25% Kill [runnable] thread
    2.25    2.00    3.00    0.38   75%  75% Destroy [dead] thread
    4.25    4.00    5.00    0.38   75%  75% Destroy [runnable] thread
    4.75    4.00    6.00    0.56   50%  37% Resume [high priority] thread
    1.54    1.00    2.00    0.50   53%  46% Thread switch

    0.20    0.00    1.00    0.31   80%  80% Scheduler lock
    0.79    0.00    1.00    0.33   78%  21% Scheduler unlock [0 threads]
    0.79    0.00    1.00    0.33   78%  21% Scheduler unlock [1 suspended]
    0.77    0.00    1.00    0.36   76%  23% Scheduler unlock [many suspended]
    0.80    0.00    1.00    0.32   79%  20% Scheduler unlock [many low prio]

    0.25    0.00    1.00    0.38   75%  75% Init mutex
    1.25    1.00    2.00    0.38   75%  75% Lock [unlocked] mutex
    1.19    1.00    2.00    0.30   81%  81% Unlock [locked] mutex
    1.00    1.00    1.00    0.00  100% 100% Trylock [unlocked] mutex
    0.88    0.00    1.00    0.22   87%  12% Trylock [locked] mutex
    0.31    0.00    1.00    0.43   68%  68% Destroy mutex
    5.00    5.00    5.00    0.00  100% 100% Unlock/Lock mutex

    0.44    0.00    1.00    0.49   56%  56% Create mbox
    0.13    0.00    1.00    0.22   87%  87% Peek [empty] mbox
    1.19    1.00    2.00    0.30   81%  81% Put [first] mbox
    0.31    0.00    1.00    0.43   68%  68% Peek [1 msg] mbox
    1.13    1.00    2.00    0.22   87%  87% Put [second] mbox
    0.06    0.00    1.00    0.12   93%  93% Peek [2 msgs] mbox
    1.19    1.00    2.00    0.30   81%  81% Get [first] mbox
    1.06    1.00    2.00    0.12   93%  93% Get [second] mbox
    1.00    1.00    1.00    0.00  100% 100% Tryput [first] mbox
    0.94    0.00    1.00    0.12   93%   6% Peek item [non-empty] mbox
    1.06    1.00    2.00    0.12   93%  93% Tryget [non-empty] mbox
    0.94    0.00    1.00    0.12   93%   6% Peek item [empty] mbox
    0.94    0.00    1.00    0.12   93%   6% Tryget [empty] mbox
    0.25    0.00    1.00    0.38   75%  75% Waiting to get mbox
    0.25    0.00    1.00    0.38   75%  75% Waiting to put mbox
    0.44    0.00    1.00    0.49   56%  56% Delete mbox
    3.50    3.00    4.00    0.50  100%  50% Put/Get mbox

    0.19    0.00    1.00    0.30   81%  81% Init semaphore
    0.81    0.00    1.00    0.31   81%  18% Post [0] semaphore
    0.94    0.00    1.00    0.12   93%   6% Wait [1] semaphore
    1.00    1.00    1.00    0.00  100% 100% Trywait [0] semaphore
    0.81    0.00    1.00    0.31   81%  18% Trywait [1] semaphore
    0.25    0.00    1.00    0.38   75%  75% Peek semaphore
    0.25    0.00    1.00    0.38   75%  75% Destroy semaphore
    3.00    3.00    3.00    0.00  100% 100% Post/Wait semaphore

    0.38    0.00    1.00    0.47   62%  62% Create counter
    0.38    0.00    1.00    0.47   62%  62% Get counter value
    0.19    0.00    1.00    0.30   81%  81% Set counter value
    1.19    1.00    2.00    0.30   81%  81% Tick counter
    0.25    0.00    1.00    0.38   75%  75% Delete counter

    0.25    0.00    1.00    0.38   75%  75% Init flag
    1.00    1.00    1.00    0.00  100% 100% Destroy flag
    1.00    1.00    1.00    0.00  100% 100% Mask bits in flag
    1.00    1.00    1.00    0.00  100% 100% Set bits in flag [no waiters]
    1.50    1.00    2.00    0.50  100%  50% Wait for flag [AND]
    1.31    1.00    2.00    0.43   68%  68% Wait for flag [OR]
    1.31    1.00    2.00    0.43   68%  68% Wait for flag [AND/CLR]
    1.38    1.00    2.00    0.47   62%  62% Wait for flag [OR/CLR]
    0.19    0.00    1.00    0.30   81%  81% Peek on flag

    0.69    0.00    1.00    0.43   68%  31% Create alarm
    1.44    1.00    2.00    0.49   56%  56% Initialize alarm
    1.00    1.00    1.00    0.00  100% 100% Disable alarm
    1.44    1.00    2.00    0.49   56%  56% Enable alarm
    0.94    0.00    1.00    0.12   93%   6% Delete alarm
    1.19    1.00    2.00    0.30   81%  81% Tick counter [1 alarm]
    3.81    3.00    4.00    0.31   81%  18% Tick counter [many alarms]
    2.00    2.00    2.00    0.00  100% 100% Tick & fire counter [1 alarm]
   17.88   17.00   18.00    0.22   87%  12% Tick & fire counters [>1 together]
    4.63    4.00    5.00    0.47   62%  37% Tick & fire counters [>1 separately]
    4.00    4.00    4.00    0.00  100% 100% Alarm latency [0 threads]
    4.00    4.00    4.00    0.00  100% 100% Alarm latency [2 threads]
    4.00    4.00    4.00    0.00  100% 100% Alarm latency [many threads]
    7.01    7.00    8.00    0.01   99%  99% Alarm -> thread resume latency

    0.00    0.00    0.00    0.00            Clock/interrupt latency

    2.71    2.00    3.00    0.00            Clock DSR latency

     204     204     204                    Worker thread stack used (stack size 1088)
            All done, main thrd : stack used   796 size  1536
            All done : Idlethread stack used   172 size  1280

Timing complete - 29330 ms total

PASS:<Basic timing OK>
EXIT:<done>
```

Name

Test Programs — Details

Test Programs

The STM324X9I-EVAL platform HAL contains some test programs which allow various aspects of the board to be tested.

Manual Test

The **manual** test is not built by default. The configuration option `CYGPKG_HAL_CORTEXM_STM32_STM324X9I_EVAL_TESTS_MANUAL` should be enabled to allow the test to be built.

This program tests various aspects of the basic platform port, e.g. flashing LEDs, checking I²C device access and that the push-button GPIO operates.

SPI Flash Test

The **m25pxx1** test is not built by default, since the standard (unmodifed) STM324x9I-EVAL platform does not provide a suitable SPI flash device. The configuration option `CYGPKG_HAL_CORTEXM_STM32_STM324X9I_EVAL_SPI1_FLASH` should be enabled to allow the test to be built if a suitably modified board is being used.

This program tests the M25Pxx compatible serial flash connected to SPI bus 1. It erases, programs and reads a number of sectors in the flash, and should therefore not be run if the flash contains data that should be retained. The `CYGPKG_IO_FLASH` package must be present to allow this test to be built.

ADC Test

The **adc1** program tests the ADC driver for the STM32. The only device connected to the ADC on the board is the potentiometer connected to ADC3 logical channel 8, named RV1 “ADC channel PF10” on the motherboard. Therefore this test primarily tests that. However, in addition it also report the values of the Vrefint and Vbat+Temp inputs that are sourced on-chip. The option `CYGBLD_HAL_CORTEXM_STM324X9I_EVAL_TEST_ADC` must be enabled to run this test since it needs human interaction.

Name

BootUp Integration — Detail

BootUp

The BootUp support for the STM324x9I-EVAL target is primarily implemented in the `stm324x9i_eval_support.c` file. The majority of the functions provided by that source file are only included when the `CYGPKG_BOOTUP` package is being used to construct the actual BootUp ROM loader binary.

The BootUp code is designed to be very simple, and it is envisaged that once its implementation has been tested and validated, the binary will only need to be installed onto a device once. Its only purpose is to allow the safe updating and startup of the main application. If the BootUp code ever needs to be replaced then it is a “factory” operation, for example using JTAG/SWD to reprogram the on-chip flash.

This platform specific documentation should be read in conjunction with the generic BootUp package and bundle image support documentation.

The BootUp package provides a basic but fully functional implementation for the platform. This has been tested to ensure that the underlying mechanism is sound. It is envisaged that the developer will customize and further extend the platform side support to meet their specific application update requirements.

BootUp loaded applications

Applications started via the BootUp loader, since they cannot include the `CYGPKG_BOOTUP` package themselves, may need access to some related configuration state. The platform is responsible for providing such “common” information. For example, the CDL option `CYGIMP_BOOTUP_RESERVED` specifies the amount of on-chip flash set aside for BootUp. Applications can then ensure that they do not interfere with the BootUp loader if using the remaining on-chip flash for their own purposes.

Warning

Care must be taken to ensure that the target application configuration matches the BootUp configuration, since it is normally expected that the applications to be loaded will be independent of the initial BootUp build environment. This includes the fundamental on-chip flash space set aside for the BootUp ROM loader code (`CYGIMP_BOOTUP_RESERVED`) as well as, when using `CYGPKG_BUNDLE` support, where the bundle image is located (selected Non-Volatile Memory (NVM) and offset/partition information). It is expected that such values, for a particular platform instance, will be *fixed* at a suitable point during development, and definitely before products are shipped. It is the responsibility of the developer to ensure a consistent configuration between the BootUp ROM loader and any applications that may be installed/started by that BootUp code.

The platform HAL header file at `<cyg/hal/plf_io.h>` defines the following convenience function for BootUp, and applications started by BootUp, to ascertain the configured off-chip bundle/update location:

```
extern struct cyg_flash_dev *hal_stm324x9i_eval_source_flash( cyg_flashaddr_t *pbase,
                                                              cyg_flashaddr_t *plimit );
```

The function will return a pointer to the relevant flash device. The passed *`pbase`* parameter is a pointer to the value to be filled with the base address for the image (or `NULL` if the value is not needed by the caller). Similarly the *`plimit`* parameter is a pointer to the value to be filled with the limiting address for any image, or `NULL` if the address is not needed.

Primarily to avoid source duplication, the `hal_stm324x9i_eval_source_flash()` function provides common run-time access to the settings derived from the CDL options `CYGIMP_BOOTUP_STM324X9I_EVAL_SOURCE`, `CYGNUM_BOOTUP_STM324X9I_EVAL_SOURCE_OFFSET` and `CYGNUM_BOOTUP_STM324X9I_EVAL_SOURCE_LIMIT`.

Bundle based applications

When the CDL option `CYGIMP_BOOTUP_STM324X9I_EVAL_BUNDLE` is enabled, the STM324x9I-EVAL platform BootUp code incorporates the `CYGPKG_BUNDLE` package and support for bundle based application distribution.

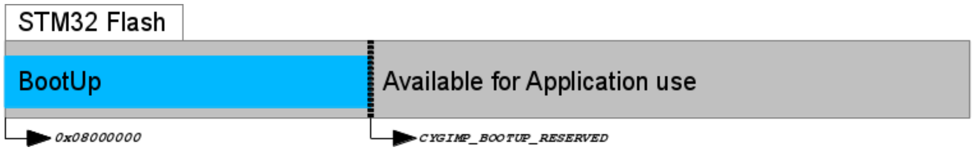
The current STM324x9I-EVAL platform BootUp bundle support is limited (by design) to starting SDRAM based applications. i.e. `CYG_HAL_STARTUP_JTAG` startup type. This has implications for early EVAL hardware containing parts that suffer from the FMC errata since that precludes the use of the off-chip PSRAM and NOR memories when targetting SDRAM for main application code+data.

Note

The (slightly misleading) `JTAG` startup type name is used for standalone SDRAM based applications for historical reasons. The `RAM` startup name is assumed by some systems to refer to an application that relies on the presence of a debug monitor. BootUp is purely a “loader” and does *NOT* provide GDB stubs, so cannot support `CYG_HAL_STARTUP_RAM` applications.

The platform HAL and `CYGPKG_BUNDLE` package provide a common set of routines shared by both BootUp and applications. This ensures that all bundle operations are carried out in a compatible and consistent manner.

Figure 305.1. On-chip flash



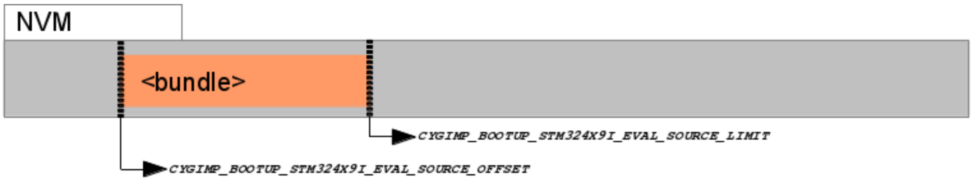
BootUp `ROMINT` application in on-chip flash

On startup the BootUp loader will use the `hal_stm324x9i_eval_source_flash()` function to ascertain the NVM memory used to hold the source bundle image. If a valid bundle image is found, then the configured “main application” item tag as specified by the CDL option `CYGNUM_BOOTUP_STM324X9I_EVAL_BUNDLE_TAG` is searched for within the bundle. If a valid matching tag item is found, then the data from that item is loaded into SDRAM and executed.

Note

This simple approach of using a fixed, pre-allocated, area for holding the bundle image simplifies the BootUp (and similarly any main application based update) code without the issues that would need to be considered if the bundle was stored in a filesystem. e.g. a JFFS2 filesystem on the SPI flash, with potentially slow JFFS2 mount performance, inability to ascertain how much “true” free space is available on the filesystem, programmatic support for deletion of “data” to free space for a bundle as part of an update, etc. The normal “lifetime” cycles of NOR flash (e.g. S25FL256S) should be more than sufficient for the “limited” number of in-field updates that may be undertaken on a specific board over its lifetime.

Figure 305.2. NVM bundle


<bundle>
NVM
CYGIMP_BOOTUP_STM324X9I_EVAL_SOURCE_LIMIT
CYGIMP_BOOTUP_STM324X9I_EVAL_SOURCE_OFFSET

Main application held in bundle stored in NVM

If a valid bundle exists and contains a valid `CYGNUM_BOOTUP_STM324X9I_EVAL_BUNDLE_TAG` item, then the BootUp loader will always start that application. The BootUp loader itself does NOT perform any update revision based automatic update support. That support is entirely within the domain of the started application program, which is responsible for all system update decisions and processing.

The only case where BootUp will initiate a system update is when a bundle is not present or is invalid, or when an otherwise valid bundle doesn't contain a valid `CYGNUM_BOOTUP_STM324X9I_EVAL_BUNDLE_TAG` item. In this case BootUp will attempt to install a bundle from external media. The current example implementation uses a FATFS formatted SD Card for this purpose.

Note

A beneficial side-effect of this approach is that it can help simplify the board production process. Boards only need to be pre-initialized with the stable BootUp binary, which can then be used to install the latest application firmware in NVM.

The example BootUp bundle support provided for this platform expects a single release bundle to be stored in the root directory of an inserted FATFS SD Card. The first file found that matches the `CYGDAT_BOOTUP_STM324X9I_EVAL_BUNDLE_PREFIX` prefix is used. Any filename text after the prefix is ignored and can contain human-readable or customer specific identification information as required. For example, assuming `CYGDAT_BOOTUP_STM324X9I_EVAL_BUNDLE_PREFIX` is configured as `“MyProductName_”`, then files named `MyProductName_1.02`, `MyProductName_1.03.B99.1234`, `MyProductName_example.bin` would all be matched.

The SD Card FATFS filesystem is mounted read-only, so any interrupted update operations (e.g. loss-of-power, reset condition) should not affect the “validity” of the FATFS filesystem held on the SD Card.

Since only a single bundle image is held in the SPI flash there is a chance for the SPI flash based bundle to be in a “corrupt” state if an update fails (power-loss, CPU reset, etc.) during an active update. However, since an update is only manually started when a validated image is available on an SDcard, if the update is interrupted the same SDcard (and field-engineer/operator) should be available to re-apply the update on the system restart. This avoids the (normal) “robustness” requirement of providing two application images to be held in the SPI flash to ensure “safe” updates.

To reiterate, the BootUp code will *ONLY* perform an update from an SD Card to the NVM when there is *NO* valid main application bundle/item pair (missing or corrupted). For in-field upgrades any update process will be instigated under the control of the BootUp started “main application”. For example, the application could use the `CYGPKG_BUNDLE` API to validate a bundle image from whatever source it has access to, and then to update the relevant NVM image itself. If the update is provided on an SD Card then (after ensuring the SD Card does contain a valid bundle image) the main application just needs to invalidate the current NVM bundle image, and then force a CPU reset to have BootUp detect the now invalid main application and apply the update. It is up to the developer to decide the best approach for their particular needs in how point-revision updates are installed, and is beyond the scope of this documentation.

The bundle implementation currently limits the number of automatic update attempts when a missing/invalid bundle is detected. This is a deliberate choice to avoid continually failing attempts that could eventually wear out a flash device. The platform `hal_stm324x9i_eval_badapp()` function implementation, when bundle support is configured, will reset the system to allow another restart attempt if the “Tamper/Key” (`CYGHWR_HAL_STM324X9I_EVAL_BUTTON_USER`) button is pressed for more than one second. This can be used to manually force another “automatic” update attempt to be started.

When BootUp has installed a bundle to the SPI flash, the last 4-bytes of the SPI flash area (partition) set aside for the bundle will be erased to 0xFFFFFFFF. This location and value can be used by the customer main application to ascertain that an install/update has just been performed (since sector erase will only occur as part of a bundle install/update). It is the responsibility of the main application to update this single location (clearing at minimum 1-bit) if it wants to track “post update” state. This can by used by the main application to acknowledge the update, and can ensure, for example, that any (potentially slow) “post update” main application specific functionality is not performed on every normal startup. For example, the main application may need to check and update the software components of attached daughter-boards from the bundle, and can use this mechanism to ensure it is only performed *once* after an update. This simple (erased flash) mechanism avoids complicated support for passing non-volatile “log” information between the seperate BootUp and main application worlds.

Note

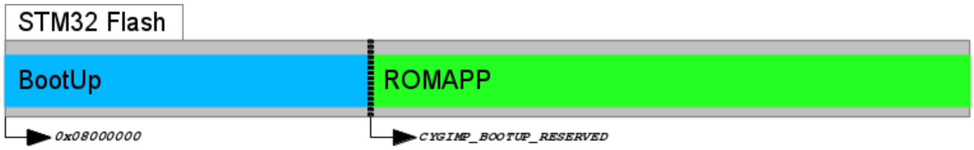
The main application should *NOT* use bit 31 of this field (treating the 32-bit value as being stored in little-endian format) since it is reserved as a flag for the BootUp loader update processing. The main application should *ALWAYS* leave bit 31 set.

On-Chip ROMAPP applications

If the CDL option `CYGIMP_BOOTUP_STM324X9I_EVAL_BUNDLE` is not enabled, then BootUp provides an alternative mechanism that supports the safe update of on-chip flash resident (`CYG_HAL_STARTUP_ROMAPP`) applications.

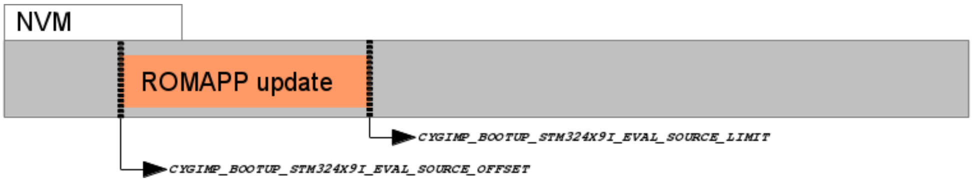
Updates using this mechanism are initiated and directed solely by the application itself. The application is responsible for locating, acquiring and verifying a new update, and placing it into NVM storage. If BootUp detects a verified update in NVM, it installs the update into the on-chip flash, overwriting and replacing the existing application. The updated application is then executed.

Figure 305.3. BootUp and Application



BootUp `ROMINT` loader and `ROMAPP` main application held in on-chip flash

Figure 305.4. Application Update image


NVM
ROMAPP update
CYGIMP_BOOTUP_STM324X9I_EVAL_SOURCE_LIMIT
CYGIMP_BOOTUP_STM324X9I_EVAL_SOURCE_OFFSET

Update for `ROMAPP` main application held in NVM

The example implementation uses a simple scheme that checks a fixed-format contiguous structure near the start of the binary application image file. Other than the fields used to identify the structure, the BootUp code does not interpret the hal_stm324x9i_eval_bootup_structure_t structure `identity` field.

Depending on how the alternative (pending update) application is downloaded and installed in the NVM, it may be more relevant to have the `tail` marker at the very end of the binary image. The developer may wish to update the build/release process so that the actual binary length is held in the application description structure, since that could avoid the overhead of unnecessary flash reads and writes when processing updates. Similarly, instead of a simple binary number being used to differentiate application images, the choice may be made to use the 64-bit UTC timestamp the application was created, or a human-readable string as the unique identification for a release. It is the responsibility of the build/release engineer to ensure individual releases are uniquely identifiable.

It is *critical* that the main application, when storing a pending update, stores the `tail` marker as the last bytes written. It is the responsibility of the main application to verify the data written, prior to placing the `tail` marker. This ensures that a partial image is not treated as a valid update. For example the sequence undertaken by the main application would be:

Table 305.2. Pending update sequence

Operation	**Details**
Invalidate “previous” alternative image	At a minimum ensure an invalid signature `tail` marker is written. Erasing the flash is normally required anyway, and would invalidate any previous image.

Operation	Details
Receive update application image and write to alternative image location	NOT writing the `tail` marker. The code that stores the application should leave a “hole” where the `tail` marker resides to ensure a partial image is not incorrectly treated as valid
Verify downloaded contents	e.g. CRC or binary comparison. Normally this would be done as individual application chunks are downloaded and written to the alternative storage
Write `tail` marker	This is the very last operation after validating that the alternative image has been stored correctly. If an error has occured during the download then not-writing the `tail` ensures that the BootUp loader will not interpret the data written as a pending update
Force system RESET to start update	e.g. using the `HAL_PLATFORM_RESET` macro

The BootUp loader code will only READ from the alternative image location. This ensures that if an in-progress update is interrupted (e.g. power-loss) then when the system restarts the BootUp code will restart the application update as required.

If the BootUp platform implementation for validating the alternative image is extended to include a CRC, or similar “slow” processing, it may be worth considering whether the main application on startup will always invalidate the `tail` marker after an update to avoid subsequent system resets having to re-validate the alternative image prior to discovering that it is the same as the current main application.

Note

We cannot have the SIGNATURE support purely conditional on the BOOTUP support; since non-BOOTUP applications need to be built leaving the space. For the moment this is only enforced for ROMAPP applications, since that is all that the simple (non-BUNDLE) BootUp update support implements.

Building BootUp

The `ROMINT` startup type is chosen for BootUp so that the loader uses the on-chip SRAM for its workspace, to avoid the overhead of managing off-chip memory where the target application will be loaded.

Example eCos configuration templates for BootUp are provided in the `misc` directory of the release. The `hostboot_ROMINT.ecm` configuration file can be used to construct a bundle based BootUp loader and `bootup_ROMINT.ecm` for the simpler on-chip ROMAPP update BootUp loader.

Building a BootUp ROM image is most conveniently done at the command line. For the `stm32429i_eval_drb` (SPI modified board), the steps needed to rebuild the bundle based ROMINT version of BootUp on linux are:

```
$ mkdir hostboot_romint
$ cd hostboot_romint
$ ecosconfig new stm32429i_eval_drb minimal
[ … ecosconfig output elided … ]
$ ecosconfig import \
  $ECOS_REPOSITORY/hal/cortexm/stm32/stm324x9i_eval/current/misc/hostboot_ROMINT.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

The steps needed to rebuild the bundle based ROMINT version of BootUp on Windows within the Shell Environment are

```
C:\Users\demo> mkdir hostboot_romint
C:\Users\demo> cd hostboot_romint
C:\Users\demo\hostboot_romint> ecosconfig new stm32429i_eval_drb minimal
[ … ecosconfig output elided … ]
C:\Users\demo\hostboot_romint> ecosconfig import \
  %ECOS_REPOSITORY%/hal/cortexm/stm32/stm324x9i_eval/current/misc/hostboot_ROMINT.ecm
C:\Users\demo\hostboot_romint> ecosconfig resolve
C:\Users\demo\hostboot_romint> ecosconfig tree
C:\Users\demo\hostboot_romint> make
```

The resulting `install/bin/bootup.bin` binary can then be programmed into the on-chip flash from address `0x08000000`.

It is expected that the BootUp binary is installed onto the STM32F427 on-chip flash either via JTAG/SWD or by utilising the on-chip `BootROM` USB based DFU process. This is a factory or in-field process requiring specific equipment/host-software.

Once BootUp is installed it is not normally expected to require updating. Its purpose is to bootstrap the main application, and provide a standard mechanism for installing the main application. The update mechanism does *NOT* provide a method for updating the BootUp loader itself. If in-field updates of the BootUp binary are necessary, this could be achieved via the STM32 on-chip `BootROM` USB based DFU process.

BootUp Test Programs

The `tests/bundle_example.c` source implements a simple example of an application that utilises the `CYGPKG_BUNDLE` package. Its code could serve as a useful starting point when adding bundle update support to your own application.

Normally a standard `CYG_HAL_STARTUP_JTAG` configured build of the **bundle_example** would be used. If the **bootup** application is used to bootstrap the processor and is built as described in Building BootUp, the eCos Configuration against which **bundle_example** is linked must also include “CRC Support” (`CYGPKG_CRC`), “Zlib compress/decompress support” (`CYGPKG_COMPRESS_ZLIB`), “File IO” (`CYGPKG_IO_FILEIO`) and “Generic FLASH memory support” (`CYGPKG_IO_FLASH`). The “ST STM32 SPI bus 1 option” (`CYGHWR_DEVS_SPI_CORTEXM_STM32_BUS1`) must also be enabled.

As well as ensuring the required packages are present in the configuration, the CDL option `CYGBLD_HAL_CORTEXM_STM324X9I_EVAL_TESTS_MANUAL`, and its sub-option `CYGBLD_HAL_CORTEXM_STM324X9I_EVAL_TEST_BOOTUP`, should be enabled to allow **bundle_example** to be built.

Note

If required, the release provides an example “default” template for the `stm32429i_eval_drb` platform in the `misc/bundle_example.ecm` file which includes all the necessary packages and defines the necessary options. This can be imported to a configuration (using either the command-line **ecosconfig import**, or the GUI **configtool** “File->Import...” support).

Once built, a raw binary copy of the application would be extracted for adding to a bundle using the **arm-eabi-objcopy** command. For example:

```
$ arm-eabi-objcopy -O binary bundle_example bundle_example.bin
```

The resulting binary could then be added to a bundle image using the host-based **bundle** tool:

```
$ bundle MyProductName_example.bin create add 0x0001:bundle_example.bin:C:md5
```

Refer to the bundle host tool section for detailed information on the bundle host tool.

In this example the bundle image would then be placed onto a suitably formatted FATFS SD Card. The bundle would then be installed into the NVM either by a pre-existing main application, or by the BootUp loader if a valid bundle is not currently installed in the NVM.

Note

When placing the bundle image onto a FATFS SD Card only the `CYGDAT_BOOTUP_STM324X9I_EVAL_BUNDLE_PREFIX` configured filename prefix is checked by the BootUp code. It is expected that only a single, suitably prefixed, bundle is present on an SD Card used for update/installation.

The provided **bundle_invalidate** test can be used during BootUp bundle testing to explicitly invalidate any NVM held bundle. This can be done to check the BootUp operation when no valid bundle containing a main application is available, e.g. to test installation from FATFS SD Card.

Chapter 306. STM32F7XX-EVAL Platform HAL

Name

`CYGPKG_HAL_CORTEXM_STM32_STM32F7XX_EVAL` — eCos Support for the STM32F7XX-EVAL Board

Description

The STM32F7XX-EVAL board has a STM32F746NG microcontroller that incorporates 1MiB of internal flash and 320KiB of internal SRAM. The board also has an additional 32MiB of external SDRAM, 2Mib of external SRAM, 16MiB NOR-flash, 64MiB Quad-SPI NOR-flash, and a variety of I/O devices and interfaces.

Since the board is equipped with an on-board ST-LINK/V2 hardware debugger interface (via the CN21 “USB ST-LINK” connector) for typical eCos development test programs are downloaded and debugged via the hardware debugger in conjunction with the relevant host-side tools. Alternatively the CN12 and CN15 connectors are available for connecting off-board hardware debuggers.

This documentation describes platform-specific elements of the STM32F7XX-EVAL board support within eCos. The STM32 variant HAL documentation covers various topics including HAL support common to STM32 variants, and on-chip device support. This document complements the STM32 documentation.

This HAL provides support for two variants of the STM32F7XX evaluation board. The original prototype board was essentially a STM324x9I-EVAL with the processor chip replaced. The STM32F746NG-EVAL2 board is a similar but reworked board with several differences. By default, this HAL, and this documentation concentrates on the STM32F746NG-EVAL2 board.

Supported Hardware

The STM32F746NG has two main on-chip memory regions. The device has a SRAM region of 320KiB present at 0x20000000, and a 1MiB FLASH region present at 0x08000000 (which is aliased to 0x00000000 during normal execution). In addition, the STM32F7XX-EVAL motherboard has 32MiB of SDRAM memory mapped to address 0x60000000, 2MiB of SRAM memory mapped to address 0x64000000, 16MiB of conventional NOR-flash memory mapped to address 0xC0000000, and a 64MiB Quad-SPI NOR-flash device which, if the relevant driver is configured, can be memory mapped to address 0x90000000.

The STM32 variant HAL includes support for the eight on-chip serial devices which are documented in the variant HAL. However, the STM32F7XX-EVAL motherboard only provides a single standard DB9 UART connector CN7.

The STM32 variant HAL also includes support for the I²C buses. Devices are instantiated for the audio codec, touch panel controller and MFX IO expander. The latter is used to control various IO pins for other devices.

Similarly the STM32 variant HAL includes support for the SPI buses. Though the motherboard does not provide any SPI devices as standard.

Device drivers are also provided for the STM32 on-chip Ethernet MAC, ADC, BXCAN and SDIO interfaces.

Additionally, support is provided for the on-chip watchdog, RTC (wallclock) and a Flash driver exists to permit management of the STM32's on-chip Flash.

Note

The STM32 variant HAL support for the SDIO interface is currently limited to supporting MMC/SD cards. If the multi-bit MMC/SD support is used it is recommended that on-chip SRAM transfer buffers are used to avoid RX overrun or TX underrun due to the slow external SDRAM access speed.

The STM32F7 processor and the STM32F7XX-EVAL board provide a wide variety of peripherals, but unless support is specifically indicated, it should be assumed that it is not included in this eCos port.

Tools

The board port is intended to work with GNU tools configured for an **arm-eabi** target. The original port was done using **arm-eabi-gcc** version 4.7.3g, **arm-eabi-gdb** version 7.6.1, and **binutils** version 2.23.2.

Name

Setup — Preparing the STM32F7XX-EVAL Board for eCos Development

Overview

Typically, since the STM32F7XX-EVAL motherboard has a built-in ST-LINK/V2 interface providing hardware debug support, eCos applications are loaded and run via the debugger **arm-eabi-gdb** or via the Eclipse IDE. The debugger then communicates with the "GDB server" provided by the relevant host ST-LINK/V2 support tool being used (e.g. OpenOCD).

Normally for release applications the `ROM` startup type would be used, with the application programmed into the on-chip flash for execution when the board boots. It is still possible to use the hardware debugging support to debug such flash-based ROM applications, and this may be the desired approach if the application is too large for execution from on-chip SRAM, or where all of the SRAM and SDRAM is required for application run-time use.

If off-chip non-volatile memory (NVM) is used to hold the main application then the board can boot from the internal flash using the BootUp ROM loader. This BootUp code will then start the main application (after an optional update sequence).

If required, it is still possible to program a GDB stub or RedBoot `ROM` image into on-chip Flash and download and debug via the serial UART (CN8). In that case, eCos applications are configured for `RAM` startup and then downloaded and run on the board via the debugger **arm-eabi-gdb**, or via the Eclipse IDE. By default for serial communications, all versions run with 8 bits, no parity, and 1 stop bit at 115200 baud. This rate can be changed in the eCos configuration used for building the `ROM` image.

Preparing ST-LINK/V2 interface

The support for using the on-chip ITM stimulus ports for diagnostic and instrumentation output requires that the ST-LINK/V2 firmware is at least version `V2.J17.SO`. The firmware for the ST-LINK/V2 interface can be checked, and updated if needed, using a tool available from STMicroelectronics. The firmware version is also reported when the **openocd** command is executed (using a suitable configuration file). For example, the following OpenOCD output reports `JTAG v17`:

```
Info : STLINK v2 JTAG v17 API v2 SWIM v0 VID 0x0483 PID 0x3748
```

Unfortunately the official firmware updater is only available for the Windows platform at the moment. From a Windows machine:

1. Ensure that the Windows PC and STM32F7XX-EVAL board are disconnected.
2. Download the STM32 ST-LINK Utility from ST's website.

   The page titled "STSW-LINK004 STM32 ST-LINK utility" provides a free download of the utility http://www.st.com/web/en/catalog/tools/PF258168

3. Install the ST-LINK Utility software on your Windows PC.

   Simply unzip the downloaded file `stsw-link004.zip` and run the `STM32 ST-LINK Utility_v3.0.0.exe` that was contained within it. Follow the on-screen instructions. This will install both the utility application and the ST-LINK/V2 USB driver.

4. Connect the STM32F7XX-EVAL board to the PC.

   Connect the STM32F7XX-EVAL board to the PC using the ST supplied mini-B USB cable. Windows should correctly identify the USB device and load the device driver. Windows Device Manager should now show "STMicroelectronics STLink dongle" under "Universal Serial Bus controllers".

5. Run the ST-LINK Utility and ensure the ST-LINK firmware is up to date.

   From the Windows "Start" menu run the "STM32 ST-LINK Utility". Click on the `connect` icon, or select `Target->Connect` from the menu. This should confirm that a successful connection can be made to the board. To update the on-board ST-

LINK/V2 firmware select `ST-LINK->Firmware Update` from the menu. In the ST-LINK dialog box that then appears click on the `Device Connect` button. This will likely result in a message “ST-Link is not in DFU mode. Please restart it.”. In this case simply disconnect the board from the PC and then reconnect it after a couple of seconds, then click the `OK` button on the message. In the original ST-Link dialog box click `Device Connect` again. The dialog box should now report the current on-board and available firmware versions, and enable you to upgrade the board by pressing the `Yes >>>>` button. We have tested the system with firmware version `V2.J17.S0` and would recommend this version as a minimum. Clicking `Yes >>>>` will cause a progress bar in the dialog to be animated and should eventually result in a “Update Successful” message. You can then close the various dialogs and exit the ST-LINK Utility. Disconnect and reconnect the board and it is now ready for use with OpenOCD.

Programming ROM images

Since the STM32F7XX-EVAL board has a built-in ST-LINK/V2 SWD interface, the micro USB host connection (CN21) and suitable host software (e.g. The OpenOCD package **openocd** tool) can be used to program the flash.

The **openocd** GDB server can directly program flash based applications from the GDB **load** command.

Note

The **openocd** command being used should have been configured and built to support the ST-LINK/V2 interface. This is achieved by specifying the **--enable-stlink** when configuring the OpenOCD build. Additional information on running **openocd** may be found in the OpenOCD notes.

For example, assuming that **openocd** is running on the same host as GDB, and is connected to the target board the following will program the “bootup.elf” application into the on-chip flash:

```
$ arm-eabi-gdb install/bin/bootup.elf
GNU gdb (eCosCentric GNU tools 4.7.3c) 7.6.1
[ … GDB output elided … ]
(gdb) target remote localhost:3333
hal_reset_vsr () at path/hal_misc.c:171
(gdb) load
Loading section .rom_vectors, size 0x14 lma 0x8000000
Loading section .text, size 0x3adc lma 0x8000018
Loading section .rodata, size 0x6c0 lma 0x8003af8
Loading section .data, size 0x6dc lma 0x80041b8
Start address 0x8000018, load size 18572
Transfer rate: 14 KB/sec, 4643 bytes/write.
(gdb)
```

Alternatively, the **openocd** telnet interface can be used to manually program the flash. By default the **openocd** session provides a command-line via port `4444`. Consult the OpenOCD documentation for more details if a non-default **openocd** configuration is being used.

With a **telnet** connection established to the **openocd** any binary data can easily be written to the on-chip flash. e.g.

```
$ telnet localhost 4444
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Open On-Chip Debugger
> flash write_image test.bin 0x08000000
wrote 32518 bytes from file test.bin in 1.073942s (29.569 KiB/s)
```

To create a binary for flash programming the **arm-eabi-objcopy** command is used. This converts the, ELF format, linked application into a raw binary. For example:

```
$ arm-eabi-objcopy -O binary programname programname.bin
```

Name

Configuration — Platform-specific Configuration Options

Overview

The STM32F7XX-EVAL board platform HAL package is loaded automatically when eCos is configured for the `stm32f746g_eval2` or `stm32f746g_proto` targets. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The STM32F7XX-EVAL board platform HAL package supports six separate startup types.

Note

Due to Cortex-M limitations regarding executable addresses, and the STM32F7 FMC memory bank address locations, it is not possible to have a configuration with code executing from SDRAM and PSRAM/NOR without runtime support for manipulating FMC_MEMRMP when moving between regions. The eCos run-time does *not* provide such support. The relevant `CYG_HAL_STARTUP` configuration should be chosen for the desired application CODE +DATA memory usage, with the other external memory being used purely for DATA storage.

ROM

This startup type can be used for finished (stand-alone) applications which will be programmed into internal flash at location 0x08000000. Data and BSS will be put into external SDRAM starting from 0x60000000. The application will be self-contained with no dependencies on services provided by other software. The program expects to boot from reset with ROM mapped at location zero. It will then transfer control to the 0x08000000 region. eCos startup code will perform all necessary hardware initialization.

The off-chip SRAM memory from 0xC8000000 is available, but is not referenced by the eCos run-time. It is available for application DATA use if required.

ROMAPP

This startup type can be used for finished applications which will be programmed into internal (on-chip) flash at the configured offset from location (0x08000000+`CYGIMP_BOOTUP_RESERVED`), and started via a suitably configured BootUp ROM loader. Data and BSS will be put into internal SRAM. The application will be self-contained with no dependencies on services provided by other software.

ROMINT

This startup type can be used for finished applications which will be programmed into internal flash at location 0x08000000. Data and BSS will be put into internal SRAM starting from 0x200002C8. Internal SRAM below this address is reserved for vector tables. The application will be self-contained with no dependencies on services provided by other software. The program expects to boot from reset with ROM mapped at location zero. It will then transfer control to the 0x08000000 region. eCos startup code will perform all necessary hardware initialization.

The off-chip SDRAM memory from 0x60000000 and off-chip SRAM memory from 0xC8000000 are available, but are not referenced by the eCos run-time so are available for application use if required. The SDRAM can be used for CODE+DATA, with the external SRAM for DATA.

JTAG

This is the startup type used to build applications that are loaded via the hardware debugger interface. The application will be self-contained with no dependencies on services provided by other software. The program expects to be loaded from

0x60000000 and entered at that address. eCos startup code will perform all necessary hardware initialization, though since the application is loaded via the hardware debugger interface the host debug environment is responsible for configuring the necessary I/O state to initialise the off-chip SDRAM.

This is the startup type normally used during application development, since the large SDRAM memory space allows for larger debug applications where compiler optimisation may be disabled, and run-time assert checking enabled.

Note

Executing code from the SDRAM memory has a performance downside. It is significantly slower than execution from on-chip SRAM or flash. If performance is an issue then hardware debugging can be used for any of the startup types if required.

The off-chip SRAM memory from 0xC8000000 is available, but is not referenced by the eCos run-time. It is available for application DATA use if required.

SRAM

This is a variation of the `JTAG` type that only uses internal memory. The application will be self-contained with no dependencies on services provided by other software. The program expects to be loaded from 0x200002C8 and entered at that address. eCos startup code will perform all necessary hardware initialization. Unlike the `JTAG` startup no explicit hardware debugger configuration is needed, since the application (like the `ROM` and `ROMINT` startup types) will initialise the off-chip SDRAM memory as required.

The off-chip SRAM memory from 0xC8000000 is also available, but is not referenced by the eCos run-time. It is available for application DATA use if required.

SRAMEXT

This is a variation of the `JTAG` type that uses the external SRAM memory addressed from 0x68000000.

The off-chip SDRAM memory from 0xC0000000 is also available, but is not referenced by the eCos run-time. It is available for application DATA use if required.

RAM

For the ST-LINK/V2 enabled STM32F7XX-EVAL platform this startup type is unlikely to be used. It is provided for completeness.

When the board has RedBoot (or a GDB stub ROM) programmed into internal Flash at location 0x08000000 then the arm-eabi-gdb debugger can communicate with the relevant UART connection to load and debug applications. An application is loaded into memory from 0x60100000. It is assumed that the hardware has already been initialized by RedBoot. By default the application will not be stand-alone, and will use the eCos virtual vectors mechanism to obtain services from RedBoot, including diagnostic output.

Note

As well as having a memory footprint cost, RedBoot use can adversely affect the real-time performance of an eCos application.

RedBoot and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building RedBoot.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a `RAM` startup, disabled otherwise. It can be manually disabled for a `RAM` startup, making the application self-contained, as a testing step before switching to `ROM` startup.

Note

Though, as previously discussed, since the option of hardware debugging is available as standard on the STM32F7XX-EVAL platform it is unlikely that the `RAM` startup type would be used for development.

SPI Driver

An SPI bus driver is available for the STM32 in the package “ST STM32 SPI driver” (`CYGPKG_DEVS_SPI_CORTEXM_STM32`).

When QSPI NOR flash support is enabled in the configuration with `CYGHWR_HAL_CORTEXM_STM32_FLASH_QSPI`, then the `m25pxx_flash_device` device is exported and can be accessed via the standard flash API. The device is given a logical base address to match its physical base address of `0x90000000` (corresponding to FMC bank 4) when it is memory mapped (if `CYGFUN_DEVS_FLASH_QSPI_CORTEXM_STM32_MEMMAPPED` is enabled in the QSPI driver, which is the default). Even if memory mapping is disabled, using the eCos Flash API will still allow the device to be read/written at that logical base address.

Consult the generic SPI driver API documentation in the eCosPro Reference Manual for further details on SPI support in eCosPro, along with the configuration options in the STM32 SPI device driver.

I²C Driver

The STM32 variant HAL provides the main I²C hardware driver itself, configured at `CYGPKG_HAL_STM32_I2C`. Since the platform uses an I²C bus 1 based I/O expander the I²C support is always enabled. The touch-panel device is instantiated and becomes available for applications from `<cyg/io/i2c.h>`.

ADC Driver

The STM32 processor variant HAL provides an ADC driver. The STM32F7XX-EVAL platform HAL enables the support for the devices ADC1, ADC2 and ADC3 and for configuration of the respective ADC device input channels.

Consult the generic ADC driver API documentation in the eCosPro Reference Manual for further details on ADC support in eCosPro, along with the configuration options in the STM32 ADC device driver.

Ethernet Driver

The Ethernet MAC is connected to a DP83848 PHY via the MII interface and thence to a RJ45 connector at CN9. By default the external 25MHz crystal is used to supply the clock and JP6 must be set to the 1-2 position, which should be the default setting. The MMIO and MDC signals must be connected to PA2 and PC1, so JP21 and JP22 should be set to the 1-2 position, which should also be the default settings.

By default solder bridges SB36, SB47 and SB9 are open, which means that MII_CRS, MII_COL and MII_RX_ER are not connected to the PHY. Without MII_CRS and MII_COL, the MAC will only operate in full duplex mode and not half duplex. This means that it will work with Ethernet switches, but not with older hubs.

CAN Driver

The STM32 has a dual BXCAN device for CAN support. This consists of a master device, BXCAN1, and a slave device, BXCAN2. If BXCAN2 is to be used, BXCAN1 must be powered and clocked, regardless of whether it is to be used for CAN traffic. BXCAN1 is the only device connected to an external D-Sub socket at CN22. It shares an IO pin with the OTG FS controller. JP8 controls connection of CAN1_RX to PA11. By default this jumper is not fitted, so one must be fitted to enable BXCAN1. Additionally, the OTG_FS1 connector at CN13 cannot now be used and must be left unconnected. This means that the OTG_FS USB controller and CAN cannot be used concurrently.

Consult the generic CAN driver API documentation in the eCosPro Reference Manual for further details on CAN support in eCosPro, along with the documentation and configuration options in the BXCAN device driver.

Flash Driver

The STM32's on-chip Flash may be programmed and managed using the Flash driver located in the “STM32 Flash memory support” (`CYGPKG_DEVS_FLASH_STM32`) package. This driver is enabled automatically if the generic “Flash device drivers” (`CYGPKG_IO_FLASH`) package is included in the eCos configuration. The driver will configure itself automatically for the size and parameters of the specific STM32 variant present on the STM32F7XX-EVAL board.

A number of aspects of Flash driver behaviour can be configured within that driver, such as program/erase parallelism and program burst size. Consult the driver for more details.

Name

SWD support — Usage

Use of JTAG/SWD for debugging

JTAG/SWD can be used to single-step and debug loaded applications, or even applications resident in ROM.

Debugging of ROM applications is only possible if using hardware breakpoints. The Cortex-M7 core of the STM32F7 only supports eight such hardware breakpoints, so they may need to be used sparingly. If using a GDB front-end such as Eclipse, check it has not set unnecessary extra breakpoints such as at `main()`. Some JTAG/SWD devices give the option of whether to set hardware or software breakpoints by default. Be sure to configure your device appropriately.

When debugging via JTAG, you are likely to need to disable the default HAL idle thread action, otherwise there may be issues where the target fails to halt and the debugging session is unreliable. More details can be found in the Cortex-M architectural HAL. This should *not* be necessary when using a SWD-based hardware debugger such as the on-board ST-LINK/V2 interface.

The default eCos configuration does not enable the use of ITM stimulus ports for the output of HAL diagnostics or Kernel instrumentation. The architecture HAL package `CYGPKG_HAL_CORTEXM` provides options to enable such use.

For HAL diagnostic (e.g. `diag_printf()`) output the architecture CDL option `CYGHWR_HAL_CORTEXM_DIAGNOSTICS_INTERFACE` should be updated to select `ITM` as the output destination. Once the `ITM` option has been configured the option `CYGHWR_HAL_CORTEXM_DIAGNOSTICS_ITM_PORT` allows the actual stimulus port used for the diagnostics to be selected.

When the Kernel instrumentation option `CYGPKG_KERNEL_INSTRUMENT` is enabled then the `CYGHWR_HAL_CORTEXM_ITM_INSTRUMENTATION` option can be enabled to direct instrumentation record output via an ITM stimulus port, rather than into a local memory buffer. The stimulus port used can be configured via the `CYGHWR_HAL_CORTEXM_ITM_INSTRUMENTATION_PORT_BASE` option.

However, when using the STM32F7XX-EVAL board via the ST-LINK/V2 interface then it is recommended that the `gdb_hwdebug_fileio` approach is used to provide access to diagnostics via the GDB debug connection. When ITM support is used it has been observed that the ST-LINK/V2 firmware can drop data, leading to the possibility of confusing output. However, with care the ITM system can be tuned to provide diagnostic and instrumentation via the host SWD debugger.

Using the ST-LINK/V2 connection allows for a single cable to provide power (JP12), hardware debug support and diagnostic output.

OpenOCD notes

The OpenOCD debugger can be configured to support the on-board ST-LINK/V2 interface available via the USB CN21 connection. When configuring the **openocd** tool build, the **configure** script can be given the option `--enable-stlink` to provide for ST-LINK support.

Two example OpenOCD configuration files, `openocd.stm32f7xx_eval.sdram.cfg` and `openocd.stm32f7xx_eval.psram.cfg`, are provided within the eCos platform HAL package in the source repository. The latter file is used for SRAMEXT startup types (although it can also be used for SRAM and ROMINT startup types) with the former used for all other startup types. These are in the directory `packages/hal/cortexm/stm32/stm32f7xx_eval/current/misc` relative to the root of your eCos installation and the appropriate OpenOCD configuration configuration file will be copied as the file *`PREFIX`*`/etc/openocd.cfg` when you build the `target.ld` for your configuration.

This configuration file can be used with OpenOCD on the host as follows:

```
$ openocd -f PREFIX/etc/openocd.cfg
Open On-Chip Debugger 0.9.0 (2015-09-18-16:19)
Licensed under GNU GPL v2
For bug reports, read
```

```
    http://openocd.org/doc/doxygen/bugs.html
Info : The selected transport took over low-level target control. The results might differ compared to plain JTAG/SWD
adapter speed: 2000 kHz
adapter_nsrst_delay: 100
none separate
Info : Unable to match requested speed 2000 kHz, using 1800 kHz
Info : Unable to match requested speed 2000 kHz, using 1800 kHz
Info : clock speed 1800 kHz
Info : STLINK v2 JTAG v17 API v2 SWIM v0 VID 0x0483 PID 0x3748
Info : using stlink api v2
Info : Target voltage: 2.886506
Info : stm32f4x.cpu: hardware has 6 breakpoints, 4 watchpoints
```

Note

Do not edit the *PREFIX*`/etc/openocd.cfg` directly for use by **openocd** as this will be over-written by the original when the eCos configuration changes and `target.ld` is rebuilt. Instead, create a copy of the file for local changes and specify the copy as the `-f` argument to **openocd**.

By default **openocd** provides a console on port `4444`, and this can be used to interact with the target system. This console interface can be used to perform debugging, program the flash, etc.

Normally **arm-eabi-gdb** is used to connect to the default GDB server port `3333` for debugging. For example:

```
(gdb) target remote localhost:3333
Remote debugging using localhost:3333
0x00000000 in ?? ()
(gdb) monitor reset halt
target state: halted
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x0800422c msp: 0x20000c80
(gdb)
```

The application can then be loaded and executed under GDB as normal. If you are using Eclipse then you can review the “Hardware Assisted Debugging” section of the “Eclipse/CDT for eCos application development” document's “Debugging eCos applications” chapter for further details.

If the HAL diagnostics are configured to use ITM, and stimulus port 31 is configured as the HAL diagnostic destination, then the configuration example above will direct OpenOCD to direct ITM output (and also DWT and ETM) to a file named `tpiu.out` in the current directory of the shell which was used to run the **openocd** command. A more specific filename can be used by adjusting the OpenOCD configuration file.

To extract the ITM output, the Cortex-M architecture HAL package provides a helper program **parseitm** in the directory `packages/hal/cortexm/arch/current/host` relative to the root of your eCos installation. It can be compiled simply with:

```
$ gcc -o parseitm parseitm.c
```

You simply run it with the desired ITM stimulus port and name of the file containing the ITM output, for example:

```
$ parseitm -p 31 -f itm.out
```

It will then echo all ITM stimulus for that port, continuing to read from the file until interrupted with Ctrl-C. Note that limited buffer space in debug hardware such as the ST-LINK can result in occasionally missed ITM data. eCosPro provides a workaround of throttling data within the `CYGHWR_HAL_CORTEXM_ITM_DIAGNOSTICS_THROTTLE` CDL configuration component in order to reduce or avoid lost ITM data. For further details, see the note in OpenOCD ITM support.

Similarly, if the eCos application is built with Kernel instrumentation enabled and configured for ITM output, then the default stimulus port 24 output can be captured. For example, assuming the application **cminfo** is the ELF file built from an eCos configuration with ITM instrumentation enabled, and is loaded and run via **openocd**, then we could run **parseitm** to capture instrumentation whilst the program executes, and then view the gathered data using the example **instdump** tool provided in the Kernel package.

```
$ parseitm -p 24 -f tpiu.out > inst.bin
^C
$ instdump -r inst.bin cminfo
```

```
Threads:
 threadid 1 threadobj 200045D0 "idle_thread"

     0:[THREAD:CREATE][THREAD 4095][TSHAL 4][TSTICK 0][ARG1:200045D0] { ts 4 microseconds }
     1:[SCHED:LOCK][THREAD 4095][TSHAL 45][TSTICK 0][ARG1:00000002] { ts 45 microseconds }
     2:[SCHED:UNLOCK][THREAD 4095][TSHAL 195][TSTICK 0][ARG1:00000002] { ts 195 microseconds }
     3:[SCHED:LOCK][THREAD 4095][TSHAL 346][TSTICK 0][ARG1:00000002] { ts 346 microseconds }
     4:[SCHED:UNLOCK][THREAD 4095][TSHAL 495][TSTICK 0][ARG1:00000002] { ts 495 microseconds }
     5:[THREAD:RESUME][THREAD 1][TSHAL 647][TSTICK 0][ARG1:200045D0][ARG2:200045D0] { ts 647 microseconds }
     6:[SCHED:LOCK][THREAD 1][TSHAL 795][TSTICK 0][ARG1:00000002] { ts 795 microseconds }
     7:[MLQ:ADD][THREAD 1][TSHAL 945][TSTICK 0][ARG1:200045D0][ARG2:0000001F] { ts 945 microseconds }
     8:[SCHED:UNLOCK][THREAD 1][TSHAL 1096][TSTICK 0][ARG1:00000002] { ts 1096 microseconds }
     9:[INTR:ATTACH][THREAD 1][TSHAL 0][TSTICK 0][ARG1:00000000] { ts 10000 microseconds }
    10:[INTR:UNMASK][THREAD 1][TSHAL 149][TSTICK 0][ARG1:00000000] { ts 10149 microseconds }
    11:[INTR:ATTACH][THREAD 1][TSHAL 305][TSTICK 0][ARG1:00000054] { ts 10305 microseconds }
    12:[INTR:UNMASK][THREAD 1][TSHAL 449][TSTICK 0][ARG1:00000054] { ts 10449 microseconds }
```

Configuration of JTAG/SWD applications

JTAG/SWD applications can be loaded directly into SRAM or SDRAM without requiring a ROM monitor. Loading can be done directly through the JTAG/SWD device, or through GDB where supported by the JTAG/SWD device.

In order to configure the application to support this mode, it is recommended to use the `JTAG` startup type which will implicitly cause two important settings to change. Firstly, `CYGSEM_HAL_USE_ROM_MONITOR` must be disabled. Secondly the `CYGDBG_HAL_DIAG_TO_DEBUG_CHAN` option should be enabled in order to prevent HAL diagnostic output being encoded into GDB ($O) packets. These configuration changes could be made by hand, but use of the `JTAG` startup type will just work.

With these changes, any diagnostic output will appear out of the configured diagnostic channel. An eCosCentric extension allows diagnostic output to appear in GDB. For this feature to work, you must enable the configuration option `CYGSEM_HAL_DIAG_TO_GDBFILEIO_CHAN` in the common HAL package. If you are using the graphical configuration tool then you should then accept any suggested solutions to the subsequent configuration conflicts. Older eCos releases also required the gdb "set hwdebug on" command to be used to enable GDB or Eclipse console output, but this is no longer required with the latest tools.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the STM32F7XX-EVAL board hardware, and should be read in conjunction with that specification. The STM32F7XX-EVAL platform HAL package complements the Cortex-M architectural HAL and the STM32 variant HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset the HAL will initialize or reinitialize many of the on-chip peripherals. There is an exception for `RAM` startup applications which depend on a ROM monitor for certain services.

For `ROM`, `ROMINT`, `SRAM`, `JTAG` and `SRAMEXT` startup types the HAL will perform additional initialization, programming the various internal registers including the PLL, peripheral clocks and GPIO pins. The details of the early hardware startup may be found in the `src/stm32f7xx_eval_misc.c` in both `hal_system_init` and `hal_platform_init`.

Memory Map

The platform HAL package provides the memory layout information needed to generate the linker script. For all the STARTUP variations the top `CYGNUM_HAL_COMMON_INTERRUPTS_STACK_SIZE` bytes of the on-chip SRAM are reserved for the interrupt stack. The remainder of the internal SRAM is available for use by applications. The key memory locations are as follows:

Internal RAM

> This is located at address `0x20000000` of the memory space, and is 320KiB in size. The eCos VSR table occupies the bottom 392 bytes of memory, with the virtual vector table starting at `0x200001AC` and extending to `0x200002AC`.
>
> This memory region comprises three contiguous memory blocks, the DTCM (Data Tightly Coupled Memory), SRAM region 1 and SRAM region 2.

External SDRAM

> This is located at address `0x60000000` of the memory space, and is 32MiB long. For `ROM` applications, all of the SDRAM is available for use. For `JTAG` applications the application is loaded from `0x80000000` with the remaining SDRAM after the code+data available for application use.
>
> For `RAM` startup applications, SDRAM below `0x60008000` is reserved for the debug monitor (e.g. RedBoot).

External SRAM

> This is located at address `0x64000000` of the memory space, and is 2MiB long. For `SRAMEXT` applications, all of the external SRAM is available for use.

Internal FLASH

> This is located at address `0x08000000` of the memory space and will be mapped to `0x00000000` at reset. This region is 1024KiB in size. `ROM` and `ROMINT` applications are by default configured to run from this memory.

External FLASH

> This is located at address `0xC0000000` of the memory space. This region is 16MiB in size.

On-Chip Peripherals

These are accessible at locations 0x40000000 and 0xE0000000 upwards. Descriptions of the contents can be found in the STM32 User Manual.

Linker Scripts

The platform linker scripts define the following symbols:

hal_vsr_table
: This defines the location of the VSR table. This is set to 0x20000000 for all startup types, and space for 98 entries is reserved.

hal_virtual_vector_table
: This defines the location of the virtual vector table used to communicate between an ROM monitor and an eCos application. This is allocated right after the VSR table, at 0x200001AC.

hal_interrupt_stack
: This defines the location of the interrupt stack. This is allocated to the top of internal SRAM, 0x20030000.

hal_startup_stack
: This defines the location of the startup stack. For all startup types it is initially allocated at the half-way point of the interrupt stack.

Diagnostic LEDs

Four LEDs are fitted on the board for diagnostic purposes and are labelled LD1 (green), LD2 (orange), LD3 (red) and LD4 (blue).

The platform HAL header file at <cyg/hal/plf_io.h> defines the following convenience function to allow LEDs LD1 and LD3 to be set:

```
extern void hal_stm32f7xx_eval_led(char c);
```

However, LEDs LD2 and LD4 cannot be set with this function as it is intended for low level control of the LEDs, but LD2 and LD4 are under MFX_GPO control. This means setting them would result in I²C transactions which in turn means the function could not be called from ISR/DSR or system-critical code.

Nevertheless, the lowest 4-bits of the argument c correspond to the LED number (with LED0/LD1 as the least significant bit). Attempting to set the bits for LD2/LD4 will have no effect.

Table 306.1. LEDs

eCos LED GPIO manifest	STM32F7 GPIO	Bit number	Board label	Colour
CYGHWR_HAL_STM32F7XX_EVAL_LED0	PF10	0	LD1	Green
CYGHWR_HAL_STM32F7XX_EVAL_LED2	PB7	2	LD3	Red

The platform HAL will automatically light LED0 when the platform initialisation is complete, however the LEDs are then free for application use.

Flash wait states

The STM32F7XX-EVAL platform HAL provides a configuration option to set the number of Flash read wait states to use: CYGNUM_HAL_CORTEXM_STM32_FLASH_WAIT_STATES. It is important to verify and if necessary update this value if changing the CPU clock (HCLK) frequency or CPU voltage. Consult the STM32 Flash programming manual (PM0081) for appropriate values for different clock speeds or voltages. The default of 5 reflects a supply voltage of 3.3V and HCLK of 168MHz.

Real-time characterization

The **tm_basic** kernel test gives statistics gathered about the real-time characterization and performance of the kernel. The sample output is shown here for information. The test was built for `SRAM` startup with optimization flag -O2, since it provides the best performance as both code and data could remain on-chip.

Example 306.1. stm32f7xx_eval Real-time characterization

```
             Startup, main thrd : stack used   352 size  1536
             Startup : Idlethread stack used    76 size  1280

eCos Kernel Timings
Notes: all times are in microseconds (.000001) unless otherwise stated

Reading the hardware clock takes 0 'ticks' overhead
... this value will be factored out of all other measurements
Clock interrupt took    3.13 microseconds (3 raw clock ticks)

Testing parameters:
   Clock samples:            32
   Threads:                  16
   Thread switches:         128
   Mutexes:                  32
   Mailboxes:                32
   Semaphores:               32
   Scheduler operations:    128
   Counters:                 32
   Flags:                    32
   Alarms:                   32
   Stack Size:             1088


                                 Confidence
     Ave     Min     Max     Var  Ave  Min  Function
  ======  ======  ======  ====== ========== ========
    2.00    1.00    3.00    0.13   87%   6% Create thread
    0.44    0.00    1.00    0.49   56%  56% Yield thread [all suspended]
    0.56    0.00    1.00    0.49   56%  43% Suspend [suspended] thread
    0.50    0.00    1.00    0.50  100%  50% Resume thread
    0.69    0.00    1.00    0.43   68%  31% Set priority
    0.25    0.00    1.00    0.38   75%  75% Get priority
    1.19    1.00    2.00    0.30   81%  81% Kill [suspended] thread
    0.44    0.00    1.00    0.49   56%  56% Yield [no other] thread
    0.69    0.00    1.00    0.43   68%  31% Resume [suspended low prio] thread
    0.50    0.00    1.00    0.50  100%  50% Resume [runnable low prio] thread
    0.56    0.00    1.00    0.49   56%  43% Suspend [runnable] thread
    0.44    0.00    1.00    0.49   56%  56% Yield [only low prio] thread
    0.44    0.00    1.00    0.49   56%  56% Suspend [runnable->not runnable]
    1.25    1.00    2.00    0.38   75%  75% Kill [runnable] thread
    1.13    1.00    2.00    0.22   87%  87% Destroy [dead] thread
    2.19    2.00    3.00    0.30   81%  81% Destroy [runnable] thread
    2.88    2.00    4.00    0.44   62%  25% Resume [high priority] thread
    0.80    0.00    2.00    0.33   78%  20% Thread switch

    0.12    0.00    1.00    0.21   88%  88% Scheduler lock
    0.36    0.00    1.00    0.46   64%  64% Scheduler unlock [0 threads]
    0.33    0.00    1.00    0.44   67%  67% Scheduler unlock [1 suspended]
    0.40    0.00    1.00    0.48   60%  60% Scheduler unlock [many suspended]
    0.33    0.00    1.00    0.44   67%  67% Scheduler unlock [many low prio]

    0.16    0.00    1.00    0.26   84%  84% Init mutex
    0.53    0.00    1.00    0.50   53%  46% Lock [unlocked] mutex
    0.50    0.00    1.00    0.50  100%  50% Unlock [locked] mutex
    0.47    0.00    1.00    0.50   53%  53% Trylock [unlocked] mutex
    0.38    0.00    1.00    0.47   62%  62% Trylock [locked] mutex
    0.22    0.00    1.00    0.34   78%  78% Destroy mutex
    3.69    3.00    4.00    0.43   68%  31% Unlock/Lock mutex

    0.28    0.00    1.00    0.40   71%  71% Create mbox
    0.16    0.00    1.00    0.26   84%  84% Peek [empty] mbox
    0.56    0.00    1.00    0.49   56%  43% Put [first] mbox
    0.16    0.00    1.00    0.26   84%  84% Peek [1 msg] mbox
    0.59    0.00    1.00    0.48   59%  40% Put [second] mbox
    0.19    0.00    1.00    0.30   81%  81% Peek [2 msgs] mbox
    0.50    0.00    1.00    0.50  100%  50% Get [first] mbox
    0.44    0.00    1.00    0.49   56%  56% Get [second] mbox
    0.47    0.00    1.00    0.50   53%  53% Tryput [first] mbox
    0.47    0.00    1.00    0.50   53%  53% Peek item [non-empty] mbox
    0.56    0.00    1.00    0.49   56%  43% Tryget [non-empty] mbox
    0.44    0.00    1.00    0.49   56%  56% Peek item [empty] mbox
    0.44    0.00    1.00    0.49   56%  56% Tryget [empty] mbox
    0.13    0.00    1.00    0.22   87%  87% Waiting to get mbox
    0.13    0.00    1.00    0.22   87%  87% Waiting to put mbox
    0.28    0.00    1.00    0.40   71%  71% Delete mbox
    2.19    2.00    3.00    0.30   81%  81% Put/Get mbox

    0.13    0.00    1.00    0.22   87%  87% Init semaphore
    0.41    0.00    1.00    0.48   59%  59% Post [0] semaphore
    0.47    0.00    1.00    0.50   53%  53% Wait [1] semaphore
    0.38    0.00    1.00    0.47   62%  62% Trywait [0] semaphore
    0.41    0.00    1.00    0.48   59%  59% Trywait [1] semaphore
    0.25    0.00    1.00    0.38   75%  75% Peek semaphore
    0.16    0.00    1.00    0.26   84%  84% Destroy semaphore
    2.00    2.00    2.00    0.00  100% 100% Post/Wait semaphore

    0.19    0.00    1.00    0.30   81%  81% Create counter
    0.06    0.00    1.00    0.12   93%  93% Get counter value
    0.13    0.00    1.00    0.22   87%  87% Set counter value
    0.53    0.00    1.00    0.50   53%  46% Tick counter
    0.16    0.00    1.00    0.26   84%  84% Delete counter

    0.16    0.00    1.00    0.26   84%  84% Init flag
    0.34    0.00    1.00    0.45   65%  65% Destroy flag
    0.38    0.00    1.00    0.47   62%  62% Mask bits in flag
    0.41    0.00    1.00    0.48   59%  59% Set bits in flag [no waiters]
    0.63    0.00    1.00    0.47   62%  37% Wait for flag [AND]
    0.66    0.00    1.00    0.45   65%  34% Wait for flag [OR]
    0.69    0.00    1.00    0.43   68%  31% Wait for flag [AND/CLR]
    0.63    0.00    1.00    0.47   62%  37% Wait for flag [OR/CLR]
    0.16    0.00    1.00    0.26   84%  84% Peek on flag

    0.38    0.00    1.00    0.47   62%  62% Create alarm
    0.66    0.00    1.00    0.45   65%  34% Initialize alarm
    0.44    0.00    1.00    0.49   56%  56% Disable alarm
    0.59    0.00    1.00    0.48   59%  40% Enable alarm
    0.44    0.00    1.00    0.49   56%  56% Delete alarm
    0.69    0.00    1.00    0.43   68%  31% Tick counter [1 alarm]
    2.31    2.00    3.00    0.43   68%  68% Tick counter [many alarms]
    0.94    0.00    1.00    0.12   93%   6% Tick & fire counter [1 alarm]
   14.03   14.00   15.00    0.06   96%  96% Tick & fire counters [>1 together]
    2.69    2.00    3.00    0.43   68%  31% Tick & fire counters [>1 separately]
    3.00    3.00    3.00    0.00  100% 100% Alarm latency [0 threads]
    2.48    2.00    3.00    0.50   52%  52% Alarm latency [2 threads]
    3.00    3.00    3.00    0.00  100% 100% Alarm latency [many threads]
    4.99    4.00    5.00    0.02   99%   0% Alarm -> thread resume latency

    0.00    0.00    0.00    0.00            Clock/interrupt latency

    1.92    1.00    2.00    0.00            Clock DSR latency

     191     160     204                    Worker thread stack used (stack size 1088)
            All done, main thrd : stack used   796 size  1536
            All done : Idlethread stack used   164 size  1280

Timing complete - 29810 ms total

PASS:<Basic timing OK>
EXIT:<done>
```

Name

Test Programs — Details

Test Programs

The STM32F7XX-EVAL platform HAL contains some test programs which allow various aspects of the board to be tested.

Manual Test

The **manual** test is not built by default. The configuration option `CYGPKG_HAL_CORTEXM_STM32_STM32F7XX_EVAL_TESTS_MANUAL` should be enabled to allow the test to be built.

This program tests various aspects of the basic platform port, e.g. flashing LEDs, checking I²C device access and that the push-button GPIO operates.

Name

BootUp Integration — Detail

BootUp

The BootUp support for the STM32F7xx-EVAL target is primarily implemented in the `stm32f7xx_eval_support.c` file. The majority of the functions provided by that source file are only included when the `CYGPKG_BOOTUP` package is being used to construct the actual BootUp ROM loader binary.

The BootUp code is designed to be very simple, and it is envisaged that once its implementation has been tested and validated, the binary will only need to be installed onto a device once. Its only purpose is to allow the safe updating and startup of the main application. If the BootUp code ever needs to be replaced then it is a “factory” operation, for example using JTAG/SWD to reprogram the on-chip flash.

This platform specific documentation should be read in conjunction with the generic BootUp package and bundle image support documentation.

The BootUp package provides a basic but fully functional implementation for the platform. This has been tested to ensure that the underlying mechanism is sound. It is envisaged that the developer will customize and further extend the platform side support to meet their specific application update requirements.

BootUp loaded applications

Applications started via the BootUp loader, since they cannot include the `CYGPKG_BOOTUP` package themselves, may need access to some related configuration state. The platform is responsible for providing such “common” information. For example, the CDL option `CYGIMP_BOOTUP_RESERVED` specifies the amount of on-chip flash set aside for BootUp. Applications can then ensure that they do not interfere with the BootUp loader if using the remaining on-chip flash for their own purposes.

Warning

Care must be taken to ensure that the target application configuration matches the BootUp configuration, since it is normally expected that the applications to be loaded will be independent of the initial BootUp build environment. This includes the fundamental on-chip flash space set aside for the BootUp ROM loader code (`CYGIMP_BOOTUP_RESERVED`) as well as, when using `CYGPKG_BUNDLE` support, where the bundle image is located (selected Non-Volatile Memory (NVM) and offset/partition information). It is expected that such values, for a particular platform instance, will be *fixed* at a suitable point during development, and definitely before products are shipped. It is the responsibility of the developer to ensure a consistent configuration between the BootUp ROM loader and any applications that may be installed/started by that BootUp code.

The platform HAL header file at `<cyg/hal/plf_io.h>` defines the following convenience function for BootUp, and applications started by BootUp, to ascertain the configured off-chip bundle/update location:

```
extern struct cyg_flash_dev *hal_stm32f7xx_eval_source_flash(cyg_flashaddr_t *pbase,cyg_flashaddr_t *plimit);
```

The function will return a pointer to the relevant flash device. The passed *`pbase`* parameter is a pointer to the value to be filled with the base address for the image (or `NULL` if the value is not needed by the caller). Similarly the *`plimit`* parameter is a pointer to the value to be filled with the limiting address for any image, or `NULL` if the address is not needed.

Primarily to avoid source duplication, the `hal_stm32f7xx_eval_source_flash()` function provides common run-time access to the settings derived from the CDL options `CYGIMP_BOOTUP_STM32F7XX_EVAL_SOURCE`, `CYGNUM_BOOTUP_STM32F7XX_EVAL_SOURCE_OFFSET` and `CYGNUM_BOOTUP_STM32F7XX_EVAL_SOURCE_LIMIT`.

Bundle based applications

When the CDL option `CYGIMP_BOOTUP_STM32F7XX_EVAL_BUNDLE` is enabled, the STM32F7xx-EVAL platform BootUp code incorporates the `CYGPKG_BUNDLE` package and support for bundle based application distribution.

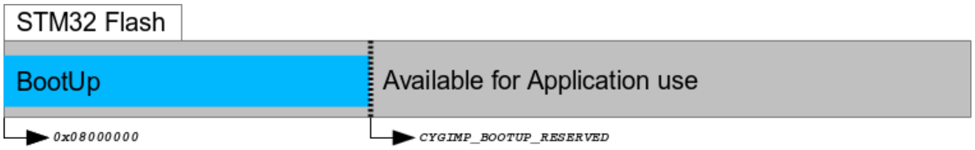
The current STM32F7xx-EVAL platform BootUp bundle support is limited (by design) to starting SDRAM based applications. i.e. `CYG_HAL_STARTUP_JTAG` startup type.

Note

The (slightly misleading) `JTAG` startup type name is used for standalone SDRAM based applications for historical reasons. The `RAM` startup name is assumed by some systems to refer to an application that relies on the presence of a debug monitor. BootUp is purely a “loader” and does *NOT* provide GDB stubs, so cannot support `CYG_HAL_STARTUP_RAM` applications.

The platform HAL and `CYGPKG_BUNDLE` package provide a common set of routines shared by both BootUp and applications. This ensures that all bundle operations are carried out in a compatible and consistent manner.

Figure 306.1. On-chip flash



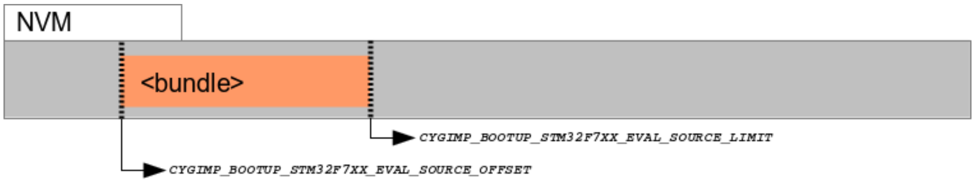
BootUp `ROMINT` application in on-chip flash

On startup the BootUp loader will use the `hal_stm32f7xx_eval_source_flash()` function to ascertain the NVM memory used to hold the source bundle image. If a valid bundle image is found, then the configured “main application” item tag as specified by the CDL option `CYGNUM_BOOTUP_STM32F7XX_EVAL_BUNDLE_TAG` is searched for within the bundle. If a valid matching tag item is found, then the data from that item is loaded into SDRAM and executed.

Note

This simple approach of using a fixed, pre-allocated, area for holding the bundle image simplifies the BootUp (and similarly any main application based update) code without the issues that would need to be considered if the bundle was stored in a filesystem. e.g. a JFFS2 filesystem on the SPI flash, with potentially slow JFFS2 mount performance, inability to ascertain how much “true” free space is available on the filesystem, programmatic support for deletion of “data” to free space for a bundle as part of an update, etc. The normal “lifetime” cycles of NOR flash (e.g. S25FL256S) should be more than sufficient for the “limited” number of in-field updates that may be undertaken on a specific board over its lifetime.

Figure 306.2. NVM bundle


NVM
<bundle>
CYGIMP_BOOTUP_STM32F7XX_EVAL_SOURCE_LIMIT
CYGIMP_BOOTUP_STM32F7XX_EVAL_SOURCE_OFFSET

Main application held in bundle stored in NVM

If a valid bundle exists and contains a valid `CYGNUM_BOOTUP_STM32F7XX_EVAL_BUNDLE_TAG` item, then the BootUp loader will always start that application. The BootUp loader itself does NOT perform any update revision based automatic update support. That support is entirely within the domain of the started application program, which is responsible for all system update decisions and processing.

The only case where BootUp will initiate a system update is when a bundle is not present or is invalid, or when an otherwise valid bundle doesn't contain a valid `CYGNUM_BOOTUP_STM32F7XX_EVAL_BUNDLE_TAG` item. In this case BootUp will attempt to install a bundle from external media. The current example implementation uses a FATFS formatted SD Card for this purpose.

Note

A beneficial side-effect of this approach is that it can help simplify the board production process. Boards only need to be pre-initialized with the stable BootUp binary, which can then be used to install the latest application firmware in NVM.

The example BootUp bundle support provided for this platform expects a single release bundle to be stored in the root directory of an inserted FATFS SD Card. The first file found that matches the `CYGDAT_BOOTUP_STM32F7XX_EVAL_BUNDLE_PREFIX` prefix is used. Any filename text after the prefix is ignored and can contain human-readable or customer specific identification information as required. For example, assuming `CYGDAT_BOOTUP_STM32F7XX_EVAL_BUNDLE_PREFIX` is configured as `“MyProductName_”`, then files named `MyProductName_1.02`, `MyProductName_1.03.B99.1234`, `MyProductName_example.bin` would all be matched.

The SD Card FATFS filesystem is mounted read-only, so any interrupted update operations (e.g. loss-of-power, reset condition) should not affect the “validity” of the FATFS filesystem held on the SD Card.

Since only a single bundle image is held in the SPI flash there is a chance for the SPI flash based bundle to be in a “corrupt” state if an update fails (power-loss, CPU reset, etc.) during an active update. However, since an update is only manually started when a validated image is available on an SDcard, if the update is interrupted the same SDcard (and field-engineer/operator) should be available to re-apply the update on the system restart. This avoids the (normal) “robustness” requirement of providing two application images to be held in the SPI flash to ensure “safe” updates.

To reiterate, the BootUp code will *ONLY* perform an update from an SD Card to the NVM when there is *NO* valid main application bundle/item pair (missing or corrupted). For in-field upgrades any update process will be instigated under the control of the BootUp started “main application”. For example, the application could use the `CYGPKG_BUNDLE` API to validate a bundle image from whatever source it has access to, and then to update the relevant NVM image itself. If the update is provided on an SD Card then (after ensuring the SD Card does contain a valid bundle image) the main application just needs to invalidate the current NVM bundle image, and then force a CPU reset to have BootUp detect the now invalid main application and apply the update. It is up to the developer to decide the best approach for their particular needs in how point-revision updates are installed, and is beyond the scope of this documentation.

The bundle implementation currently limits the number of automatic update attempts when a missing/invalid bundle is detected. This is a deliberate choice to avoid continually failing attempts that could eventually wear out a flash device. The platform `hal_stm32f7xx_eval_badapp()` function implementation, when bundle support is configured, will reset the system to allow another restart attempt if the “Wakeup/Tamper” (`CYGHWR_HAL_STM32F7XX_EVAL_BUTTON_USER`) button is pressed for more than one second. This can be used to manually force another “automatic” update attempt to be started.

When BootUp has installed a bundle to the SPI flash, the last 4-bytes of the SPI flash area (partition) set aside for the bundle will be erased to 0xFFFFFFFF. This location and value can be used by the customer main application to ascertain that an install/update has just been performed (since sector erase will only occur as part of a bundle install/update). It is the responsibility of the main application to update this single location (clearing at minimum 1-bit) if it wants to track “post update” state. This can by used by the main application to acknowledge the update, and can ensure, for example, that any (potentially slow) “post update” main application specific functionality is not performed on every normal startup. For example, the main application may need to check and update the software components of attached daughter-boards from the bundle, and can use this mechanism to ensure it is only performed *once* after an update. This simple (erased flash) mechanism avoids complicated support for passing non-volatile “log” information between the seperate BootUp and main application worlds.

Note

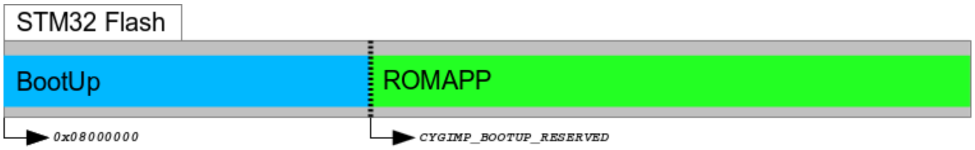
The main application should *NOT* use bit 31 of this field (treating the 32-bit value as being stored in little-endian format) since it is reserved as a flag for the BootUp loader update processing. The main application should *ALWAYS* leave bit 31 set.

On-Chip ROMAPP applications

If the CDL option `CYGIMP_BOOTUP_STM32F7XX_EVAL_BUNDLE` is not enabled, then BootUp provides an alternative mechanism that supports the safe update of on-chip flash resident (`CYG_HAL_STARTUP_ROMAPP`) applications.

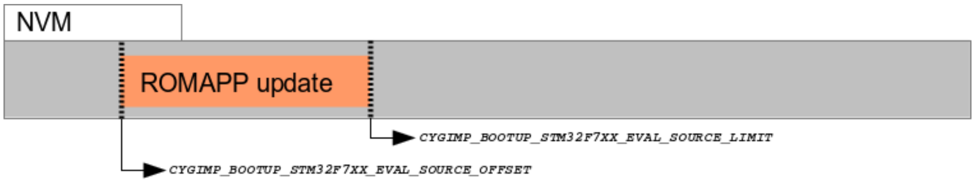
Updates using this mechanism are initiated and directed solely by the application itself. The application is responsible for locating, acquiring and verifying a new update, and placing it into NVM storage. If BootUp detects a verified update in NVM, it installs the update into the on-chip flash, overwriting and replacing the existing application. The updated application is then executed.

Figure 306.3. BootUp and Application



BootUp `ROMINT` loader and `ROMAPP` main application held in on-chip flash

Figure 306.4. Application Update image


NVM
ROMAPP update
CYGIMP_BOOTUP_STM32F7XX_EVAL_SOURCE_LIMIT
CYGIMP_BOOTUP_STM32F7XX_EVAL_SOURCE_OFFSET

Update for `ROMAPP` main application held in NVM

The example implementation uses a simple scheme that checks a fixed-format contiguous structure near the start of the binary application image file. Other than the fields used to identify the structure, the BootUp code does not interpret the hal_stm32f7xx_eval_bootup_structure_t structure `identity` field.

Depending on how the alternative (pending update) application is downloaded and installed in the NVM, it may be more relevant to have the `tail` marker at the very end of the binary image. The developer may wish to update the build/release process so that the actual binary length is held in the application description structure, since that could avoid the overhead of unnecessary flash reads and writes when processing updates. Similarly, instead of a simple binary number being used to differentiate application images, the choice may be made to use the 64-bit UTC timestamp the application was created, or a human-readable string as the unique identification for a release. It is the responsibility of the build/release engineer to ensure individual releases are uniquely identifiable.

It is *critical* that the main application, when storing a pending update, stores the `tail` marker as the last bytes written. It is the responsibility of the main application to verify the data written, prior to placing the `tail` marker. This ensures that a partial image is not treated as a valid update. For example the sequence undertaken by the main application would be:

Table 306.2. Pending update sequence

Operation	Details
Invalidate “previous” alternative image	At a minimum ensure an invalid signature `tail` marker is written. Erasing the flash is normally required anyway, and would invalidate any previous image.
Receive update application image and write to alternative image location	NOT writing the `tail` marker. The code that stores the application should leave a “hole” where the `tail` marker resides to ensure a partial image is not incorrectly treated as valid

Operation	Details
Verify downloaded contents	e.g. CRC or binary comparison. Normally this would be done as individual application chunks are downloaded and written to the alternative storage
Write `tail` marker	This is the very last operation after validating that the alternative image has been stored correctly. If an error has occured during the download then not-writing the `tail` ensures that the BootUp loader will not interpret the data written as a pending update
Force system RESET to start update	e.g. using the `HAL_PLATFORM_RESET` macro

The BootUp loader code will only READ from the alternative image location. This ensures that if an in-progress update is interrupted (e.g. power-loss) then when the system restarts the BootUp code will restart the application update as required.

If the BootUp platform implementation for validating the alternative image is extended to include a CRC, or similar “slow” processing, it may be worth considering whether the main application on startup will always invalidate the `tail` marker after an update to avoid subsequent system resets having to re-validate the alternative image prior to discovering that it is the same as the current main application.

Note

We cannot have the SIGNATURE support purely conditional on the BOOTUP support; since non-BOOTUP applications need to be built leaving the space. For the moment this is only enforced for ROMAPP applications, since that is all that the simple (non-BUNDLE) BootUp update support implements.

Building BootUp

The `ROMINT` startup type is chosen for BootUp so that the loader uses the on-chip SRAM for its workspace, to avoid the overhead of managing off-chip memory where the target application will be loaded.

Example eCos configuration templates for BootUp are provided in the `misc` directory of the release. The `hostboot_ROMINT.ecm` configuration file can be used to construct a bundle based BootUp loader and `bootup_ROMINT.ecm` for the simpler on-chip ROMAPP update BootUp loader.

Building a BootUp ROM image is most conveniently done at the command line. For the `stm32f746g_eval2`, the steps needed to rebuild the bundle based ROMINT version of BootUp on linux are:

```
$ mkdir hostboot_romint
$ cd hostboot_romint
$ ecosconfig new stm32f746g_eval2 minimal
[ … ecosconfig output elided … ]
$ ecosconfig import $ECOS_REPOSITORY/hal/cortexm/stm32/stm32f7xx_eval/current/misc/hostboot_ROMINT.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

The steps needed to rebuild the bundle based ROMINT version of BootUp on Windows within the Shell Environment are

```
C:\Users\demo> mkdir hostboot_romint
C:\Users\demo> cd hostboot_romint
C:\Users\demo\hostboot_romint> ecosconfig new stm32f746g_eval2 minimal
[ … ecosconfig output elided … ]
C:\Users\demo\hostboot_romint> ecosconfig import \
    %ECOS_REPOSITORY%/hal/cortexm/stm32/stm32f7xx_eval/current/misc/hostboot_ROMINT.ecm
C:\Users\demo\hostboot_romint> ecosconfig resolve
C:\Users\demo\hostboot_romint> ecosconfig tree
C:\Users\demo\hostboot_romint> make
```

The resulting `install/bin/bootup.bin` binary can then be programmed into the on-chip flash from address `0x08000000`.

It is expected that the BootUp binary is installed onto the STM32F427 on-chip flash either via JTAG/SWD or by utilising the on-chip `BootROM` USB based DFU process. This is a factory or in-field process requiring specific equipment/host-software.

Once BootUp is installed it is not normally expected to require updating. Its purpose is to bootstrap the main application, and provide a standard mechanism for installing the main application. The update mechanism does *NOT* provide a method for updating the BootUp loader itself. If in-field updates of the BootUp binary are necessary, this could be achieved via the STM32 on-chip `BootROM` USB based DFU process.

BootUp Test Programs

The `tests/bundle_example.c` source implements a simple example of an application that utilises the `CYGPKG_BUNDLE` package. Its code could serve as a useful starting point when adding bundle update support to your own application.

Normally a standard `CYG_HAL_STARTUP_JTAG` configured build of the **bundle_example** would be used. If the **bootup** application is used to bootstrap the processor and is built as described in Building BootUp, the eCos Configuration against which **bundle_example** is linked must also include “CRC Support” (`CYGPKG_CRC`), “Zlib compress/decompress support” (`CYGPKG_COMPRESS_ZLIB`), “File IO” (`CYGPKG_IO_FILEIO`) and “Generic FLASH memory support” (`CYGPKG_IO_FLASH`).

As well as ensuring the required packages are present in the configuration, the CDL option `CYGBLD_HAL_CORTEXM_STM32F7XX_EVAL_TESTS_MANUAL`, and its sub-option `CYGBLD_HAL_CORTEXM_STM32F7XX_EVAL_TEST_BOOTUP`, should be enabled to allow **bundle_example** to be built.

Note

If required, the release provides an example “default” template for the `stm32f746g_eval2` platform in the `misc/bundle_example.ecm` file which includes all the necessary packages and defines the necessary options. This can be imported to a configuration (using either the command-line **ecosconfig import**, or the GUI **configtool** “File->Import...” support).

Once built, a raw binary copy of the application would be extracted for adding to a bundle using the **arm-eabi-objcopy** command. For example:

```
$ arm-eabi-objcopy -O binary bundle_example bundle_example.bin
```

The resulting binary could then be added to a bundle image using the host-based **bundle** tool:

```
$ bundle MyProductName_example.bin create add 0x0001:bundle_example.bin:C:md5
```

Refer to the bundle host tool section for detailed information on the bundle host tool.

In this example the bundle image would then be placed onto a suitably formatted FATFS SD Card. The bundle would then be installed into the NVM either by a pre-existing main application, or by the BootUp loader if a valid bundle is not currently installed in the NVM.

Note

When placing the bundle image onto a FATFS SD Card only the `CYGDAT_BOOTUP_STM32F7XX_EVAL_BUNDLE_PREFIX` configured filename prefix is checked by the BootUp code. It is expected that only a single, suitably prefixed, bundle is present on an SD Card used for update/installation.

The provided **bundle_invalidate** test can be used during BootUp bundle testing to explicitly invalidate any NVM held bundle. This can be done to check the BootUp operation when no valid bundle containing a main application is available, e.g. to test installation from FATFS SD Card.

Chapter 307. STM32L476-DISCO Platform HAL

Name

`CYGPKG_HAL_CORTEXM_STM32_STM32L476_DISCO` — eCos Support for the STM32L476-DISCO Board

Description

This documentation describes the platform-specific elements of the ST STM32L476 board support within eCos. It should be read in conjunction with the STM32 variant HAL section, which covers the common functionality shared by all STM32 variants, including eCos HAL features and on-chip device support.

The board is equipped with an on-board ST-LINK/V2 hardware debugger interface (via the CN1 “USB ST-LINK” connector), which is typically used for eCos application development.

Supported Hardware

The STM32L476VG has two main on-chip memory regions. The device has a SRAM region of 96KiB present at 0x20000000, and a 1MiB FLASH region present at 0x08000000 (which is aliased to 0x00000000 during normal execution). A 128Mbit N25Q SPI flash device it available through the QSPI controller.

The STM32 variant HAL includes support for the six on-chip serial devices. These consist of three USARTs, two UARTs and a LPUART. These are all supported by a common driver and are documented in the variant HAL. However, the STM32L476-DISCO motherboard has no direct UART connectors. It is possible to use USART1 on PIO pins PB6 and PB7 which are connected to pins 15 and 16 of the P1 header.

The STM32 variant HAL also includes support for the I²C buses. A number of I²C devices are present on the board, but none are currently supported by eCosPro. Connections for external I²C devices on bus 1 can be made either via the CN2 I²C extension connector or pins 15 and 16 of P1. Note that these are the same pins as used by USART1, so these two devices cannot be used simultaneously.

Similarly the STM32 variant HAL includes support for the SPI buses. There are a number of SPI devices on the board, but none are currently supported by eCosPro. External access to SPI bus 1 can be had via P2 header pins 15 to 18 which are connected to PIO pins PE12-15. However, these pins are also used for the QSPI memory, so it is not possible to use SPI and QSPI simultaneously.

Device drivers are also provided for the STM32 on-chip, ADC devices. There are no usable analog inputs on the board, only the internal sources are generally available. Only a limited number of PIO pins that connect to the ADCs are accessible. These are mainly limited to PA0-3 and PA5 on the P1 header, which are shared with the joystick switch. If the LCD is removed then further pins are accessible via the socket.

Additionally, support is provided for the on-chip watchdog, RTC (wallclock) and a Flash driver exists to permit management of the STM32's on-chip Flash.

The STM32L4 processor and the STM32L476-DISCO board provide a wide variety of peripherals, but unless support is specifically indicated, it should be assumed that it is not included.

Tools

The board port is intended to work with GNU tools configured for an **arm-eabi** target. The original port was done using **arm-eabi-gcc** version 4.7.3j, **arm-eabi-gdb** version 7.8.2, and **binutils** version 2.23.2.

Name

Setup — Preparing the STM32L476-DISCO Board for eCos Development

Overview

Typically, since the STM32L476-DISCO motherboard has a built-in ST-LINK/V2-1 interface providing hardware debug support, eCos applications are loaded and run via the debugger **arm-eabi-gdb** or via the Eclipse IDE. The debugger then communicates with the “GDB server” provided by the relevant host ST-LINK/V2-1 support tool being used (e.g. OpenOCD).

Normally for release applications the `ROM` startup type would be used, with the application programmed into the on-chip flash for execution when the board boots. It is still possible to use the hardware debugging support to debug such flash-based ROM applications, and this may be the desired approach if the application is too large for execution from on-chip SRAM, or where all of the SRAM is required for application run-time use.

If off-chip Non-Volatile Memory (NVM) is used to hold the main application then the board can boot from the internal flash using a suitable boot loader. For example, the eCosPro BootUp ROM loader, where the BootUp code can start the main application (after an optional update sequence).

If required, it is still possible to program a GDB stub or RedBoot `ROM` image into on-chip Flash and download and debug via a serial connection (using USART3 on J8). In that case, eCos applications are configured for `RAM` startup and then downloaded and run on the board via the debugger **arm-eabi-gdb**, or via the Eclipse IDE as appropriate. However, the space available to applications with this approach is so limited as to make it essentially impractical.

Preparing ST-LINK/V2-1 interface

Use of the ST-LINK with STM32L4 microcontrollers and OpenOCD currently requires that the ST-LINK/V2-1 firmware is version `V2.J20.XXX` and not a more recent version. The firmware for the ST-LINK/V2-1 interface can be checked, and updated if needed, using a tool available from STMicroelectronics. The firmware version is also reported when the **openocd** command is executed (using a suitable configuration file). For example, the following OpenOCD output reports `JTAG v20`:

```
Info : STLINK v2 JTAG v20 API v2 SWIM v11 VID 0x0483 PID 0x374B
```

Future revisions of ST-LINK firmware may restore compatibility, but would require testing to confirm this. The user should refer to the ST “ST-LINK/V2-1 firmware upgrade” (RN0093) Release Note, which provides details on upgrading the ST-Link firmware on Linux, Mac OS X and Windows hosts.

Programming ROM images

Since the STM32L476-DISCO board has a built-in ST-LINK/V2-1 SWD interface, the USB host connection (CN1) and suitable host software (e.g. The OpenOCD package **openocd** tool) can be used to program the flash.

The **openocd** GDB server can directly program flash based applications from the GDB **load** command.

Note

The **openocd** command being used should have been configured and built to support the ST-LINK/V2-1 interface. This is achieved by specifying the **--enable-stlink** when configuring the OpenOCD build. Additional information on running **openocd** may be found in the OpenOCD notes.

For example, assuming that **openocd** is running on the same host as GDB, and is connected to the target board the following will program the “bootup.elf” application into the on-chip flash:

```
$ arm-eabi-gdb install/bin/bootup.elf
GNU gdb (eCosCentric GNU tools 4.7.3j) 7.8.2
```

```
[ … GDB output elided … ]
(gdb) target remote localhost:3333
hal_reset_vsr () at path/hal_misc.c:171
(gdb) load
Loading section .rom_vectors, size 0x14 lma 0x8000000
Loading section .text, size 0x3adc lma 0x8000018
Loading section .rodata, size 0x6c0 lma 0x8003af8
Loading section .data, size 0x6dc lma 0x80041b8
Start address 0x8000018, load size 18572
Transfer rate: 14 KB/sec, 4643 bytes/write.
(gdb)
```

Alternatively, the **openocd** telnet interface can be used to manually program the flash. By default the **openocd** session provides a comand-line via port `4444`. Consult the OpenOCD documentation for more details if a non-default **openocd** configuration is being used.

With a **telnet** connection established to the **openocd** any binary data can easily be written to the on-chip flash. e.g.

```
$ telnet localhost 4444
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Open On-Chip Debugger
> flash write_image test.bin 0x08000000
wrote 32518 bytes from file test.bin in 1.073942s (29.569 KiB/s)
```

To create a binary for flash programming the **arm-eabi-objcopy** command is used. This converts the, ELF format, linked application into a raw binary. For example:

```
$ arm-eabi-objcopy -O binary programname programname.bin
```

Name

Configuration — Platform-specific Configuration Options

Overview

The STM32L476-DISCO board platform HAL package `CYGPKG_HAL_CORTEXM_STM32_STM32L476_DISCO` is loaded automatically when eCos is configured for the `stm32l476_disco` target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The STM32L476-DISCO board platform HAL package supports four separate startup types:

ROM

> This startup type can be used for finished applications which will be programmed into internal flash at location 0x08000000. Data and BSS will be put into internal SRAM starting from 0x20000288. Internal SRAM below this address is reserved for vector tables. The application will be self-contained with no dependencies on services provided by other software. The program expects to boot from reset with ROM mapped at location zero. It will then transfer control to the 0x08000000 region. eCos startup code will perform all necessary hardware initialization.

ROMAPP

> This startup type can be used for finished applications which will be programmed into internal flash at location 0x08008000. Data and BSS will be put into internal SRAM starting from 0x20000288. Internal SRAM below this address is reserved for vector tables. The application will be self-contained with no dependencies on services provided by other software. The program expects to boot from reset with ROM mapped at location zero. It will then transfer control to the 0x08000000 region. eCos startup code will perform all necessary hardware initialization.
>
> This startup type is identical to the ROM startup with the exception of the flash base address. It is used for applications that can be started or updated by BootUp.

SRAM

> This startup type can be used for finished applications which will be loaded into internal SRAM via a JTAG interfae. The application will be self-contained with no dependencies on services provided by other software. The program expects to be loaded from 0x20000288 and entered at that address. eCos startup code will perform all necessary hardware initialization.

RAM

> When the board has RedBoot (or a GDB stub ROM) programmed into internal Flash at location 0x08000000 then the arm-eabi-gdb debugger can communicate with a suitably configured UART connection to load and debug applications. An application is loaded into memory from 0x20001000. It is assumed that the hardware has already been initialized by RedBoot. By default the application will *not* be stand-alone, and will use the eCos virtual vectors mechanism to obtain services from RedBoot, including diagnostic output.

Warning

RedBoot can have an adverse affect on the real-time performance of applications.

RedBoot and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building RedBoot.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a `RAM` startup, disabled otherwise. It can be manually disabled for a `RAM` startup, making the application self-contained, as a testing step before switching to `ROM` startup.

Note

Though, as previously discussed, since the option of hardware debugging is available as standard on the STM32L476-DISCO platform, and space in the SRAM is limited, it is unlikely that the `RAM` startup type would be used for development.

SPI Driver

An SPI bus driver is available for the STM32 in the package “ST STM32 SPI driver” (`CYGPKG_DEVS_SPI_CORTEXM_STM32`).

No SPI devices are instantiated for this platform by default.

Consult the generic SPI driver API documentation in the eCosPro Reference Manual for further details on SPI support in eCosPro, along with the configuration options in the STM32 SPI device driver.

I^{2}C Driver

The STM32 variant HAL provides the main I^{2}C hardware driver itself, configured at `CYGPKG_HAL_STM32_I2C`. However, the platform I^{2}C support can also be configured separately at `CYGPKG_HAL_CORTEXM_STM32_STM32L476_DISCO_I2C`. This enables I^{2}C bus 1. A CAT2C128 EEPROM is available on this bus and is instantiated with the name `hal_stm32l476_disco_eeprom`. The instantiated device is available for applications via `<cyg/io/i2c.h>`.

ADC Driver

The STM32 processor variant HAL provides an ADC driver. The STM32L476-DISCO platform HAL enables the support for the devices ADC1, ADC2 and ADC3 and for configuration of the respective ADC device input channels.

Consult the generic ADC driver API documentation in the eCosPro Reference Manual for further details on ADC support in eCosPro, along with the configuration options in the STM32 ADC device driver.

Flash Driver

The STM32's on-chip Flash may be programmed and managed using the Flash driver located in the “STM32 Flash memory support” (`CYGPKG_DEVS_FLASH_STM32`) package. This driver is enabled automatically if the generic “Flash device drivers” (`CYGPKG_IO_FLASH`) package is included in the eCos configuration. The driver will configure itself automatically for the size and parameters of the specific STM32 variant present on the STM32L476-DISCO board.

A number of aspects of Flash driver behaviour can be configured within that driver, such as program/erase parallelism and program burst size. Consult the driver for more details.

QSPI Flash Driver

When QSPI NOR flash support is enabled in the configuration with `CYGHWR_HAL_CORTEXM_STM32_FLASH_QSPI`, then the `m25pxx_flash_device` device is exported and can be accessed via the standard flash API. The device is given a logical base address to match its physical base address of `0x90000000` (corresponding to FMC bank 4) when it is memory mapped (if `CYGFUN_DEVS_FLASH_QSPI_CORTEXM_STM32_MEMMAPPED` is enabled in the QSPI driver, which is the default). Even if memory mapping is disabled, using the eCos Flash API will still allow the device to be read/written at that logical base address.

Name

SWD support — Usage

Use of JTAG/SWD for debugging

JTAG/SWD can be used to single-step and debug loaded applications, or even applications resident in ROM.

Debugging of ROM applications is only possible if using hardware breakpoints. The Cortex-M7 core of the STM32L476 only supports eight such hardware breakpoints, so they may need to be used sparingly. If using a GDB front-end such as Eclipse, check it has not set unnecessary extra breakpoints such as at `main()`. Some JTAG/SWD devices give the option of whether to set hardware or software breakpoints by default. Be sure to configure your device appropriately.

When debugging via JTAG, you are likely to need to disable the default HAL idle thread action, otherwise there may be issues where the target fails to halt and the debugging session is unreliable. More details can be found in the Cortex-M architectural HAL. This should *not* be necessary when using a SWD-based hardware debugger such as the on-board ST-LINK/V2-1 interface.

The default eCos configuration does not enable the use of ITM stimulus ports for the output of HAL diagnostics or Kernel instrumentation. The architecture HAL package `CYGPKG_HAL_CORTEXM` provides options to enable such use.

For HAL diagnostic (e.g. `diag_printf()`) output the architecture CDL option `CYGHWR_HAL_CORTEXM_DIAGNOSTICS_INTERFACE` should be updated to select `ITM` as the output destination. Once the `ITM` option has been configured the option `CYGHWR_HAL_CORTEXM_DIAGNOSTICS_ITM_PORT` allows the actual stimulus port used for the diagnostics to be selected.

When the Kernel instrumentation option `CYGPKG_KERNEL_INSTRUMENT` is enabled then the `CYGHWR_HAL_CORTEXM_ITM_INSTRUMENTATION` option can be enabled to direct instrumentation record output via an ITM stimulus port, rather than into a local memory buffer. The stimulus port used can be configured via the `CYGHWR_HAL_CORTEXM_ITM_INSTRUMENTATION_PORT_BASE` option.

However, when using the STM32L476-DISCO board via the ST-LINK/V2-1 interface then it is recommended that the `gdb_hwdebug_fileio` approach is used to provide access to diagnostics via the GDB debug connection. When ITM support is used it has been observed that the ST-LINK/V2-1 firmware can drop data, leading to the possibility of confusing output. However, with care the ITM system can be tuned to provide diagnostic and instrumentation via the host SWD debugger.

Using the ST-LINK/V2-1 connection allows for a single cable to provide power, hardware debug support and diagnostic output.

OpenOCD notes

The OpenOCD debugger can be configured to support the on-board ST-LINK/V2-1 interface available via the USB CN14 connection. When configuring the **openocd** tool build, the **configure** script can be given the option `--enable-stlink` to provide for ST-LINK support.

An example OpenOCD configuration file `openocd.stm32l476_disco.cfg` is provided within the eCos platform HAL package in the source repository. This will be in the directory `packages/hal/cortexm/stm32/stm32l476_disco/current/misc` relative to the root of your eCos installation.

This configuration file can be used with OpenOCD on the host as follows:

```
$ openocd -f openocd.stm32l476_disco.cfg
Open On-Chip Debugger 0.10.0-dev-00371-g81631e4 (2016-09-08-17:23)
Licensed under GNU GPL v2
For bug reports, read
  http://openocd.org/doc/doxygen/bugs.html
Info : The selected transport took over low-level target control. The results might differ compared to plain JTAG/SWD
adapter speed: 480 kHz
adapter_nsrst_delay: 100
srst_only separate srst_nogate srst_open_drain connect_assert_srst
Info : clock speed 480 kHz
Info : STLINK v2 JTAG v20 API v2 SWIM v4 VID 0x0483 PID 0x374B
Info : using stlink api v2
Info : Target voltage: 3.227369
Info : stm32l4x.cpu: hardware has 6 breakpoints, 4 watchpoints
```

By default **openocd** provides a console on port `4444`, and this can be used to interact with the target system. This console interface can be used to perform debugging, program the flash, etc.

Normally **arm-eabi-gdb** is used to connect to the default GDB server port `3333` for debugging. For example:

```
(gdb) target remote localhost:3333
Remote debugging using localhost:3333
0x00000000 in ?? ()
(gdb)
```

The application can then be loaded and executed under GDB as normal. If you are using Eclipse then, if required, you can define a “preload” gdb macro to emit any necessary commands to OpenOCD. See the “Hardware Assisted Debugging” section of the “Eclipse/CDT for eCos application development” document's “Debugging eCos applications” chapter.

If the HAL diagnostics are configured to use ITM, and stimulus port 31 is configured as the HAL diagnostic destination, then the configuration example above will direct OpenOCD to direct ITM output (and also DWT and ETM) to a file named `tpiu.out` in the current directory of the shell which was used to run the **openocd** command. A more specific filename can be used by adjusting the OpenOCD configuration file.

To extract the ITM output, the Cortex-M architecture HAL package provides a helper program **parseitm** in the directory `packages/hal/cortexm/arch/current/host` relative to the root of your eCos installation. It can be compiled simply with:

```
$ gcc -o parseitm parseitm.c
```

You simply run it with the desired ITM stimulus port and name of the file containing the ITM output, for example:

```
$ parseitm -p 31 -f itm.out
```

It will then echo all ITM stimulus for that port, continuing to read from the file until interrupted with Ctrl-C. Note that limited buffer space in debug hardware such as the ST-LINK can result in occasionally missed ITM data. eCosPro provides a workaround of throttling data within the `CYGHWR_HAL_CORTEXM_ITM_DIAGNOSTICS_THROTTLE` CDL configuration component in order to reduce or avoid lost ITM data. For further details, see the note in OpenOCD ITM support.

Similarly, if the eCos application is built with Kernel instrumentation enabled and configured for ITM output, then the default stimulus port 24 output can be captured. For example, assuming the application **cminfo** is the ELF file built from an eCos configuration with ITM instrumentation enabled, and is loaded and run via **openocd**, then we could run **parseitm** to capture instrumentation whilst the program executes, and then view the gathered data using the example **instdump** tool provided in the Kernel package.

```
$ parseitm -p 24 -f tpiu.out > inst.bin
^C
$ instdump -r inst.bin cminfo
Threads:
 threadid 1 threadobj 200045D0 "idle_thread"

     0:[THREAD:CREATE][THREAD 4095][TSHAL 4][TSTICK 0][ARG1:200045D0] { ts 4 microseconds }
     1:[SCHED:LOCK][THREAD 4095][TSHAL 45][TSTICK 0][ARG1:00000002] { ts 45 microseconds }
     2:[SCHED:UNLOCK][THREAD 4095][TSHAL 195][TSTICK 0][ARG1:00000002] { ts 195 microseconds }
     3:[SCHED:LOCK][THREAD 4095][TSHAL 346][TSTICK 0][ARG1:00000002] { ts 346 microseconds }
     4:[SCHED:UNLOCK][THREAD 4095][TSHAL 495][TSTICK 0][ARG1:00000002] { ts 495 microseconds }
     5:[THREAD:RESUME][THREAD 1][TSHAL 647][TSTICK 0][ARG1:200045D0][ARG2:200045D0] { ts 647 microseconds }
     6:[SCHED:LOCK][THREAD 1][TSHAL 795][TSTICK 0][ARG1:00000002] { ts 795 microseconds }
     7:[MLQ:ADD][THREAD 1][TSHAL 945][TSTICK 0][ARG1:200045D0][ARG2:0000001F] { ts 945 microseconds }
     8:[SCHED:UNLOCK][THREAD 1][TSHAL 1096][TSTICK 0][ARG1:00000002] { ts 1096 microseconds }
     9:[INTR:ATTACH][THREAD 1][TSHAL 0][TSTICK 0][ARG1:00000000] { ts 10000 microseconds }
    10:[INTR:UNMASK][THREAD 1][TSHAL 149][TSTICK 0][ARG1:00000000] { ts 10149 microseconds }
    11:[INTR:ATTACH][THREAD 1][TSHAL 305][TSTICK 0][ARG1:00000054] { ts 10305 microseconds }
    12:[INTR:UNMASK][THREAD 1][TSHAL 449][TSTICK 0][ARG1:00000054] { ts 10449 microseconds }
```

Configuration of JTAG/SWD applications

JTAG/SWD applications can be loaded directly into SRAM or flash without requiring a ROM monitor. Loading can be done directly through the JTAG/SWD device, or through GDB where supported by the JTAG/SWD device.

In order to configure the application to support these modes, it is recommended to use the `SRAM`, `ROM` or `ROMAPP` startup types which will implicitly cause two important settings to change. Firstly, `CYGSEM_HAL_USE_ROM_MONITOR` must be disabled. Secondly the `CYGDBG_HAL_DIAG_TO_DEBUG_CHAN` option should be enabled in order to prevent HAL diagnostic output being encoded into GDB ($O) packets. These configuration changes could be made by hand, but use of the aformentioned startup types will just work.

With these changes, any diagnostic output will appear out of the configured diagnostic channel. An eCosCentric extension allows diagnostic output to appear in GDB. For this feature to work, you must enable the configuration option `CYGSEM_HAL_DIAG_TO_GDBFILEIO_CHAN` in the common HAL package. If you are using the graphical configuration tool then you should then accept any suggested solutions to the subsequent configuration conflicts. Older eCos releases also required the gdb "set hwdebug on" command to be used to enable GDB or Eclipse console output, but this is no longer required with the latest tools.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the STM32L476-DISCO board hardware, and should be read in conjunction with that specification. The STM32L476-DISCO platform HAL package complements the Cortex-M architectural HAL and the STM32 variant HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset the HAL will initialize or reinitialize many of the on-chip peripherals. There is an exception for `RAM` startup applications which depend on a ROM monitor for certain services.

For `ROM`, `ROMAPP` and `SRAM` startup types the HAL will perform additional initialization, programming the various internal registers including the PLL, peripheral clocks and GPIO pins. The details of the early hardware startup may be found in the `src/stm32l476_disco_misc.c` in both `hal_system_init` and `hal_platform_init`.

Memory Map

The platform HAL package provides the memory layout information needed to generate the linker script. For all the STARTUP variations the top `CYGNUM_HAL_COMMON_INTERRUPTS_STACK_SIZE` bytes of the on-chip SRAM are reserved for the interrupt stack. The remainder of the internal SRAM is available for use by applications. The key memory locations are as follows:

Internal SRAM
: This is located at address `0x20000000` of the memory space, and is 192KiB in size. The eCos VSR table occupies the bottom 392 bytes of memory, with the virtual vector table starting at `0x20000188` and extending to `0x20000288`.

Internal FLASH
: This is located at address `0x08000000` of the memory space and will be mapped to `0x00000000` at reset. This region is 1024KiB in size. `ROM` and `ROMAPP` applications are by default configured to run from this memory.

On-Chip Peripherals
: These are accessible at locations `0x40000000` and `0xE0000000` upwards. Descriptions of the contents can be found in the STM32 User Manual.

Linker Scripts

The platform linker scripts define the following symbols:

hal_vsr_table
: This defines the location of the VSR table. This is set to `0x20000000` for all startup types, and space for 98 entries is reserved.

hal_virtual_vector_table
: This defines the location of the virtual vector table used to communicate between an ROM monitor and an eCos application. This is allocated right after the VSR table, at `0x20000188`.

hal_interrupt_stack
: This defines the location of the interrupt stack. This is allocated to the top of internal SRAM, from `0x20018000` down.

hal_startup_stack
: This defines the location of the startup stack. For all startup types it is initially allocated at the half-way point of the interrupt stack.

Flash wait states

The STM32L476-DISCO platform HAL provides a configuration option to set the number of Flash read wait states to use: `CYGNUM_HAL_CORTEXM_STM32_FLASH_WAIT_STATES`. It is important to verify and if necessary update this value if changing the CPU clock (HCLK) frequency or CPU voltage. Consult the relevant STM32 datasheets and programming manuals for the STM32L476 parts for appropriate values for different clock speeds or voltages. The default of 4 reflects a supply voltage in Vcore range 1 and HCLK of 80MHz.

Real-time characterization

The **tm_basic** kernel test gives statistics gathered about the real-time characterization and performance of the kernel. The sample output is shown here for information. The test was built for `SRAM` startup with optimization flag -O2, since it provides the best performance as both code and data could remain on-chip.

Example 307.1. stm32l476_disco Real-time characterization

```
             Startup, main thrd : stack used   360 size  1536
             Startup : Idlethread stack used    76 size  1280

eCos Kernel Timings
Notes: all times are in microseconds (.000001) unless otherwise stated

Reading the hardware clock takes 0 'ticks' overhead
... this value will be factored out of all other measurements
Clock interrupt took   13.03 microseconds (13 raw clock ticks)

Testing parameters:
   Clock samples:            32
   Threads:                  16
   Thread switches:         128
   Mutexes:                  32
   Mailboxes:                32
   Semaphores:               32
   Scheduler operations:    128
   Counters:                 32
   Flags:                    32
   Alarms:                   32
   Stack Size:             1088


                                 Confidence
     Ave     Min     Max     Var  Ave  Min  Function
  ======  ======  ======  ====== ========== ========
INFO:<Ctrl-C disabled until test completion>
    7.13    6.00    8.00    0.66   37%  25% Create thread
    2.00    2.00    2.00    0.00  100% 100% Yield thread [all suspended]
    1.88    1.00    2.00    0.22   87%  12% Suspend [suspended] thread
    1.69    1.00    2.00    0.43   68%  31% Resume thread
    2.50    2.00    4.00    0.56   93%  56% Set priority
    0.38    0.00    1.00    0.47   62%  62% Get priority
    4.63    4.00    7.00    0.63   93%  50% Kill [suspended] thread
    1.63    1.00    2.00    0.47   62%  37% Yield [no other] thread
    2.56    2.00    3.00    0.49   56%  43% Resume [suspended low prio] thread
    2.06    2.00    3.00    0.12   93%  93% Resume [runnable low prio] thread
    2.19    2.00    3.00    0.30   81%  81% Suspend [runnable] thread
    2.00    2.00    2.00    0.00  100% 100% Yield [only low prio] thread
    1.50    1.00    2.00    0.50  100%  50% Suspend [runnable->not runnable]
    4.50    4.00    6.00    0.56   93%  56% Kill [runnable] thread
    3.88    3.00    5.00    0.33   75%  18% Destroy [dead] thread
    8.31    8.00   10.00    0.47   75%  75% Destroy [runnable] thread
    9.88    9.00   12.00    0.55   56%  31% Resume [high priority] thread
    3.02    3.00    5.00    0.03   99%  99% Thread switch
    0.36    0.00    1.00    0.46   64%  64% Scheduler lock
    1.49    1.00    2.00    0.50   50%  50% Scheduler unlock [0 threads]
    1.52    1.00    2.00    0.50   51%  48% Scheduler unlock [1 suspended]
    1.41    1.00    2.00    0.48   59%  59% Scheduler unlock [many suspended]
    1.42    1.00    2.00    0.49   57%  57% Scheduler unlock [many low prio]

    0.41    0.00    1.00    0.48   59%  59% Init mutex
    1.97    1.00    2.00    0.06   96%   3% Lock [unlocked] mutex
    2.03    2.00    3.00    0.06   96%  96% Unlock [locked] mutex
    1.78    1.00    2.00    0.34   78%  21% Trylock [unlocked] mutex
    1.66    1.00    2.00    0.45   65%  34% Trylock [locked] mutex
    0.53    0.00    1.00    0.50   53%  46% Destroy mutex
   12.09   12.00   13.00    0.17   90%  90% Unlock/Lock mutex

    1.00    1.00    1.00    0.00  100% 100% Create mbox
    0.34    0.00    1.00    0.45   65%  65% Peek [empty] mbox
    2.03    2.00    3.00    0.06   96%  96% Put [first] mbox
    0.38    0.00    1.00    0.47   62%  62% Peek [1 msg] mbox
    2.03    2.00    3.00    0.06   96%  96% Put [second] mbox
    0.31    0.00    1.00    0.43   68%  68% Peek [2 msgs] mbox
    1.94    1.00    3.00    0.18   87%   9% Get [first] mbox
    1.91    1.00    2.00    0.17   90%   9% Get [second] mbox
    1.75    1.00    2.00    0.38   75%  25% Tryput [first] mbox
    1.69    1.00    2.00    0.43   68%  31% Peek item [non-empty] mbox
    1.81    1.00    2.00    0.31   81%  18% Tryget [non-empty] mbox
    1.75    1.00    2.00    0.38   75%  25% Peek item [empty] mbox
    1.72    1.00    3.00    0.45   65%  31% Tryget [empty] mbox
    0.34    0.00    1.00    0.45   65%  65% Waiting to get mbox
    0.38    0.00    1.00    0.47   62%  62% Waiting to put mbox
    0.63    0.00    1.00    0.47   62%  37% Delete mbox
    8.78    8.00    9.00    0.34   78%  21% Put/Get mbox

    0.38    0.00    1.00    0.47   62%  62% Init semaphore
    1.59    1.00    2.00    0.48   59%  40% Post [0] semaphore
    1.81    1.00    2.00    0.31   81%  18% Wait [1] semaphore
    1.50    1.00    2.00    0.50  100%  50% Trywait [0] semaphore
    1.59    1.00    2.00    0.48   59%  40% Trywait [1] semaphore
    0.50    0.00    1.00    0.50  100%  50% Peek semaphore
    0.50    0.00    1.00    0.50  100%  50% Destroy semaphore
    7.63    7.00    8.00    0.47   62%  37% Post/Wait semaphore

    0.81    0.00    1.00    0.31   81%  18% Create counter
    0.59    0.00    1.00    0.48   59%  40% Get counter value
    0.44    0.00    1.00    0.49   56%  56% Set counter value
    2.00    2.00    2.00    0.00  100% 100% Tick counter
    0.44    0.00    1.00    0.49   56%  56% Delete counter

    0.41    0.00    1.00    0.48   59%  59% Init flag
    1.72    1.00    3.00    0.45   65%  31% Destroy flag
    1.53    1.00    2.00    0.50   53%  46% Mask bits in flag
    1.84    1.00    3.00    0.32   78%  18% Set bits in flag [no waiters]
    2.44    2.00    3.00    0.49   56%  56% Wait for flag [AND]
    2.41    2.00    3.00    0.48   59%  59% Wait for flag [OR]
    2.53    2.00    4.00    0.53   96%  50% Wait for flag [AND/CLR]
    2.50    2.00    3.00    0.50  100%  50% Wait for flag [OR/CLR]
    0.38    0.00    1.00    0.47   62%  62% Peek on flag

    1.22    1.00    2.00    0.34   78%  78% Create alarm
    2.63    2.00    3.00    0.47   62%  37% Initialize alarm
    1.53    1.00    2.00    0.50   53%  46% Disable alarm
    2.59    2.00    4.00    0.52   53%  43% Enable alarm
    1.69    1.00    2.00    0.43   68%  31% Delete alarm
    2.16    2.00    3.00    0.26   84%  84% Tick counter [1 alarm]
   10.34   10.00   11.00    0.45   65%  65% Tick counter [many alarms]
    3.50    3.00    5.00    0.53   96%  53% Tick & fire counter [1 alarm]
   51.81   51.00   53.00    0.36   75%  21% Tick & fire counters [>1 together]
   11.69   11.00   12.00    0.43   68%  31% Tick & fire counters [>1 separately]
   12.00   12.00   12.00    0.00  100% 100% Alarm latency [0 threads]
   10.06   10.00   12.00    0.12   95%  95% Alarm latency [2 threads]
   11.20   10.00   12.00    0.52   54%  12% Alarm latency [many threads]
   18.02   18.00   20.00    0.03   99%  99% Alarm -> thread resume latency

    1.00    1.00    1.00    0.00            Clock/interrupt latency

    4.96    4.00    6.00    0.00            Clock DSR latency

     181     152     212                    Worker thread stack used (stack size 1088)
            All done, main thrd : stack used   796 size  1536
            All done : Idlethread stack used   164 size  1280

Timing complete - 29810 ms total

PASS:<Basic timing OK>
EXIT:<done>
```

Name

Test Programs — Details

Test Programs

The STM32L476-DISCO platform HAL contains some test programs which allow various aspects of the board to be tested.

ADC Test

The **adc1** program tests the ADC driver for the STM32. In order to run this test a potentiometer needs to be connected to the board; outer pins to 3V3 and GND (P1:1 and P1:2 are convienient), and the center pin to P1:10, which is attached to PA0. This test primarily tests that the external input functions correctly. However, in addition it also reports the values of the Vrefint, Vbat and Vts inputs that are sourced on-chip. The option `CYGBLD_HAL_CORTEXM_STM32L476_DISCO_TESTS_ADC` must be enabled to run this test since it needs human interaction.

Name

BootUp Integration — Detail

BootUp

The BootUp support for the STM32L476-DISCO target is primarily implemented in the `stm32l476_disco_support.c` file. The majority of the functions provided by that source file are only included when the `CYGPKG_BOOTUP` package is being used to construct the actual BootUp ROM loader binary.

The BootUp code is designed to be very simple, and it is envisaged that once its implementation has been tested and validated, the binary will only need to be installed onto a device once. Its only purpose is to allow the safe updating and startup of the main application. If the BootUp code ever needs to be replaced then it is a “factory” operation, for example using JTAG to re-program the on-chip flash.

This platform specific documentation should be read in conjunction with the generic BootUp package.

The BootUp package provides a basic but fully functional implementation for the platform. This has been tested to ensure that the underlying mechanism is sound. It is envisaged that the developer will customize and further extend the platform side support to meet their specific application update requirements.

BootUp loaded applications

Applications started via the BootUp loader, since they cannot include the `CYGPKG_BOOTUP` package themselves, may need access to some related configuration state. The platform is responsible for providing such “common” information. For example, the CDL option `CYGIMP_BOOTUP_RESERVED` specifies the amount of on-chip flash set aside for BootUp. Applications can then ensure that they do not interfere with the BootUp loader if using the remaining on-chip flash for their own purposes.

Warning

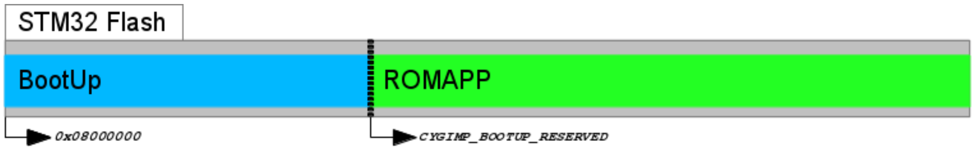
Care must be taken to ensure that the target application configuration matches the BootUp configuration, since it is normally expected that the applications to be loaded will be independent of the initial BootUp build environment. This includes the fundamental on-chip flash space set aside for the BootUp ROM loader code (`CYGIMP_BOOTUP_RESERVED`). It is expected that such values, for a particular platform instance, will be *fixed* at a suitable point during development, and definitely before products are shipped. It is the responsibility of the developer to ensure a consistent configuration between the BootUp ROM loader and any applications that may be installed/started by that BootUp code.

On-Chip ROMAPP applications

BootUp provides an alternative mechanism that supports the safe update of on-chip flash resident (`CYG_HAL_STARTUP_ROMAPP`) applications.

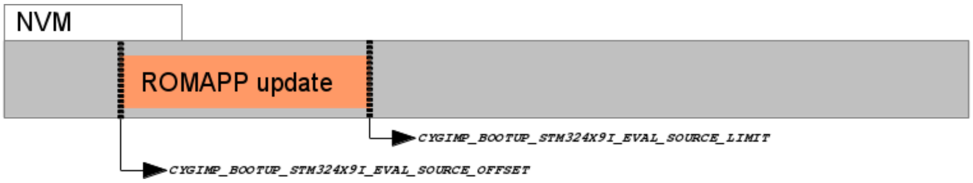
Updates using this mechanism are initiated and directed solely by the application itself. The application is responsible for locating, acquiring and verifying a new update, and placing it into Non-Volatile Memory (NVM) storage. If BootUp detects a verified update in NVM, it installs the update into the on-chip flash, overwriting and replacing the existing application. The updated application is then executed.

Figure 307.1. BootUp and Application



BootUp `ROM` loader and `ROMAPP` main application held in on-chip flash

Figure 307.2. Application Update image


NVM
ROMAPP update
CYGIMP_BOOTUP_STM324X9I_EVAL_SOURCE_LIMIT
CYGIMP_BOOTUP_STM324X9I_EVAL_SOURCE_OFFSET

Update for `ROMAPP` main application held in NVM

The example implementation uses a simple scheme that checks a fixed-format contiguous structure near the start of the binary application image file. Other than the fields used to identify the structure, the BootUp code does not interpret the hal_stm32l476_disco_bootup_structure_t in any other way.

Depending on how the alternative (pending update) application is downloaded and installed in the NVM, it may be more relevant to have the `tail` marker at the very end of the binary image. The developer may wish to update the build/release process so that the actual binary length is held in the application description structure, since that could avoid the overhead of unnecessary flash reads and writes when processing updates. Similarly, instead of a simple binary number being used to differentiate application images, the choice may be made to use the 64-bit UTC timestamp the application was created, or a human-readable string as the unique identification for a release. It is the responsibility of the build/release engineer to ensure individual releases are uniquely identifiable.

It is *critical* that the main application, when storing a pending update, stores the `tail` marker as the last bytes written. It is the responsibility of the main application to verify the data written, prior to placing the `tail` marker. This ensures that a partial image is not treated as a valid update. For example the sequence undertaken by the main application would be:

Table 307.1. Pending update sequence

Operation	**Details**
Invalidate “previous” alternative image	At a minimum ensure an invalid signature `tail` marker is written. Erasing the flash is normally required anyway, and would invalidate any previous image.
Receive update application image and write to alternative image location	NOT writing the `tail` marker. The code that stores the application should leave a “hole” where the `tail` marker resides to ensure a partial image is not incorrectly treated as valid
Verify downloaded contents	e.g. CRC or binary comparison. Normally this would be done as individual application chunks are downloaded and written to the alternative storage
Write `tail` marker	This is the very last operation after validating that the alternative image has been stored correctly. If an error has occured during the download then not-writing the `tail` ensures that the BootUp loader will not interpret the data written as a pending update
Force system RESET to start update	e.g. using the `HAL_PLATFORM_RESET` macro

The BootUp loader code will only READ from the alternative image location. This ensures that if an in-progress update is interrupted (e.g. power-loss) then when the system restarts the BootUp code will restart the application update as required.

If the BootUp platform implementation for validating the alternative image is extended to include a CRC, or similar “slow” processing, it may be worth considering whether the main application on startup will always invalidate the `tail` marker after an update to avoid subsequent system resets having to re-validate the alternative image prior to discovering that it is the same as the current main application.

Note

We cannot have the SIGNATURE support purely conditional on the BOOTUP support; since non-BOOTUP applications need to be built leaving the space. For the moment this is only enforced for ROMAPP applications, since that is all that the simple BootUp update support implements.

Building BootUp

The `ROM` startup type is chosen for BootUp so that the loader uses the on-chip SRAM for its workspace.

Example eCos configuration templates for BootUp are provided in the `misc` directory of the release. The `bootup_ROM.ecm` configuration file can be used to configure the BootUp loader.

Building a BootUp ROM image is most conveniently done at the command line. The steps needed to rebuild the ROM version of BootUp on linux are:

```
$ mkdir bootup_rom
$ cd bootup_rom
$ ecosconfig new stm32l476-disco minimal
[ … ecosconfig output elided … ]
$ ecosconfig import $ECOS_REPOSITORY/hal/cortexm/stm32/stm32l476_disco/current/misc/bootup_ROM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

The steps needed to rebuild the bundle based ROM version of BootUp on Windows within the Shell Environment are:

```
C:\Users\demo> mkdir bootup_rom
C:\Users\demo> cd bootup_rom
C:\Users\demo\bootup_rom> ecosconfig new stm32429i_eval_drb minimal
[ … ecosconfig output elided … ]
C:\Users\demo\bootup_rom> ecosconfig import %ECOS_REPOSITORY%/hal/cortexm/stm32/stm32l476_disco/current/misc/bootup_rom.ecm
C:\Users\demo\bootup_rom> ecosconfig resolve
C:\Users\demo\bootup_rom> ecosconfig tree
C:\Users\demo\bootup_rom> make
```

The resulting `install/bin/bootup.bin` binary can then be programmed into the on-chip flash from address `0x08000000`.

It is expected that the BootUp binary is installed onto the STM32L476 on-chip flash either via JTAG or by utilising the on-chip `BootROM` USB based DFU process. This is a factory or in-field process requiring specific equipment/host-software.

Once BootUp is installed it is not normally expected to require updating. Its purpose is to bootstrap the main application, and provide a standard mechanism for installing the main application. The update mechanism does *NOT* provide a method for updating the BootUp loader itself. If in-field updates of the BootUp binary are necessary, this could be achieved via the STM32 on-chip `BootROM` USB based DFU process.

Altinit Test

This application is used to test BootUp support for updating a ROMAPP application from the QSPI flash. Since there is no Ethernet available on the board, this test uses the application already programmed in to the on-chip flash.

The test expects a ROMAPP application to be stored in the on-chip flash. It erases the alternate application in the QSPI flash, copies the ROMAPP application from on-chip flash into QSPI, and then invalidates the signature of the application in on-chip flash. When BootUp starts it will discover that the main application is invalid, copy the alternate application from the QSPI and then run it.

An example eCos configuration template for building the altinit application is provided in the `misc` directory. The steps needed to build this application are as follows:

```
$ mkdir altinit
$ cd altinit
$ ecosconfig new stm32l476-disco kernel
[ … ecosconfig output elided … ]
```

```
$ ecosconfig import $ECOS_REPOSITORY/hal/cortexm/stm32/stm32l476_disco/current/misc/altinit_SRAM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
$ make -C hal/cortexm/stm32/stm32l476_disco/current tests
```

Following this, the **altinit** executable can be found in `install/tests/hal/cortexm/stm32/stm32l476_disco/current/tests/altinit`. running this will produce the following output:

```
INFO:<STM32L4_M25P64_ALTERNATIVE_OFFSET 00008000>
INFO:<STM32L4_M25P64_ALTERNATIVE_MAXLEN 000F8000>
INFO:<CYGNUM_BOOTUP_SIGNATURE_OFFSET 8>
INFO:<CYGNUM_BOOTUP_SIGNATURE_LENGTH 16>
INFO:<Erasing M25P64 area for alternative image>
INFO:<Copy main image to alt>
INFO:<Set signature in alt image>
INFO:<Invalidate main image>
INFO:<Update done>
```

The erase and copy operations here may take a some time, do not reset the board until the program is finished.

Chapter 308. BCM943362WCD4 Platform HAL

Name

`CYGPKG_HAL_CORTEXM_STM32_BCM943362WCD4` — eCos Support for the BCM943362WCD4 WICED Module

Description

This documentation describes the platform-specific elements of the STM32F205 based BCM943362WCD4 WICED module support within eCos. It should be read in conjunction with the STM32 variant HAL section, which covers the common functionality shared by all STM32 variants, including eCos HAL features and on-chip device support.

For this platform, the expected eCos development model is that programs may be downloaded and debugged via a hardware JTAG debugger. Nevertheless it is still possible to program a GDB stub image into the on-chip Flash and download and debug eCos applications with the GDB debugger via available UART pins.

Supported Hardware

The BCM943362WCD4 module consists primarily of a STM32F205RG CPU and a WM-N-BM-02 SiP (BCM43362 Wi-Fi) module.

The base BCM943362WCD4 module exposes a set of I/O pins that are used to provide functionality based on the configuration of the STM32F2 host CPU on the module. This platform HAL optionally provides the ability to target the module as installed on the BCM943362WCD4_EVB development kit (a BCM943362WCD4 module installed on a BCM9WCD1EVAL1 motherboard).

Note

The BCM9WCD1EVAL1 motherboard is intended for evaluating 31-pin WICED modules, and currently supports the following modules: BCM943362WCD2, BCM943362WCD4, BCM943362WCD6, BCM943362WCD8 and BCM9WCDUSI09.

The STM32F205RG has two main on-chip memory regions. The device has a SRAM region of 128KiB present at 0x20000000, and a 1MiB FLASH region present at 0x08000000 (which is aliased to 0x00000000 during normal execution).

The STM32 variant HAL includes support for the six on-chip serial devices which are documented in the variant HAL, however it is assumed that only USART1 is available. There is no connection for hardware flow control (RTS/CTS) lines for USART1.

Device drivers are also provided for the STM32 on-chip SPI interfaces, watchdog, RTC (wallclock) and Flash.

Tools

The board port is intended to work with GNU tools configured for an **arm-eabi** target. The original port was done using **arm-eabi-gcc** version 4.7.3g, **arm-eabi-gdb** version 7.6.1, and **binutils** version 2.32.2.

Name

Setup — Preparing the Broadcom BCM943362WCD4 for eCos Development

Overview

Given the limited available RAM memory, it is expected that the most common development method is to use JTAG for development, either by loading smaller applications into the on-chip SRAM, or by programming larger applications directly into on-chip Flash. In the first case, eCos applications should be configured for the `SRAM` startup type, and in the second case for `ROM` startup type.

Nevertheless, it is still possible to program a GDB stub ROM image into on-chip Flash and download and debug via a serial UART, if pins for the UART are available. In that case, eCos applications are configured for `RAM` startup and then downloaded and run on the board via the debugger **arm-eabi-gdb**, or via the Eclipse IDE. For serial communications, all versions run with 8 bits, no parity, and 1 stop bit at 115200 baud. This rate can be changed in the eCos configuration used for building the GDB stub ROM image.

HAL startup types

The following startup types may be selected for applications:

Configuration	Description
SRAM	Stand-alone programs running from on-chip SRAM, loaded via JTAG hardware debugger
ROM	Stand-alone programs running from on-chip FLASH
ROMAPP	Stand-alone programs running from an offset into the on-chip FLASH, that are started by a separate boot loader
RAM	Programs loading via a GDB stub ROM into on-chip RAM, which rely on a debug monitor

Further details are available later in this manual.

Programming ROM images

To program ROM startup applications into Flash, including the GDB stub ROM, a JTAG debugger that understands the STM32 flash may be used, such as the OpenOCD tool. The **openocd** GDB server can directly program flash based applications from the GDB **load** command.

For example, assuming that **openocd** is running on the same host as GDB, and is connected to the target board the following will program the “bootup.elf” application into the on-chip flash:

```
$ arm-eabi-gdb install/bin/bootup.elf
GNU gdb (eCosCentric GNU tools 4.7.3g) 7.6.1
[ … GDB output elided … ]
(gdb) target remote localhost:3333
hal_reset_vsr () at path/hal_misc.c:171
(gdb) load
Loading section .rom_vectors, size 0x14 lma 0x8000000
Loading section .text, size 0x3adc lma 0x8000018
Loading section .rodata, size 0x6c0 lma 0x8003af8
Loading section .data, size 0x6dc lma 0x80041b8
Start address 0x8000018, load size 18572
Transfer rate: 14 KB/sec, 4643 bytes/write.
(gdb)
```

Alternatively, the **openocd** telnet interface can be used to manually program the flash. By default the **openocd** session provides a comand-line via port `4444`. Consult the OpenOCD documentation for more details if a non-default **openocd** configuration is being used.

With a **telnet** connection established to the **openocd** any binary data can easily be written to the on-chip flash. e.g.

```
$ telnet localhost 4444
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Open On-Chip Debugger
> flash write_image test.bin 0x08000000
wrote 32518 bytes from file test.bin in 1.073942s (29.569 KiB/s)
```

To create a binary for flash programming the **arm-eabi-objcopy** command is used. This converts the, ELF format, linked application into a raw binary. For example:

```
$ arm-eabi-objcopy -O binary programname programname.bin
```

Rebuilding the GDB stub

Should it prove necessary to rebuild a GDB stub ROM binary, this is done most conveniently at the command line. For the `bcm943362wcd4_evb` platform the steps needed are:

```
$ mkdir gdbstub_bcm943362wcd4_evb
$ cd gdbstub_bcm943362wcd4_evb
$ ecosconfig new bcm943362wcd4_evb stubs
[ … ecosconfig output elided … ]
$ ecosconfig tree
$ make
```

At the end of the build, the `install/bin` subdirectory should contain the file `gdb_module.bin`. This may be programmed to the board using the above procedure.

Name

Configuration — Platform-specific Configuration Options

Overview

The Broadcom BCM943362WCD4 module platform HAL package is loaded automatically when eCos is configured for the `bcm943362wcd4` or `bcm943362wcd4_evb` targets. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The Broadcom BCM943362WCD4 module platform HAL package supports four separate startup types:

SRAM

> This is the startup type used to build applications that are loaded via a JTAG interface. The application will be self-contained with no dependencies on services provided by other software. The program expects to be loaded from `hal_virtual_vector_table_end` and entered at that address. Memory below `hal_virtual_vector_table_end` is set aside for vector tables. eCos startup code will perform all necessary hardware initialization.

ROM

> This startup type can be used for finished applications which will be programmed into internal Flash ROM at location `0x08000000`. Data and BSS will be put into internal SRAM starting from `hal_virtual_vector_table_end`. The application will be self-contained with no dependencies on services provided by other software. The program expects to boot from reset with ROM mapped at location zero. It will then transfer control to the `0x08000000` region. eCos startup code will perform all necessary hardware initialization.

ROMAPP

> This startup type can be used for finished applications which will be programmed into internal Flash ROM at the `CYGIMP_BOOTUP_RESERVED` configured offset. This is a variation of the `ROM` startup type for applications that are started by a smaller boot loader application. eCos startup code will perform all necessary hardware initialization.

RAM

> This is the startup type which is used if relying on a GDB stub ROM image programmed into internal Flash to download and run applications into SRAM via arm-eabi-gdb and a serial UART. RAM from `0x20000000` to `0x20001000` is reserved for the GDB stub, but then the RAM startup application may be loaded into memory from `0x20001000` and debugged using GDB. It is assumed that the hardware has already been initialized by the GDB stub ROM. By default the application will use the eCos virtual vectors mechanism to obtain services from the GDB stub ROM, including diagnostic output.

Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building the GDB stub ROM (or RedBoot).

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a `RAM` startup, and disabled otherwise. It can be manually disabled for a `RAM` startup, making the application self-contained, as a testing step before switching to `ROM` startup.

If the application does not rely on a ROM monitor for diagnostic services then the serial port will be claimed for HAL diagnostic output.

UART Serial Driver

The BCM943362WCD4 module uses the STM32's internal UART serial support. The HAL diagnostic interface, used for both polled diagnostic output and GDB stub communication, is only expected to be available to be used on the USART1 port (counting the first UART as UART1).

As well as the polled HAL diagnostic interface, there is also a `CYGPKG_IO_SERIAL_CORTEXM_STM32` package which contains all the code necessary to support interrupt-driven operation with greater functionality. All six UARTs can be supported by this driver. For the BCM943362WCD4 module however the available I/O pins impose a limit on the available functionality.

Note

It is not recommended to use this driver with a port at the same time as using that port for HAL diagnostic I/O.

This driver is not active until the `CYGPKG_IO_SERIAL_DEVICES` configuration option (within the generic serial driver support package `CYGPKG_IO_SERIAL`) is enabled in the configuration. By default this will only enable support in the driver for the USART1 port (the same as the HAL diagnostic interface), but the default configuration can be modified to enable support for other serial ports. Note that in this package, serial port numbering starts at 0, rather than 1. So for example, to enable the serial driver for ports USART1 and USART2, enable the configuration options “ST STM32 serial port 0 driver” (`CYGPKG_IO_SERIAL_CORTEXM_STM32_SERIAL0`) and “ST STM32 serial port 1 driver” (`CYGPKG_IO_SERIAL_CORTEXM_STM32_SERIAL1`).

SPI Driver

An SPI bus driver is available for the STM32 in the package “ST STM32 SPI driver” (`CYGPKG_DEVS_SPI_CORTEXM_STM32`).

For the base BCM943362WCD4 module SPI bus 1 is configured for off-module SPI connections. If SPI is being used to communicate with the on-module WM-N-BM-02 SiP tne SPI bus 2 is also configured. Suitable device entries are created as appropriate for application use.

When `CYGPKG_HAL_CORTEXM_STM32_BCM943362WCD4_SPI_ACCESS` is configured then the SPI2 device `spi_device_wm_n_bm_02` is enumerated.

When targetting the BCM9WCD1EVAL1 based `bcm943362wcd4_evb` platform then the SPI1 device `m25pxx_spi_device` is enumerated.

To disable support for both the above SPI devices, the platform HAL contains an option “SPI devices” (`CYGPKG_HAL_CORTEXM_STM32_BCM943362WCD4_SPI`) which can be disabled. No other SPI devices are instantiated.

Consult the generic SPI driver API documentation in the eCosPro Reference Manual for further details on SPI support in eCosPro, along with the configuration options in the STM32 SPI device driver.

Flash Driver

The STM32's on-chip Flash may be programmed and managed using the Flash driver located in the “STM32 Flash memory support” (`CYGPKG_DEVS_FLASH_STM32`) package. This driver is enabled automatically if the generic “Flash device drivers” (`CYGPKG_IO_FLASH`) package is included in the eCos configuration.

The driver will configure itself automatically for the size and parameters of the specific STM32 variant present on the BCM943362WCD4 module.

A number of other aspects of Flash driver behaviour can be configured within that driver, such as program/erase parallelism and program burst size. Consult the driver for more details.

Name

JTAG support — Usage

Use of JTAG for debugging

JTAG can be used to single-step and debug loaded applications, or even applications resident in ROM, including the GDB stub ROM.

Debugging of ROM applications is only possible if using hardware breakpoints. The Cortex-M3 core of the STM32 only supports six such hardware breakpoints, so they should be used sparingly. If using a GDB front-end such as Eclipse, check it has not set unnecessary extra breakpoints such as at `main()`. Some JTAG devices give the option of whether to set hardware or software breakpoints by default. Be sure to configure your device appropriately.

The base BCM943362WCD4 module exposes the STM32F205RG JTAG/SWD signals on the module connectors TP6..TP10. A suitable hardware connection would be required to allow JTAG debugging. The BCM943362WCD4_EVB kit provides a standard 20-pin ARM JTAG header J8, but by default the module JTAG connection is configured to use the BCM9WCD1EVAL1 motherboard FT2232 connection presented via the USB J5 connector. Direct support for this USB interface is provided by the WICED-SDK supplied OpenOCD binary.

When debugging via JTAG, you are likely to need to disable the default HAL idle thread action, otherwise there may be issues where the target fails to halt and the debugging session is unreliable. More details can be found in the Cortex-M architectural HAL.

Normally a notable disadvantage with JTAG debugging is that it does not allow thread-aware debugging, such as the ability to inspect different eCos threads or their stack backtraces, set thread-specific breakpoints, and so on. Fortunately the Ronetix PEEDI JTAG unit does support thread-aware debugging of eCos applications, however extra configuration steps are required. Consult the PEEDI documentation for more details as usage is beyond the scope of this document.

OpenOCD notes

The following OpenOCD documentation uses as an example the BCM943362WCD4_EVB J5 USB JTAG connection. An OpenOCD configuration that supports the `ft2232` interface and understands the `ft2232_layout BCM9WCD1EVAL1` configuration must be used, Such an **openocd** is pre-built and available in the WICED-SDK. For example WICED-SDK revision 3.1.2 the necessary host binaries can be found in the directory `WICED-SDK-3.1.2/tools/OpenOCD`.

An example OpenOCD configuration file `openocd.bcm943362wcd4_evb.cfg` is provided within the eCos platform HAL package in the source repository. This will be in the directory `packages/hal/cortexm/stm32/bcm943362wcd4/`*`VERSION`*`/misc` relative to the root of your eCos installation.

This configuration file can be used with the WICED-SDK supplied OpenOCD on the host as follows:

```
$ ./Linux64/openocd-all-brcm-libftdi -f openocd.bcm943362wcd4_evb.cfg
Open On-Chip Debugger 0.8.0-dev-00139-g4505978-dirty (2013-08-26-15:55)
Licensed under GNU GPL v2
For bug reports, read
    http://openocd.sourceforge.net/doc/doxygen/bugs.html
Info : only one transport option; autoselect 'jtag'
trst_and_srst separate srst_nogate trst_push_pull srst_push_pull connect_assert_srst
adapter speed: 1000 kHz
adapter_nsrst_delay: 100
jtag_ntrst_delay: 100
jtag_init
Info : max TCK change to: 30000 kHz
Info : clock speed 1000 kHz
Polling target stm32f2xxx.cpu failed, GDB will be halted. Polling again in 100ms
Info : JTAG tap: stm32f2xxx.cpu tap/device found: 0x4ba00477 (mfg: 0x23b, part: 0xba00, ver: 0x4)
Info : JTAG tap: stm32f2xxx.bs tap/device found: 0x06411041 (mfg: 0x020, part: 0x6411, ver: 0x0)
Info : Selecting JTAG transport command set.
Info : AP INIT COMPLETE
Info : stm32f2xxx.cpu: hardware has 6 breakpoints, 4 watchpoints
Polling target stm32f2xxx.cpu succeeded again
Info : JTAG tap: stm32f2xxx.cpu tap/device found: 0x4ba00477 (mfg: 0x23b, part: 0xba00, ver: 0x4)
Info : JTAG tap: stm32f2xxx.bs tap/device found: 0x06411041 (mfg: 0x020, part: 0x6411, ver: 0x0)
Info : Selecting JTAG transport command set.
Info : AP INIT COMPLETE
Info : Selecting JTAG transport command set.
Info : AP INIT COMPLETE
```

By default **openocd** provides a console on port `4444`, and this can be used to interact with the target system. This console interface can be used to perform debugging, program the flash, etc.

Normally **arm-eabi-gdb** is used to connect to the default GDB server port `3333` for debugging. For example:

```
(gdb) target remote localhost:3333
Remote debugging using localhost:3333
0x00000000 in ?? ()
(gdb)
```

The application can then be loaded and executed under GDB as normal. If you are using Eclipse then, if required, you can define a “preload” gdb macro to emit any necessary commands to OpenOCD. See the “Hardware Assisted Debugging” section of the “Eclipse/CDT for eCos application development” document's “Debugging eCos applications” chapter.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the Broadcom BCM943362WCD4 module hardware and should be read in conjunction with the specification for that device. The BCM943362WCD4 platform HAL package complements the Cortex-M architectural HAL and the STM32 variant HAL. It provides functionality which is specific to the target module, and optionally, specifically supported motherboards.

Targetting the eCos platform `bcm943362wcd4` will configure eCos for a stand-alone BCM943362WCD4 module with no assumption made about the I/O connected to the module TP pins.

Targetting the `bcm943362wcd4_evb` platform will configure eCos for a BCM9WCD1EVAL1 motherboard based module, and provide access to the LEDs, switches, thermistor and SPI flash device available on that motherboard.

Startup

Following a hard or soft reset the HAL will initialize or reinitialize many of the on-chip peripherals. There is an exception for `RAM` startup applications which depend on a ROM monitor for certain services.

For the stand-alone application startup types, the HAL will perform additional initialization, programming the various internal registers including the PLL, peripheral clocks and GPIO pins. The details of the early hardware startup may be found in the `src/bcm943362wcd4_misc.c` in both the `hal_system_init` and `hal_platform_init` functions.

Memory Map

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

Internal RAM
: This is located at address `0x20000000` of the memory space, and is 128KiB in size. The eCos VSR table always occupies the initial bytes at the base of this memory, followed by the optional virtual vector table depending on the eCos configuration. The top `CYGNUM_HAL_COMMON_INTERRUPTS_STACK_SIZE` bytes of memory are reserved for the interrupt stack. The remainder of on-chip SRAM is available for use by applications.

Internal FLASH
: This is located at address `0x08000000` of the memory space and will be mapped to `0x00000000` at reset. This region is 1MiB in size. and `ROM` applications are by default configured to run from this memory.

On-Chip Peripherals
: These are accessible at locations `0x40000000` and `0xE0000000` upwards. Descriptions of the contents can be found in the relevant STM32 User Manual.

Linker Scripts

The platform linker scripts define the following symbols:

hal_vsr_table
: This defines the location of the VSR table. This is set to `0x20000000` for all startup types, and space for `CYGNUM_HAL_VSR_COUNT` entries is reserved to match the use of a STM32F2 processor.

hal_virtual_vector_table
: This defines the location of the virtual vector table used to communicate between an ROM monitor and an eCos application. This is allocated right after the VSR table, and is normally `0x20000184` for STM32F2 targets.

The CDL option CYGSEM_HAL_VIRTUAL_VECTOR_SUPPORT defines whether this virtual vector support is needed. If not defined then the table is zero sized.

hal_virtual_vector_table_end
This defines the location of the end of the (optional) virtual vector table.

hal_interrupt_stack
This defines the location of the interrupt stack. For all startup types this is allocated to the top of internal SRAM, at 0x20020000.

hal_startup_stack
This defines the location of the startup stack. For all startup types it is initially allocated at the half-way point of the interrupt stack. The size of the interrupt stack is defined by the CDL option CYGNUM_HAL_COMMON_INTERRUPTS_STACK_SIZE.

Flash wait states

The BCM943362WCD4 platform HAL provides a configuration option to set the number of Flash read wait states to use: CYGNUM_HAL_CORTEXM_STM32_FLASH_WAIT_STATES. It is important to verify and if necessary update this value if changing the CPU clock (HCLK) frequency or CPU voltage. Consult the STM32 Flash programming manual (PM0059) for appropriate values for different clock speeds or voltages. The default of 3 reflects a supply voltage of 3.3V and HCLK of 120MHz.

Diagnostic LEDs

The platform HAL header file at <cyg/hal/plf_io.h> defines the following convenience function to allow any LEDs to be set:

```
extern void hal_bcm943362wcd4_led(unsigned char c);
```

The low-order bits of the argument c correspond to individual LEDs.

The default BCM943362WCD4 module support does not provide LEDs, since the relevant off-module GPIO signals are not defined. However, when targetting the BCM9WCD1EVAL1 motherboard two LEDs are fitted for diagnostic purposes: D1 (red) and D2 (green). These LEDs are free for application use. The bcm943362wcd4_evb**manual** test provides a simple example of changing the LED state.

Real-time characterization

The **tm_basic** kernel test gives statistics gathered about the real-time characterization and performance of the kernel. The sample output is shown here for information. The test was built for the bcm943362wcd4_evb platform using the SRAM startup type and optimization flag -O2.

Example 308.1. bcm943362wcd4 Real-time characterization

```
             Startup, main thrd : stack used   360 size  1536
             Startup : Idlethread stack used    76 size  1280

eCos Kernel Timings
Notes: all times are in microseconds (.000001) unless otherwise stated

Reading the hardware clock takes 0 'ticks' overhead
... this value will be factored out of all other measurements
Clock interrupt took    7.06 microseconds (7 raw clock ticks)

Testing parameters:
   Clock samples:            32
   Threads:                   8
   Thread switches:         128
   Mutexes:                  16
   Mailboxes:                16
   Semaphores:               16
   Scheduler operations:    128
   Counters:                 16
   Flags:                    16
   Alarms:                   16
   Stack Size:             1088


                                 Confidence
     Ave     Min     Max     Var  Ave  Min  Function
  ======  ======  ======  ====== ========== ========
INFO:<Ctrl-C disabled until test completion>
    5.63    5.00    6.00    0.47   62%  37% Create thread
    1.00    1.00    1.00    0.00  100% 100% Yield thread [all suspended]
    1.38    1.00    2.00    0.47   62%  62% Suspend [suspended] thread
    1.38    1.00    2.00    0.47   62%  62% Resume thread
    1.88    1.00    2.00    0.22   87%  12% Set priority
    0.25    0.00    1.00    0.38   75%  75% Get priority
    3.63    3.00    4.00    0.47   62%  37% Kill [suspended] thread
    1.25    1.00    2.00    0.38   75%  75% Yield [no other] thread
    2.00    2.00    2.00    0.00  100% 100% Resume [suspended low prio] thread
    1.25    1.00    2.00    0.38   75%  75% Resume [runnable low prio] thread
    2.00    2.00    2.00    0.00  100% 100% Suspend [runnable] thread
    1.00    1.00    1.00    0.00  100% 100% Yield [only low prio] thread
    1.38    1.00    2.00    0.47   62%  62% Suspend [runnable->not runnable]
    4.00    4.00    4.00    0.00  100% 100% Kill [runnable] thread
    3.00    3.00    3.00    0.00  100% 100% Destroy [dead] thread
    5.75    5.00    6.00    0.38   75%  25% Destroy [runnable] thread
    6.50    6.00    7.00    0.50  100%  50% Resume [high priority] thread
    2.28    2.00    3.00    0.40   71%  71% Thread switch

    0.21    0.00    1.00    0.33   78%  78% Scheduler lock
    1.11    1.00    2.00    0.19   89%  89% Scheduler unlock [0 threads]
    1.13    1.00    2.00    0.22   87%  87% Scheduler unlock [1 suspended]
    1.14    1.00    2.00    0.24   85%  85% Scheduler unlock [many suspended]
    1.13    1.00    2.00    0.22   87%  87% Scheduler unlock [many low prio]

    0.38    0.00    1.00    0.47   62%  62% Init mutex
    1.44    1.00    2.00    0.49   56%  56% Lock [unlocked] mutex
    2.00    2.00    2.00    0.00  100% 100% Unlock [locked] mutex
    1.44    1.00    2.00    0.49   56%  56% Trylock [unlocked] mutex
    1.31    1.00    2.00    0.43   68%  68% Trylock [locked] mutex
    0.44    0.00    1.00    0.49   56%  56% Destroy mutex
    7.00    7.00    7.00    0.00  100% 100% Unlock/Lock mutex

    0.69    0.00    1.00    0.43   68%  31% Create mbox
    0.50    0.00    1.00    0.50  100%  50% Peek [empty] mbox
    1.63    1.00    2.00    0.47   62%  37% Put [first] mbox
    0.25    0.00    1.00    0.38   75%  75% Peek [1 msg] mbox
    1.63    1.00    2.00    0.47   62%  37% Put [second] mbox
    0.00    0.00    0.00    0.00  100% 100% Peek [2 msgs] mbox
    1.56    1.00    2.00    0.49   56%  43% Get [first] mbox
    1.50    1.00    2.00    0.50  100%  50% Get [second] mbox
    1.31    1.00    2.00    0.43   68%  68% Tryput [first] mbox
    1.31    1.00    2.00    0.43   68%  68% Peek item [non-empty] mbox
    1.44    1.00    2.00    0.49   56%  56% Tryget [non-empty] mbox
    1.25    1.00    2.00    0.38   75%  75% Peek item [empty] mbox
    1.31    1.00    2.00    0.43   68%  68% Tryget [empty] mbox
    0.25    0.00    1.00    0.38   75%  75% Waiting to get mbox
    0.31    0.00    1.00    0.43   68%  68% Waiting to put mbox
    0.56    0.00    1.00    0.49   56%  43% Delete mbox
    4.94    4.00    5.00    0.12   93%   6% Put/Get mbox

    0.31    0.00    1.00    0.43   68%  68% Init semaphore
    1.00    1.00    1.00    0.00  100% 100% Post [0] semaphore
    1.44    1.00    2.00    0.49   56%  56% Wait [1] semaphore
    1.19    1.00    2.00    0.30   81%  81% Trywait [0] semaphore
    1.13    1.00    2.00    0.22   87%  87% Trywait [1] semaphore
    0.44    0.00    1.00    0.49   56%  56% Peek semaphore
    0.38    0.00    1.00    0.47   62%  62% Destroy semaphore
    4.69    4.00    5.00    0.43   68%  31% Post/Wait semaphore

    0.56    0.00    1.00    0.49   56%  43% Create counter
    0.44    0.00    1.00    0.49   56%  56% Get counter value
    0.38    0.00    1.00    0.47   62%  62% Set counter value
    1.63    1.00    2.00    0.47   62%  37% Tick counter
    0.31    0.00    1.00    0.43   68%  68% Delete counter

    0.38    0.00    1.00    0.47   62%  62% Init flag
    1.38    1.00    2.00    0.47   62%  62% Destroy flag
    1.13    1.00    2.00    0.22   87%  87% Mask bits in flag
    1.44    1.00    2.00    0.49   56%  56% Set bits in flag [no waiters]
    2.00    2.00    2.00    0.00  100% 100% Wait for flag [AND]
    1.81    1.00    2.00    0.31   81%  18% Wait for flag [OR]
    2.00    2.00    2.00    0.00  100% 100% Wait for flag [AND/CLR]
    1.94    1.00    2.00    0.12   93%   6% Wait for flag [OR/CLR]
    0.25    0.00    1.00    0.38   75%  75% Peek on flag

    1.00    1.00    1.00    0.00  100% 100% Create alarm
    2.00    2.00    2.00    0.00  100% 100% Initialize alarm
    1.19    1.00    2.00    0.30   81%  81% Disable alarm
    2.00    2.00    2.00    0.00  100% 100% Enable alarm
    1.31    1.00    2.00    0.43   68%  68% Delete alarm
    1.00    1.00    1.00    0.00  100% 100% Tick counter [1 alarm]
    5.00    5.00    5.00    0.00  100% 100% Tick counter [many alarms]
    2.94    2.00    3.00    0.12   93%   6% Tick & fire counter [1 alarm]
   25.00   25.00   25.00    0.00  100% 100% Tick & fire counters [>1 together]
    6.44    6.00    7.00    0.49   56%  56% Tick & fire counters [>1 separately]
    6.00    6.00    6.00    0.00  100% 100% Alarm latency [0 threads]
    6.00    6.00    6.00    0.00  100% 100% Alarm latency [2 threads]
    5.83    5.00    6.00    0.28   82%  17% Alarm latency [many threads]
   10.01   10.00   11.00    0.01   99%  99% Alarm -> thread resume latency

    0.00    0.00    0.00    0.00            Clock/interrupt latency

    2.75    2.00    3.00    0.00            Clock DSR latency

     208     180     212                    Worker thread stack used (stack size 1088)
            All done, main thrd : stack used   804 size  1536
            All done : Idlethread stack used   164 size  1280

Timing complete - 29320 ms total

PASS:<Basic timing OK>
EXIT:<done>
```

Name

Test Programs — Details

Test Programs

The BCM943362WCD4 platform HAL contains a test suitable for the `bcm943362wcd4_evb` platform, that allows various aspects of that board to be tested.

Manual Test

The **manual** test is only built by default when targetting a BCM9WCD1EVAL1 motherboard based BCM943362WCD4 module (e.g. the BCM943362WCD4_EVB development kit).

This program tests various aspects of the basic platform port. The basic test can be used to validate the LED and push-button GPIO operation. Depending on the eCos configuration further testing of the flash (detecting on-chip and motherboard SPI devices) and ADC device access is performed.

When flash support is configured the test will display the memory address ranges for the flash areas.

Note

Whereas the on-chip flash is directly addressable, the off-chip SPI flash is given a logical address for use through the flash API but it is not actually memory mapped.

When ADC support is configured the **manual** program tests the ADC driver for the STM32. The only device connected to the ADC on the board is the thermistor connected to ADC1 logical channel 3, named TH1 on the motherboard. In addition the test also report the values of the Temperature, Vrefint and Vbat inputs that are sourced on-chip.

Chapter 309. BCM943364WCD1 Platform HAL

Name

`CYGPKG_HAL_CORTEXM_STM32_BCM943364WCD1` — eCos Support for the BCM943364WCD1 WICED Module

Description

This documentation describes the platform-specific elements of the STM32F411 based BCM943364WCD1 WICED module support within eCos. It should be read in conjunction with the STM32 variant HAL section, which covers the common functionality shared by all STM32 variants, including eCos HAL features and on-chip device support.

For this platform, the expected eCos development model is that programs may be downloaded and debugged via a hardware JTAG debugger. Nevertheless it is still possible to program a GDB stub image into the on-chip Flash and download and debug eCos applications with the GDB debugger via available UART pins.

Supported Hardware

The BCM943364WCD1 module consists primarily of a STM32F411RE CPU and a BCM43364 Wi-Fi CoB.

The base BCM943364WCD1 module exposes a set of I/O pins that are used to provide functionality based on the configuration of the STM32F4 host CPU on the module. This platform HAL optionally provides the ability to target the module as installed on the BCM943364WCD1_EVB development kit (a BCM943364WCD1 module installed on a BCM9WCD9EVAL1 motherboard).

Note

The BCM9WCD9EVAL1 motherboard is intended for evaluating 44-pin WICED modules.

The STM32F411RE has two main on-chip memory regions. The device has a SRAM region of 128KiB present at 0x20000000, and a 512KiB FLASH region present at 0x08000000 (which is aliased to 0x00000000 during normal execution).

The STM32 variant HAL includes support for the three on-chip serial devices which are documented in the variant HAL, however it is assumed that only USART1 is available.

Device drivers are also provided for the STM32 on-chip SPI interfaces, I²C interfaces, watchdog, RTC (wallclock) and Flash.

Tools

The board port is intended to work with GNU tools configured for an **arm-eabi** target. The original port was done using **arm-eabi-gcc** version 4.7.3j, **arm-eabi-gdb** version 7.6.1, and **binutils** version 2.32.2.

Name

Setup — Preparing the Broadcom BCM943364WCD1 for eCos Development

Overview

Given the limited available RAM memory, it is expected that the most common development method is to use JTAG for development, either by loading smaller applications into the on-chip SRAM, or by programming larger applications directly into on-chip Flash. In the first case, eCos applications should be configured for the `SRAM` startup type, and in the second case for `ROM` startup type.

Nevertheless, it is still possible to program a GDB stub ROM image into on-chip Flash and download and debug via a serial UART, if pins for the UART are available. In that case, eCos applications are configured for `RAM` startup and then downloaded and run on the board via the debugger **arm-eabi-gdb**, or via the Eclipse IDE. For serial communications, all versions run with 8 bits, no parity, and 1 stop bit at 115200 baud. This rate can be changed in the eCos configuration used for building the GDB stub ROM image.

HAL startup types

The following startup types may be selected for applications:

Configuration	Description
SRAM	Stand-alone programs running from on-chip SRAM, loaded via JTAG hardware debugger
ROM	Stand-alone programs running from on-chip FLASH
ROMAPP	Stand-alone programs running from an offset into the on-chip FLASH, that are started by a separate boot loader
RAM	Programs loading via a GDB stub ROM into on-chip RAM, which rely on a debug monitor

Further details are available later in this manual.

Programming ROM images

To program ROM startup applications into Flash, including the GDB stub ROM, a JTAG debugger that understands the STM32 flash may be used, such as the OpenOCD tool. The **openocd** GDB server can directly program flash based applications from the GDB **load** command.

For example, assuming that **openocd** is running on the same host as GDB, and is connected to the target board the following will program the "bootup.elf" application into the on-chip flash:

```
$ arm-eabi-gdb install/bin/bootup.elf
GNU gdb (eCosCentric GNU tools 4.7.3g) 7.6.1
[ … GDB output elided … ]
(gdb) target remote localhost:3333
hal_reset_vsr () at path/hal_misc.c:171
(gdb) load
Loading section .rom_vectors, size 0x14 lma 0x8000000
Loading section .text, size 0x3adc lma 0x8000018
Loading section .rodata, size 0x6c0 lma 0x8003af8
Loading section .data, size 0x6dc lma 0x80041b8
Start address 0x8000018, load size 18572
Transfer rate: 14 KB/sec, 4643 bytes/write.
(gdb)
```

Alternatively, the **openocd** telnet interface can be used to manually program the flash. By default the **openocd** session provides a comand-line via port `4444`. Consult the OpenOCD documentation for more details if a non-default **openocd** configuration is being used.

With a **telnet** connection established to the **openocd** any binary data can easily be written to the on-chip flash. e.g.

```
$ telnet localhost 4444
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Open On-Chip Debugger
> flash write_image test.bin 0x08000000
wrote 32518 bytes from file test.bin in 1.073942s (29.569 KiB/s)
```

To create a binary for flash programming the **arm-eabi-objcopy** command is used. This converts the, ELF format, linked application into a raw binary. For example:

```
$ arm-eabi-objcopy -O binary programname programname.bin
```

Rebuilding the GDB stub

Should it prove necessary to rebuild a GDB stub ROM binary, this is done most conveniently at the command line. For the `bcm943364wcd1_evb` platform the steps needed are:

```
$ mkdir gdbstub_bcm943364wcd1_evb
$ cd gdbstub_bcm943364wcd1_evb
$ ecosconfig new bcm943364wcd1_evb stubs
[ … ecosconfig output elided … ]
$ ecosconfig tree
$ make
```

At the end of the build, the `install/bin` subdirectory should contain the file `gdb_module.bin`. This may be programmed to the board using the above procedure.

Name

Configuration — Platform-specific Configuration Options

Overview

The Broadcom BCM943364WCD1 module platform HAL package is loaded automatically when eCos is configured for the `bcm943364wcd1` or `bcm943364wcd1_evb` targets. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The Broadcom BCM943364WCD1 module platform HAL package supports four separate startup types:

SRAM

> This is the startup type used to build applications that are loaded via a JTAG interface. The application will be self-contained with no dependencies on services provided by other software. The program expects to be loaded from `hal_virtual_vector_table_end` and entered at that address. Memory below `hal_virtual_vector_table_end` is set aside for vector tables. eCos startup code will perform all necessary hardware initialization.

ROM

> This startup type can be used for finished applications which will be programmed into internal Flash ROM at location `0x08000000`. Data and BSS will be put into internal SRAM starting from `hal_virtual_vector_table_end`. The application will be self-contained with no dependencies on services provided by other software. The program expects to boot from reset with ROM mapped at location zero. It will then transfer control to the `0x08000000` region. eCos startup code will perform all necessary hardware initialization.

ROMAPP

> This startup type can be used for finished applications which will be programmed into internal Flash ROM at the `CYGIMP_BOOTUP_RESERVED` configured offset. This is a variation of the `ROM` startup type for applications that are started by a smaller boot loader application. eCos startup code will perform all necessary hardware initialization.

RAM

> This is the startup type which is used if relying on a GDB stub ROM image programmed into internal Flash to download and run applications into SRAM via arm-eabi-gdb and a serial UART. RAM from `0x20000000` to `0x20001000` is reserved for the GDB stub, but then the RAM startup application may be loaded into memory from `0x20001000` and debugged using GDB. It is assumed that the hardware has already been initialized by the GDB stub ROM. By default the application will use the eCos virtual vectors mechanism to obtain services from the GDB stub ROM, including diagnostic output.

Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building the GDB stub ROM (or RedBoot).

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a `RAM` startup, and disabled otherwise. It can be manually disabled for a `RAM` startup, making the application self-contained, as a testing step before switching to `ROM` startup.

If the application does not rely on a ROM monitor for diagnostic services then the serial port will be claimed for HAL diagnostic output.

UART Serial Driver

The BCM943364WCD1 module uses the STM32's internal UART serial support. The HAL diagnostic interface, used for both polled diagnostic output and GDB stub communication, is only expected to be available to be used on the USART1 port (counting the first UART as UART1).

As well as the polled HAL diagnostic interface, there is also a CYGPKG_IO_SERIAL_CORTEXM_STM32 package which contains all the code necessary to support interrupt-driven operation with greater functionality. All three UARTs can be supported by this driver. For the BCM943364WCD1 module however the available I/O pins impose a limit on the available functionality.

Note

It is not recommended to use this driver with a port at the same time as using that port for HAL diagnostic I/O.

This driver is not active until the CYGPKG_IO_SERIAL_DEVICES configuration option (within the generic serial driver support package CYGPKG_IO_SERIAL) is enabled in the configuration. By default this will only enable support in the driver for the USART1 port (the same as the HAL diagnostic interface), but the default configuration can be modified to enable support for other serial ports. Note that in this package, serial port numbering starts at 0, rather than 1. So for example, to enable the serial driver for ports USART1 and USART2, enable the configuration options “ST STM32 serial port 0 driver” (CYGPKG_IO_SERIAL_CORTEXM_STM32_SERIAL0) and “ST STM32 serial port 1 driver” (CYGPKG_IO_SERIAL_CORTEXM_STM32_SERIAL1).

SPI Driver

An SPI bus driver is available for the STM32 in the package “ST STM32 SPI driver” (CYGPKG_DEVS_SPI_CORTEXM_STM32).

For the base BCM943364WCD1 module SPI bus 1 and bus 2 are configured for off-module SPI connections.

When targetting the BCM9WCD9EVAL1 based bcm943364wcd1_evb platform then the SPI1 flash memory device m25pxx_spi_device is enumerated.

To disable support for both the above SPI devices, the platform HAL contains an option “SPI devices” (CYGPKG_HAL_CORTEXM_STM32_BCM943364WCD1_SPI) which can be disabled. No other SPI devices are instantiated.

Consult the generic SPI driver API documentation in the eCosPro Reference Manual for further details on SPI support in eCosPro, along with the configuration options in the STM32 SPI device driver.

Flash Driver

The STM32's on-chip Flash may be programmed and managed using the Flash driver located in the “STM32 Flash memory support” (CYGPKG_DEVS_FLASH_STM32) package. This driver is enabled automatically if the generic “Flash device drivers” (CYGPKG_IO_FLASH) package is included in the eCos configuration.

The driver will configure itself automatically for the size and parameters of the specific STM32 variant present on the BCM943364WCD1 module.

A number of other aspects of Flash driver behaviour can be configured within that driver, such as program/erase parallelism and program burst size. Consult the driver for more details.

Name

JTAG support — Usage

Use of JTAG for debugging

JTAG can be used to single-step and debug loaded applications, or even applications resident in ROM, including the GDB stub ROM.

Debugging of ROM applications is only possible if using hardware breakpoints. The Cortex-M3 core of the STM32 only supports six such hardware breakpoints, so they should be used sparingly. If using a GDB front-end such as Eclipse, check it has not set unnecessary extra breakpoints such as at `main()`. Some JTAG devices give the option of whether to set hardware or software breakpoints by default. Be sure to configure your device appropriately.

The base BCM943364WCD1 module exposes the STM32F411RE JTAG/SWD signals on the module connectors TP7..TP11. A suitable hardware connection would be required to allow JTAG debugging. The BCM943364WCD1_EVB kit provides a standard 20-pin ARM JTAG header J3, but by default the module JTAG connection is configured to use the BCM9WCD9EVAL1 motherboard FT2232 connection presented via the USB J4 connector. Direct support for this USB interface is provided by the WICED-SDK supplied OpenOCD binary.

When debugging via JTAG, you are likely to need to disable the default HAL idle thread action, otherwise there may be issues where the target fails to halt and the debugging session is unreliable. More details can be found in the Cortex-M architectural HAL.

Normally a notable disadvantage with JTAG debugging is that it does not allow thread-aware debugging, such as the ability to inspect different eCos threads or their stack backtraces, set thread-specific breakpoints, and so on. Fortunately the Ronetix PEEDI JTAG unit does support thread-aware debugging of eCos applications, however extra configuration steps are required. Consult the PEEDI documentation for more details as usage is beyond the scope of this document.

OpenOCD notes

The following OpenOCD documentation uses as an example the BCM943364WCD1_EVB J4 USB JTAG connection. An OpenOCD configuration that supports the `ft2232` interface and understands the `ft2232_layout BCM9WCD1EVAL1` configuration must be used, Such an **openocd** is pre-built and available in the WICED-SDK. For example WICED-SDK revision 3.5.1 the necessary host binaries can be found in the directory `WICED-SDK-3.5.1/tools/OpenOCD`.

An example OpenOCD configuration file `openocd.bcm943364wcd1_evb.cfg` is provided within the eCos platform HAL package in the source repository. This will be in the directory `packages/hal/cortexm/stm32/bcm943364wcd1/`*`VERSION`*`/misc` relative to the root of your eCos installation.

This configuration file can be used with the WICED-SDK supplied OpenOCD on the host as follows:

```
$ ./Linux64/openocd-all-brcm-libftdi -f openocd.bcm943364wcd1_evb.cfg
Open On-Chip Debugger 0.9.0-00029-g33ca6ac-dirty (2015-05-28-14:31)
Licensed under GNU GPL v2
For bug reports, read
    http://openocd.sourceforge.net/doc/doxygen/bugs.html
Info : only one transport option; autoselect 'jtag'
trst_and_srst separate srst_nogate trst_push_pull srst_push_pull connect_assert_srst
adapter speed: 2000 kHz
adapter_nsrst_delay: 100
jtag_ntrst_delay: 100
cortex_m reset_config sysresetreq
jtag_init
Warn : Using DEPRECATED interface driver 'ft2232'
Info : Consider using the 'ftdi' interface driver, with configuration files in interface/ftdi/...
Info : max TCK change to: 30000 kHz
Info : clock speed 2000 kHz
Info : JTAG tap: stm32f4x.cpu tap/device found: 0x4ba00477 (mfg: 0x23b, part: 0xba00, ver: 0x4)
Info : JTAG tap: stm32f4x.bs tap/device found: 0x06431041 (mfg: 0x020, part: 0x6431, ver: 0x0)
Info : stm32f4x.cpu: hardware has 6 breakpoints, 4 watchpoints
```

By default **openocd** provides a console on port `4444`, and this can be used to interact with the target system. This console interface can be used to perform debugging, program the flash, etc.

Normally **arm-eabi-gdb** is used to connect to the default GDB server port `3333` for debugging. For example:

```
(gdb) target remote localhost:3333
Remote debugging using localhost:3333
0x00000000 in ?? ()
(gdb)
```

The application can then be loaded and executed under GDB as normal. If you are using Eclipse then, if required, you can define a “preload” gdb macro to emit any necessary commands to OpenOCD. See the “Hardware Assisted Debugging” section of the “Eclipse/CDT for eCos application development” document's “Debugging eCos applications” chapter.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the Broadcom BCM943364WCD1 module hardware and should be read in conjunction with the specification for that device. The BCM943364WCD1 platform HAL package complements the Cortex-M architectural HAL and the STM32 variant HAL. It provides functionality which is specific to the target module, and optionally, specifically supported motherboards.

Targetting the eCos platform `bcm943364wcd1` will configure eCos for a stand-alone BCM943364WCD1 module with no assumption made about the I/O connected to the module TP pins.

Targetting the `bcm943364wcd1_evb` platform will configure eCos for a BCM9WCD9EVAL1 motherboard based module, and provide access to the LEDs, switches, thermistor and SPI flash device available on that motherboard.

Startup

Following a hard or soft reset the HAL will initialize or reinitialize many of the on-chip peripherals. There is an exception for `RAM` startup applications which depend on a ROM monitor for certain services.

For the stand-alone application startup types, the HAL will perform additional initialization, programming the various internal registers including the PLL, peripheral clocks and GPIO pins. The details of the early hardware startup may be found in the `src/bcm943364wcd1_misc.c` in both the `hal_system_init` and `hal_platform_init` functions.

Memory Map

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

Internal RAM
: This is located at address `0x20000000` of the memory space, and is 128KiB in size. The eCos VSR table always occupies the initial bytes at the base of this memory, followed by the optional virtual vector table depending on the eCos configuration. The top `CYGNUM_HAL_COMMON_INTERRUPTS_STACK_SIZE` bytes of memory are reserved for the interrupt stack. The remainder of on-chip SRAM is available for use by applications.

Internal FLASH
: This is located at address `0x08000000` of the memory space and will be mapped to `0x00000000` at reset. This region is 512KiB in size. and `ROM` applications are by default configured to run from this memory.

On-Chip Peripherals
: These are accessible at locations `0x40000000` and `0xE0000000` upwards. Descriptions of the contents can be found in the relevant STM32 User Manual.

Linker Scripts

The platform linker scripts define the following symbols:

hal_vsr_table
: This defines the location of the VSR table. This is set to `0x20000000` for all startup types, and space for `CYGNUM_HAL_VSR_COUNT` entries is reserved to match the use of a STM32F2 processor.

hal_virtual_vector_table
: This defines the location of the virtual vector table used to communicate between an ROM monitor and an eCos application. This is allocated right after the VSR table, and is normally `0x20000198` for STM32F4 targets.

The CDL option `CYGSEM_HAL_VIRTUAL_VECTOR_SUPPORT` defines whether this virtual vector support is needed. If not defined then the table is zero sized.

hal_virtual_vector_table_end
: This defines the location of the end of the (optional) virtual vector table.

hal_interrupt_stack
: This defines the location of the interrupt stack. For all startup types this is allocated to the top of internal SRAM, at `0x20020000`.

hal_startup_stack
: This defines the location of the startup stack. For all startup types it is initially allocated at the half-way point of the interrupt stack. The size of the interrupt stack is defined by the CDL option `CYGNUM_HAL_COMMON_INTERRUPTS_STACK_SIZE`.

Flash wait states

The BCM943364WCD1 platform HAL provides a configuration option to set the number of Flash read wait states to use: `CYGNUM_HAL_CORTEXM_STM32_FLASH_WAIT_STATES`. It is important to verify and if necessary update this value if changing the CPU clock (HCLK) frequency or CPU voltage. Consult the STM32 Flash programming manual (PM0059) for appropriate values for different clock speeds or voltages. The default of `3` reflects a supply voltage of 3.3V and HCLK of 100MHz.

Diagnostic LEDs

The platform HAL header file at `<cyg/hal/plf_io.h>` defines the following convenience function to allow any LEDs to be set:

```
extern void hal_bcm943364wcd1_led(unsigned char c);
```

The low-order bits of the argument `c` correspond to individual LEDs.

The default BCM943364WCD1 module support does not provide LEDs, since the relevant off-module GPIO signals are not defined. However, when targetting the BCM9WCD9EVAL1 motherboard two LEDs are fitted for diagnostic purposes: D4 (red) and D3 (green). These LEDs are free for application use. The `bcm943364wcd1_evb`**manual** test provides a simple example of changing the LED state.

Real-time characterization

The **tm_basic** kernel test gives statistics gathered about the real-time characterization and performance of the kernel. The sample output is shown here for information. The test was built for the `bcm943364wcd1_evb` platform using the `SRAM` startup type and optimization flag -O2.

Example 309.1. bcm943364wcd1 Real-time characterization

```
             Startup, main thrd : stack used   352 size  1536
             Startup : Idlethread stack used    84 size  1280

eCos Kernel Timings
Notes: all times are in microseconds (.000001) unless otherwise stated

Reading the hardware clock takes 0 'ticks' overhead
... this value will be factored out of all other measurements
Clock interrupt took    8.50 microseconds (8 raw clock ticks)

Testing parameters:
   Clock samples:            32
   Threads:                   8
   Thread switches:         128
   Mutexes:                  16
   Mailboxes:                16
   Semaphores:               16
   Scheduler operations:    128
   Counters:                 16
   Flags:                    16
   Alarms:                   16
   Stack Size:             1088


                                 Confidence
     Ave     Min     Max     Var  Ave  Min  Function
  ======  ======  ======  ====== ========== ========
INFO:<Ctrl-C disabled until test completion>
    7.13    7.00    8.00    0.22   87%  87% Create thread
    1.50    1.00    2.00    0.50  100%  50% Yield thread [all suspended]
    1.50    1.00    2.00    0.50  100%  50% Suspend [suspended] thread
    1.50    1.00    2.00    0.50  100%  50% Resume thread
    2.25    2.00    3.00    0.38   75%  75% Set priority
    0.38    0.00    1.00    0.47   62%  62% Get priority
    4.50    4.00    5.00    0.50  100%  50% Kill [suspended] thread
    1.50    1.00    2.00    0.50  100%  50% Yield [no other] thread
    2.50    2.00    3.00    0.50  100%  50% Resume [suspended low prio] thread
    1.63    1.00    2.00    0.47   62%  37% Resume [runnable low prio] thread
    2.13    2.00    3.00    0.22   87%  87% Suspend [runnable] thread
    1.38    1.00    2.00    0.47   62%  62% Yield [only low prio] thread
    1.50    1.00    2.00    0.50  100%  50% Suspend [runnable->not runnable]
    4.75    4.00    5.00    0.38   75%  25% Kill [runnable] thread
    3.88    3.00    4.00    0.22   87%  12% Destroy [dead] thread
    7.25    7.00    8.00    0.38   75%  75% Destroy [runnable] thread
    8.13    8.00    9.00    0.22   87%  87% Resume [high priority] thread
    2.84    2.00    4.00    0.29   82%  17% Thread switch

    0.37    0.00    1.00    0.46   63%  63% Scheduler lock
    1.35    1.00    2.00    0.46   64%  64% Scheduler unlock [0 threads]
    1.35    1.00    2.00    0.46   64%  64% Scheduler unlock [1 suspended]
    1.35    1.00    2.00    0.46   64%  64% Scheduler unlock [many suspended]
    1.35    1.00    2.00    0.46   64%  64% Scheduler unlock [many low prio]

    0.44    0.00    1.00    0.49   56%  56% Init mutex
    1.81    1.00    2.00    0.31   81%  18% Lock [unlocked] mutex
    2.06    2.00    3.00    0.12   93%  93% Unlock [locked] mutex
    1.81    1.00    2.00    0.31   81%  18% Trylock [unlocked] mutex
    1.69    1.00    2.00    0.43   68%  31% Trylock [locked] mutex
    0.44    0.00    1.00    0.49   56%  56% Destroy mutex
    9.06    9.00   10.00    0.12   93%  93% Unlock/Lock mutex

    0.75    0.00    1.00    0.38   75%  25% Create mbox
    0.19    0.00    1.00    0.30   81%  81% Peek [empty] mbox
    2.00    2.00    2.00    0.00  100% 100% Put [first] mbox
    0.19    0.00    1.00    0.30   81%  81% Peek [1 msg] mbox
    2.00    2.00    2.00    0.00  100% 100% Put [second] mbox
    0.31    0.00    1.00    0.43   68%  68% Peek [2 msgs] mbox
    1.94    1.00    2.00    0.12   93%   6% Get [first] mbox
    1.94    1.00    2.00    0.12   93%   6% Get [second] mbox
    1.00    1.00    1.00    0.00  100% 100% Tryput [first] mbox
    1.44    1.00    2.00    0.49   56%  56% Peek item [non-empty] mbox
    1.75    1.00    2.00    0.38   75%  25% Tryget [non-empty] mbox
    1.56    1.00    2.00    0.49   56%  43% Peek item [empty] mbox
    1.50    1.00    2.00    0.50  100%  50% Tryget [empty] mbox
    0.38    0.00    1.00    0.47   62%  62% Waiting to get mbox
    0.25    0.00    1.00    0.38   75%  75% Waiting to put mbox
    1.00    1.00    1.00    0.00  100% 100% Delete mbox
    6.25    6.00    7.00    0.38   75%  75% Put/Get mbox

    0.44    0.00    1.00    0.49   56%  56% Init semaphore
    1.44    1.00    2.00    0.49   56%  56% Post [0] semaphore
    2.00    2.00    2.00    0.00  100% 100% Wait [1] semaphore
    1.38    1.00    2.00    0.47   62%  62% Trywait [0] semaphore
    1.50    1.00    2.00    0.50  100%  50% Trywait [1] semaphore
    0.50    0.00    1.00    0.50  100%  50% Peek semaphore
    0.50    0.00    1.00    0.50  100%  50% Destroy semaphore
    5.50    5.00    6.00    0.50  100%  50% Post/Wait semaphore

    0.75    0.00    1.00    0.38   75%  25% Create counter
    0.50    0.00    1.00    0.50  100%  50% Get counter value
    0.31    0.00    1.00    0.43   68%  68% Set counter value
    1.88    1.00    2.00    0.22   87%  12% Tick counter
    0.44    0.00    1.00    0.49   56%  56% Delete counter

    0.50    0.00    1.00    0.50  100%  50% Init flag
    1.94    1.00    2.00    0.12   93%   6% Destroy flag
    1.50    1.00    2.00    0.50  100%  50% Mask bits in flag
    1.81    1.00    2.00    0.31   81%  18% Set bits in flag [no waiters]
    2.50    2.00    3.00    0.50  100%  50% Wait for flag [AND]
    2.38    2.00    3.00    0.47   62%  62% Wait for flag [OR]
    2.50    2.00    3.00    0.50  100%  50% Wait for flag [AND/CLR]
    2.31    2.00    3.00    0.43   68%  68% Wait for flag [OR/CLR]
    0.31    0.00    1.00    0.43   68%  68% Peek on flag

    1.19    1.00    2.00    0.30   81%  81% Create alarm
    2.38    2.00    3.00    0.47   62%  62% Initialize alarm
    1.44    1.00    2.00    0.49   56%  56% Disable alarm
    2.50    2.00    3.00    0.50  100%  50% Enable alarm
    1.31    1.00    2.00    0.43   68%  68% Delete alarm
    2.19    2.00    3.00    0.30   81%  81% Tick counter [1 alarm]
    6.56    6.00    7.00    0.49   56%  43% Tick counter [many alarms]
    3.75    3.00    4.00    0.38   75%  25% Tick & fire counter [1 alarm]
   31.38   31.00   32.00    0.47   62%  62% Tick & fire counters [>1 together]
    8.00    8.00    8.00    0.00  100% 100% Tick & fire counters [>1 separately]
    8.00    8.00    8.00    0.00  100% 100% Alarm latency [0 threads]
    7.23    6.00    8.00    0.38   73%   1% Alarm latency [2 threads]
    7.13    6.00    8.00    0.43   63%  11% Alarm latency [many threads]
   12.01   12.00   13.00    0.01   99%  99% Alarm -> thread resume latency

    0.00    0.00    0.00    0.00            Clock/interrupt latency

    3.66    2.00    4.00    0.00            Clock DSR latency

     192     172     204                    Worker thread stack used (stack size 1088)
            All done, main thrd : stack used   796 size  1536
            All done : Idlethread stack used   172 size  1280

Timing complete - 29320 ms total

PASS:<Basic timing OK>
EXIT:<done>
```

Name

Test Programs — Details

Test Programs

The BCM943364WCD1 platform HAL contains a test suitable for the `bcm943364wcd1_evb` platform, that allows various aspects of that board to be tested.

Manual Test

The **manual** test is only built by default when targetting a BCM9WCD9EVAL1 motherboard based BCM943364WCD1 module (e.g. the BCM943364WCD1_EVB development kit).

This program tests various aspects of the basic platform port. The basic test can be used to validate the LED and push-button GPIO operation. Depending on the eCos configuration further testing of the flash (detecting on-chip and motherboard SPI devices) and ADC device access is performed.

When flash support is configured the test will display the memory address ranges for the flash areas.

Note

Whereas the on-chip flash is directly addressable, the off-chip SPI flash is given a logical address for use through the flash API but it is not actually memory mapped.

When ADC support is configured the **manual** program tests the ADC driver for the STM32. The only device connected to the ADC on the board is the thermistor connected to ADC1 logical channel 3, named TH1 on the motherboard. In addition the test also report the values of the Temperature, Vrefint and Vbat inputs that are sourced on-chip.

Wi-Fi firmware

With a suitable eCos configuration the **manual** application can be used to initialise the off-chip SPI flash memory on the BCM9WCD9EVAL1 motherboard. The use of the off-chip flash will normally be required for this platform due to the limited on-chip flash space with which to hold both an application and the required radio firmware binary.

The following example uses the command-line **ecosconfig** tool, though, as always, the necessary configuration changes can be performed using the graphical **configtool** tool if desired. This configuration is just being used to build a `DIRECT` application containing a copy of the firmware binary, and to allow that image to be written to the off-chip flash memory. It is not envisaged that real-world applications for this platform will make use of the `DIRECT` mode due to the limited space remaining for applications.

Prior to creating the actual configuration some non-default options should be placed into a file suitable for importing. These are used to ensure the correct configuration for the firmware update build of the **manual** application. This example assumes the following has been placed into the file `tmp.ecm`.

```
cdl_option CYG_HAL_STARTUP { user_value ROM };
cdl_option CYGFUN_NET_WIFI_BROADCOM_WWD_RESOURCES_INDIRECT { user_value 0 };
cdl_option CYGHWR_HAL_CORTEXM_DIAGNOSTICS_INTERFACE { user_value "gdb_hwdebug_fileio" };
```

Once the configuration fragment file has been created then the application can be configured and built using the following command sequence:

```
$ ecosconfig new bcm943364wcd1_evb lwip_eth
$ ecosconfig add CYGPKG_IO CYGPKG_IO_FLASH CYGPKG_CRC
$ ecosconfig import tmp.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
$ make tests
```

The resulting `install/tests/hal/cortexm/stm32/bcm943364wcd1/current/tests/manual` application will then validate, and update appropriately if required, the off-chip firmware image. The firmware binary image included in the application is as defined by the selected WWD SDK. A successful run of the application will be similar to:

```
INFO:<Manual Test>
FLASH: 0x08000000-0x0807ffff, 4 x 0x4000 blocks, 1 x 0x10000 blocks, 3 x 0x20000 blocks
FLASH: 0xc0000000-0xc00fffff, 16 x 0x10000 blocks
INFO:<WWD configuration PLATFORM "BCM943364WCD1_EVB" SDK "3.5.2">
INFO:<WWD firmware image required and platform is INDIRECT capable>
INFO:<DIRECT F/W CRC 4585C974 mismatch against INDIRECT F/W CRC 48D488E3>
... Unlocking from 0xc0000000-0xc007ffff: ........
... Erase from 0xc0000000-0xc007ffff: ........
PASS:<Erased INDIRECT area C0001000-C0080000>
... Program from 0x080070dc-0x0805e5b0 to 0xc0001000: ......
PASS:<Programmed INDIRECT area>
... Locking from 0xc0000000-0xc007ffff: ........
PASS:<Current F/W image written to INDIRECT area with CRC 4585C974>
[ … rest of output elided … ]
```

Re-running the application will report that the off-chip image matches:

```
INFO:<Manual Test>
FLASH: 0x08000000-0x0807ffff, 4 x 0x4000 blocks, 1 x 0x10000 blocks, 3 x 0x20000 blocks
FLASH: 0xc0000000-0xc00fffff, 16 x 0x10000 blocks
INFO:<WWD configuration PLATFORM "BCM943364WCD1_EVB" SDK "3.5.2">
INFO:<WWD firmware image required and platform is INDIRECT capable>
INFO:<DIRECT F/W CRC 4585C974 image already installed in INDIRECT area>
[ … rest of output elided … ]
```

Subsequently, normal WWD Wi-Fi applications will use the `CYGFUN_NET_WIFI_BROADCOM_WWD_RESOURCES_INDIRECT` enabled support to allow all of the on-chip memory for application use; with the code loading the firmware binary from the off-chip storage.

Chapter 310. STM32L4R9-DISCO Platform HAL

Name

`CYGPKG_HAL_CORTEXM_STM32_STM32L4R9_DISCO` — eCos Support for the STM32L4R9-DISCO Board

Description

This documentation describes the platform-specific elements of the ST STM32L4R9I_DISCO board support within eCos. It should be read in conjunction with the STM32 variant HAL section, which covers the common functionality shared by all STM32 variants, including eCos HAL features and on-chip device support.

The board is equipped with an on-board ST-Link interface, which is typically used for eCos application development. There is also a TAG connector (CN8) available for a direct SWD debug connection.

Supported Hardware

The STM32L4R9AI has two main on-chip memory regions. The device has a SRAM region of 640KiB present at 0x20000000, and a 2MiB FLASH region present at 0x08000000 (which is aliased to 0x00000000 during normal execution). A 512Mbit MX25LM51245G Octo SPI flash device is available through the OCTOSPI controller. A 16Mbit PSRAM device provides an off-chip RAM area after the initialisaton of the system (the MFX I/O expander is used to configure the H/W appropriately).

The STM32 variant HAL includes support for the six on-chip serial devices. These consist of three USARTs, two UARTs and a LPUART. These are all supported by a common driver and are documented in the variant HAL. However, the STM32L4R9-DISCO motherboard only makes use of two of these. USART2 is routed to the on-board ST-Link and presented as a CDC-ACM device on the CN13 USB connector. USART3 is routed to the DSI with its hardware RTS/CTS flow control lines.

The STM32 variant HAL also includes support for the I²C buses. Support is provided for the on-board MFX I/O expander and audio codec I²C devices, with both being on bus 1. The descriptors are exported in the normal way via `<cyg/io/i2c.h>`, with the respective names `hal_st_mfx` and `hal_stm32l4r9_disco_cs42l51`.

Similarly the STM32 variant HAL includes support for the SPI buses. Though the board does not provide any SPI devices as standard.

Device drivers are also provided for the STM32 on-chip, ADC devices. The internal channels are available, with some I/O pins on the connectors available for analog input.

Additionally, support is provided for the on-chip watchdog, RTC (wallclock) and a Flash driver exists to permit management of the STM32's on-chip Flash and the off-chip Quad SPI NOR flash.

The STM32L4 processor and the STM32L4R9-DISCO board provide a wide variety of peripherals, but unless support is specifically indicated, it should be assumed that it is not included.

Tools

The board port is intended to work with GNU tools configured for an **arm-eabi** target. The original port was done using **arm-eabi-gcc** version 7.3.0c, **arm-eabi-gdb** version 8.1, and **binutils** version 2.23.2.

Name

Setup — Preparing the STM32L4R9-DISCO Board for eCos Development

Overview

The STM32L4R9-DISCO motherboard incorporates a ST-Link interface, which is used to provide hardware debug support for eCos application development. Applications can be loaded, run and debugged either via the command line GDB debugger **arm-eabi-gdb**, or via the Eclipse IDE. These host tools communicate with the target via a “GDB server” intermediary that supports a specific JTAG-based hardware debugger (e.g. **JLinkGDBServer** for the Segger J-Link, or **OpenOCD** for the J-Link and other hardware debuggers).

Normally for release applications the `ROM` startup type would be used, with the application programmed into the on-chip flash for execution when the board boots. It is still possible to use the hardware debugging support to debug such flash-based ROM applications, and this may be the desired approach if the application is too large for execution from on-chip SRAM, or where all of the SRAM is required for application run-time use.

If off-chip Non-Volatile Memory (NVM) is used to hold the main application then the board can boot from the internal flash using a suitable boot loader. For example, the eCosPro BootUp ROM loader, where the BootUp code can start the main application (after an optional update sequence).

If required, it is still possible to program a GDB stub or RedBoot `ROM` image into on-chip Flash and download and debug via a serial connection (using USART1). In that case, eCos applications are configured for `RAM` startup and then downloaded and run on the board via the debugger **arm-eabi-gdb**, or via the Eclipse IDE as appropriate. However, the space available to applications with this approach is so limited as to make it essentially impractical.

Programming ROM images

The JTAG connection and suitable host software (e.g. The OpenOCD package **openocd** tool, or the Segger J-Link **JLinkGDBServer**) can be used to program the flash.

Both these GDB servers can directly program flash based applications from the GDB **load** command. For example, assuming that **openocd** is running on the same host as GDB, and is connected to the target board the following will program the “bootup.elf” application into the on-chip flash:

```
$ arm-eabi-gdb install/bin/bootup.elf
GNU gdb (eCosCentric GNU tools 7.3.0c) 8.1
[ … GDB output elided … ]
(gdb) target remote localhost:3333
hal_reset_vsr () at path/hal_misc.c:171
(gdb) load
Loading section .rom_vectors, size 0x14 lma 0x8000000
Loading section .text, size 0x3adc lma 0x8000018
Loading section .rodata, size 0x6c0 lma 0x8003af8
Loading section .data, size 0x6dc lma 0x80041b8
Start address 0x8000018, load size 18572
Transfer rate: 14 KB/sec, 4643 bytes/write.
(gdb)
```

If using **JLinkGDBServer**, the approach is identical, apart from using port 2231 to connect to the server.

Alternatively, the **openocd** telnet interface can be used to manually program the flash. By default the **openocd** session provides a comand-line via port `4444`. Consult the OpenOCD documentation for more details if a non-default **openocd** configuration is being used.

With a **telnet** connection established to the **openocd** any binary data can easily be written to the on-chip flash. e.g.

```
$ telnet localhost 4444
```

```
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Open On-Chip Debugger
> flash write_image test.bin 0x08000000
wrote 32518 bytes from file test.bin in 1.073942s (29.569 KiB/s)
```

To create a binary for flash programming the **arm-eabi-objcopy** command is used. This converts the, ELF format, linked application into a raw binary. For example:

```
$ arm-eabi-objcopy -O binary programname programname.bin
```

Name

Configuration — Platform-specific Configuration Options

Overview

The STM32L4R9-DISCO board platform HAL package `CYGPKG_HAL_CORTEXM_STM32_STM32L4R9_DISCO` is loaded automatically when eCos is configured for the `stm32l4r9_disco` target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The STM32L4R9-DISCO board platform HAL package supports four separate startup types:

ROM
: This startup type can be used for finished applications which will be programmed into internal flash at location 0x08000000. Data and BSS will be put into internal SRAM starting from 0x200002D8. Internal SRAM below this address is reserved for vector tables. The application will be self-contained with no dependencies on services provided by other software. The program expects to boot from reset with ROM mapped at location zero. It will then transfer control to the 0x08000000 region. eCos startup code will perform all necessary hardware initialization.

ROMAPP
: This startup type can be used for finished applications which will be programmed into internal flash at location 0x08008000. Data and BSS will be put into internal SRAM starting from 0x200002D8. Internal SRAM below this address is reserved for vector tables. The application will be self-contained with no dependencies on services provided by other software. The program expects to boot from reset with ROM mapped at location zero. It will then transfer control to the 0x08000000 region. eCos startup code will perform all necessary hardware initialization.

  This startup type is identical to the ROM startup with the exception of the flash base address. It is used for applications that can be started or updated by BootUp.

SRAM
: This startup type can be used for finished applications which will be loaded into internal SRAM via a JTAG interface. The application will be self-contained with no dependencies on services provided by other software. The program expects to be loaded from 0x200002D8 and entered at that address. eCos startup code will perform all necessary hardware initialization.

RAM
: When the board has RedBoot (or a GDB stub ROM) programmed into internal Flash at location 0x08000000 then the arm-eabi-gdb debugger can communicate with a suitably configured UART connection to load and debug applications. An application is loaded into memory from 0x20001000. It is assumed that the hardware has already been initialized by RedBoot. By default the application will *not* be stand-alone, and will use the eCos virtual vectors mechanism to obtain services from RedBoot, including diagnostic output.

  ### Warning

  RedBoot can have an adverse affect on the real-time performance of applications.

It should be noted that due to the MFX I/O expander needing to be used to configure the hardware for correct PSRAM operation there is no direct startup type for loading and executing applications from PSRAM. However, there is nothing to stop a suitable boot loader from initialising the hardware appropriately and application code subsequently being loaded into and executed from the PSRAM.

RedBoot and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building RedBoot.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option CYGSEM_HAL_USE_ROM_MONITOR should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, as a testing step before switching to ROM startup.

Note

Though, as previously discussed, since the option of hardware debugging is available as standard on the STM32L4R9-DISCO platform, and space in the SRAM is limited, it is unlikely that the RAM startup type would be used for development.

SPI Driver

An SPI bus driver is available for the STM32 in the package “ST STM32 SPI driver” (CYGPKG_DEVS_SPI_CORTEXM_STM32).

No SPI devices are instantiated for this platform by default.

Consult the generic SPI driver API documentation in the eCosPro Reference Manual for further details on SPI support in eCosPro, along with the configuration options in the STM32 SPI device driver.

I²C Driver

The STM32 variant HAL provides the main I²C hardware driver itself, configured at CYGPKG_HAL_STM32_I2C. However, the platform I²C support can also be configured separately at CYGPKG_HAL_CORTEXM_STM32_STM32L4R9_DISCO_I2C. This enables I²C bus 1. Some board H/W features are routed via the I²C MFXv3 I/O expander device, and as such, the support for bus 1 is enabled by default. An audio codec is also available on this bus and is instantiated with the name hal_stm32l4r9_disco_cs42l51. The instantiated device is available for applications via <cyg/io/i2c.h>.

ADC Driver

The STM32 processor variant HAL provides an ADC driver. The STM32L4R9-DISCO platform HAL enables the support for the device ADC1 and for configuration of the respective ADC device input channels.

Consult the generic ADC driver API documentation in the eCosPro Reference Manual for further details on ADC support in eCosPro, along with the configuration options in the STM32 ADC device driver.

Flash Driver

The STM32's on-chip Flash may be programmed and managed using the Flash driver located in the “STM32 Flash memory support” (CYGPKG_DEVS_FLASH_STM32) package. This driver is enabled automatically if the generic “Flash device drivers” (CYGPKG_IO_FLASH) package is included in the eCos configuration. The driver will configure itself automatically for the size and parameters of the specific STM32 variant present on the STM32L4R9-DISCO board.

A number of aspects of Flash driver behaviour can be configured within that driver, such as program/erase parallelism and program burst size. Consult the driver for more details.

OCTOSPI Flash Driver

When OCTOSPI NOR flash support is enabled in the configuration with CYGHWR_HAL_CORTEXM_STM32_FLASH_OCTOSPI, then the cyg_stm32_octospi1_device device is exported and can be accessed via the standard flash API. The device is given a logical base address to match its physical base address of 0x90000000 (corresponding to FMC bank 4) when it is memory mapped (if CYGFUN_DEVS_FLASH_OCTOSPI1_CORTEXM_STM32_MEMMAPPED is enabled in the OCTOSPI driver, which is not the default). When memory mapping is disabled, using the eCos Flash API will still allow the device to be read/written at that logical base address.

Name

SWD support — Usage

Use of JTAG for debugging

JTAG can be used to single-step and debug loaded applications, or even applications resident in ROM.

Debugging of ROM applications is only possible if using hardware breakpoints. The Cortex-M4 core of the STM32L4R9 only supports a limited number of such hardware breakpoints, so they may need to be used sparingly. If using a GDB front-end such as Eclipse, check it has not set unnecessary extra breakpoints such as at `main()`. Some JTAG devices give the option of whether to set hardware or software breakpoints by default. Be sure to configure your device appropriately.

When debugging via JTAG, you are likely to need to disable the default HAL idle thread action, otherwise there may be issues where the target fails to halt and the debugging session is unreliable. More details can be found in the Cortex-M architectural HAL.

J-Link Support

The preferred debug device for this board is the on-board ST-Link debug interface probe. This also has the benefit fof routing USART2 as a standard USB CDC-ACM terminal.

Initialization scripts that support debugging the board via either the OpenOCD or J-Link GDB servers can be found in the directory `packages/hal/cortexm/stm32/stm32l4r9_disco/`*`VERSION`*`/misc` relative to the root of your eCos installation.

For **JLinkGDBServer**, the file `stm32l4r9_disco.jlink`, provides initialization for the board and should be used in the following command line:

```
    $ JLinkGDBServer -device STM32L4R9AI -xc stm32l4r9_disco.jlink
```

Similarly OpenOCD may be invoked with the following command line:

```
    $ openocd -f openocd.stm32l4r9_disco.cfg
```

Configuration of JTAG applications

JTAG/SWD applications can be loaded directly into SRAM or flash without requiring a ROM monitor. Loading can be done directly through the JTAG/SWD device, or through GDB where supported by the JTAG/SWD device.

In order to configure the application to support these modes, it is recommended to use the `SRAM`, `ROM` or `ROMAPP` startup types which will implicitly cause two important settings to change. Firstly, `CYGSEM_HAL_USE_ROM_MONITOR` must be disabled. Secondly the `CYGDBG_HAL_DIAG_TO_DEBUG_CHAN` option should be enabled in order to prevent HAL diagnostic output being encoded into GDB ($O) packets. These configuration changes could be made by hand, but use of the aformentioned startup types will just work.

With these changes, any diagnostic output will appear out of the configured diagnostic channel, by default USART2 on ST-Link USB CN13. An eCosCentric extension allows diagnostic output to appear on the GDB console, or on the Eclipse console. To enable this feature, you must set the configuration option `CYGSEM_HAL_DIAG_TO_GDBFILEIO_CHAN` in the common HAL package. If you are using the graphical configuration tool then you should then accept any suggested solutions to the subsequent configuration conflicts. Older eCos releases also required the gdb "set hwdebug on" command to be used to enable GDB or Eclipse console output, but this is no longer required with the latest tools.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the STM32L4R9-DISCO board hardware, and should be read in conjunction with that specification. The STM32L4R9-DISCO platform HAL package complements the Cortex-M architectural HAL and the STM32 variant HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset the HAL will initialize or reinitialize many of the on-chip peripherals. There is an exception for `RAM` startup applications which depend on a ROM monitor for certain services.

For `ROM`, `ROMAPP` and `SRAM` startup types the HAL will perform additional initialization, programming the various internal registers including the PLL, peripheral clocks and GPIO pins. The details of the early hardware startup may be found in the `src/stm32l4r9_disco_misc.c` in both `hal_system_init` and `hal_platform_init`.

Memory Map

The platform HAL package provides the memory layout information needed to generate the linker script. For all the STARTUP variations the top `CYGNUM_HAL_COMMON_INTERRUPTS_STACK_SIZE` bytes of the on-chip SRAM are reserved for the interrupt stack. The remainder of the internal SRAM is available for use by applications. The key memory locations are as follows:

Internal SRAM

This is located at address `0x20000000` of the memory space, and is 640KiB in size. The eCos VSR table occupies the bottom 444 bytes of memory, with the virtual vector table starting at `0x200001BC` and extending to `0x200002D8`.

Internal FLASH

This is located at address `0x08000000` of the memory space and will be mapped to `0x00000000` at reset. This region is 2048KiB in size. `ROM` and `ROMAPP` applications are by default configured to run from this memory.

OCTOSPI NOR Flash

The OCTOSPI NOR flash is accessible through the flash API. It is partitioned between the alternate application image and test space for the JFFS2 flash file system. The alternate application image occupies the first 1Mbyte of the OCTOSPI flash. The JFFS2 test space currently occupies the next 256Kbytes. The space allocated for the alternate application image may be adjusted by changing `STM32L4_BOOTUP_ALTERNATIVE_OFFSET` and `STM32L4_BOOTUP_ALTERNATIVE_MAXLEN` in `plf_arch.h`. The JFFS2 test space is defined in `__STM32L4R9_DISCO_FLASHTEST_OCTOSPI` in `plf_io.h`. Applications would not normally use this to define their JFFS2 filesystem location, but use the device/offset/length placement device format in the filesystem `mount()` call.

On-Chip Peripherals

These are accessible at locations `0x40000000` and `0xE0000000` upwards. Descriptions of the contents can be found in the STM32 User Manual.

Linker Scripts

The platform linker scripts define the following symbols:

hal_vsr_table

This defines the location of the VSR table. This is set to `0x20000000` for all startup types, and space for 111 entries is reserved.

hal_virtual_vector_table

This defines the location of the virtual vector table used to communicate between an ROM monitor and an eCos application. This is allocated right after the VSR table, at `0x200002BC`.

hal_interrupt_stack This defines the location of the interrupt stack. This is allocated to the top of internal SRAM, from `0x200A0000` down.

hal_startup_stack This defines the location of the startup stack. For all startup types it is initially allocated at the half-way point of the interrupt stack.

Flash wait states

The STM32L4R9-DISCO platform HAL provides a configuration option to set the number of Flash read wait states to use: `CYGNUM_HAL_CORTEXM_STM32_FLASH_WAIT_STATES`. It is important to verify and if necessary update this value if changing the CPU clock (HCLK) frequency or CPU voltage. Consult the relevant STM32 datasheets and programming manuals for the STM32L476 parts for appropriate values for different clock speeds or voltages. The default of 5 reflects a supply voltage in Vcore range 1 and HCLK of 120MHz.

Real-time characterization

The **tm_basic** kernel test gives statistics gathered about the real-time characterization and performance of the kernel. The sample output is shown here for information. The test was built for `SRAM` startup with optimization flag -O2, since it provides the best performance as both code and data could remain on-chip.

Example 310.1. stm32l4r9_disco Real-time characterization

```
Configured
Testing parameters:
   Clock samples:            32
   Threads:                  64
   Thread switches:         128
   Mutexes:                  32
   Mailboxes:                32
   Semaphores:               32
   Scheduler operations:    128
   Counters:                 32
   Flags:                    32
   Alarms:                   32
   Stack Size:             1088

             Startup, main thrd : stack used   356 size  1536
             Startup : Idlethread stack used    76 size  1280

eCos Kernel Timings
Notes: all times are in microseconds (.000001) unless otherwise stated

Reading the hardware clock takes 0 'ticks' overhead
... this value will be factored out of all other measurements
Clock interrupt took    7.09 microseconds (7 raw clock ticks)

Testing parameters:
   Clock samples:            32
   Threads:                  64
   Thread switches:         128
   Mutexes:                  32
   Mailboxes:                32
   Semaphores:               32
   Scheduler operations:    128
   Counters:                 32
   Flags:                    32
   Alarms:                   32
   Stack Size:             1088


                                 Confidence
     Ave     Min     Max     Var  Ave  Min  Function
```

```
  ======  ======  ======  ====== ========== ========
INFO:<Ctrl-C disabled until test completion>
    7.67    6.00   10.00    1.14   39%  45% Create thread
    1.50    1.00    2.00    0.50  100%  50% Yield thread [all suspended]
    1.44    1.00    2.00    0.49   56%  56% Suspend [suspended] thread
    1.34    1.00    2.00    0.45   65%  65% Resume thread
    2.00    2.00    2.00    0.00  100% 100% Set priority
    0.00    0.00    0.00    0.00  100% 100% Get priority
    3.67    3.00    4.00    0.44   67%  32% Kill [suspended] thread
    1.41    1.00    2.00    0.48   59%  59% Yield [no other] thread
    2.09    2.00    3.00    0.17   90%  90% Resume [suspended low prio] thread
    1.38    1.00    2.00    0.47   62%  62% Resume [runnable low prio] thread
    1.88    1.00    2.00    0.22   87%  12% Suspend [runnable] thread
    1.44    1.00    2.00    0.49   56%  56% Yield [only low prio] thread
    1.42    1.00    2.00    0.49   57%  57% Suspend [runnable->not runnable]
    3.72    3.00    4.00    0.40   71%  28% Kill [runnable] thread
    3.36    3.00    4.00    0.46   64%  64% Destroy [dead] thread
    6.25    6.00    7.00    0.38   75%  75% Destroy [runnable] thread
    6.64    6.00    8.00    0.48   60%  37% Resume [high priority] thread
    2.47    2.00    3.00    0.50   53%  53% Thread switch

    0.27    0.00    1.00    0.39   73%  73% Scheduler lock
    1.23    1.00    2.00    0.36   76%  76% Scheduler unlock [0 threads]
    1.23    1.00    2.00    0.36   76%  76% Scheduler unlock [1 suspended]
    1.26    1.00    2.00    0.38   74%  74% Scheduler unlock [many suspended]
    1.27    1.00    2.00    0.39   73%  73% Scheduler unlock [many low prio]

    0.31    0.00    1.00    0.43   68%  68% Init mutex
    1.63    1.00    2.00    0.47   62%  37% Lock [unlocked] mutex
    1.81    1.00    2.00    0.31   81%  18% Unlock [locked] mutex
    1.44    1.00    2.00    0.49   56%  56% Trylock [unlocked] mutex
    1.38    1.00    2.00    0.47   62%  62% Trylock [locked] mutex
    0.28    0.00    1.00    0.40   71%  71% Destroy mutex
    7.00    7.00    7.00    0.00  100% 100% Unlock/Lock mutex

    0.53    0.00    1.00    0.50   53%  46% Create mbox
    0.22    0.00    1.00    0.34   78%  78% Peek [empty] mbox
    1.84    1.00    2.00    0.26   84%  15% Put [first] mbox
    0.31    0.00    1.00    0.43   68%  68% Peek [1 msg] mbox
    2.00    2.00    2.00    0.00  100% 100% Put [second] mbox
    0.41    0.00    1.00    0.48   59%  59% Peek [2 msgs] mbox
    1.97    1.00    2.00    0.06   96%   3% Get [first] mbox
    1.84    1.00    2.00    0.26   84%  15% Get [second] mbox
    1.59    1.00    2.00    0.48   59%  40% Tryput [first] mbox
    1.50    1.00    2.00    0.50  100%  50% Peek item [non-empty] mbox
    1.56    1.00    2.00    0.49   56%  43% Tryget [non-empty] mbox
    1.50    1.00    2.00    0.50  100%  50% Peek item [empty] mbox
    1.38    1.00    2.00    0.47   62%  62% Tryget [empty] mbox
    0.38    0.00    1.00    0.47   62%  62% Waiting to get mbox
    0.06    0.00    1.00    0.12   93%  93% Waiting to put mbox
    0.41    0.00    1.00    0.48   59%  59% Delete mbox
    5.09    5.00    6.00    0.17   90%  90% Put/Get mbox

    0.19    0.00    1.00    0.30   81%  81% Init semaphore
    1.38    1.00    2.00    0.47   62%  62% Post [0] semaphore
    1.56    1.00    2.00    0.49   56%  43% Wait [1] semaphore
    1.41    1.00    2.00    0.48   59%  59% Trywait [0] semaphore
    1.34    1.00    2.00    0.45   65%  65% Trywait [1] semaphore
    0.31    0.00    1.00    0.43   68%  68% Peek semaphore
    0.28    0.00    1.00    0.40   71%  71% Destroy semaphore
    4.94    4.00    5.00    0.12   93%   6% Post/Wait semaphore

    0.56    0.00    1.00    0.49   56%  43% Create counter
    0.41    0.00    1.00    0.48   59%  59% Get counter value
    0.19    0.00    1.00    0.30   81%  81% Set counter value
    1.75    1.00    2.00    0.38   75%  25% Tick counter
    0.28    0.00    1.00    0.40   71%  71% Delete counter
    0.31    0.00    1.00    0.43   68%  68% Init flag
    1.53    1.00    2.00    0.50   53%  46% Destroy flag
    1.31    1.00    2.00    0.43   68%  68% Mask bits in flag
    1.50    1.00    2.00    0.50  100%  50% Set bits in flag [no waiters]
    2.06    2.00    3.00    0.12   93%  93% Wait for flag [AND]
    2.09    2.00    3.00    0.17   90%  90% Wait for flag [OR]
    2.06    2.00    3.00    0.12   93%  93% Wait for flag [AND/CLR]
    2.03    2.00    3.00    0.06   96%  96% Wait for flag [OR/CLR]
    0.22    0.00    1.00    0.34   78%  78% Peek on flag

    1.00    1.00    1.00    0.00  100% 100% Create alarm
    2.13    2.00    3.00    0.22   87%  87% Initialize alarm
    1.31    1.00    2.00    0.43   68%  68% Disable alarm
    2.25    2.00    3.00    0.38   75%  75% Enable alarm
    1.47    1.00    2.00    0.50   53%  53% Delete alarm
    1.91    1.00    2.00    0.17   90%   9% Tick counter [1 alarm]
    9.00    9.00    9.00    0.00  100% 100% Tick counter [many alarms]
    2.84    2.00    3.00    0.26   84%  15% Tick & fire counter [1 alarm]
   46.97   46.00   47.00    0.06   96%   3% Tick & fire counters [>1 together]
   10.25   10.00   11.00    0.38   75%  75% Tick & fire counters [>1 separately]
    6.00    6.00    6.00    0.00  100% 100% Alarm latency [0 threads]
    6.00    6.00    6.00    0.00  100% 100% Alarm latency [2 threads]
    5.77    5.00    6.00    0.36   76%  23% Alarm latency [many threads]
   10.01   10.00   11.00    0.01   99%  99% Alarm -> thread resume latency

    0.00    0.00    0.00    0.00            Clock/interrupt latency

    2.74    2.00    3.00    0.00            Clock DSR latency

     175     132     220                    Worker thread stack used (stack size 1088)
            All done, main thrd : stack used   704 size  1536
            All done : Idlethread stack used   172 size  1280

Timing complete - 29740 ms total

PASS:<Basic timing OK>
EXIT:<done>
```

Name

Test Programs — Details

Test Programs

The STM32L4R9-DISCO platform HAL contains some test programs which allow various aspects of the board to be tested.

ADC Test

The **adc1** program tests the ADC driver for the STM32. The test reports the values of the Vrefint, Vbat and Vts inputs that are sourced on-chip. The option `CYGBLD_HAL_CORTEXM_STM32L4R9_DISCO_TESTS_ADC` must be enabled to run this test since it needs human interaction.

Name

BootUp Integration — Detail

BootUp

The BootUp support for the STM32L4R9-DISCO target is primarily implemented in the `stm32l4r9_disco_support.c` file. The majority of the functions provided by that source file are only included when the `CYGPKG_BOOTUP` package is being used to construct the actual BootUp ROM loader binary.

The BootUp code is designed to be very simple, and it is envisaged that once its implementation has been tested and validated, the binary will only need to be installed onto a device once. Its only purpose is to allow the safe updating and startup of the main application. If the BootUp code ever needs to be replaced then it is a “factory” operation, for example using JTAG to re-program the on-chip flash.

This platform specific documentation should be read in conjunction with the generic BootUp package.

The BootUp package provides a basic but fully functional implementation for the platform. This has been tested to ensure that the underlying mechanism is sound. It is envisaged that the developer will customize and further extend the platform side support to meet their specific application update requirements.

BootUp loaded applications

Applications started via the BootUp loader, since they cannot include the `CYGPKG_BOOTUP` package themselves, may need access to some related configuration state. The platform is responsible for providing such “common” information. For example, the CDL option `CYGIMP_BOOTUP_RESERVED` specifies the amount of on-chip flash set aside for BootUp. Applications can then ensure that they do not interfere with the BootUp loader if using the remaining on-chip flash for their own purposes.

Warning

Care must be taken to ensure that the target application configuration matches the BootUp configuration, since it is normally expected that the applications to be loaded will be independent of the initial BootUp build environment. This includes the fundamental on-chip flash space set aside for the BootUp ROM loader code (`CYGIMP_BOOTUP_RESERVED`). It is expected that such values, for a particular platform instance, will be *fixed* at a suitable point during development, and definitely before products are shipped. It is the responsibility of the developer to ensure a consistent configuration between the BootUp ROM loader and any applications that may be installed/started by that BootUp code.

On-Chip ROMAPP applications

BootUp provides an alternative mechanism that supports the safe update of on-chip flash resident (`CYG_HAL_STARTUP_ROMAPP`) applications.

Updates using this mechanism are initiated and directed solely by the application itself. The application is responsible for locating, acquiring and verifying a new update, and placing it into Non-Volatile Memory (NVM) storage. If BootUp detects a verified update in NVM, it installs the update into the on-chip flash, overwriting and replacing the existing application. The updated application is then executed.

The example implementation uses a simple scheme that checks a fixed-format contiguous structure near the start of the binary application image file. Other than the fields used to identify the structure, the BootUp code does not interpret the hal_stm32l4r9_disco_bootup_structure_t in any other way.

Depending on how the alternative (pending update) application is downloaded and installed in the NVM, it may be more relevant to have the `tail` marker at the very end of the binary image. The developer may wish to update the build/release process so that the actual binary length is held in the application description structure, since that could avoid the overhead of unnecessary flash reads and writes when processing updates. Similarly, instead of a simple binary number being used to differentiate application images, the choice may be made to use the 64-bit UTC timestamp the application was created, or a human-readable string as the unique identification for a release. It is the responsibility of the build/release engineer to ensure individual releases are uniquely identifiable.

It is *critical* that the main application, when storing a pending update, stores the `tail` marker as the last bytes written. It is the responsibility of the main application to verify the data written, prior to placing the `tail` marker. This ensures that a partial image is not treated as a valid update. For example the sequence undertaken by the main application would be:

Table 310.1. Pending update sequence

Operation	Details
Invalidate “previous” alternative image	At a minimum ensure an invalid signature `tail` marker is written. Erasing the flash is normally required anyway, and would invalidate any previous image.
Receive update application image and write to alternative image location	NOT writing the `tail` marker. The code that stores the application should leave a “hole” where the `tail` marker resides to ensure a partial image is not incorrectly treated as valid
Verify downloaded contents	e.g. CRC or binary comparison. Normally this would be done as individual application chunks are downloaded and written to the alternative storage
Write `tail` marker	This is the very last operation after validating that the alternative image has been stored correctly. If an error has occured during the download then not-writing the `tail` ensures that the BootUp loader will not interpret the data written as a pending update
Force system RESET to start update	e.g. using the `HAL_PLATFORM_RESET` macro

The BootUp loader code will only READ from the alternative image location. This ensures that if an in-progress update is interrupted (e.g. power-loss) then when the system restarts the BootUp code will restart the application update as required.

If the BootUp platform implementation for validating the alternative image is extended to include a CRC, or similar “slow” processing, it may be worth considering whether the main application on startup will always invalidate the `tail` marker after an update to avoid subsequent system resets having to re-validate the alternative image prior to discovering that it is the same as the current main application.

Note

We cannot have the SIGNATURE support purely conditional on the BOOTUP support; since non-BOOTUP applications need to be built leaving the space. For the moment this is only enforced for ROMAPP applications, since that is all that the simple BootUp update support implements.

Building BootUp

The `ROM` startup type is chosen for BootUp so that the loader uses the on-chip SRAM for its workspace.

Example eCos configuration templates for BootUp are provided in the `misc` directory of the release. The `bootup_ROM.ecm` configuration file can be used to configure the BootUp loader.

Building a BootUp ROM image is most conveniently done at the command line. The steps needed to rebuild the ROM version of BootUp on linux are:

```
$ mkdir bootup_rom
$ cd bootup_rom
$ ecosconfig new stm32l4r9_disco minimal
[ … ecosconfig output elided … ]
$ ecosconfig import $ECOS_REPOSITORY/hal/cortexm/stm32/stm32l4r9_disco/current/misc/bootup_ROM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

The steps needed to rebuild the bundle based ROM version of BootUp on Windows within the Shell Environment are:

```
C:\Users\demo> mkdir bootup_rom
C:\Users\demo> cd bootup_rom
C:\Users\demo\bootup_rom> ecosconfig new stm32l4r9_disco minimal
[ … ecosconfig output elided … ]
C:\Users\demo\bootup_rom> ecosconfig import %ECOS_REPOSITORY%/hal/cortexm/stm32/stm32l4r9_disco/current/misc/bootup_ROM.ecm
C:\Users\demo\bootup_rom> ecosconfig resolve
C:\Users\demo\bootup_rom> ecosconfig tree
C:\Users\demo\bootup_rom> make
```

The resulting `install/bin/bootup.bin` binary can then be programmed into the on-chip flash from address `0x08000000`.

It is expected that the BootUp binary is installed onto the STM32L4S5VI on-chip flash either via JTAG or by utilising the on-chip `BootROM` USB based DFU process. This is a factory or in-field process requiring specific equipment/host-software.

Once BootUp is installed it is not normally expected to require updating. Its purpose is to bootstrap the main application, and provide a standard mechanism for installing the main application. The update mechanism does *NOT* provide a method for updating the BootUp loader itself. If in-field updates of the BootUp binary are necessary, this could be achieved via the STM32 on-chip `BootROM` USB based DFU process.

Altinit Test

This application is used to test BootUp support for updating a ROMAPP application from the QSPI flash. Since there is no Ethernet available on the board, this test uses the application already programmed in to the on-chip flash.

The test expects a ROMAPP application to be stored in the on-chip flash. It erases the alternate application in the QSPI flash, copies the ROMAPP application from on-chip flash into QSPI, and then invalidates the signature of the application in on-chip flash. When BootUp starts it will discover that the main application is invalid, copy the alternate application from the QSPI and then run it.

An example eCos configuration template for building the altinit application is provided in the `misc` directory. The steps needed to build this application are as follows:

```
$ mkdir altinit
$ cd altinit
$ ecosconfig new stm32l4r9_disco kernel
[ … ecosconfig output elided … ]
$ ecosconfig import $ECOS_REPOSITORY/hal/cortexm/stm32/stm32l4r9_disco/current/misc/altinit_SRAM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
$ make -C hal/cortexm/stm32/stm32l4r9_disco/current tests
```

Following this, the **altinit** executable can be found in `install/tests/hal/cortexm/stm32/stm32l4r9_disco/current/tests/altinit`. Running this will produce the following output:

```
INFO:<STM32L4_BOOTUP_ALTERNATIVE_OFFSET 00008000>
INFO:<STM32L4_BOOTUP_ALTERNATIVE_MAXLEN 000F8000>
INFO:<CYGNUM_BOOTUP_SIGNATURE_OFFSET 8>
INFO:<CYGNUM_BOOTUP_SIGNATURE_LENGTH 16>
INFO:<Erasing MX25L128 area for alternative image>
INFO:<Copy main image to alt>
INFO:<Set signature in alt image>
INFO:<Invalidate main image>
INFO:<Update done>
```

The erase and copy operations here may take a some time, do not reset the board until the program is finished.

Chapter 311. STM32L4R9-EVAL Platform HAL

This documentation is still under development.

Chapter 312. NXP i.MX RT10XX Variant HAL

Name

`CYGPKG_HAL_CORTEXM_IMX` — eCos Support for the NXP i.MX RT10XX Microprocessor Family

Description

The NXP i.MX RT10XX series of Cortex-M microcontrollers is supported by eCos with an eCos processor variant (VAR) HAL and a number of device drivers supporting some of the on-chip peripherals. These include device drivers for the on-chip flash, serial, I²C, SPI, Ethernet, CAN, PWM and watchdog devices. In addition it provides common functionality and definitions that RT10XX based platform ports may require, as well as definitions useful to application developers. Throughout this document this processor family will just be referred to as *i.MX* or *IMX*, without any numerical designations.

This documentation covers the i.MX functionality provided but should be read in conjunction with the specific HAL documentation for the platform port. That documentation will cover issues that are platform-specific and are not covered here, and may also describe differences that override or supersede what the IMX variant HAL provides. The areas that are specific to platform HALs and not the IMX variant HAL include:

- memory map and related configuration and setup
- Clock parameters
- Pin multiplexing and GPIO setup
- Any special handling for external interrupts, or additional interrupts
- Diagnostic I/O baud rates
- Additional diagnostic I/O devices, if any
- LED/LCD control

Name

On-chip Subsystems and Peripherals — Hardware Support

Hardware support

The IMX family contains many on-chip peripherals.

On-chip memory

The IMX parts include on-chip SRAM (`OCRAM`). The SRAM can vary in size from as little as 4KiB to 1MiB. Support is also available for external FLASH and SDRAM.

Typically, an eCos platform HAL port will expect a standalone application image to be programmed into boot memory, and the board would boot this image from reset.

Note

The i.MX RT ROM bootloader will only load and execute SRAM (OCRAM) or memory-mapped flash applications. If your final application is linked for SRAM then creating a bootable image as described in the relevant platform specific documentation is sufficient. For applications that are linked to execute from external memory (e.g. SDRAM) then the first stage of booting an application will be via execution of a second-level boot loader application started by the ROM bootloader. For eCos this second-level is normally BootUp (`CYGPKG_BOOTUP`) or RedBoot (`CYGPKG_REDBOOT`), though it can just as well be a custom application if required.

The i.MX RT ROM bootloader will parse a table describing the initial I/O setup and memory destination for the application image (copying the image if required). This table is known as the DCD (Device Configuration Data).

For embedded development it is normally expected that H/W debugging (JTAG/SWD) via a suitable adapter is used to aid application development of standalone applications. For standalone applications (no dependency on a ROM monitor) there is **no** application binary difference whether the application is started from the i.MX RT ROM bootloader process or loaded via a H/W debugger interface; with the exception that the H/W debugger may need to perform the same actions as the DCD table prior to loading an executable to ensure the MCU setup is the same as execution via the ROM bootloader.

An alternative to H/W debugging using an external adapter is for RedBoot to be installed as the application started by the IMX boot process. RedBoot provides GDB stub functionality so it is then possible to download and debug stand-alone and eCos applications via the gdb debugger using serial interfaces or other debug channels. If RedBoot is present it may also be used to manage the external flash memory. For RedBoot based production purposes, applications are programmed into external FLASH and may be self-booting. Applications may also be loaded into memory using a RedBoot startup script. RedBoot has a memory and performance impact on applications if they are configured to use RedBoot provided functionality. The use of the simple BootUp world instead of RedBoot means that there is no performance or memory impact on the final **standalone** application started by the boot process.

Cache Handling

The RT10XX variants contain an instruction and data cache controller defined as part of the Cortex-M architectural specification. Support for this controller is supplied by the architecture HAL.

The variant HAL provides a mechanism for setting aside a, configurable size, block of `OCRAM` as uncached for device driver and DMA usage.

OCOTP (On-Chip One-Time-Programmable) fuses

The HAL provides for basic `OCOTP` operations. Details are supplied later.

Serial I/O

The IMX variant HAL supports basic polled HAL diagnostic I/O over any of the on-chip serial devices. There is also a fully interrupt-driven serial device driver suitable for eCos applications for all on-chip serial devices. The serial driver consists of an eCos package: `CYGPKG_IO_SERIAL_NXP_LPUART` which provides support for the IMX on-chip serial devices. Using the HAL diagnostic I/O support, any of these devices can be used by the ROM monitor or RedBoot for communication with GDB. If a device is needed by the application, either directly or via the serial driver, then it cannot also be used for GDB communication using the HAL I/O support. An alternative serial port should be used instead.

The HAL defines CDL interfaces, `CYGINT_HAL_CORTEXM_IMX_UART1` to `CYGINT_HAL_CORTEXM_IMX_UART8` for each of the available UARTs. The platform HAL CDL should contain an **implements** directive for each such UART that is available for use on the board. This will enable use of the UART for diagnostic use.

Interrupts

The IMX HAL relies on the architectural HAL to provide support for the interrupts directly routed to the NVIC. The `cyg/hal/var_intr.h` header defines the vector mapping for these.

Pin Multiplexing and GPIO

The variant HAL provides support for packaging the pin, GPIO and daisy chain configurations of an external line into 32-bit descriptors that can then be used with macros to configure the pin and set and read its value. Details are supplied later.

Ethernet

eCos includes a device driver for the on-chip `ENET` Ethernet controllers. This is located in the package `CYGPKG_DEVS_ETH_FREESCALE_ENET` (“Freescale ENET ethernet driver”).

This variant HAL provides some common support for `ENET` interfaces via the initialisation helper function `CYGHWR_FREESCALE_ENET_PLATFORM_INIT(base)` and the IEEE MAC acquisition function `CYGHWR_FREESCALE_ENET_PLATFORM_MAC(base, p_enaddr)`.

Note

The `CYGHWR_FREESCALE_ENET_PLATFORM_MAC()` will attempt to use the `OCOTP` fuses set aside for holding MAC addresses, but from the factory these are unset. In which case for the first `ENET` controller the code will fall back to providing a MAC address based on the `OCOTP` unique ID field.

It is the responsibility of the developer to implement any factory production steps required to fuse a unique MAC address into the relevant `OCOTP` locations as required.

CAN

eCos includes a device driver for the on-chip `FlexCAN` controllers. This is located in the package `CYGPKG_DEVS_CAN_NXP_FLEXCAN` (“NXP FlexCAN driver”).

Watchdog

eCos includes device drivers for the on-chip watchdog in the IMX family. This is located in the package `CYGPKG_DEVICES_WALLCLOCK_NXP` ("NXP wallclock driver").

Clock Control

The platform HAL must provide the input oscillator frequency (`CYGHWR_HAL_CORTEXM_IMX_OSC_MAIN`) in its CDL file. Under normal circumstances this is set to 24MHz.

The actual values of the clock frequencies, is stored in a global structure, `hal_imx_clock`, which contains the PLL frequencies, PFDs and root clocks. The clock supplied to the SysTick timer is also assigned to `hal_cortexm_systick_clock`. These variables are set by examining the actual hardware register so they reflect settings made by any bootloader or JTAG adaptor.

Note that when changing or configuring any of these clock settings, you should consult the relevant processor datasheet as there may be both upper and lower constraints on the frequencies of some clock signals, including intermediate clocks. There are also some clocks where, while there is no strict constraint, clock stability is improved if values are chosen wisely. Finally, be aware that increasing clock speeds using this package may have an effect on platform specific properties, such as memory timings which may have to be adjusted accordingly.

Name

Hardware Configuration Support on IMX Processors — Details

Synopsis

#include <cyg/hal/hal_io.h>

`cyg_uint32 desc = CYGHWR_HAL_IMX_PAD(pad, mode);`

`cyg_uint32 desc = CYGHWR_HAL_IMX_SNVS_PAD(pad, mode);`

`CYGHWR_HAL_IMX_PAD_SET (desc);`

`cyg_uint32 desc = CYGHWR_HAL_IMX_DAISY(reg, value);`

`CYGHWR_HAL_IMX_DAISY_SET (desc);`

`cyg_uint32 desc = CYGHWR_HAL_IMX_CLOCK_GATE(reg, gate);`

`CYGHWR_HAL_IMX_CLOCK_ENABLE (desc);`

`CYGHWR_HAL_IMX_CLOCK_DISABLE (desc);`

`cyg_uint32 desc = CYGHWR_HAL_IMX_GPIO(ctlr, pin, mode);`

`CYGHWR_HAL_IMX_GPIO_SET (desc);`

`CYGHWR_HAL_IMX_GPIO_OUT (desc, val);`

`CYGHWR_HAL_IMX_GPIO_IN (desc, val);`

`CYGHWR_HAL_IMX_GPIO_INTSTAT (desc, status);`

`CYGHWR_HAL_IMX_GPIO_INTMASK (desc, enable);`

`CYGHWR_HAL_IMX_GPIO_INTCLR (desc);`

Description

The IMX HAL provides a number of macros to support the encoding of various hardware configuration options into a 32-bit descriptor. This is useful to drivers and other packages that need to configure the hardware.

PAD Multiplexing

Many of the IO pads on the chip can be connected to a variety of different devices and have a variety of properties that need to be configured. A standard pad descriptor is created by the `CYGHWR_HAL_IMX_PAD()` macro. For the `SNVS` pads connected to `GPIO5` the `CYGHWR_HAL_IMX_SNVS_PAD()` is used instead. The `pad` argument defines the pad to be configured, and follows the hardware naming convention, for example `AD_B0(12)` or `SD_B1(3)`. The `mode` argument is the terminal part of a `CYGHWR_HAL_IMX_PAD_MODE_XXXX` macro; this may be one supplied in the HAL, or one constructed by the driver or application. This macro is defined as a combination of the `CYGHWR_HAL_IMX_PAD_MODE_` definitions in `cyg/hal/var_io.h`, which correspond to the definitions in the hardware pad MUX and CTL registers.

The macro `CYGHWR_HAL_IMX_PAD_SET()` is used to configure a pad according to the descriptor.

The following example shows how the LPUART pads are configured.

```
#define CYGHWR_HAL_IMX_PAD_MODE_LPUART_PAD(__alt)       \
        CYGHWR_HAL_IMX_PAD_MODE_MUX(__alt)      |       \
        CYGHWR_HAL_IMX_PAD_MODE_PUE             |       \
        CYGHWR_HAL_IMX_PAD_MODE_PKE             |       \
        CYGHWR_HAL_IMX_PAD_MODE_PUS_47K_PU      |       \
        CYGHWR_HAL_IMX_PAD_MODE_DSE(6)          |       \
        CYGHWR_HAL_IMX_PAD_MODE_SPEED_MED2

#define CYGHWR_HAL_IMX_LPUART1_TX       CYGHWR_HAL_IMX_PAD( AD_B0(12), LPUART_PAD(2) )
#define CYGHWR_HAL_IMX_LPUART1_RX       CYGHWR_HAL_IMX_PAD( AD_B0(13), LPUART_PAD(2) )
```

PAD Daisy Chaining

Some device IO lines can connect to multiple pads, the daisy chain registers select which pad is to be used for the device. A daisy chain descriptor is created by the CYGHWR_HAL_IMX_DAISY macro. The reg argument selects which daisy chain device line is to be programmed, and the val argument defines the pad selection. Both of these arguments are fragments of macros defined in cyg/hal/var_io.h. Only registers and values currently used are defined there, new ones can be added there or defined in the driver or application.

The CYGHWR_HAL_IMX_DAISY_SET macro is called to program the configuration from a descriptor into the hardware.

The following example shows some daisy chain descriptor definitions.

```
#  define CYGHWR_HAL_NXP_LPUART3_TXD_DAISY CYGHWR_HAL_IMX_DAISY(LPUART3_TX, AD_B1_06_ALT2)

#  define CYGHWR_HAL_NXP_LPUART3_RXD_DAISY CYGHWR_HAL_IMX_DAISY(LPUART3_RX, AD_B1_07_ALT2)

#  define CYGHWR_HAL_NXP_LPUART3_CTS_DAISY CYGHWR_HAL_IMX_DAISY(LPUART3_CTS_B, AD_B1_04_ALT2)
#  define CYGHWR_HAL_NXP_LPUART3_RTS_DAISY CYGHWR_HAL_IMX_DAISY_NONE
```

Clock Gating Control

Most device clocks are controlled by a gate that needs to be switched on or off. The HAL provides macros which may be used to enable or disable peripheral clocks. Effectively this indicates whether the peripheral is powered on (enabled) or powered down (disabled), and so may be used to ensure unused peripherals are turned off to save power. The macro CYGHWR_HAL_IMX_CLOCK_GATE() defines a clock gate descriptor. The reg argument gives the CCGR register to be used and the gate argument selects the clock gate bit in that register. Clock gate descriptors are defined in cyg/hal/var_io.h and new values will be added there as needed.

The macros CYGHWR_HAL_IMX_CLOCK_ENABLE() and CYGHWR_HAL_IMX_CLOCK_DISABLE() enable and disable the clock described by the descriptor. At present clocks are either fully on or fully off, the option to switch clocks off in WAIT mode is not implemented.

It is important to remember that before a peripheral can be used, it must be enabled. It is safe to re-enable a peripheral that is already enabled, although usually a device driver will only do so once in its initialization. eCos will automatically initialize some peripheral blocks where it needs to use the associated peripherals, and in eCos-supplied device drivers which are included in the eCos configuration. However this should not be relied on - it is always safest to enable the peripheral clocks anyway just in case.

GPIO

A descriptor is created by the CYGHWR_HAL_IMX_GPIO() macro. The ctlr argument selects the GPIO controller while the pin argument selects the GPIO pin in the controller. The mode argument configures the pin within the GPIO controller.

For basic I/O the supplied mode is either IN or OUT. For input pins an interrupt configuration mode can us used instead (with IN implied). At present the options LOW_LEVEL, HIGH_LEVEL, RISING_EDGE, FALLING_EDGE and BOTH_EDGES are available to define which input transitions/states will generate an interrupt event.

For example, the descriptor for GPIO output control of GPIO1 pin 9 could be declared as follows:

```
#define OUTPUT_EXAMPLE CYGHWR_HAL_IMX_GPIO(1, 9, OUT)
```

Correspondingly, a descriptor for polled input of GPIO5 pin 0 can simply be declared using IN for the mode field:

```
#define INPUT_EXAMPLE CYGHWR_HAL_IMX_GPIO(5, 0, IN)
```

For polled **or** interrupt driven input the relevant interrupt detection can be specifiued as the mode field. e,g:

```
#define INPUT_EXAMPLE CYGHWR_HAL_IMX_GPIO(5, 0, BOTH_EDGES)
```

In addition to GPIO configuration, the matching pad will need to be configured using a PAD descriptor as detailed above in PAD Multiplexing.

Prior to a pin being accessed for GPIO then the pin **and** its corresponding pad will need to be configured at run-time. With appropriate descriptor values defined this is done via the SET calls. For example, using the manifests from above:

```
   // One-time initialisation of output pin:
   CYGHWR_HAL_IMX_PAD_SET(OUTPUT_PAD);
   CYGHWR_HAL_IMX_GPIO_SET(OUTPUT_EXAMPLE);

   // One-time initialisation of input pin:
   CYGHWR_HAL_IMX_PAD_SET(INPUT_PAD)
   CYGHWR_HAL_IMX_GPIO_SET(INPUT_EXAMPLE);
```

Subsequent to the setting of the I/O configuration the GPIO pin can then be accessed as configured.

If a GPIO pin has been configured as an output then its value may be set using CYGHWR_HAL_IMX_GPIO_OUT(). For example:

```
   CYGHWR_HAL_IMX_GPIO_OUT(OUTPUT_EXAMPLE, 0); // set LOW
   some_app_processing();
   CYGHWR_HAL_IMX_GPIO_OUT(OUTPUT_EXAMPLE, 1); // set HIGH
```

Similarly the current value of an input pin can be read using CYGHWR_HAL_IMX_GPIO_IN(). For example:

```
   int bstat;
   CYGHWR_HAL_IMX_GPIO_IN(INPUT_EXAMPLE, &bstat);
```

Note

Normally the variant HAL will manage the masking and acknowledgment of interrupts via the standard kernel interrupt support API, and so the application level code does not normally need to access the low-level INTMASK and INTCLR functions directly since they will be called by the kernel as appropriate.

For completeness the low-level helpers exposed to the standard interrupt support are documented here.

The CYGHWR_HAL_IMX_GPIO_INTMASK() parameter enable controls the masking/unmasking of the interrupt associated with the pin descriptor. A non-zero value will enable the source, with 0 disabling the source.

If required, the current active “interrupt asserted” state of a pin can be interrogated using CYGHWR_HAL_IMX_GPIO_INTSTAT().

The CYGHWR_HAL_IMX_GPIO_INTCLR function can be used to acknowledge an active interrupt.

Since most of the individual GPIO pin interrupt sources are multiplexed through the Cortex-M NVIC the variant HAL provides support for demultiplexing the combined GPIO interrupts to individual logical vectors. The CDL option CYGSEM_HAL_CORTEXM_IMX_GPIO_INT_DEMUX controls whether the demux feature is implemented. It is enabled by default when the CYGPKG_KERNEL is configured. This feature avoids application code having to manage their own demux support when interrupt support is required for multiple pins on a specific GPIO controller.

For example, the physical GPIO2 pin 5 is actually multiplexed via the hardware CYGNUM_HAL_INTERRUPT_GPIO2_COMBO_0_15 interrupt vector. If the HAL demux support is enabled then that specific pin source can be supported individually via the CYGNUM_HAL_INTERRUPT_GPIO2_INT5 logical vector manifest, with the other 15 sources multiplexed onto the underlying NVIC interrupt also available via their own logical vector numbers.

Name

OCOTP Support on IMX Processors — Details

Synopsis

#include <cyg/hal/hal_io.h>

cyg_bool success = **cyg_imx_ocotp_timing**();

cyg_bool success = **cyg_imx_ocotp_read**(*addr*, *p_value*);

cyg_bool success = **cyg_imx_ocotp_write**(*addr*, *value*, *key*);

cyg_imx_ocotp_reload();

Description

The IMX HAL provides a number of functions to support interaction with the `OCOTP` (On-Chip One-Time-Programmable) fuses.

OCOTP Initialisation

Prior to accessing the `OCOTP` the function `cyg_imx_ocotp_timing()` should be used to configure the required timing setup. If the function returns `false` then an error has occurred and the `OCOTP` API should not be used.

OCOTP Reading

The `addr` is constructed using the `CYGHWR_HAL_IMX_HW_OCOTP_ADDR(bank, word)` macro to convert a bank# and word# into a register offset.

The passed `p_value` parameter is then filled with the `OCOTP` fuse value corresponding to the passed `addr`.

OCOTP Writing

Note

Due to the fact that `OCOTP` updates can affect how the CPU operates from boot, the writing of fuses (setting from `0` to `1`) is **disabled** by default.

If `OCOTP` write support is required then the CDL option `CYGHWR_HAL_CORTEXM_IMX_OCOTP_WRITE` should be explicitly enabled.

When the `OCOTP` write support is configured then the function `cyg_imx_ocotp_write()` can be used to set a specific `value` for the specified fuse location. The `key` parameter is an extra safety measure since the write will only be performed if `key` is `CYGHWR_HAL_IMX_HW_OCOTP_WRITE_KEY`. This is to minimise the chance for errant execution of the write function due to some application flaw resulting in erroneous `OCOTP` updates. The source code is constructed so that the 32-bit key value does **not** appear in the binary. Application users of the `OCOTP` write support should ideally implement similar functionality. e.g.

```
static const cyg_uint32 keyth = (0xDEAD0000 | (CYGHWR_HAL_IMX_HW_OCOTP_WRITE_KEY >> 16));
static const cyg_uint32 keybh = (0xC0DE0000 | (CYGHWR_HAL_IMX_HW_OCOTP_WRITE_KEY & 0xFFFF));

cyg_uint32 bitshigh;
cyg_uint32 bitslow;

HAL_READ_UINT32(&keyth, bitshigh);
HAL_READ_UINT32(&keybh, bitslow);
```

```
cyg_uint32 writekey = (((bitshigh & 0xFFFF) << 16) | (bitslow & 0xFFFF));

if (cyg_imx_ocotp_write(CYGHWR_HAL_IMX_HW_OCOTP_ADDR(bank, word), bitmask, writekey)) {
  // success // OCOTP updated
} else {
  // failure
}
```

After updating the fuses the `cyg_imx_ocotp_reload()` **should** be called to reload the shadow registers. This ensures other code, that may directly access the shadow state, will see the updated fuse state.

Name

BootUp second level boot loader — Bootstrap

BootUp

BootUp (CYGPKG_BOOTUP) is a lightweight second-level boot loader package, which can be extended with VAR and PLF specific features as required (e.g. firmware updates, secure boot, etc.).

The BootUp support for the i.MX RT targets is primarily implemented in the imx_support.c file. The functions are normally only included when the CYGPKG_BOOTUP package is being used to construct the actual BootUp loader binary.

The BootUp code is designed to be very simple, and it is envisaged that once an implementation has been defined the binary will only need to be installed onto a device once.

Alternative

An alternative to using BootUp to start applications is to use a SRAM (OCRAM) based RedBoot to manage the flash space, and to use the RedBoot features to load and start the main application.

The benefit of RedBoot is the ability for command-line interaction with the management of the stored application. The downside of RedBoot is the code+data space overhead and potential for run-time performance costs when using (via the virtual vector interface) RedBoot features or debugging an application via RedBoot. The BootUp package offers a light weight solution without impacting the memory space or performance of the final application, and is better suited to a H/W debugger development environment.

Boot details

Currently, its only purpose is to allow the startup of the main application when linked for external memory (e.g. SDRAM) since the i.MX RT ROM Bootloader will only load and execute applications to SRAM (OCRAM). If an SRAM based application is to be booted then the BootUp intermediate (second-level) code is not needed since the CPU can directly load and start the SRAM application from the bootable NVM.

This platform specific documentation should be read in conjunction with the generic BootUp package documentation.

The BootUp package provides a basic but fully functional implementation for the platform. It is envisaged that the developer will customize and further extend the platform side support to meet their specific application identification and update requirements.

The BootUp binary iself can be installed on *any* i.MX RT bootable media, and is not restricted to being placed into FlexSPI flash.

On execution BootUp will copy the configured final application from its Non-Volatile-Memory (NVM) location to its destination address and start the application. The configuration option CYGIMP_BOOTUP_IMX_SOURCE selects where the second-level BootUp code will look for the final application image. The available options depend on the configured target CPU, and whether the variant/platform specific BootUp has support for the specific storage medium.

The following diagrams give an overview of the first-level (on-chip) ROM Bootloader and second-level (SRAM) boot sequence. Blocks shown in green indicate running code. The blue arrows indicate a memory copy operation. The red arrows indicate a switch in execution. The BootUp application image stored in the bootable NVM is expected to be a valid image prefixed with FCFB, BootData, DCD and IVT information as required for the on-chip ROM Bootloader. This example is for the final application binary to be executed from SDRAM.

Figure 312.1. On-chip ROM Bootloader executes

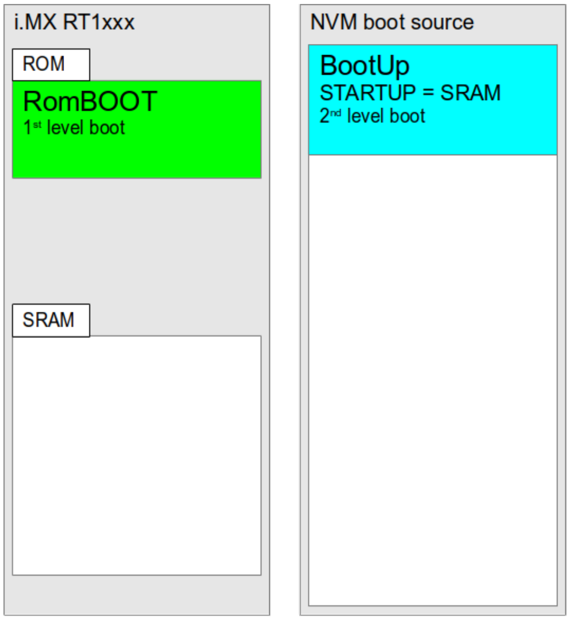


Figure 312.2. On-chip ROM Bootloader copies second-level boot code from NVM to on-chip SRAM

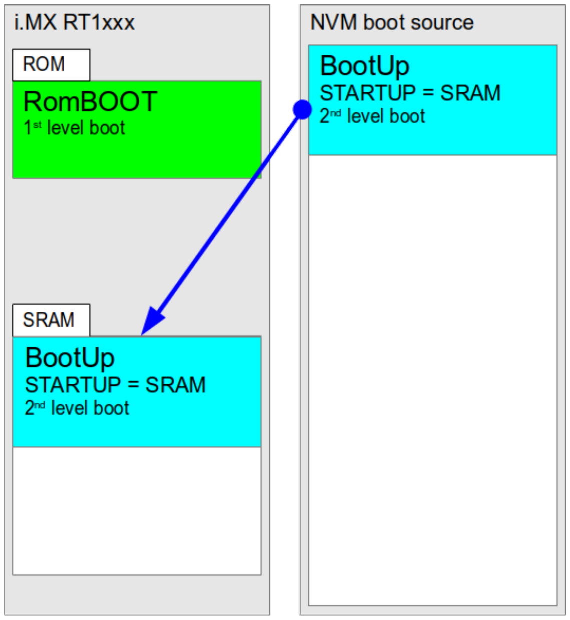


Figure 312.3. SRAM loaded second-level boot code is executed

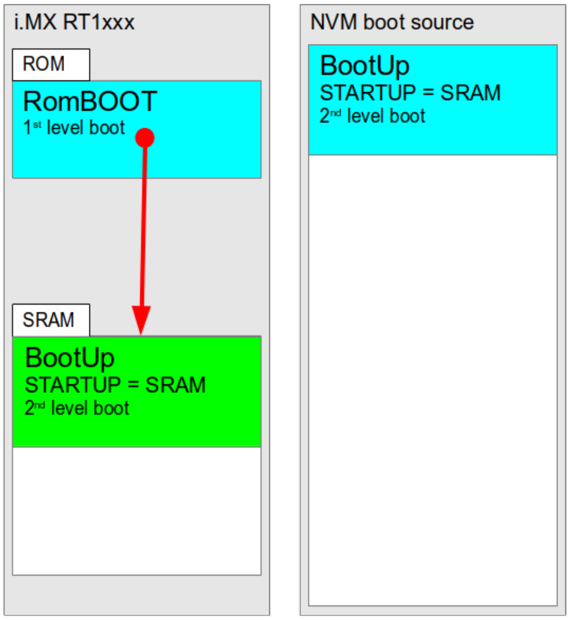


Figure 312.4. Final application is located in NVM

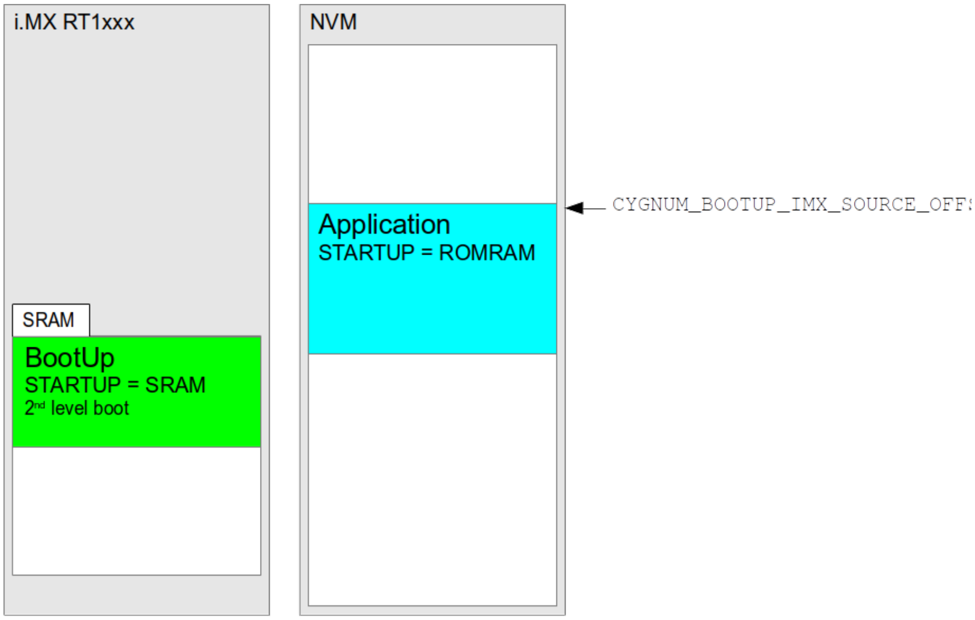


Figure 312.5. Second-level boot copies application from NVM to SDRAM

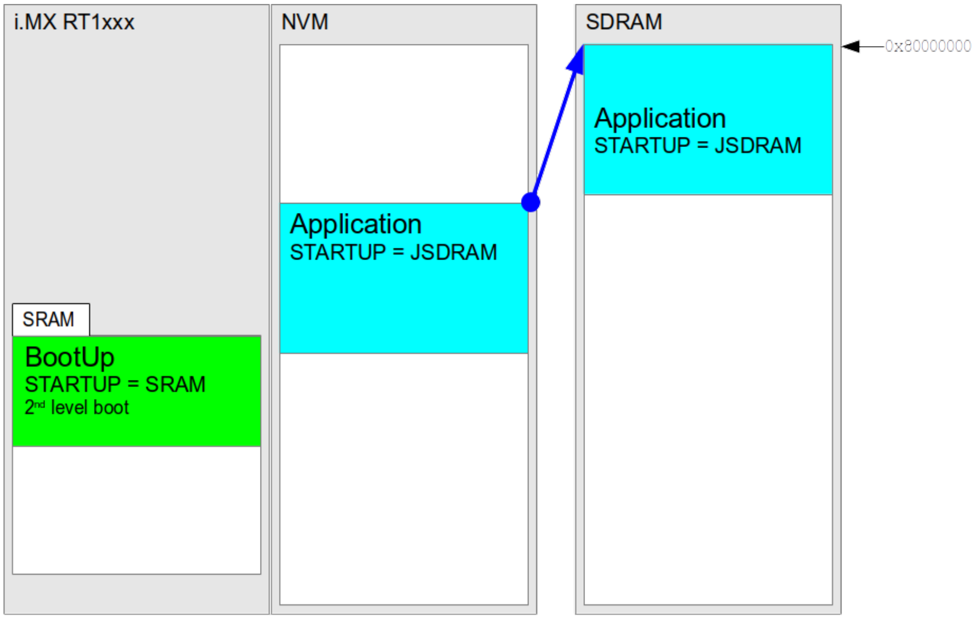
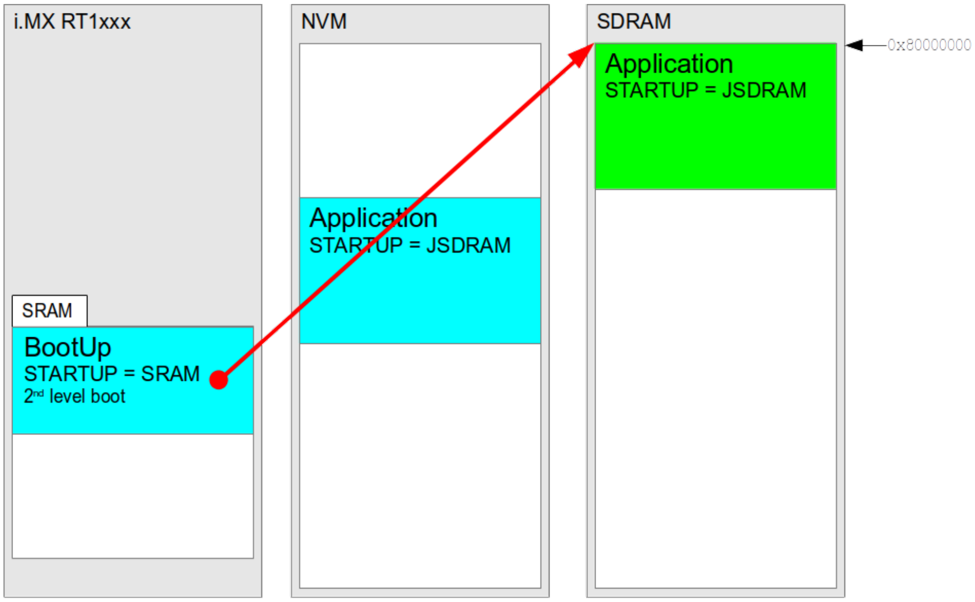


Figure 312.6. Application is started



The example above shows an application with the platform startup type `JSDRAM`, but the application could also be the special-case `RBRAM` startup type for a RAM based RedBoot image, or some other platform specific SDRAM or other external-memory startup type as required.

Building BootUp

Building a BootUp loader image is most conveniently done at the command line. The steps needed to rebuild the `SRAM` version of BootUp are:

```
$ mkdir bootup_SRAM
$ cd bootup_SRAM
$ ecosconfig new TARGET minimal
[ … ecosconfig output elided … ]
$ ecosconfig import $ECOS_REPOSITORY/packages/hal/cortexm/imx/var/current/misc/bootup_SRAM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

Where *TARGET* is replaced with the required i.MX target platform name, e.g. mimxrt1064_evk, mimxrt1050_evk, etc.

The resulting install/bin/bootup.bin binary can then be packaged into a bootable image (see platform specific documentation), with that bootable image subsequently programmed into a suitable non-volatile memory as supported by the i.MX RT on-chip ROM Bootloader. To be clear, the BootUp binary itself needs to be wrapped with the descriptors required by the i.MX RT ROM Bootloader **but** the binary started by BootUp is just the unmodified final application binary.

The example bootup_SRAM.ecm is configured to expect to find the application stored in the selected FlexSPI flash from the address offset CYGNUM_BOOTUP_IMX_SOURCE_OFFSET.

The option CYGIMP_BOOTUP_IMX_SOURCE selects the NVM location used by BootUp. Currently this is limited to the FlexSPI flash interface(s), and will default to the controller instance used by the ROM Bootloader to match the device where the second-level boot loader is stored.

For convenience, prebuilt wrapped BootUp images named qspi_bootup.bin are provided with eCosPro releases in the loaders/*target* subdirectory of your eCosPro installation alongside the unwrapped BootUp images as well as various RedBoot images.

Application Signature

The BootUp loader will only copy and start execution of binaries with a suitable “signature” block. For the i.MX targets the location of the descriptor block is fixed at the offset of 8-bytes within the binary image (CYGNUM_BOOTUP_SIGNATURE_OFFSET).

The binary image descriptor provides “magic” identifier values used to detect a complete signature block, and the descriptor holds the necessary information required by BootUp. e.g. a length field specifying the number of bytes in the binary image.

If the binary image at offset CYGNUM_BOOTUP_IMX_SOURCE_OFFSET is not valid then BootUp will enter the hal_imx_badapp() function. The VAR HAL provides a default implementation which is a simple infinite loop, but the weak VAR HAL definition can be overridden by a PLF specific implementation if required.

Chapter 313. NXP MIMXRT1xxx-EVK Platform HAL

Name

CYGPKG_HAL_CORTEXM_MIMXRT1XXX_EVK — eCos Support for the IMXRT1050-EVKB and MIMXRT1064-EVK boards

Description

This document covers the configuration and usage of eCos on the NXP MIMXRT10XX-EVK evaluation kits. This includes the RT1052 and RT1064 on which this support has been tested. Boards containing other devices in the RT10XX family should be supportable with minimal effort.

For typical eCos development it is expected that programs will be downloaded and debugged via a hardware debugger (JTAG/SWD) attached to either the standard ARM 20-pin JTAG connector or via the on-board CMSIS-DAP USB socket. Use of a hardware debugging interface avoids the requirement for (and cost overheads of) a debug monitor application to be present on the platform.

Note

As shipped the EVK boards are configured for H/W debugging via the onboard CMSIS-DAP interface. If the standard ARM 20-pin 2.54mm IDC J21 connector is to be used the jumpers may need to be re-configured. For example, the MIMXRT1064-EVK requires jumpers J47..J50 to be disconnected. The relevant NXP schematic or board documentation should be referenced to correctly configure the target board if using an external H/W debugger connected via J21.

The Startup section below gives an overview of the various STARTUP types that can be configured. The selected STARTUP type defines where the final application binary will be linked to execute from, as well as control some run-time features of the final application (e.g. standalone or dependant on a ROM monitor, etc.).

Supported Hardware

The variant HAL includes support for the on-chip serial devices which are documented in the variant HAL. LPUART1 is connected to the CMSIS socket where it is available as a CDC/ACM interface. There is no support for hardware flow control in this device. LPUART1 is configured as the default diagnostics console.

Device drivers provide support for the I²C interfaces, which are instantiated by the platform (PLF) HAL. These have been tested using external I²C devices.

Device drivers provide support for the SPI interfaces, which are instantiated by the platform HAL. These have been tested using external SPI devices.

Support is available for the FlexSPI controller(s) with attached QSPI device(s). In the case of the MIMXRT1064-EVK board this support is provided for the SiP QSPI device as well as the external ISSI IS25WP064D QSPI device. For the IMXRT1050-EVKB the FlexSPI support requires board modifications to use the on-board QSPI ISSI IS25WP064D device in place of the default HyperFlash device.

Normally the first few blocks of the bootable QSPI NOR flash are set aside for a bootable application image (e.g. standalone SRAM application, 2nd-level boot loader, or a (RBSRAM) RedBoot debug monitor image). The bootable application image requires specific descriptor structures to be prefixed to allow the i.MX RT ROM Bootloader to start the application, as documented in the flashimg_rt10 section. When using RedBoot (either a directly booted RBSRAM build, or a RBRAM build loaded via BootUp or a RBSRAM RedBoot), the last couple of blocks of the flash device are used to hold the RedBoot **fconfig** and FIS information. The remaining blocks are free for use by application code. When not using RedBoot all of the flash beyond the i.MX RT ROM Bootloader required (DCD+image) is available for application use.

Warning

The IMXRT1050-EVKB and MIMXRT1064-EVK boards ship with J1 configured to power the board via the CMSIS-DAP debug connector. Unfortunately, depending on the host connection, that connection is insufficient to power the

USB and CAN tranceivers, and the Ethernet PHY, such that USB, CAN and Ethernet are **unlikely** to operate correctly when the board is powered via that debug USB connection.

It is **important** to power the board via an external PSU via J2, with the J1 pins 1-2 connected as appropriate if CAN, USB or Ethernet are to be used.

Support for USB host and peripheral mode is provided by EHCI drivers. Host mass storage and CDC-ACM class drivers are present. The peripheral port is configured by default to provide a CDC-ACM device.

Support for Ethernet and CAN (**not** CAN-FD) is provided by configuring the respective device drivers.

Note

The `IMXRT1050-EVKB` and `MIMXRT1064-EVK` boards seen to date do **not** have the CAN connector J11 populated. So a suitable connector will need to be soldered to provide access to the on-board CAN transceiver.

Tools

The board port is intended to work with GNU tools configured for an **arm-eabi** target. The original port was done using **arm-eabi-gcc** version 7.3.0e, **arm-eabi-gdb** version 8.1, and **binutils** version 2.30.

`flashimg_rt10`

The i.MX RT ROM Bootloader will only bootstrap applications prefixed with headers describing the application to be started. A suitable bootable image can be created using the `flashimg_rt10` tool to prefix the eCos executable binary with the required descriptors.

The tool will by default create an image suitable to be booted from an SDcard.

To create a QSPI boot image from a binary image the `--qspi` command-line option should be used:

```
$ flashimg_rt10 --qspi image.bin qspi_boot.bin
```

Note

Only binaries linked to execute from the On-Chip SRAM are supported as bootable applications.

Application Boot Location Examples

For the `mimxrt1050_evk` and `mimxrt1064_evk` targets the code executed at boot can be loaded from flash using the ROM Bootloader FlexSPI NOR flash support. Refer to the NXP H/W board documentation and relevant i.MX RT Processor Reference Manual for details on the available boot-from-reset configurations.

The following section just highlights where images are stored in the FlexSPI NOR for some, common, example startup configurations.

Note

The following examples are **not** exhaustive. For example, a custom second-level bootloader can be used to start an application. These are just common examples.

For FlexSPI configured boots the `mimxrt1050_evk` target only has the FlexSPI1 flash mapped from address `0x60000000`. However, the FlexSPI boot option for the `mimxrt1064_evk` can only boot from the FlexSPI2 (SiP) flash mapped from address

`0x70000000`, but any second-level boot loader or application started has access to the external FlexSPI1 flash mapped from `0x60000000`.

If an application binary is linked for SDRAM execution then it needs to be loaded via a second-level boot loader. Only the second-level bootloader (as started by the i.MX RT ROM Bootloader) needs to be created using the `flashimg_rt10` tool. The final application does not need to be converted. When using BootUp as the second-level boot loader the SDRAM application stored in NVM will be a simple, raw, binary located at a **fixed** location. When using RedBoot as the second-level loader the application image stored in the NVM will be the larger ELF image, with FIS controlling where the image is located.

In the following figures **BLUE** is used to indicate an image at its storage location, with **GREEN** indicating an application at its final execution location. For applications that can be loaded via a RedBoot debug session **YELLOW** is used.

Direct standalone application startup

If the final application loads and executes from On-Chip SRAM, or is itself a second-level boot loader executing from SRAM then the suitable wrapped (QSPI bootable) image is placed from offset `0` of the respective bootable flash device.

The following figures highlight the RT1064 and RT1052 locations.

Figure 313.1. Standalone `mimxrt1064_evk` SRAM application

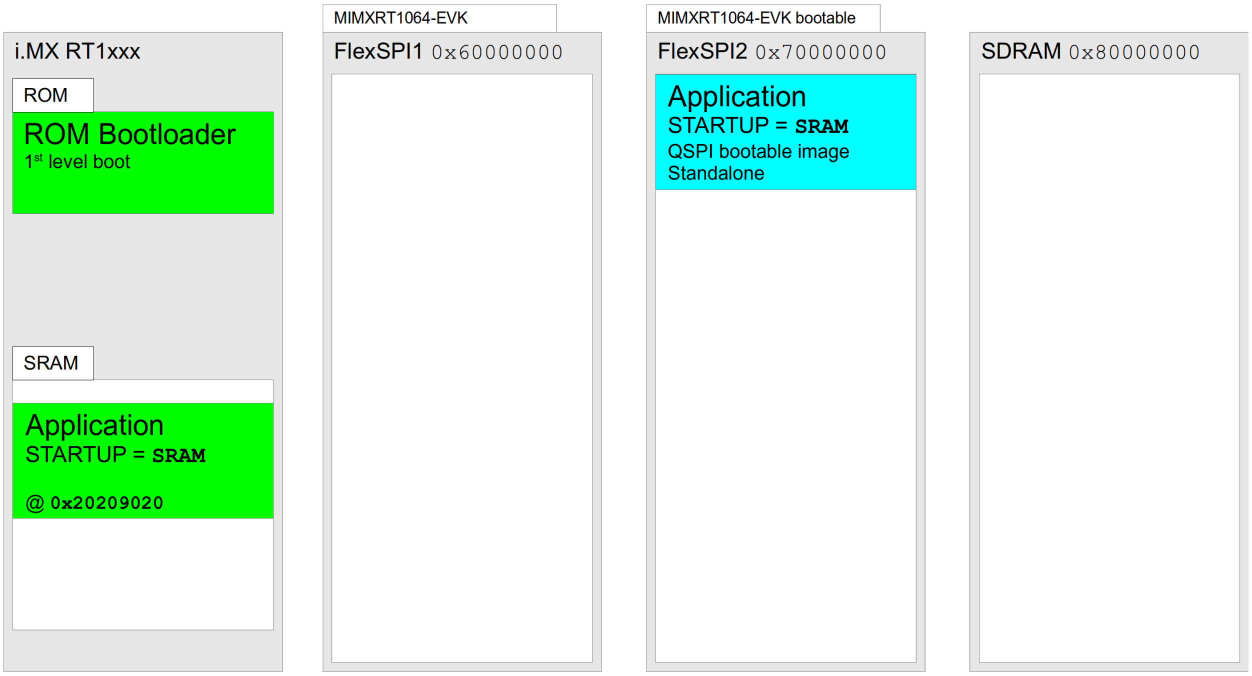
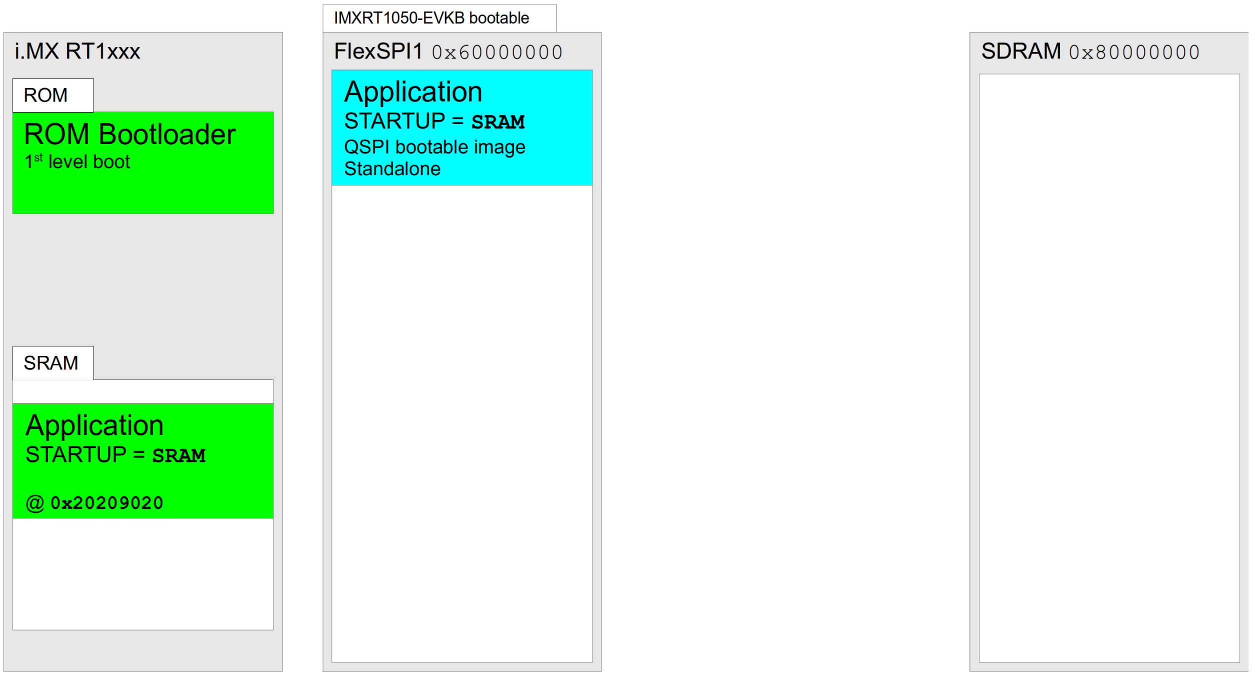


Figure 313.2. Standalone `mimxrt1050_evk` `SRAM` application



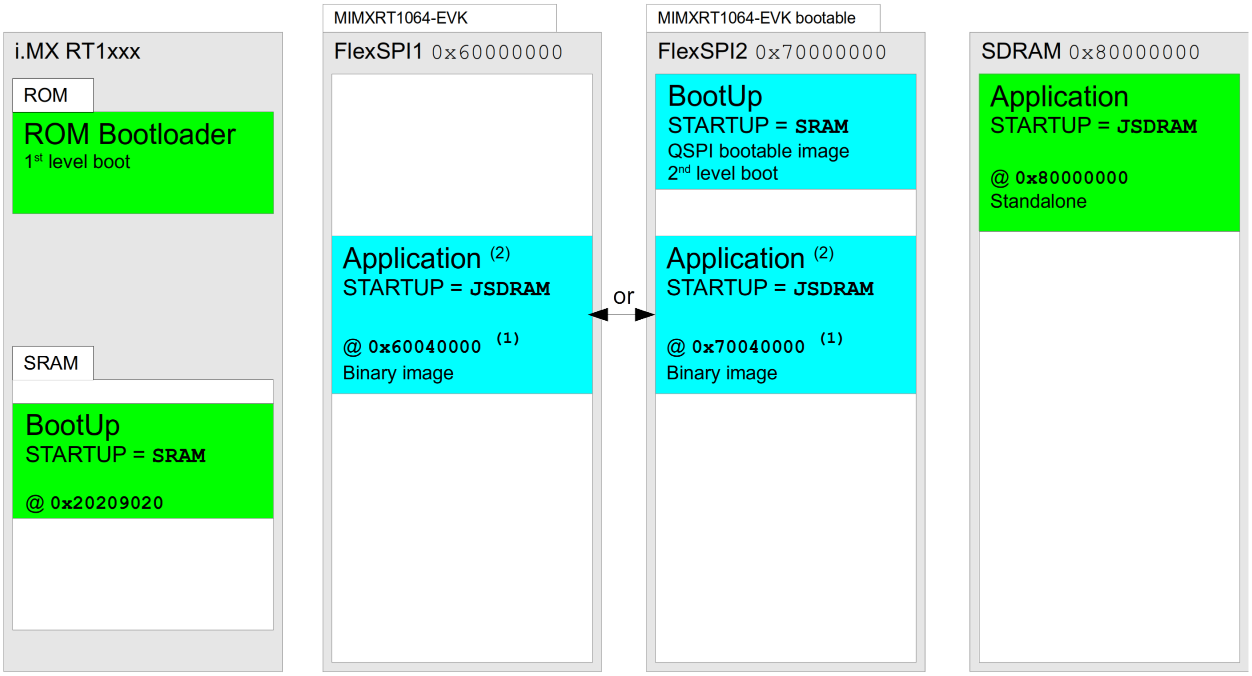
Since the RedBoot specific `RBSRAM` is just a variation of a SRAM application, for the figures above we can replace the `SRAM` Application with a `RBSRAM` RedBoot QSPI bootable image. Examples of which are shown below.

BootUp used to start application

For systems where a standalone, external-SDRAM, application is to be started at boot then the BootUp second-level boot loader can be used.

Normally the final application would be a `JSDRAM` startup application as shown for the RT1064 and RT1050 platforms respectively.

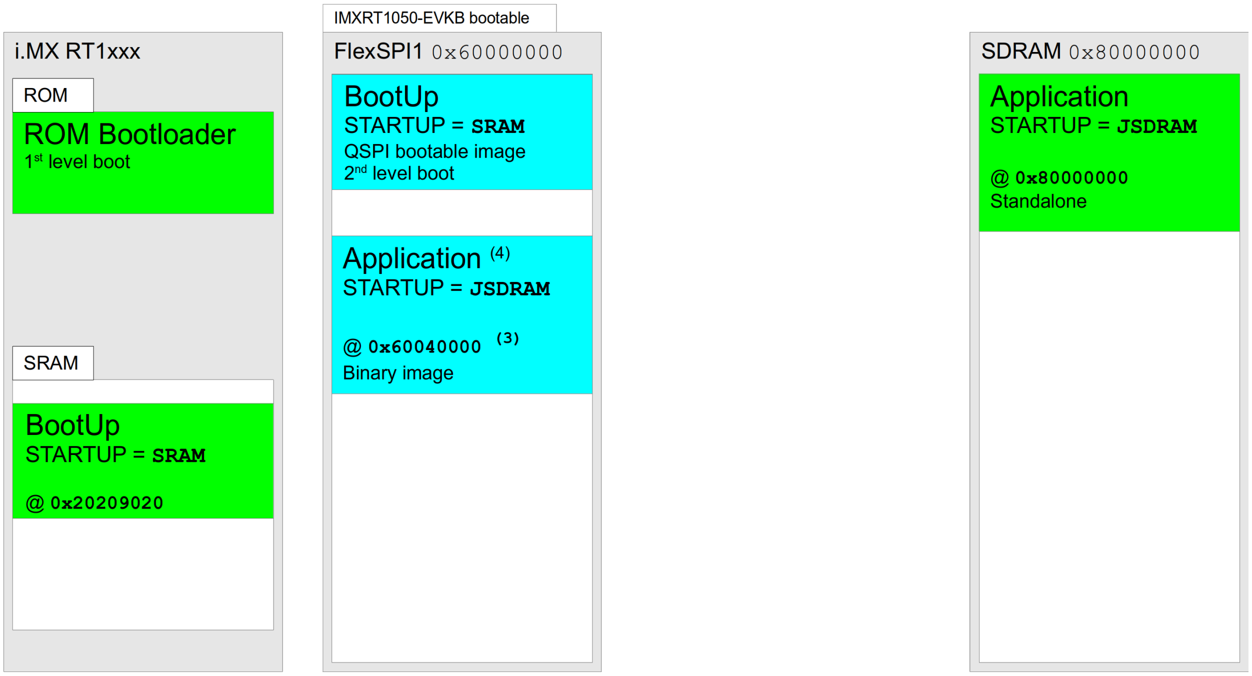
Figure 313.3. Standalone `mimxrt1064_evk` `JSDRAM` application



(1) address shown is the result of adding the default `CYGNUM_BOOTUP_IMX_SOURCE_OFFSET` offset onto the relevant FlexSPI# flash base

(2) default `CYGIMP_BOOTUP_IMX_SOURCE` is FlexSPI2, but BootUp can use FlexSPI1 if required

Figure 313.4. Standalone `mimxrt1050_evk` `JSDRAM` application

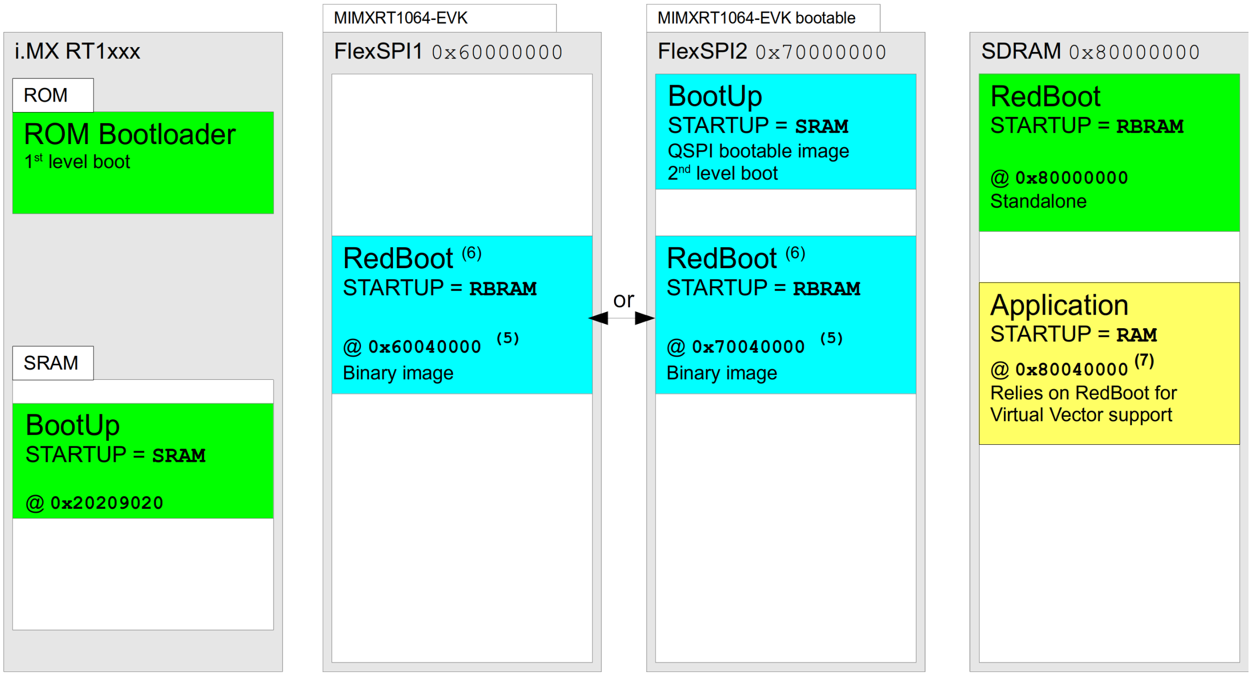


(3) address shown is the result of adding the default `CYGNUM_BOOTUP_IMX_SOURCE_OFFSET` offset onto the FlexSPI1 base address

(4) CYGIMP_BOOTUP_IMX_SOURCE is FlexSPI1

However, if a SDRAM based RedBoot is required (RBRAM startup) then BootUp can be used to load and start the SDRAM RedBoot, which can then subsequently be used to load RAM startup applications that make use of RedBoot functionality via the virtual vector support. The following figures highlight that the SDRAM configured RedBoot requires a second-level boot loader to copy and start the RedBoot instance from SDRAM:

Figure 313.5. Standalone mimxrt1064_evk RBRAM application

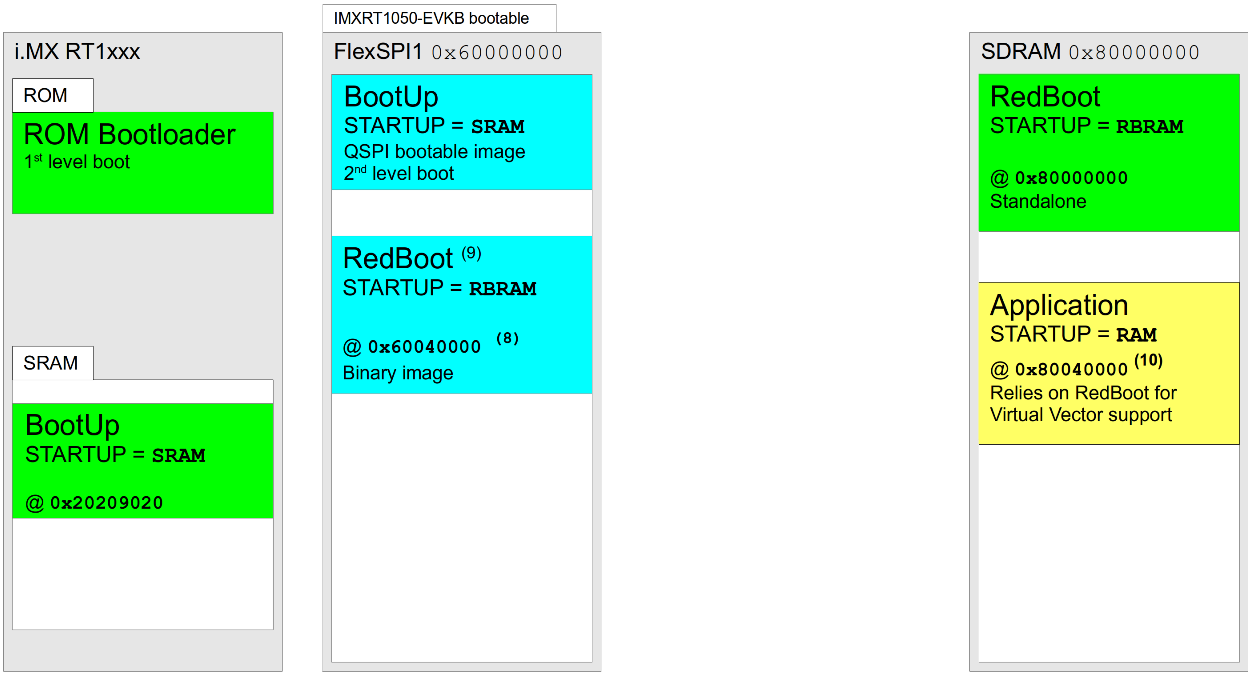


(5) address shown is the result of adding the default CYGNUM_BOOTUP_IMX_SOURCE_OFFSET offset onto the relevant FlexSPI# flash base

(6) default CYGIMP_BOOTUP_IMX_SOURCE is FlexSPI2, but BootUp can use FlexSPI1 if required

(7) default CYGMEM_REGION_redboot_SIZE offset into SDRAM defined in the “mlt_mimxrt1xxx_evk.h” header with app loaded by RedBoot from FIS or via GDB stubs

Figure 313.6. Standalone `mimxrt1050_evk` `RBRAM` application



(8) address shown is the result of adding the default `CYGNUM_BOOTUP_IMX_SOURCE_OFFSET` offset onto the FlexSPI1 flash base

(9) `CYGIMP_BOOTUP_IMX_SOURCE` is FlexSPI1

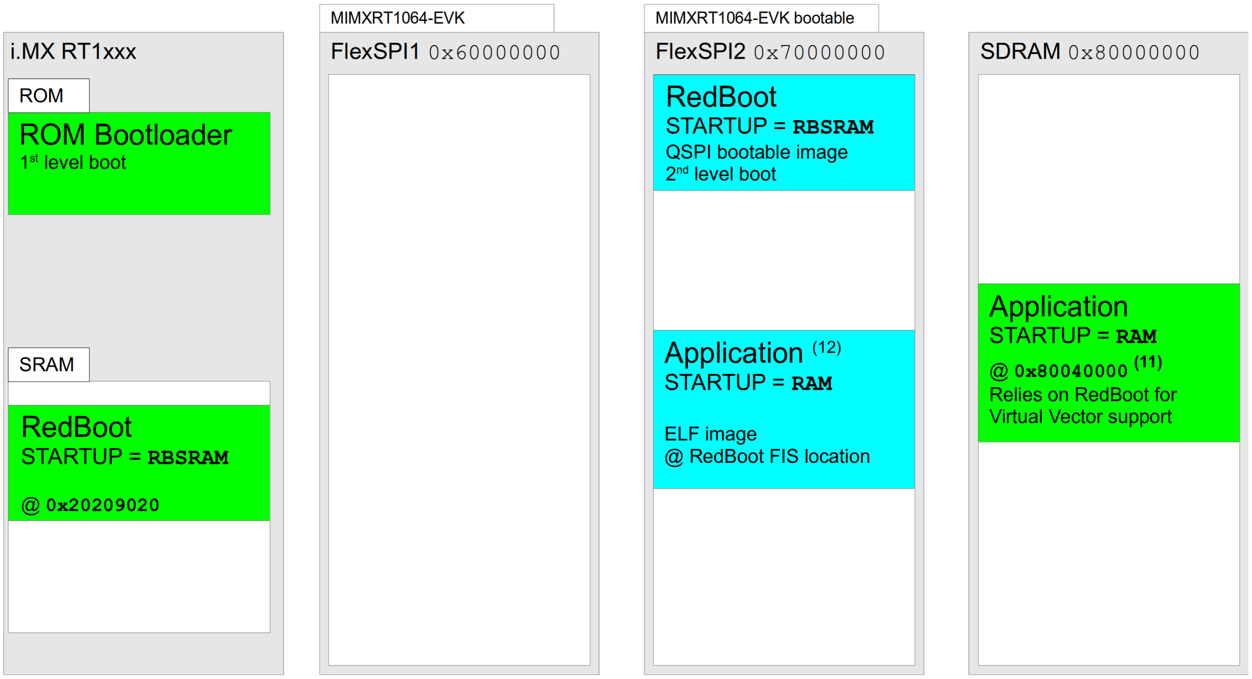
(10) default `CYGMEM_REGION_redboot_SIZE` offset into SDRAM defined in the `"mlt_mimxrt1xxx_evk.h"` header with app loaded by RedBoot from FIS or via GDB stubs

RedBoot used to start application

When RedBoot is being used to load an application, the storage location of the ELF image held in flash is controlled by the RedBoot FIS support. The following figure shows a SRAM based RedBoot setup, but the same FIS held ELF image is also applicable to external-SDRAM (`RBRAM`) RedBoot instances.

Only the RT1064 platform is shown in the figures below, but as per the standalone SRAM examples above, for the RT1052 based target the bootable second-level loader image and stored application image will be held in the FlexSPI1 flash memory.

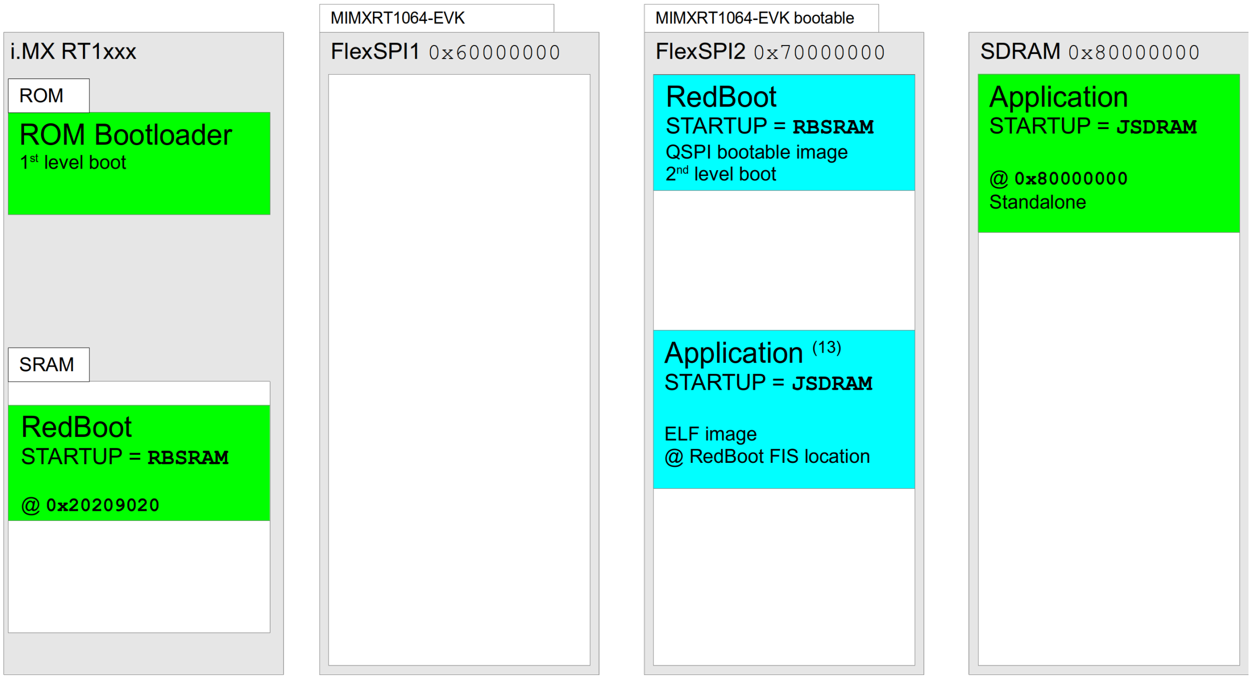
Figure 313.7. `mimxrt1064_evk` SRAM RedBoot and `RAM` application



(11) address shown is the result of adding the default `CYGMEM_REGION_redboot_SIZE` offset onto the SDRAM base address, where that offset manifest is defined in the “`mlt_mimxrt1xxx_evk.h`” header, and the app subsequently loaded by RedBoot from FIS or via GDB stubs

(12) figure shows RedBoot managed application stored in flash, but app can also be loaded via the GDB stubs debug connection

Figure 313.8. `mimxrt1064_evk` SRAM RedBoot and `JSDRAM` application



(13) figure shows RedBoot managed application stored in flash, but app can also be loaded via the GDB stubs debug connection

Name

Setup — Preparing the MIMXRT1xxx-EVK Board for eCos Development

Overview

In a development environment the EVK board may be programmed via a JTAG/SWD interface or via RedBoot loaded from boot memory.

When debugging via JTAG, you may need to disable the default HAL idle thread action, otherwise there may be issues where the target fails to halt and the debugging session is unreliable. More details can be found in the Cortex-M architectural HAL. If you are debugging via SWD this should not be necessary. When using hardware debug we **recommend** that the board SW7 DIP switches (1234) are used to select SD card boot (ON OFF ON OFF). Without an SD card installed, this will have the useful side effect of causing the processor to pause in the ROM loader. It avoids potential issues caused by any pre-installed firmware re-configuring the SoC as part of the ROM Bootloader process.

For debugging, applications are loaded and then executed on the board via the debugger **arm-eabi-gdb**, or via the Eclipse IDE. The following describes setting up to use OpenOCD with GDB.

OpenOCD

Note

As mentioned in the Overview section, if there is unknown firmware already installed in the bootable QSPI then it is worth ensuring that it does not interfere with the H/W debug session by changing the SW7 bootstrap selection to disable QSPI execution, until a known eCos application has been installed in the bootable flash.

To debug via OpenOCD the `etc/openocd.cfg` from the build install directory should be used. Running OpenOCD on a host connected to the board via a suitable cable should produce something similar to the following:

```
$ openocd -f openocd.cfg
Open On-Chip Debugger 0.11.0-2 (eCosCentric 2021-06-15)
Licensed under GNU GPL v2
For bug reports, read
  http://openocd.org/doc/doxygen/bugs.html
Info : auto-selecting first available session transport "swd". To override use 'transport select <transport>'.
Info : Listening on port 6666 for tcl connections
Info : Listening on port 4444 for telnet connections
Info : CMSIS-DAP: SWD  Supported
Info : CMSIS-DAP: FW Version = 1.10
Info : CMSIS-DAP: Interface Initialised (SWD)
Info : SWCLK/TCK = 0 SWDIO/TMS = 1 TDI = 0 TDO = 0 nTRST = 0 nRESET = 1
Info : CMSIS-DAP: Interface ready
Info : clock speed 1000 kHz
Info : SWD DPIDR 0x0bd11477
Info : imxrt1050.cpu: hardware has 8 breakpoints, 4 watchpoints
Info : starting gdb server for imxrt1050.cpu on 3333
Info : Listening on port 3333 for gdb connections
```

When subsequently connecting via GDB, the output similar to the following will be seen from the OpenOCD session:

```
Info : accepting 'gdb' connection on tcp/3333

Initialising CPU...
target halted due to debug-request, current mode: Thread
xPSR: 0x21000000 pc: 0x2020e3a8 psp: 0x2023f728
force hard breakpoints
Note: Breakpoints limited to 8 hardware breakpoints
Disable Caches
Initialising SDRAM
```

```
semihosting is enabled
```

As can be see from the “Initialising SDRAM” above, the default CDL behaviour is to install the SDRAM initialising OpenOCD configuration script. This ensures that applications configured for the `JSDRAM` or `RBRAM` startup types can be loaded directly via GDB and OpenOCD for debugging, without requiring staging via a second-level boot loader.

See OpenOCD scripts for notes regarding the OpenOCD scripts.

PEEDI

For the Ronetix PEEDI, the `peedi.mimxrt1064_evk.cfg` or `peedi.imxrt1050_evkb.cfg` file respectively should be used to setup and configure the `MIMXRT1064-EVK` and `IMXRT1050-EVKB` hardware to an appropriate state to load programs. These files only perform basic initialization by default, leaving application code to initialize PLLs and other clocks. However, these files do include an initialization section to configure SDRAM.

The configuration files also contains an option to define whether hardware or software breakpoints are used by default, using the `CORE0_BREAKMODE` directive in the `[PLATFORM_Cortex-M_SWD]` section. Edit this file if you wish to use hardware break points, and remember to restart the PEEDI to make the changes take effect.

Note

Normally, when shipped, the EVK platforms are configured to use the on-board CMSIS-DAP H/W debug interface via the micro-USB connector. If an external H/W debugger is being attached via the standard ARM 20-pin 2.54mm IDC debug connector (labelled `J21` on RevA1 boards) then the corresponding H/W debug signal jumpers need to be adjusted accordingly. Please refer to the schematic or board documentation. For example, on the RevA1 `MIMXRT1064-EVK` board jumpers `J47` through `J50` control access to the external H/W debug signals.

On the PEEDI, debugging can be performed either via the telnet interface or using **arm-eabi-gdb** and the GDB interface. In the case of the latter, **arm-eabi-gdb** needs to connect to TCP port 9000 on the PEEDI's IP address. For example:

```
(gdb) target remote 111.222.333.444:9000
```

By default when the PEEDI is powered up, the target will always run the initialization section of the configuration file, and halts the target. This behaviour is repeated with the PEEDI **reset** command.

If the board is reset (either with the '**reset**', or by pressing the reset button) and the '**go**' command is then given, then the board will boot as normal and run from the contents of the flash.

Consult the PEEDI documentation for information on other features.

H/W Debugging

Currently the standalone startup types, where H/W debug can be used, default to outputting all diagnostics information via LPUART1. The default communications parameters are 115200 baud, no parity, 1 stop bit. It is recommended that LPUART1 be accessed via the virtual CDC-ACM communication device presented by the CMSIS-DAP port.

It is possible to arrange for diagnostics to be output via the H/W debug GDB connection and appear on the gdb console. This requires the configuration option `CYGFUN_HAL_GDB_FILEIO` in the common HAL package to be enabled. This has two sub-options, `CYGSEM_HAL_DIAG_TO_GDBFILEIO_CHAN` and `CYGSEM_HAL_DIAG_VIA_GDB_FILEIO_IMMEDIATE`, that are enabled by default when `CYGFUN_HAL_GDB_FILEIO` is enabled and both should remain enabled. In this case, when **arm-eabi-gdb** is attached to the PEEDI, the following gdb command must be issued:

```
(gdb) set hwdebug on
```

Eclipse users can do this by creating a GDB command file with the contents:

```
define preload
  set hwdebug on
end
```

This will be referenced from their Eclipse debug launch configuration. Using GDB command files is described in more detail in the "Eclipse/CDT for eCos application development" manual.

OpenOCD scripts

The OpenOCD `.cfg` files supplied in the platform package `misc` directory assume that the debug session is being used to load/start a new application. If the developer wants to attach a H/W debug session to a running system (debugging a live system) then a `noinit` configuration file that connects but does not modify the hardware state should be used instead.

For historical reasons the scripts are prefixed with `openocd.mimxrt1050-evk`, but are also used for the i.MX RT1064 targets (e.g. `mimxrt1064_evk`). Similarly, internally the OpenOCD configuration scripts use the `imxrt1050` name to identity CPU information messages generated by OpenOCD. The OpenOCD naming can be ignored.

A full overview of all of OpenOCD is beyond the scope of this document, and the developer is referred to the 3rd-party OpenOCD website for detailed documentation.

Installing RedBoot

RedBoot allows for S/W based debugging of applications over the LPUART1 diagnostic channel, or an Ethernet connection, via the GDB stubs support built into RedBoot. RedBoot is normally used to load+debug applications when a H/W debugger interface is **not** available. Since the IMX EVK boards provide an on-board CMSIS-DAP then RedBoot is not strictly required for debugging since a H/W debugger is always present. However, RedBoot also provides some functionality that may be required for the target use case (e.g. RBL (`CYGPKG_RBL`)).

Warning

The default RedBoot configuration enables Ethernet support to allow debug sessions over Ethernet. See the Supported Hardware warning about powering the board from an external PSU if RedBoot-with-Ethernet is being installed.

The following RedBoot installation process uses a H/W debug session with a `RBSRAM` RedBoot image to provide the flash access support for updating the bootable QSPI device. This approach of using a on-chip SRAM based RedBoot to provide flash write capability is applicable to any binary data that the developer wishes to install in flash (and in the case of the MIMXRT1064-EVK target the SiP QSPI as well as the external QSPI). For example, the SRAM based RedBoot can be used to write the BootUp bootable QSPI image as well as the binary (`JSDRAM`) application that BootUp will start.

Prebuilt RedBoot binaries are provided in the `loaders/mimxrt1xxx_evk` subdirectory of the eCosPro release installation. The `host` directory within this subdirectory includes the `openocd.cfg` OpenOCD configuration file used to load and execute RedBoot such that the QSPI flash boot image can be programmed into the bootable FlexSPI QSPI flash memory of the i.MX RT10XX boards.

Notes:

1. The IMXRT1050-EVKB hardware **must** be modified from its default (as shipped) HyperFlash configuration to enable QSPI access. This is documented in the relevant "MIMXRT1050-EVK Board Hardware User Guide". It involves removing and moving some 0 Ω resistors on the underside of the board. The MIMXRT1064-EVK requires no such modification since the external QSPI is already the default, and the RT1064 boots from the internal SiP flash anyway.

2. Detailed instructions for rebuilding and programming your own version of RedBoot may be found in the section called “Rebuilding and Installing RedBoot” below.

In one shell window, start OpenOCD as described in the section called “OpenOCD” above, replacing `openocd.cfg` with the OpenOCD configuration file `loaders/mimxrt1xxx_evk/host/openocd.cfg` provided within the eCosPro installation directory.

Within a second shell window, open a suitable terminal application and connect to the CDC-ACM device provided by the CMSIS-DAP debug USB interface.

Within a third shell window, as illustrated in the figure below, change to the `loaders/mimxrt1xxx_evk` subdirectory of the eCosPro release installation, run cksum to note the checksum and size of the QSPI boot image, and load the (`RBSRAM`) RedBoot ELF prebuilt image into the target using OpenOCD and GDB.

Figure 313.9. Checksum of QSPI image and Execution of RedBoot

```
$ cd eCos-4.x.y/loaders/mimxrt1050_evk
$ cksum qspi_boot.bin
2434204827 111164 qspi_boot.bin
$ arm-eabi-gdb redboot_RBSRAM.elf
GNU gdb (eCosCentric GNU tools 7.3.0e) 8.1
Copyright (C) 2018 Free Software Foundation, Inc.
... elided ...
Reading symbols from redboot_RBSRAM.elf...done.
(gdb) tar extended-rem localhost:3333
Remote debugging using localhost:3333
0x2020c330 in hal_delay_us (us=100) at .../packages/hal/cortexm/arch/current/src/hal_misc.c:609
(gdb) load
Loading section .rom_vectors, size 0x8 lma 0x20209020
Loading section .text, size 0xc080 lma 0x20209028
Loading section .rodata, size 0x36f8 lma 0x202150a8
Loading section .data, size 0x634 lma 0x202187a0
Start address 0x20209028, load size 64948
Transfer rate: 42 KB/sec, 9278 bytes/write.
(gdb) cont
Continuing.
```

From the cksum result above you can see the checksum and size of the QSPI boot image which are `2434204827` and `111164` respectively. On the second shell window running the terminal application, RedBoot output similar to the following should appear.

Figure 313.10. RedBoot Output

```
RedBoot(tm) bootstrap and debug environment [RBSRAM]
eCosCentric certified release, version 4.6.6 - built 15:15:21, Mar 17 2022

Copyright (C) 2000-2009 Free Software Foundation, Inc.
Copyright (C) 2003-2022 eCosCentric Limited
The RedBoot bootloader is a component of the eCos real-time operating system.
Want to know more? Visit www.ecoscentric.com for everything eCos & RedBoot related.
This is free software, covered by the eCosPro Non-Commercial Public License
and eCos Public License. You are welcome to change it and/or distribute copies
of it under certain conditions. Under the license terms, RedBoot's source code
and full license terms must have been made available to you.
Redboot comes with ABSOLUTELY NO WARRANTY.

Platform: NXP MIMXRT1050-EVK (Cortex-M7)
RAM: 0x20200000-0x2023f000 [0x20220b40-0x2023b000 available]
     0x20240000-0x20280000 [0x20240000-0x20280000 available]
     0x80000000-0x82000000 [0x80000000-0x82000000 available]
FLASH: 0x60000000-0x607fffff, 2048 x 0x1000 blocks
RedBoot>
```

The naming and RAM and FLASH areas reported will depend on the target board.

RedBoot is now running on the NXP EVK and may be interacted with through the terminal application. The RedBoot output shows the available RAM that can be used as a temporary load buffer for the RedBoot QSPI boot image. In the example above the spaces

0x20240000-0x20280000 and 0x80000000-0x82000000 are available. The flash can now be initialised using RedBoot with the following flash initialisation issued at the **RedBoot>** prompt within the terminal application.

Note

Change your responses below accordingly if using DHCP/BOOTP or provide your own IP addresses and netmask for the gateway, server and local address of NXP EVK.

Figure 313.11. Initialise Flash

```
RedBoot> fconfig -i
Initialize non-volatile configuration - continue (y/n)? y
Run script at boot: false
Use BOOTP for network configuration: false
Gateway IP address: 192.168.1.1
Local IP address: 192.168.1.5
Local IP address mask: 255.255.255.0
Default server IP address: 192.168.1.2
Console baud rate: 115200
GDB connection port: 9000
Force console for special debug messages: false
Network debug at boot time: false
Default network device: enet0_eth
Update RedBoot non-volatile configuration - continue (y/n)? y
... Erase from 0x607fe000-0x607fefff: .
... Program from 0x20231000-0x20232000 to 0x607fe000: .
RedBoot>
```

Once the flash is initialised perform the following actions, as illustrated in Figure 313.12, “Loading RedBoot QSPI boot image into memory”:

1. Temporarily interrupt RedBoot execution by using **ctrl-C** within the third shell window (in which GDB is executing), returning to the **(gdb)** prompt.
2. Load the prepared QSPI boot image into available SRAM using the **restore** command. In this example location 0x20240000 was used.
3. Resume execution of RedBoot.

The steps above are just a simple mechanism to use the GDB debug connection to load an arbitrary binary from the host machine into the target board memory.

Figure 313.12. Loading RedBoot QSPI boot image into memory

```
...
(gdb) cont
Continuing.
^C
Program received signal SIGINT, Interrupt.
0x2020c330 in hal_delay_us (us=76) at .../packages/hal/cortexm/arch/current/src/hal_misc.c:609
(gdb) restore qspi_boot.bin binary 0x20240000
Restoring binary file qspi_boot.bin into memory (0x20240000 to 0x202524ac)
(gdb) cont
Continuing.
```

Within the RedBoot terminal, confirm the image has been loaded correctly into memory by performing a checksum of the memory region into which the QSPI boot image was loaded by GDB and compare these results with the results from the cksum application as illustrated in Figure 313.9, “Checksum of QSPI image and Execution of RedBoot”. The RedBoot cksum command requires the base address of the memory region (0x20240000 as used above) as well as the size of the region (111164 in our example).

Figure 313.13. RedBoot cksum of memory image

```
RedBoot> cksum -b 0x20240000 -l 111164
POSIX cksum = 2434204827 111164 (0x9117049b 0x000124ac)
```

As you can see the checksum values `2434204827` from both sources match. Instruct RedBoot to program the bootable QSPI flash with the image in memory as shown below.

Note

For the IMXRT1050-EVKB external QSPI the bootable flash base address is `0x60000000`. For the MIMXRT1064-EVK SiP flash the base address is `0x70000000`.

Figure 313.14. Program RedBoot into QSPI from memory image

```
RedBoot> fis write -f 0x60000000 -b 0x20240000 -l 111164
* CAUTION * about to program FLASH
            at 0x60000000..0x60012fff from 0x20240000 - continue (y/n)? y
... Erase from 0x60000000-0x60012fff: ...................
... Program from 0x20240000-0x20253000 to 0x60000000: ...................
```

At this point RedBoot has been programmed into the QSPI flash. The GDB session as well as OpenOCD must now be terminated using the **`ctrl-C`** keystroke combination in each window to interrupt and terminate the application. Assuming the SW7 is set to `OFF`, `OFF`, `ON`, `OFF` for QSPI boot selection then on the next hardware reset the system will boot (the `RBSRAM`) RedBoot from the QSPI copy automatically. The output illustrated in Figure 313.10, "RedBoot Output" will again be seen in the window executing the terminal application and normal interaction with RedBoot may again occur.

Finally, terminate the terminal application to release the CDC-ACM interface and enable both GDB and Eclipse to make use of the interface to download and debug user applications.

Rebuilding and Installing RedBoot

The following process is actually applicable to any standalone application that needs to be programmed into the FlexSPI QSPI flash memory and started from CPU reset, not just the special `RBSRAM` RedBoot startup type. For example, a different second-level boot loader or the final standalone (`SRAM`) application. The `RBSRAM` is for RedBoot configurations that execute from SRAM and can be directly started by the i.MX RT ROM bootloader, with `RBRAM` being used for RedBoot configurations that execute from SDRAM and are started via a second-level boot loader (e.g. BootUp, a `RBSRAM` RedBoot or a customer specific boot loader application). See the section called "Startup" for an overview of the different application startup types.

See the Notes: from the previous section with regards to IMXRT1050-EVKB QSPI support.

RedBoot can be configured to reside+execute from SRAM (OCRAM) or the external SDRAM. A `RBSRAM` RedBoot application should be configured and built using the `redboot_RBSRAM.ecm` file provided in the `packages/hal/cortexm/imx/mimxrt1xxx_evk/current/misc` subdirectory of the eCosPro installation. Correspondingly a SDRAM resident RedBoot application should be configured and built using the `redboot_RBRAM.ecm` configuration fragment.

However, since the decision to keep a single `RAM` startup type for applications loaded into SDRAM and executed under RedBoot regardless of whether a SRAM or SDRAM RedBoot is active means that the RedBoot SRAM allocation is **always** set aside. This means that there is **no** substantive benefit from an SDRAM based RedBoot currently. So the following example just documents the `RBSRAM` case.

Should it prove necessary to rebuild a RedBoot binary, this is done most conveniently at the command line. The steps needed to rebuild the SRAM version of RedBoot are:

```
$ mkdir redboot_rbsram
$ cd redboot_rbsram
$ ecosconfig new mimxrt1050_evk redboot
[ … ecosconfig output elided … ]
```

```
$ ecosconfig import $ECOS_REPOSITORY/hal/cortexm/imx/mimxrt1xxx_evk/VERSION/misc/redboot_RBSRAM.ecm
$ ecosconfig tree
$ make
```

The standard make build process will create a binary version of the RedBoot application alongside the ELF version used for debugging. The `install/bin` subdirectory should contain the file `redboot.bin` which will be used when building the boot image. The example above targets the `mimxrt1050_evk` platform, but the same process can be used by replacing with the `mimxrt1064_evk` platform name.

Create a QSPI boot image from this `RBSRAM` RedBoot binary as follows:

```
$ flashimg_rt10 --qspi install/bin/redboot.bin install/bin/qspi_boot.bin
```

Notes:

1. Prebuilt host executables of the **flashimg_rt10** are located in the `loaders/mimxrt1050_evk/host` subdirectory of the eCosPro installation directory. Ensure these executables are either in the host's PATH or provide the full or relative path to the executable relevant to your host's operating system when executing the above command.

2. The **--qspi** parameter is **required** to ensure the correct layout and contents of boot ROM descriptor structures to allow the application to be booted from the QSPI device.

3. If using the prebuilt RedBoot images provided in the release, the filenames `redboot_RBSRAM.elf`, `redboot_RBSRAM.bin` and its flashimg_rt10 derivative `qspi_boot.bin` within the `loaders/mimxrt1050_evk` subdirectory should be referenced in place of the files `install/bin/redboot.elf`, `install/bin/redboot.bin` and `install/bin/qspi_boot.bin` respectively in the examples below.

The simplest approach for installing the boot image is to use the onboard CMSIS-DAP debug interface and GDB to load the boot image into memory and use the `RBSRAM` RedBoot itself to initialise the flash. See the OpenOCD section for an overview of starting the necessary GDB server, and the Installing RedBoot section for an example of using RedBoot to flash an image.

Further application installation examples

The following are just examples of some possible use cases that may helpful in providing an overview for developers.

SRAM application

Since SRAM resident applications, configured with the `SRAM` startup type, do not need a second-stage boot loader they can be packaged as a bootable image and flashed to the start of the respective flash memory directly.

The `RBSRAM` (SRAM resident) RedBoot is just a special-case of an SRAM application since it needs to ensure SRAM and SDRAM are shared between itself and the non-standalone `RAM` startup applications that it hosts. Aside: The `RBSRAM` RedBoot can be used to start a standalone (e.g. `JSDRAM`) application and so can just act as a second-stage boot loader if required where the executed application makes no use of RedBoot functionality after it has been started, **but** note that such standalone applications can then not be debugged via the RedBoot GDB stubs connection.

To boot **any** application built from a `SRAM` startup type configuration you first need to extract the binary image from the generated ELF file. e.g. replacing *`app.elf`* and the destination binary name as appropriate:

```
$ arm-eabi-objcopy -O binary app.elf app.bin
```

Then, as described in the flashimg_rt10 section, you convert that raw binary into a bootable image:

```
$ flashimg_rt10 --qspi app.bin qspi_boot.bin
```

The resulting image can then be written to the relevant bootable flash device as demonstrated in the Installing RedBoot section, with the filenames replaced accordingly.

BootUp started application

When the user wants to start an external SDRAM based application at boot a SRAM based second-level boot loader like BootUp is required. BootUp provides a simpler, quicker, smaller implementation than RedBoot. It can be extended with customer or platform features as required; but the basic implementation simply copies the application from its stored NVM (Non-Volatile Memory) location to SDRAM and executes it.

As documented in the VAR BootUp section the command-line BootUp build creates the `bootup.bin` image. So the `arm-eabi-objcopy` step as documented above for SRAM applications is already done. However the `flashimg_rt10` step should still be performed to create a bootable image. The resulting QSPI bootable image should be written to the start of the flash.

For the actual application, created from a `JSDRAM` configured build, just like the SRAM approach, the generated ELF file should be converted to a binary for flashing.

```
$ arm-eabi-objcopy -O binary sdramapp.elf sdramapp.bin
```

That binary needs no further processing, and should be written as-is to the relevant flash memory at offset `CYGNUM_BOOTUP_IMX_SOURCE_OFFSET`. The use of a H/W debug session and a `RBSRAM` RedBoot provides a mechanism for writing arbitrary images to flash.

On subsequent target resets, the system ROM Bootloader will execute the BootUp code, which in turn will find, load and start the SDRAM application.

For subsequent updates of the main SDRAM based application only the application binary from offset `CYGNUM_BOOTUP_IMX_SOURCE_OFFSET` onwards need be replaced. The BootUp installation itself is normally just a one-time process.

As previously noted, the SDRAM based `RBRAM` is just a special-case startup type used when building a SDRAM resident RedBoot. If using the `RBRAM` RedBoot then its binary should be written to the flash from offset `CYGNUM_BOOTUP_IMX_SOURCE_OFFSET`.

RedBoot started application

For details of using RedBoot to start the final SDRAM based application refer to the Executing Programs from RedBoot documentation.

Name

Configuration — Platform-specific Configuration Options

Overview

The MIMXRT1xxx-EVK board platform HAL package is loaded automatically when eCos is configured for a suitable target, e.g. `mimxrt1050_evk` or `mimxrt1064_evk`. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The MIMXRT1xxx-EVK board platform HAL package supports six separate startup types:

JTAG

> This is the default startup type. It is used to build applications that are loaded via a H/W debug interface. The application will be self-contained with no dependencies on services provided by other software. The program expects to be loaded from 0x20209000 and entered at 0x20209008. eCos startup code will perform all necessary hardware initialization.
>
> Even though this startup type is the default, it is not normally expected to be used in the field. It is normally used for testing and development on uninitialised boards.

SRAM

> This startup type is currently essentially equivalent to the JTAG startup type in memory layout and usage. This startup is intended to be used for standalone applications, either loaded from an external memory device such as an SD card or FlexSPI flash at boot time, or via a H/W debug interface.
>
> The program expects to be loaded from 0x20209020, and the eCos startup code will perform all necessary hardware initialisation. The difference in load address from the JTAG startup is to allow space for the boot ROM configuration structures required when the application is packaged into a boot image using the `flashimg_rt10` tool.
>
> This startup type should be configured for standalone applications to execute from SRAM. All of the SRAM is available for application use, and the external SDRAM is unassigned and not managed by eCos, but is available for application use.

JSDRAM

> This startup is intended to be used for standalone applications, either loaded from an external memory device such as an SD card or FlexSPI flash at boot time via a second-level boot loader, or via a H/W debug interface into external SDRAM.
>
> The application will be self-contained with no dependencies on services provided by other software. The program expects to be loaded from 0x80000000 and entered at 0x80000018 (based on the current application signature block size). eCos startup code will perform all necessary hardware initialization. JSDRAM applications can only be loaded once the SDRAM has been initialised, either when loaded from a boot location in conjunction with a valid IVT+DCD via a second-level boot loader, or when loaded via a suitable H/W debug session directly.
>
> This startup type should be configured for standalone applications to execute from SDRAM. All of the SDRAM and SRAM is available for application use.

RBRAM

> This is a special case startup type intended for SDRAM based RedBoot applications. It is essentially equivalent to the `JSDRAM` startup type, but with a section from the start of SRAM and SDRAM allocated to RedBoot for its code+data storage. The remaining SRAM and SDRAM is set aside for applications loaded/executed via the RedBoot instance. As mentioned, this startup is intended to be used for the standalone RedBoot, either loaded at boot from a memory device such as the FlexSPI flash via a second-level boot loader, or via a H/W debug interface.

The program expects to be loaded from 0x80000000, and the eCos startup code will perform all necessary hardware initialisation.

RBSRAM

This is a special case startup type intended for SRAM based RedBoot applications. It is essentially equivalent to the `SRAM` startup type, but with the bottom of the SRAM allocated to RedBoot, along with an (unused) section at the start of SDRAM, for the RedBoot code+data requirements. The remaining SRAM and SDRAM are set aside for applications loaded/executed via the RedBoot instance. As mentioned, this startup is intended to be used for the standalone RedBoot, either loaded at boot from a memory device such as the FlexSPI flash, or via a H/W debug interface.

The program expects to be loaded from 0x20209020, and the eCos startup code will perform all necessary hardware initialisation. The difference in load address from the `JTAG` startup is to allow space for the boot ROM configuration structures required when the application is packaged into a boot image. e.g. as by the **flashimg_rt10** tool.

RAM

This startup type is for applications that are loaded via RedBoot into external SDRAM. They rely on services supplied by RedBoot. RAM applications can only be loaded via RedBoot.

Note

Due to the `RBSRAM` startup type RedBoot code+data occupying SRAM, only the upper section of SRAM is available for `RAM` applications.

Similarly due to the `RBRAM` startup type RedBoot code+data occupying the start of SDRAM, the available SDRAM for `RAM` applications starts from the offset `CYGMEM_REGION_redboot_SIZE`.

The decision to set aside the SRAM and SDRAM space for RedBoot in both `RBRAM` **and** `RBSRAM` RedBoot configurations was taken to allow the `RAM` startup applications to be loaded irrespective of whether RedBoot is executing from SRAM or SDRAM.

As highlighted in the VAR On-chip memory section, the i.MX RT ROM bootloader cannot directly boot external-SDRAM applications. If the final application is a `JSDRAM` standalone application, or a `RBRAM` RedBoot, then a second-level boot loader is required. The BootUp (`CYGPKG_BOOTUP`) application is a lightweight second-level loader implementation, with the VAR BootUp section providing an overview. Alternatively a `RBSRAM` RedBoot could be used to boot the final `RBRAM` RedBoot if really required, but since the SRAM cost of both RedBoot configurations are the same it is expected that if RedBoot is required then a `RBSRAM` version is used (as can be directly booted by the i.MX RT ROM bootloader). The only benefit for a SDRAM based RedBoot would be **if** the `RBRAM` **and** `RAM` startup types were modified to not make use of the SRAM, but that would preclude using `RAM` startup applications under a `RBSRAM` RedBoot.

UART Serial Driver

The MIMXRT1050-EVK board uses the RT10XX internal UART serial support. The HAL diagnostic interface, used for both polled diagnostic output and GDB stub communication, is only expected to be available to be used on the LPUART1 port.

As well as the polled HAL diagnostic interface, there is also a `CYGPKG_IO_SERIAL_NXP_LPUART` package which contains all the code necessary to support interrupt-driven operation with greater functionality.

It is not recommended to use the interrupt-driven serial driver with a port at the same time as using that port for HAL diagnostic I/O.

This driver is not active until the `CYGPKG_IO_SERIAL_DEVICES` configuration option within the generic serial driver support package `CYGPKG_IO_SERIAL` is enabled in the configuration. By default this will only enable support in the driver for the LPUART1 port (the same as the HAL diagnostic interface), but the default configuration can be modified to enable support for other serial ports.

SPI Driver

An SPI bus driver is available in the package "NXP LPSPI Support" (CYGPKG_DEVS_SPI_NXP_LPSPI).

Consult the generic SPI driver API documentation in the eCosPro Reference Manual for further details on SPI support in eCosPro, along with the configuration options in the NXP SPI device driver.

I²C Driver

Support for NXP I²C busses is provided by the "NXP LPI2C Support" package (CYGPKG_DEVS_I2C_NXP_LPI2C). The variant HAL causes two buses to be instantiated. These have been tested using external I²C devices.

Flash Driver

The external FlexSPI Flash, and in the case of RT1064 boards the FlexSPI2 attached SiP Flash. may be programmed and managed using the Flash driver located in the "NXP FlexSPI Support" (CYGPKG_DEVS_FLASH_NXP_FLEXSPI) package. This driver is enabled automatically if the generic "Flash device drivers" (CYGPKG_IO_FLASH) package is included in the eCos configuration. The driver will configure itself automatically for the size and parameters of the specific flash variant present on the IMXRT1050-EVKB and MIMXRT1064-EVK boards.

Ethernet Driver

The EVK boards use the internal ENET Ethernet device attached to an external Micrel KSZ8081RNB PHY. The CYGPKG_DEVS_ETH_FREESCALE_ENET package, in conjunction with the VAR HAL, contains all the code necessary to support this device and the platform HAL package contains definitions that customize the driver to the board. The driver is not active until the generic Ethernet support package, CYGPKG_IO_ETH_DRIVERS, is included in the configuration.

This PLF HAL provides support for enforcing the start-of-day PHY pin strapping for correct operation.

CAN Driver

The iMX RT1xxx devices have multiple FlexCAN interfaces. Device support is via the NXP FlexCAN CAN Driver package.

The EVK boards have a single CAN connector (unpopulated by default) on J11 that is configured as FlexCAN2 for RT1052 boards, and FlexCAN3 for RT1064 boards.

Consult the generic Chapter 90, *CAN Support* documentation for further details on use of the CAN API, CAN configuration and device drivers.

Watchdog Driver

The board uses the RT10XX Watchdog timer 1. The CYGPKG_DEVICES_WATCHDOG_ARM_IMX package contains all the code necessary to support this device. Within that package the CYGNUM_DEVS_WATCHDOG_ARM_IMX_DESIRED_TIMEOUT_MS configuration option controls the watchdog timeout, and by default will force a reset of the board upon timeout. This driver is not active until the generic watchdog device support package, CYGPKG_IO_WATCHDOG, is included in the configuration.

PWM Driver

Support for the NXP FlexPWM devices is provided by the "NXP PWM Support" package (CYGPKG_DEVS_PWM_NXP) which needs to be used in conjunction with the CYGPKG_IO_PWM generic PWM package. Refer to the documentation for that package for usage details.

The RT10XX contains four FlexPWM devices, each of which contains four independent submodules. Each submodule has two semi-independent output lines that can be routed to a variety of pads. Each submodule is presented as a separate PWM device and have names such as "pwm1.0" for FlexPWM 1 submodule 0 or "pwm3.2" for FlexPWM 3 submodule 2. The output lines are mapped on to channel 0 for output A and channel 1 for output B. These outputs are semi-independent in that they must share a period, but may have different duty cycles.

USB Support

Support for both Host and Peripheral mode operation is provided by the USB protocol stack plus EHCI host and peripheral drivers (`CYGPKG_DEVS_USB_EHCI` and `CYGPKG_DEVS_USB_PCD_EHCI`).

Host mode is supported for USB2 which is connected to the USB_HOST microab receptacle. Class support is available for mass storage devices, and CDC-ACM serial.

Peripheral mode is supported for USB1 which is connected to the USB_OTG microab receptacle. Note that OTG mode is not supported. By default this peripheral port is configured as a CDC-ACM device.

USB configuration is handled by the "RT10XX USB controller configuration" package (`CYGPKG_DEVS_USB_RT10XX`). Here both the host and peripheral drivers are instantiated along with the CDC-ACM peripheral serial device if configured.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the IMXRT1050-EVKB and MIMXRT1064-EVK board hardware, and should be read in conjunction with that specification. The MIMXRT1xxx-EVK platform HAL package complements the Cortex-M architectural HAL and the i.MX variant HAL. It provides functionality which is specific to the target boards.

Startup

For the `SRAM`, `JTAG`, `JSDRAM`, `RBRAM` and `RBSRAM` startups, the HAL will perform initialization, programming the various internal registers including the PLL, peripheral clocks and pin multiplexing. The details of the early hardware startup may be found in the `src/imx_misc.c` in both `hal_system_init()` and `hal_platform_init()`.

Note

Some of the initial I/O run-time configuration is performed by the iMX boot ROM parsing the IVT+DCD binary structures that describe a bootable image. The relevant i.MX RTxxx PRM (Processor Reference Manual) documentation should be consulted for a detailed overview if required.

Memory Map

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory areas are as follows:

Internal SRAM

This is located at address 0x20200000 of the memory space, and is 512KiB in size for RT105x CPUs and 1MiB in size for the RT1064. The eCos VSR table occupies the bottom 704 bytes, with the virtual vector table starting at 0x200002C0 and extending to 0x200003C0. Depending on the startup type the top of SRAM may have `CYGNUM_HAL_COMMON_INTERRUPTS_STACK_SIZE` bytes reserved for the interrupt stack, as well as an uncached area for driver/DMA use. The remainder of internal SRAM is available for use by applications.

External SDRAM

This is located at address 0x800000000 of the memory space. This region is 32MiB in size. Standalone `JSDRAM` or RedBoot loaded `RAM` applications are by default configured to run from this memory. This memory is only available after either A) an application has been loaded from a boot device, since it is initialized by the DCD that is part of the boot image or B) configured by the H/W debugger connection script.

For example, if RedBoot is installed as the bootable application then the DCD prefixed to the RedBoot application configures the SDRAM as needed.

On-Chip Peripherals

These are accessible at locations 0x40000000 and 0xE0000000 upwards. Descriptions of the contents can be found in the i.MX RT10XX User Manual.

Flash

For the RT1052 and RT1064 the external (off-chip) QSPI flash is mapped from address 0x60000000. For the RT1052 this external flash is the bootable flash device.

For the RT1064 the “external” (SiP) QSPI attached to FlexSPI2 is mapped from address 0x70000000. For the RT1064 this SiP flash is the bootable flash device.

Linker Scripts

The platform linker script defines the following symbols:

hal_vsr_table
This defines the location of the VSR table. This is set to 0x20200000 for all startup types, and space for 176 entries is reserved.

hal_virtual_vector_table
This defines the location of the virtual vector table used to communicate between a ROM monitor and an eCos application. If required this is allocated right after the VSR table, at 0x202002C0.

hal_interrupt_stack
This defines the location of the interrupt stack. This is allocated to the top of application available SRAM or SDRAM depending on the startup type.

hal_startup_stack
This defines the location of the startup stack. For all startup types it is initially allocated at the half-way point of the interrupt stack.

Real-time characterization

The tm_basic kernel test gives statistics gathered about the real-time characterization and performance of the kernel. The sample output is shown here for information. The test was built for JTAG startup on a MIMXRT1050-EVK board.

Example 313.1. MIMXRT1050-EVK Real-time characterization

```
Configured
Testing parameters:
   Clock samples:            32
   Threads:                  25
   Thread switches:         128
   Mutexes:                1165
   Mailboxes:               340
   Semaphores:             2040
   Scheduler operations:    128
   Counters:                680
   Flags:                  1360
   Alarms:                  582
   Stack Size:             1088

             Startup, main thrd : stack used   356 size  2048
             Startup : Idlethread stack used    76 size  1280

eCos Kernel Timings
Notes: all times are in microseconds (.000001) unless otherwise stated

Reading the hardware clock takes 0 'ticks' overhead
... this value will be factored out of all other measurements
Clock interrupt took    7.84 microseconds (7 raw clock ticks)

Testing parameters:
   Clock samples:            32
   Threads:                  25
   Thread switches:         128
   Mutexes:                  32
   Mailboxes:                32
   Semaphores:               32
   Scheduler operations:    128
   Counters:                 32
   Flags:                    32
   Alarms:                   32
   Stack Size:             1088


                                 Confidence
     Ave     Min     Max     Var  Ave  Min  Function
  ======  ======  ======  ====== ========== ========
INFO:<Ctrl-C disabled until test completion>
    6.64    5.00    8.00    0.72   76%   8% Create thread
    1.44    1.00    2.00    0.49   56%  56% Yield thread [all suspended]
    1.56    1.00    2.00    0.49   56%  44% Suspend [suspended] thread
    1.36    1.00    2.00    0.46   64%  64% Resume thread
    2.04    2.00    3.00    0.08   96%  96% Set priority
    0.32    0.00    1.00    0.44   68%  68% Get priority
    3.20    3.00    5.00    0.34   84%  84% Kill [suspended] thread
    1.48    1.00    2.00    0.50   52%  52% Yield [no other] thread
    1.96    1.00    3.00    0.15   88%   8% Resume [suspended low prio] thread
    1.40    1.00    2.00    0.48   60%  60% Resume [runnable low prio] thread
    2.04    2.00    3.00    0.08   96%  96% Suspend [runnable] thread
    1.52    1.00    2.00    0.50   52%  48% Yield [only low prio] thread
    1.40    1.00    2.00    0.48   60%  60% Suspend [runnable->not runnable]
    3.16    3.00    5.00    0.28   88%  88% Kill [runnable] thread
    3.52    3.00    5.00    0.54   96%  52% Destroy [dead] thread
    7.52    7.00    8.00    0.50   52%  48% Destroy [runnable] thread
    6.80    6.00    9.00    0.51   60%  32% Resume [high priority] thread
    2.34    2.00    4.00    0.46   66%  66% Thread switch

    0.33    0.00    1.00    0.44   67%  67% Scheduler lock
    1.21    1.00    2.00    0.33   78%  78% Scheduler unlock [0 threads]
    1.26    1.00    2.00    0.38   74%  74% Scheduler unlock [1 suspended]
    1.24    1.00    2.00    0.37   75%  75% Scheduler unlock [many suspended]
    1.26    1.00    2.00    0.38   74%  74% Scheduler unlock [many low prio]

    0.50    0.00    1.00    0.50  100%  50% Init mutex
    1.81    1.00    3.00    0.36   75%  21% Lock [unlocked] mutex
    1.88    1.00    2.00    0.22   87%  12% Unlock [locked] mutex
    1.59    1.00    2.00    0.48   59%  40% Trylock [unlocked] mutex
    1.53    1.00    2.00    0.50   53%  46% Trylock [locked] mutex
    0.44    0.00    1.00    0.49   56%  56% Destroy mutex
   10.00   10.00   10.00    0.00  100% 100% Unlock/Lock mutex

    0.56    0.00    1.00    0.49   56%  43% Create mbox
    0.38    0.00    1.00    0.47   62%  62% Peek [empty] mbox
    1.81    1.00    2.00    0.31   81%  18% Put [first] mbox
    0.38    0.00    1.00    0.47   62%  62% Peek [1 msg] mbox
    1.69    1.00    3.00    0.47   62%  34% Put [second] mbox
    0.38    0.00    1.00    0.47   62%  62% Peek [2 msgs] mbox
    1.78    1.00    3.00    0.39   71%  25% Get [first] mbox
    1.72    1.00    2.00    0.40   71%  28% Get [second] mbox
    1.59    1.00    2.00    0.48   59%  40% Tryput [first] mbox
    1.53    1.00    2.00    0.50   53%  46% Peek item [non-empty] mbox
    1.56    1.00    2.00    0.49   56%  43% Tryget [non-empty] mbox
    1.38    1.00    2.00    0.47   62%  62% Peek item [empty] mbox
    1.50    1.00    2.00    0.50  100%  50% Tryget [empty] mbox
    0.47    0.00    1.00    0.50   53%  53% Waiting to get mbox
    0.50    0.00    1.00    0.50  100%  50% Waiting to put mbox
    0.53    0.00    1.00    0.50   53%  46% Delete mbox
    6.66    6.00    7.00    0.45   65%  34% Put/Get mbox

    0.50    0.00    1.00    0.50  100%  50% Init semaphore
    1.44    1.00    2.00    0.49   56%  56% Post [0] semaphore
    1.59    1.00    2.00    0.48   59%  40% Wait [1] semaphore
    1.34    1.00    2.00    0.45   65%  65% Trywait [0] semaphore
    1.38    1.00    2.00    0.47   62%  62% Trywait [1] semaphore
    0.44    0.00    1.00    0.49   56%  56% Peek semaphore
    0.44    0.00    1.00    0.49   56%  56% Destroy semaphore
    6.31    6.00    7.00    0.43   68%  68% Post/Wait semaphore
    0.75    0.00    1.00    0.38   75%  25% Create counter
    0.47    0.00    1.00    0.50   53%  53% Get counter value
    0.38    0.00    1.00    0.47   62%  62% Set counter value
    1.81    1.00    2.00    0.31   81%  18% Tick counter
    0.50    0.00    1.00    0.50  100%  50% Delete counter

    0.41    0.00    1.00    0.48   59%  59% Init flag
    1.47    1.00    3.00    0.53   56%  56% Destroy flag
    1.25    1.00    2.00    0.38   75%  75% Mask bits in flag
    1.53    1.00    2.00    0.50   53%  46% Set bits in flag [no waiters]
    1.91    1.00    3.00    0.23   84%  12% Wait for flag [AND]
    1.97    1.00    3.00    0.12   90%   6% Wait for flag [OR]
    1.91    1.00    3.00    0.23   84%  12% Wait for flag [AND/CLR]
    1.94    1.00    2.00    0.12   93%   6% Wait for flag [OR/CLR]
    0.34    0.00    1.00    0.45   65%  65% Peek on flag

    0.88    0.00    1.00    0.22   87%  12% Create alarm
    2.47    2.00    3.00    0.50   53%  53% Initialize alarm
    1.47    1.00    2.00    0.50   53%  53% Disable alarm
    2.38    2.00    3.00    0.47   62%  62% Enable alarm
    1.59    1.00    2.00    0.48   59%  40% Delete alarm
    1.84    1.00    2.00    0.26   84%  15% Tick counter [1 alarm]
    7.34    7.00    8.00    0.45   65%  65% Tick counter [many alarms]
    2.97    2.00    3.00    0.06   96%   3% Tick & fire counter [1 alarm]
   42.19   42.00   43.00    0.30   81%  81% Tick & fire counters [>1 together]
    8.47    8.00    9.00    0.50   53%  53% Tick & fire counters [>1 separately]
    7.00    7.00    7.00    0.00  100% 100% Alarm latency [0 threads]
    6.59    6.00    7.00    0.49   58%  41% Alarm latency [2 threads]
    6.67    6.00    7.00    0.44   67%  32% Alarm latency [many threads]
   11.02   11.00   13.00    0.03   99%  99% Alarm -> thread resume latency

    0.00    0.00    0.00    0.00            Clock/interrupt latency

    3.08    2.00    4.00    0.00            Clock DSR latency

     180     148     228                    Worker thread stack used (stack size 1088)
            All done, main thrd : stack used   704 size  2048
            All done : Idlethread stack used   172 size  1280

Timing complete - 29820 ms total

PASS:<Basic timing OK>
EXIT:<done>
```

Platform specific tests

A single platform specific test is available.

platform

The `platform` test is a simple validity check application. It tests application access to some basic functionality, as well as providing some diagnostic information on system settings.

The test source is set up for automated testing (e.g. as used in the eCosCentric test farm). However, the `__MANUAL` manifest can be manually enabled to provide some extra testing requiring user interaction. Currently `__MANUAL` controls access to a simple polled and interrupt-driven GPIO `SW8` (USER_BUTTON) test. That specific test case implements an example of using the demultiplexed GPIO interrupt support.

Part LXXIX. H8300 Architecture

Table of Contents

Chapter 314. H8/300 Architectural Support

Name

Overview — eCos Support for the H8/300 Family of Processors

Description

The H8/300 family includes the H8/300H and H8S processors. These processors execute a largely common instruction set and have the following common features:

- Eight general purpose 32-bit registers, ER0 to ER7,which may also be addressed as 16 16-bit and 16 8-bit registers. Register ER7 is also used as the stack pointer. In addition there is a 24-bit program counter, an 8-bit condition code register, and on the H8S an 8 bit extended control register.
- A linear address space, limited to 24-bits, matching the size of the PC. Hence the processor can address 16 megabytes of memory.
- No separate address space for I/O operations. Instead devices are accessed just like memory via the main address and data buses.
- A variable-length instruction set with a variety of different addressing modes.
- The H8/300H has a simple single-level interrupt system while the H8S can support an 8-level prioritized system via the extended control register.
- The H8S has support for single instruction tracing.

The architectural HAL provides support for those features which are common to all members of the H8/300 families, and for certain features which are present on some but not all members. A typical eCos configuration will also contain a variant HAL package with support code for a family of processors, possibly a processor HAL package with support for one specific processor, and a platform HAL which contains the code needed for a specific hardware platform. For example the variant or processor HAL may define the exact interrupt controller hardware that is available, and the platform HAL will define the external interrupt vector connections.

Name

Options — Configuring the H8/300 Architectural Package

Loading and Unloading the Package

The H8/300 architectural HAL package CYGPKG_HAL_H8300 should be loaded automatically when eCos is configured for H8/300-based target hardware. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Stacks

By default the architectural HAL provides a single block of memory to act as both the startup stack and the interrupt stack. The variant, processor or platform HAL may override this. For example if there are several banks of RAM with different performance characteristics it may be desirable to place the interrupt stack in fast RAM rather than in ordinary RAM.

The assembler startup code sets the stack pointer to the startup stack before switching to C code. This stack is used for all HAL initialization, running any C++ static constructors defined either by eCos or by the application, and the cyg_start entry point. In configurations containing the eCos kernel cyg_start will enable interrupts, activate the scheduler and threads will then run on their own stacks. In non-kernel single-threaded applications the whole system continues to run on the startup stack.

When an interrupt occurs the default behaviour is to switch to a separate interrupt stack. This behaviour is controlled by the common HAL configuration option CYGIMP_HAL_COMMON_INTERRUPTS_USE_INTERRUPT_STACK. It reduces the stack requirements of all threads in the system, at the cost of some extra instructions during interrupt handling. In kernel configurations the startup stack is no longer used once the scheduler starts running so its memory can be reused for the interrupt stack. To handle the possibility of nested interrupts the interrupt handling code will detect if it is already on the interrupt stack, so in non-kernel configurations it is also safe to use the same area of memory for both startup and interrupt stacks. This leads to the following scenarios:

1. If interrupt stacks are enabled via CYGIMP_HAL_COMMON_INTERRUPTS_USE_INTERRUPT_STACK and the interrupt stack is not provided by the variant, processor or platform HAL then a single block of memory will be used for both startup and interrupt stacks. The size of this block is determined by the common HAL configuration option CYGNUM_HAL_COMMON_INTERRUPTS_STACK_SIZE, with a default value CYGNUM_HAL_DEFAULT_INTERRUPT_STACK_SIZE provided by the H8300 architectural HAL.
2. If the use of an interrupt stack is disabled then the H8/300 architectural HAL will provide just the startup stack, unless this is done by the variant, processor or platform HAL. The size of the startup stack is still controlled by CYGIMP_HAL_COMMON_INTERRUPTS_USE_INTERRUPT_STACK.
3. Otherwise the interrupt and/or startup stacks are provided by other packages and it is up to those packages to provide configuration options for setting the sizes.

SCI Baud Rate

The architecture HAL provides a polled driver for the SCI serial device that is common to all H8/300 implementations. This is used by RedBoot for user interaction and by eCos applications for diagnostic output. Configuration of the device is handled in the variant HAL, but the baud rate is set in the architecture HAL using the CYGNUM_HAL_H8300_SCI_BAUD_RATE option.

Interrupt Vector Hook

The H8/300 architecture does not provide any mechanism for determining the source of an interrupt or exception other than the address of the routine to which control is vectored. Since eCos uses common code for most vectors, and demultiplexes the source later, this makes handling difficult. To overcome this, all interrupt and exception vectors are pointers to entries in a table of single instruction JSR instructions which all call the common handling code. The stacked return address can then be used to synthesize a vector number that is used by the common handling code.

By default this JSR hook table is stored in on-chip RAM (it is actually placed by the platform specific .ldi file). This placement makes this vectoring fast and of minimal impact on performance. It also allows individual vectors to be replaced if desired -- although this functionality is also provided in a more convenient form by the standard eCos VSR table that is also placed in on-chip RAM. If the option CYGSEM_HAL_H8300_VECTOR_HOOK is disabled, then the hook table will be stored in ROM, which will free space in the on-chip RAM but will increase interrupt processing time and remove the ability to revector exceptions at this level (but won't affect the VSR table).

If the option CYGSEM_HAL_H8300_SAVE_STUB_VECTOR is enabled then the vector hook table will be saved by an eCos application before being replaced with its own table. This is theoretically to permit RedBoot to coexist with eCos. In practice the common VSR table mechanism handles this in a more portable manner.

Other Options

The H8/300 architectural HAL package does not define any other configuration options that can be manipulated by the user.

Name

HAL Port — Implementation Details

Description

This documentation explains how the eCos HAL specification has been mapped onto H8/300 hardware, and should be read in conjunction with that specification. It also describes how variant, processor and platform HALs can modify the default behaviour.

eCos support for any given target will involve either three or four HAL packages: the architectural HAL, the platform HAL, the variant HAL, and optionally a processor HAL. This package, the architectural HAL, provides code and definitions that are applicable to all H8/300 processors. The platform HAL provides support for one specific board, or possibly for a number of almost-identical boards. The processor HAL, if present, serves mainly to provide details of on-chip peripherals including the interrupt controller. The variant HAL provides functionality that is common to a group of processors, for example all H8S processors have very similar UARTs and hence can share HAL diagnostic code. There is no fixed specification of what should go into the variant HAL versus the processor HAL. For simplicity the description below only refers to variant HALs, but the work may actually happen in a processor HAL instead.

As a design goal lower-level HALs can always override functionality that is normally provided higher up. For example the architectural HAL will provide the required eCos `HAL_LSBIT_INDEX` and `HAL_MSBIT_INDEX` macros, but these can be provided lower down instead. In some areas such as handling context switching the architectural HAL will usually provide the basic functionality but it may be extended by lower HALs. The architecture HAL consequently contains a large number of macros at both C and assembler level that variant HALs are expected to supply functionality for.

The architectural HAL provides header files `cyg/hal/hal_arch.h`, `cyg/hal/hal_intr.h`, `cyg/hal/hal_cache.h`, `cyg/hal/hal_io.h` and `cyg/hal/arch.inc`. These automatically include an equivalent header file from the variant HAL, for example `cyg/hal/var_arch.h`. The variant HAL header will in turn include processor and platform-specific headers. This means that application developers and other packages can simply include the architectural HAL headers without needing to know about variants or platforms. It also allows the variant and platform HALs to override architectural settings.

Data Types

For eCos purposes all H8/300 processors are big-endian and 32-bit, so the default data types in `cyg/infra/cyg_type.h` are used. Some variants have external bus widths less than 32-bit, but this does not affect the architectural HAL.

Startup and Exception Vectors

The conventional bootstrap mechanism involves a table of exception vectors at the base of memory. The first two words of this table are reset entry points for power-on reset and manual reset. In a typical embedded system the hardware is arranged such that non-volatile flash memory is found at location 0x0 so it is the start of flash that contains the exception vectors and the boot code. The table of exception vectors is used subsequently for interrupt handling and for hardware exceptions.

The exact hardware details, the various startup types, the steps needed for low-level hardware initialization, and so on are not known to the architectural HAL. Hence although the architectural HAL does provide the basic framework for startup, much of the work is done via macros provided by lower-level HAL packages and those macros are likely to depend on various configuration options. Rather than try to enumerate all the various combinations here it is better to look at the actual code in `vectors.S` and in appropriate variant, processor or platform HALs. `vectors.S` is responsible for any low-level initialization that needs to happen. This includes setting up a standard C environment with the stack pointer set to the startup stack in working RAM, making sure all statically initialized global variables have the correct values, and that all uninitialized global variables are zeroed. Once the C environment has been set up the code jumps to `cyg_start()` which completes the initialization and jumps to the application entry point.

Interrupt Handling

The H8/300 architecture reserves a vector table area of memory for exception vectors. These are used for internal and external interrupts, exceptions, software traps, and special operations such as reset handling. Some of the vectors have well-defined uses.

However when it comes to interrupt handling the details will depend on the processor variant and on the platform, and the appropriate package documentation should be consulted for full details.

The default behaviour is for all exceptions and interrupts to be vectored from the hardware vector table via the JSR hook table to a piece of trampoline code. This saves the CPU state on the stack and decodes the hook table return address into a simple vector number. This is then used to index the VSR table and fetch the address of the Vector Service Routine for that exception. This is then called with the vector number in ER1.

The standard eCos macros HAL_VSR_GET and HAL_VSR_SET just manipulate one of the entries in the VSR table. Hence it is possible to replace the default handlers for exceptions and traps in addition to interrupt handlers. hal_intr.h provides #define's for the more common exception vectors, and additional ones can be provided by the platform or variant. It is the responsibility of the platform or variant HAL to initialize the table, and to provide the HAL_VSR_SET_TO_ECOS_HANDLER macro since that requires knowledge of the default table entries.

At the architecture level there is no fixed mapping between VSR and ISR vectors. Instead that is left to the variant or platform HAL. The architectural HAL does provide default implementations of HAL_INTERRUPT_ATTACH, HAL_INTERRUPT_DETACH and HAL_INTERRUPT_IN_USE since these just involve updating a static table.

By default the interrupt state control macros HAL_DISABLE_INTERRUPTS, HAL_RESTORE_INTERRUPTS, HAL_ENABLE_INTERRUPTS and HAL_QUERY_INTERRUPTS are implemented by the variant HAL for the different processor variants, and involve updating either the condition code or extended control registers.

HAL_DISABLE_INTERRUPTS has no effect on non-maskable interrupts. This causes a problem because parts of the system assume that all normal interrupt sources are affected by this macro. If the target hardware can raise non-maskable interrupts then it is the responsibility of application code to install a suitable VSR and handle non-maskable interrupts entirely within the application, bypassing the usual eCos ISR and DSR mechanisms.

The architectural HAL does not provide any support for the interrupt controller management macros like HAL_INTERRUPT_MASK. These can only be implemented on a per-variant, per-processor or per-platform basis.

Exception Handling

Synchronous exception handling is done in much the same way as interrupt handling.

The details of exception handling vary from one variant to the next depending on the interrupt control mode. The architectural HAL makes no attempt to cope with these differences, and it is the responsibility of the variants to provide more advanced support. Otherwise if an exception needs to be handled in a very specific way then it is up to the application to install a suitable VSR and handle the exception directly.

Stacks and Stack Sizes

cyg/hal/hal_arch.h defines values for minimal and recommended thread stack sizes, CYGNUM_HAL_STACK_SIZE_MINIMUM and CYGNUM_HAL_STACK_SIZE_TYPICAL. These values are specific to the current configuration, and are affected mainly by options related to interrupt handling.

By default eCos uses a separate interrupt stack, although this can be disabled through the configuration option CYGIMP_HAL_COMMON_INTERRUPTS_USE_INTERRUPT_STACK. When an interrupt or exception occurs eCos will save the context on the current stack and then switch to the interrupt stack before calling the appropriate ISR interrupt handler. This means that thread stacks can be significantly smaller because there is no need to worry about interrupt handling overheads, just the thread context. However switching the stack does require some extra work and hence increases the interrupt latency. Disabling the interrupt stack removes this processing overhead but requires larger stack sizes. It depends on the application whether or not this is a sensible trade off.

By default eCos does not allow nested interrupts, but this can be controlled via the configuration option CYGSEM_HAL_COMMON_INTERRUPTS_ALLOW_NESTING. Supporting nested interrupts requires larger thread stacks, especially if the separate interrupt stack is also disabled. It may also require additional support from the variant and platform HALs. Note that at present this support is not complete in any variant, so interrupt nesting is currently disabled.

The H8/300 is somewhat register-poor, and although the calling conventions are register-oriented, a lot of use is made of stack space. In particular register contents must be spilled to the stack frequently, and the return address is pushed rather than ending up in a link register. To allow for this the recommended minimum stack sizes are a little bit larger than for some other architectures. Variant HALs cannot directly affect these stack sizes.

Usually the H8/300 architectural HAL will provide a single block of memory which acts as both the startup and interrupt stack, and there are configuration options to control the size of this block.

Thread Contexts and Setjmp/Longjmp

A typical thread context consists of the following:

1. The integer context. This consists of the data registers `ER0` to `ER6`. The stack pointer register `ER7` does not have to be always saved explicitly since it is implicit in the pointer to the saved context.

   The caller-save registers are `ER0` to `ER2`, and the condition code register. The remaining registers are callee-save. The result is passed back via `ER0`.

2. The condition code register, the program counter, and extended status register `EXR` on H8S. These are special because when an interrupt occurs the hardware automatically pushes these onto the stack, but exactly what gets pushed depends on the variant.

`setjmp` and `longjmp` only deal with the callee-save registers. The variant HAL package can override the default implementations if necessary.

When porting to a new H8/300 variant, the variant HAL must define a number of assembler-level macros to customize the behaviour of the architecture HAL to the variant. These are too numerous to specify in detail here and the reader is directed to look at the existing HAL ports for examples.

Bit Indexing

For performance reasons the `HAL_LSBIT_INDEX` and `HAL_MSBIT_INDEX` macros are implemented using functions containing inline assembler. A variant HAL can override the default definitions if, for example, the variant has special instructions to perform these operations.

Idle Thread Processing

The default `HAL_IDLE_THREAD_ACTION` implementation is a no-op. A variant HAL may override this, for example to put the processor into sleep mode. Alternative implementations should consider exactly how this macro gets used in eCos kernel code.

Clock Support

The architectural HAL cannot provide the required clock support because it does not know what timer hardware may be available on the target hardware. Instead this is left to either the variant or platform HAL, depending on whether the processor has a suitable on-chip timer or whether an off-chip timer has to be used.

HAL I/O

The H8/300 architecture does not have a separate I/O bus. Instead all hardware is assumed to be memory-mapped. Further it is assumed that all peripherals on the memory bus are wired appropriately for a big-endian processor and that there is no need for any byte swapping. Hence the various HAL macros for performing I/O simply involve pointers to volatile memory.

The variant, processor and platform equivalents of the `cyg/hal/hal_io.h` header will typically also provide details of some or all of the peripherals, for example register offsets and the meaning of various bits in those registers.

Diagnostic Support

The architecture HAL provides an implementation of the SCI serial device that is common to all H8/300 microcontrollers. However, it is the responsibility of the the variant or platform HAL to provide the register definitions and to instantiate the devices by calling `cyg_hal_plf_sci_init()`.

SMP Support

The H8/300 port does not have SMP support.

Debug Support

The H8300 architectural HAL package provides basic support only for gdb stubs. There is no support for more advanced debug features like hardware watchpoints. Trace-based single step is supported on the H8S variant.

Other Functionality

The H8/300 architectural HAL only implements the functionality provided by the eCos HAL specification and does not export any extra functionality.

Part LXXX. i386 Architecture

Table of Contents

Chapter 315. I386 PC Support

Name

eCos Support for the i386 PC — Overview

Description

This document covers the eCos support for all i386 based PCs. This configuration of eCos should run on all i386/486/Pentium motherboards and PC compatible embedded devices.

For typical eCos development, a RedBoot image is programmed onto a disk device and the board will boot this image from reset. RedBoot provides gdb stub functionality so it is then possible to download and debug stand-alone and eCos applications via the gdb debugger. This can happen over either a serial line or over ethernet.

Supported Hardware

eCos runs the i386 CPU in 32-bit protected mode. The segment registers are initialized to provide a flat 32-bit address space and the MMU is not enabled. Coherence between the cache and device memory is handled entirely by the hardware.

There is a serial driver `CYGPKG_IO_SERIAL_GENERIC_16X5X` which supports the 16X5X UARTs used by the PC. The `CYGPKG_IO_SERIAL_I386_PC` package provides customization of this generic driver to the PC hardware. These devices can be used by RedBoot for communication with the host. If any of these devices is needed by the application, either directly or via the serial driver, then it cannot also be used for RedBoot communication. Another communication channel such as ethernet should be used instead. The serial driver packages are loaded automatically when configuring for the PC target.

Supported Ethernet devices include the Intel i82559, Intel i82544, National Semiconductor DP83816 and RealTek RTL8139. Each of these devices is supported by a generic device driver plus a package that customizes it to the PC hardware environment.

eCos manages the standard PC priority interrupt controller. PIT timer 0 is used to implement the eCos system clock and the microsecond delay function. eCos assumes that the PCI bus will be configured by the BIOS.

Tools

The i386 port is intended to work with GNU tools configured for an i386-elf target. The original port was undertaken using i386-elf-gcc version 3.2.1, i386-elf-gdb version 5.3, and binutils version 2.13.1.

Name

Setup — Preparing a PC for eCos Development

Overview

In a typical development environment, the PC boots into the RedBoot monitor from a floppy disk. eCos applications are configured for RAM startup and then downloaded and run on the board via the debugger **i386-elf-gdb**. Preparing for development therefore involves writing a suitable RedBoot image onto a floppy disk.

The following RedBoot configurations are supported:

Configuration	Description	Use	File
FLOPPY	RedBoot booted from a floppy disk and running in the bottom 640k of RAM	`redboot_FLOPPY.ecm`	`redboot_FLOPPY.bin`
GRUB	RedBoot loaded by the GRUB bootloader into memory above 0x00100000	`redboot_GRUB.ecm`	`redboot_GRUB.bin`
ROM	RedBoot booted directly from ROM (experimental and unsupported)	`redboot_ROM.ecm`	`redboot_ROM.bin`
FLOPPY_SMP	RedBoot booted from a floppy disk and running in the bottom 640k of RAM. This version supports dual processor systems.	`redboot_FLOPPY_SMP.ecm`	`redboot_FLOPPY_SMP.bin`
GRUB_SMP	RedBoot loaded by the GRUB bootloader into memory above 0x00100000. This version supports dual processor systems.	`redboot_GRUB_SMP.ecm`	`redboot_GRUB_SMP.bin`

For serial communications, all versions run with 8 bits, no parity, and 1 stop bit at 38400 baud. RedBoot also supports ethernet communication and flash management.

Initial Installation

Floppy Installation

RedBoot takes the form of a self-booting image that must be written onto a formatted floppy disk. The process will erase any file system or data that already exists on that disk, so proceed with caution.

For Red Hat Linux users, writing the RedBoot image to floppy disk this can be achieved using the following command:

```
$ dd conv=sync if=install/bin/redboot.bin of=/dev/fd0H1440
```

For Windows users with recent versions of Cygwin, the raw floppy device should be accessible as `/dev/fd0`. Users with older versions of Cygwin may need to mount the floppy drive explicitly using the command:

```
$ mount -f -b //./a: /dev/fd0
```

To actually install the boot image on the floppy under Cygwin, use the command:

```
$ dd conv=sync if=install/bin/redboot.bin of=/dev/fd0
```

Insert this floppy in the A: drive of the PC to be used as a target and ensure that the BIOS is configured to boot from A: by default. On reset, the PC will boot from the floppy and the target will be ready for GDB debug sessions via either serial line, or the ethernet interface if it is installed.

NOTE

Unreliable floppy media may cause the write to silently fail. This can be determined if the RedBoot image does not correctly boot. In such cases, the floppy should be (unconditionally) reformatted using the **fdformat** command on Linux, or **format a: /u** on DOS/Windows.

VMWare Installation

The PC platform HAL may also be run under VMWare (Player, Server, etc). This can provide an initial development environment using simulated i386 PC hardware. The setup is similar to a floppy installation. However no IDE devices should be included in the guest definition due to issues with VMWare. The AMD Lance ethernet device is also recommended as the mechanism to communicate to the simulated PC hardware. The remainder of this sub-section describes how to set up a suitable virtual machine under VMWare that will be bootstrapped by RedBoot and allow for debugging of i386 PC applications either over the network or using serial communications.

First create a RedBoot floppy bootstrap image with the AMD Lance ethernet device included:

```
% ecosconfig new pc_vmWare redboot
% ecosconfig import $ECOS_REPOSITORY/hal/i386/pc/VERSION/misc/redboot_FLOPPY.ecm
% ecosconfig tree
% make
```

Create a VMWare virtual machine definition with no SCSI or IDE interfaces, 1MB memory, a single serial interface to a file, NAT Network adaptor and a floppy. Copy the `redboot.bin` you created above into place on the VMWare host and point the floppy image to this file. Example `.vmx` and `.vmxf` files for this configuration may be found in the `misc` subdirectory of the PC hal (`$ECOS_REPOSITORY/packages/hal/i386/pc/`). You may use other types of Network connections or serial ports as required.

To configure the Network connection to be an AMD Lance ethernet, you must edit the `.vmx` if you created your own VM definition and set `ethernet0.virtualDev` to `vlance`.

Note

The example `.vmx` file is intended for a Windows VMWare host so the setting `serial0.fileName` also must be adjusted accordingly.

If you intend to develop on a different host from the VMWare host, you may also wish to set up a NAT port forward from the VMWare host to the PC running RedBoot. An example Linux `nat.conf` NAT configuration file may also be found in the `misc` directory. For Windows, the Virtual Network Editor is recommended.

When the VM is powered on in this configuration, you will be able to download and debug executables through either the network connection or through the virtualised serial port using RedBoot.

GRUB Installation

If RedBoot is built with the GRUB startup type, it is configured to be loaded by the GRUB bootloader.

GRUB is an open source boot loader that supports many different operating systems. It is available from http://www.gnu.org/software/grub. The latest version of GRUB should be downloaded from there and installed. GRUB is now the default bootloader for most Linux distributions and therefore is already installed in many systems.

To install GRUB on a floppy disk from Linux you need to execute the following commands, with a fresh floppy diskette in the main drive:

```
$ mformat a:
$ mount /mnt/floppy
$ grub-install --root-directory=/mnt/floppy '(fd0)'
Probing devices to guess BIOS drives. This may take a long time.
Installation finished. No error reported.
This is the contents of the device map /mnt/floppy/boot/grub/device.map.
Check if this is correct or not. If any of the lines is incorrect,
fix it and re-run the script `grub-install'.

(fd0) /dev/fd0
$ cp $ECOS_REPOSITORY/packages/hal/i386/pc/current/misc/redboot_menu.lst /mnt/floppy/boot/grub/menu.lst
$ umount /mnt/floppy
```

The file `redboot_menu.lst` is a GRUB menu configuration file. It contains a menu item to load RedBoot from the floppy diskette. Alternatively you can use the command-line interface of GRUB to input commands yourself.

To install RedBoot on the diskette, execute the following command:

```
$ mcopy redboot_GRUB.img a:/boot/redboot
```

Insert this floppy in the A: drive of the PC to be used as a target and ensure that the BIOS is configured to boot from A: by default. On reset, the PC will boot from the floppy into GRUB which will display its Boot Menu. If left for 30 seconds it will boot into RedBoot automatically. However, typing a return on the keyboard will cause it to boot RedBoot immediately.

To install GRUB on a hard disk, refer to the GRUB documentation. Be warned, however, that if you get this wrong it may compromise any existing bootloader that exists on the hard disk and may make any other operating systems unbootable. Practice on floppy disks or sacrificial hard disks first. On machines already running a GRUB-booted Linux you can just add your own menu items to the `/boot/grub/menu.lst` file that already exists.

Rebuilding RedBoot

Should it prove necessary to rebuild a RedBoot binary, this is achieved most conveniently at the command line. The steps needed to rebuild the the FLOPPY version of RedBoot for the PC are:

```
$ mkdir redboot_pc_floppy
$ cd redboot_pc_floppy
$ ecosconfig new TARGET redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/i386/pc/VERSION/misc/redboot_FLOPPY.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

When following the above instructions, the user should adopt one of i386 PC target names detailed in the following section to build RedBoot with support for a specific ethernet adapter. At the end of the build the `install/bin` subdirectory should contain the file `redboot.bin`.

The steps needed to rebuild the the GRUB version of RedBoot for the PC are:

```
$ mkdir redboot_pc_grub
$ cd redboot_pc_grub
$ ecosconfig new pc redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/i386/pc/current/misc/redboot_GRUB.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

At the end of the build the `install/bin` subdirectory should contain the file `redboot.img.`.

Name

Configuration — Platform-specific Configuration Options

Overview

The PC platform HAL package is loaded automatically when eCos is configured for a `pc` target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The PC platform HAL package accommodates four separate startup types:

RAM

> This is the startup type which is normally used during application development. The board has RedBoot running in low memory and boots into that initially. i386-elf-gdb is then used to load a RAM startup application into memory above 0x00200000 and debug it. It is assumed that the hardware has already been initialized by RedBoot. By default the application will use the eCos virtual vectors mechanism to obtain certain services from RedBoot, including diagnostic output.

FLOPPY

> This startup type can be used for finished applications which will be loaded from a floppy disk by the BIOS. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization. Such applications are limited to running only within the bottom 640k of RAM.

GRUB

> This startup type can be used for finished applications which can be loaded using the GRUB boot loader. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization. These applications are loaded above the 0x00100000 boundary and therefore have all of the upper RAM area available.
>
> The load address for GRUB applications is 0x00108000 while that for RAM applications is 0x00200000. This allows a GRUB-loaded RedBoot to occupy the first 1MB of upper memory and while allowing RAM applications to be loaded beyond the second 1MB boundary, avoiding any potential clashes.

ROM

> This startup type can be used for eCos applications booting directly from ROM. The ROM startup code for the i386 PC target is experimental at present and ROM startup is therefore unsupported in this release.

Floppy Startup Installation

If an application is built with a startup type of FLOPPY, then it is configured to be a self-booting image that must be written onto a formatted floppy disk. This will erase any existing file system or data that is already on the disk, so proceed with caution.

To write an application to floppy disk, it must first be converted to a pure binary format. This is done with the following command:

```
$ i386-elf-objcopy -O binary app.elf app.bin
```

Here `app.elf` is the final linked application executable, in ELF format (it may not have a `.elf` extension). The file `app.bin` is the resulting pure binary file. This must be written to the floppy disk with the following command:

```
$ dd conv=sync if=app.bin of=/dev/fd0
```

Insert this floppy in the A: drive of the PC to be used as a target and ensure that the BIOS is configured to boot from A: by default. On reset, the PC will boot from the floppy and the eCos application will load itself and execute immediately.

NOTE

Unreliable floppy media may cause the write to silently fail. This can be determined if the application image does not correctly boot. In such cases, the floppy should be (unconditionally) reformatted using the **fdformat** command on Linux, or **format a: /u** on DOS/Windows. If this fails, try a different disk.

GRUB Bootloader Support (version 0.97)

If an application is built with the GRUB startup type, it is configured to be loaded by the GRUB bootloader.

GRUB is an open source boot loader that supports many different operating systems. It is available from http://www.gnu.org/software/grub. The latest version of GRUB should be downloaded from there and installed. GRUB is now the default bootloader for Linux and therefore is already installed in many installations.

To install GRUB on a floppy disk from Linux you need to execute the following commands:

```
$ mformat a:
$ mount /mnt/floppy
$ grub-install --root-directory=/mnt/floppy '(fd0)'
Probing devices to guess BIOS drives. This may take a long time.
Installation finished. No error reported.
This is the contents of the device map /mnt/floppy/boot/grub/device.map.
Check if this is correct or not. If any of the lines is incorrect,
fix it and re-run the script `grub-install'.

(fd0) /dev/fd0
$ cp $ECOS_REPOSITORY/packages/hal/i386/pc/VERSION/misc/menu.lst /mnt/floppy/boot/grub
$ umount /mnt/floppy
```

The file `menu.lst` is an example GRUB menu configuration file. It contains menu items to load some of the standard eCos tests from floppy or from partition zero of the first hard disk. You should, of course, customize this file to load your own application. Alternatively you can use the command-line interface of GRUB to input commands yourself.

Applications can be installed, or updated simply by copying them to the floppy disk at the location expected by the `menu.lst` file. For booting from floppy disks it is recommended that the executable be stripped of all debug and symbol table information before copying. This reduces the size of the file and can make booting faster.

To install GRUB on a hard disk, refer to the GRUB documentation. Be warned, however, that if you get this wrong it may compromise any existing bootloader that exists on the hard disk and may make any other operating systems unbootable. Practice on floppy disks or sacrificial hard disks first. On machines already running a GRUB-booted Linux you can just add your own menu items to the `/boot/grub/menu.lst` file that already exists.

NOTE

Certain distributions of Linux, including Red Hat Linux, supply a version of GRUB which references a configuration file named `grub.conf` rather than `menu.lst`.

GRUB 2 Bootloader Support (version 1.98)

If an application is built with the GRUB startup type, it is configured to be loaded by the GRUB bootloader.

GRUB 2 is an open source boot loader that supports many different operating systems. GRUB 2 is the sucessor to the legacy GRUB boot loader. It has been rewritten and requires a new configuration file that is different from the legacy GRUB loader. It is available from http://www.gnu.org/software/grub. The latest version of GRUB should be downloaded from there and installed. Cygwin users will need to install the GRand Unified Bootloader package on their system.

To install GRUB 2 on a disk drive from Cygwin you will first need to format your disk. Begin by launching a windows Command Prompt with ADMINISTRATOR priviledges. This example assumes that your destination drive is E and the filesystem type is FAT32.

```
$ format e: /fs:fat32 /q

The type of the file system is FAT32.
WARNING, ALL DATA ON NON-REMOVABLE DISK
DRIVE E: WILL BE LOST!
Proceed with Format (Y/N)? y
QuickFormatting 512M
Initializing the File Allocation Table (FAT)...
Volume label (11 characters, ENTER for none)?
Format complete.
```

When your drive is finished formatting, Cygwin will automatically mount the drive under /cygdrive/e. Next, launch a bash shell. Cygwin users should run their shell as ADMINISTRATOR otherwise the grub-install program will fail. To install GRUB we need to know the device name for your disk. A second disk drive is usually called /dev/sdb. The C: drive is usually called /dev/sda. Determine your device name by reading the /proc/partitions file. This example uses /dev/sdb as input to the GRUB install program.

```
$ cat /proc/partitions
major minor  #blocks  name
    8     0 156290904 sda
    8     1    102400 sda1
    8     2 156185600 sda2
    8    16 156290904 sdb
    8    17    524288 sdb1

$ grub-install --root-directory=/cygdrive/e /dev/sdb
Installation finished. No error reported.

$ cp $ECOS_REPOSITORY/packages/hal/i386/pc/current/misc/grub.cfg /cygdrive/e/boot/grub

$ cp redboot.elf /cygdrive/e/boot
```

After installing GRUB 2, we need to a copy a configuration file for grub to use. The file `grub.cfg` is an example configuration file that loads redboot.elf. You should, of course, customize this file to load your own application. Alternatively you can use the command-line interface of GRUB to input commands yourself. Applications can be installed or updated simply by copying them to the location expected by the `grub.cfg` file. In this example the /boot directory is used.

When installing GRUB on a hard disk, refer to the GRUB documentation. Be warned, however, that if you get this wrong it may compromise any existing bootloader that exists on the hard disk and may make any other operating systems unbootable. Practice on a spare disk or sacrificial hard disks first.

Debugging FLOPPY and GRUB Applications

When RedBoot loads an application it also provides debugging services in the form of GDB remote protocol stubs. When an application is loaded stand-alone from a floppy disk, or by GRUB, these services are not present. To allow these application to be debugged, it is possible to include GDB stubs into the application.

To do this, set the "Support for GDB stubs" (`CYGDBG_HAL_DEBUG_GDB_INCLUDE_STUBS`) configuration option. Following this any application built will allow GDB to connect to the debug serial port (by default serial device 0, also known as COM1) whenever the application takes an exception, or if a Control-C is typed to the debug port. Ethernet debugging is not supported.

The option "Enable initial breakpoint" (`CYGDBG_HAL_DEBUG_GDB_INITIAL_BREAK`) causes the HAL to take a breakpoint immediately before calling cyg_start(). This gives the developer a chance to set any breakpoints or inspect the system state before it proceeds. The configuration sets this option by default if GDB stubs are included, and this is not a RedBoot build. To make the application execute immediately either disable this option, or disable `CYGDBG_HAL_DEBUG_GDB_INCLUDE_STUBS`.

RedBoot and Virtual Vectors

If the application is intended to act as a monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building RedBoot.

If the application is supposed to make use of services provided by a monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, as a testing step before switching to FLOPPY or GRUB startup.

If the application does not rely on a monitor for diagnostic services then the serial port will be claimed for HAL diagnostics.

Ethernet Drivers

eCos is designed to support typical embedded development and production boards. These usually have a fixed set of hardware devices which are either on-chip or are soldered to the board. The PC target is unusual in that is does not have a fixed ethernet device, instead a variety PCI ethernet cards may be installed in the PCI card slots.

eCos supports different ethernet cards by defining a separate target configuration for each supported device. An instance of eCos or RedBoot configured to use one device will not work with a different ethernet device installed. The following table lists the targets and driver packages for the supported devices.

Device	Target	Driver Packages	Cards
Intel i82559	pc_i82559	CYGPKG_DEVS_ETH_INTEL_I82559, CYGPKG_DEVS_ETH_I386_PC_I82559	Intel Pro 10/100
RealTek RTL8139	pc_rltk8139	CYGPKG_DEVS_ETH_RLTK_8139, CYGPKG_DEVS_ETH_I386_PC_RLTK8139	D-Link DFE-538TX
National Semiconductor DP83816	pc_dp83816	CYGPKG_DEVS_ETH_NS_DP83816, CYGPKG_DEVS_ETH_I386_PC_DP83816	Netgear FA311
Intel i82544	pc_i82544	CYGPKG_DEVS_ETH_INTEL_I82544, CYGPKG_DEVS_ETH_I386_PC_I82544	Intel Pro 1000
AMD Lance PCNet32	pc_vmWare	CYGPKG_DEVS_ETH_AMD_LANCEPCI, CYGPKG_DEVS_ETH_I386_PC_LANCEPCI	VMWare vlance

Entries in the cards column are examples only. There are for example many cards that contain the RealTek RTL8139 or a compatible device. Also be aware that manufacturers may change the device on a particular card to a totally different one without changing the model number.

System Clock

By default, the system clock interrupts once every 10ms, corresponding to a 100Hz clock. This can be changed by the configuration option `CYGNUM_HAL_RTC_DENOMINATOR` which corresponds to the clock frequency. Other clock-related settings are recalculated automatically if the denominator is changed.

Compiler Flags

The platform HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos. There are no i386 specific flags that need to be specified for a PC platform.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the PC hardware, and should be read in conjunction with that specification. The PC platform HAL package complements the i386 architectural HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset the HAL will initialize or reinitialize the on-chip peripherals that are used by eCos. There is an exception for RAM startup applications which depend on a monitor for certain services.

Linker Scripts and Memory Maps

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

Low RAM

This is the original 640KB of PC RAM between 0x00000000 and 0x000a0000. For FLOPPY startup this is all the RAM that is available since the BIOS operations used to load the executable off the diskette can only write to this memory region.

The lower few K bytes of this region are allocated to special uses as follows:

Base	Size	Purpose
0x00000000	0x1000	The BIOS stores various system parameters in this area, and it is left untouched in case these values are of use.
0x00001000	0x800	The Interrupt Descriptor Table (IDT). Space for a full sized 256 entry table is left, although a full size table is not normally created.
0x00001800	0x400	The Vector Service Routine (VSR) table. There is space here for 256 entries, matching the IDT, although not all will be used.
0x00001c00	0x400	The Virtual Vector Table. This is used to pass control between RedBoot and the application for access to services and debugging.
0x00002000	0x1000	This area is used in SMP configurations to start up and synchronize the slave CPUs. It is unused in non-SMP configurations.
0x00003000	0xbd000	Any FLOPPY startup application will load itself at 0x00003000 and use RAM up to the 0x000a0000 boundary. In RAM and GRUB startup configurations this area is unused.

Reserved Region

The region between 0x000a0000 and 0x00100000 is reserved for ROMs, devices and display memory. The only thing of real interest here is the character display buffer at 0x000B8000.

Main Memory

The region above 0x00100000 is the main memory area. This is where RAM and GRUB startup applications are loaded. The upper limit of this region depends on the amount of RAM fitted and is determined at runtime by querying the BIOS.

PCI Devices

The exact memory region used to map PCI devices is largely dependent on the BIOS, but is usually placed above the 0xD0000000 boundary. Drivers for PCI devices will usually determine the location of any device memory regions by querying the device configuration.

IO Ports

The i386 architecture defines a separate address space for IO device registers. These are accessed by the IO instructions. All the standard PC devices are available in this space, and any PCI devices that define IO ports will also be allocated here by the BIOS. In eCos these ports are accessed using the `HAL_READ_XXX()` and `HAL_WRITE_XXX()` macros defined in the `hal_io.h` header.

SMP Support

The i386 HAL contains support for Symmetric Multi-Processing (SMP). The HAL expects the machine to be running under a multiprocessor-aware BIOS and expects to find an MP configuration table from which to configure itself. The HAL also switches over to using the per-CPU APICs and the shared IOAPIC for interrupt control in preference to the standard PIC.

SMP support in eCos is enabled by setting the `CYGPKG_KERNEL_SMP_SUPPORT` configuration option. This will cause the HAL-level support to be enabled. If applications are to be run under RedBoot then an SMP-aware RedBoot must be used. The `FLOPPY_SMP` and `GRUB_SMP` configurations of RedBoot supply this.

Other Issues

The PC platform HAL does not affect the implementation of other parts of the eCos HAL specification. The generic i386 variant HAL, and the I386 architectural HAL documentation should be consulted for further details.

Chapter 316. STPC Atlas Support

Name

CYGPKG_HAL_I386_STPC_ATLAS — eCos Support for the STPC Atlas Processor

Description

The STPC Atlas is an x86 core PC compatible system-on-chip intended for embedded applications. The central processor is largely 486-compatible and can run at up to 133MHz. The chip includes an integrated SDRAM controller, a VGA/SVGA graphics controller with TFT panel support, two serial ports, a parallel port, keyboard, mouse and USB host interfaces, support for PCI, PCMCIA and ISA buses as well as a local bus, and interrupt controller, timers, and DMA engines as per the standard PC architecture.

The STPC Atlas variant HAL package CYGPKG_HAL_I386_STPC_ATLAS provides support for all platforms based around this chip. It complements the I386 architectural HAL CYGPKG_HAL_I386. An eCos configuration for an STPC Atlas-based platform should also include a platform HAL package to support board-level details like the nature of the external memory chips.

Configuration

The STPC Atlas variant HAL package should be loaded automatically when eCos is configured for appropriate target hardware. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

The package does not contain any configuration options.

The HAL Port

This section describes how the STPC Atlas variant HAL package implements parts of the eCos HAL specification. It should be read in conjunction with similar sections from the architectural and platform HAL documentation.

HAL I/O

The cyg/hal/var_io.h header provides definitions for the on-chip peripherals. This header file is automatically included by the architectural cyg/hal/hal_io.h so other packages and application code will usually only include the latter. It is also necessary to include cyg/hal/hal_intr.h. The definitions largely follow the STPC documentation, so for example the ISA Port_B register has a definition HAL_STPC_ATLAS_Port_B, and there also definitions HAL_STPC_ATLAS_Port_B_PE and HAL_STPC_ATLAS_Port_B_SE for the PE and SE bits.

STPC Atlas registers can be accessed in a variety of ways. Some of them can be accessed directly via the x86 in and out instructions, and the eCos macros HAL_READ_UINT8, HAL_WRITE_UINT8, etc. can be used for these. However there are also memory-mapped registers, registers accessed indirectly via the IDX and DATA registers, local bus registers, PCI registers, and so on. In an attempt to reduce confusion various suffixes are used, and in some cases utility macros are provided to access the registers:

Type	Suffix	Size	Access using
Normal I/O port	None	8, 16 or 32	HAL_READ_UINT8, HAL_WRITE_UINT8 etc.
Indexed via 0x22/0x23	_IDX	8 bits	HAL_STPC_ATLAS_READ_IDX and HAL_STPC_ATLAS_WRITE_IDX
Host bus	_HB	32 bits	HAL_STPC_ATLAS_READ_HB and HAL_STPC_ATLAS_WRITE_HB
Local bus	_LB	16 bits	HAL_STPC_ATLAS_READ_LB and HAL_STPC_ATLAS_WRITE_LB
PC	_PCI	32 bits	HAL_STPC_ATLAS_READ_PCI and HAL_STPC_ATLAS_WRITE_PCI

Type	Suffix	Size	Access using
PCMCIA	_PCMCIA	8 bits	HAL_STPC_ATLAS_READ_PCMCIA and HAL_STPC_ATLAS_WRITE_PCMCIA
Memory	_MEM	32 bits	as C pointers

Accessing IDX, HB, LB, PCI and PCMCIA registers involves non-atomic sequences of operations so to avoid concurrency problems the associated macros briefly disable interrupts. If this is known to be unnecessary, for example because the relevant code runs during system initialization before interrupts are enabled, then INTS_UNSAFE variants of the macros such as HAL_STPC_ATLAS_READ_IDX_INTS_UNSAFE can be used instead.

Interrupts

The STPC Atlas variant HAL provides default implementations of the HAL macros related to the interrupt controller: HAL_INTERRUPT_ACKNOWLEDGE, HAL_INTERRUPT_MASK, HAL_INTERRUPT_UNMASK, HAL_INTERRUPT_CONFIGURE and HAL_INTERRUPT_SET_LEVEL. The platform HAL can override these definitions if platform-specific macros are more appropriate. It is up to the platform HAL to define the interrupt vector numbers. The SET_LEVEL macro is a no-op so there is no support for prioritizing interrupts.

Clock and Profiling Support

The STPC Atlas variant HAL provides default definitions of the clock-related macros HAL_CLOCK_INITIALIZE, HAL_CLOCK_RESET, HAL_CLOCK_READ and HAL_CLOCK_LATENCY. The implementation uses the processor's PIT0 timer since that is the only on-chip timer which can generate interrupts. The platform HAL determines the default clock frequency, and can override any of these definitions if required. If the variant HAL clock macros should be used then the platform HAL should implement the CDL interface CYGINT_HAL_I386_STPC_ATLAS_STANDARD_CLOCK.

When the variant HAL's clock macros are enabled the package will also provide profiling timer support.

Idle Thread Processing

The variant HAL defines a macro HAL_IDLE_THREAD_ACTION which gets invoked automatically by the kernel's idle thread. This macro executes a hlt instruction, suspending the CPU until the next interrupt and thus reducing power consumption. The platform HAL can override this definition if necessary.

Other Functionality

The variant HAL defines a HAL_PLATFORM_RESET macro which resets the processor using functionality provided by the STPC Atlas' keyboard/mouse controller. It also provides a HAL_DELAY_US macro which works in terms of a simple busy loop, so it does not depend on PIT0 having been started.

The implementation of other parts of the HAL specification is unaffected, and no additional functionality is provided.

Part LXXXI. M68000 / ColdFire Architecture

Table of Contents

Chapter 317. M68000 / ColdFire Architectural Support

Name

Overview — eCos Support for the M68K Family of Processors

Description

The original Motorola 68000 processor was released in 1979, and featured the following:

- Eight general purpose 32-bit data registers, %D0 to %D7. Seven 32-bit address registers %A0 to %A6, with %A7 dedicated as the stack pointer. A 16-bit status register.
- A linear address space, limited to 24-bits because the chip package only had 24 address pins. Hence the processor could address 16 megabytes of memory.
- No separate address space for I/O operations. Instead devices are accessed just like memory via the main address and data buses.
- 16-bit external data bus, even though the data registers were 32 bits wide.
- A CISC variable-length instruction set with no less than 14 different addressing modes (although of course the terms RISC and CISC were not yet in common use).
- Separate supervisor and user modes. The processor actually has two distinct stack pointer registers %A7, and the mode determines which one gets used.
- An interrupt subsystem with support for vectored and prioritized interrupts.

The 68000 processor was used in several microcomputers of its time, including the original Apple Macintosh, the Commodore Amiga, and the Atari ST. Over the years numerous variants have been developed. The core instruction set has remained essentially unchanged. Some of the variants have additional instructions. The development of MMUs led to changes in exception handling. In more recent variants, notably the Freescale ColdFire family, some infrequently used instructions and addressing modes have been removed.

- The 68008 reduced the widths of the external data and address buses to 8 bits and 20 bits respectively, giving the processor slow access to only one megabyte.
- The 68010 (1982) added virtual memory support.
- In the 68020 (1984) both the address and data buses were made 32-bits wide. A 256-byte instruction cache was added, as were some new instructions and addressing modes.
- The 68030 (1987) included an on-chip mmu and a 256-byte data cache.
- The 68040 (1991) added hardware floating point (previous processors relied on an external coprocessor or on software emulation). It also had larger caches and an improved mmu.
- The 68060 (1994) involved an internally very different superscalar implementation of the architecture, but few changes at the interface level. It also contained support for power management.
- There have been numerous 683xx variants for embedded use, with on-chip peripherals like UARTs and timers. The cpu core of these variants is also known as cpu32.
- The MCFxxxx ColdFire series (1995) resembles a stripped-down 68060, with some instructions and addressing modes removed to allow for a much smaller and more efficient implementation. Various hardware units such as the and FPU and MMU have become optional.

eCos only provides support for some of these variants, although it should be possible to add support for additional variants with few or no changes to the architectural HAL package.

The architectural HAL provides support for those features which are common to all members of the 68000 and ColdFire families, and for certain features which are present on some but not all members. A typical eCos configuration will also contain: a variant HAL package with support code for a family of processors, for example MCFxxxx; possibly a processor HAL package with support for one specific processor, for example the MCF5272; and a platform HAL which contains the code needed for a specific hardware platform such as the m5272c3.

Name

Options — Configuring the M68K Architectural Package

Loading and Unloading the Package

The M68K architectural HAL package `CYGPKG_HAL_M68K` should be loaded automatically when eCos is configured for M68K-based target hardware. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware. `CYGPKG_HAL_M68K` serves primarily as a container for lower-level HALs and has only a small number of configuration options.

Stacks

By default the architectural HAL provides a single block of memory to act as both the startup stack and the interrupt stack. The variant, processor or platform HAL may override this. For example if there are several banks of RAM with different performance characteristics it may be desirable to place the interrupt stack in fast RAM rather than in ordinary RAM.

The assembler startup code sets the stack pointer to the startup stack before switching to C code. This stack used for all HAL initialization, running any C++ static constructors defined either by eCos or by the application, and the `cyg_start` entry point. In configurations containing the eCos kernel `cyg_start` will enable interrupts, activate the scheduler and threads will then run on their own stacks. In non-kernel single-threaded applications the whole system continues to run on the startup stack.

When an interrupt occurs the default behaviour is to switch to a separate interrupt stack. This behaviour is controlled by the common HAL configuration option `CYGIMP_HAL_COMMON_INTERRUPTS_USE_INTERRUPT_STACK`. It reduces the stack requirements of all threads in the system, at the cost of some extra instructions during interrupt handling. In kernel configurations the startup stack is no longer used once the scheduler starts running so its memory can be reused for the interrupt stack. To handle the possibility of nested interrupts the interrupt handling code will detect if it is already on the interrupt stack, so in non-kernel configurations it is also safe to use the same area of memory for both startup and interrupt stacks. This leads to the following scenarios:

1. If interrupt stacks are enabled via `CYGIMP_HAL_COMMON_INTERRUPTS_USE_INTERRUPT_STACK` and the interrupt stack is not provided by the variant, processor or platform HAL then a single block of memory will be used for both startup and interrupt stacks. The size of this block is determined by the common HAL configuration option `CYGNUM_HAL_COMMON_INTERRUPTS_STACK_SIZE`, with a default value `CYGNUM_HAL_DEFAULT_INTERRUPT_STACK_SIZE` provided by the M68K architectural HAL.

2. If the use of an interrupt stack is disabled then the M68K architectural HAL will provide just the startup stack, unless this is done by the variant, processor or platform HAL. The size of the startup stack is controlled by `CYGNUM_HAL_M68K_STARTUP_STACK_SIZE`.

3. Otherwise the interrupt and/or startup stacks are provided by other packages and it is up to those packages to provide configuration options for setting the sizes.

Floating Point Support

There are many variants of the basic M68K architecture. Some of these have hardware floating point support. Originally this came in the form of a separate 68881 coprocessor, but with modern variants it will be part of the main processor chip. If the processor does not have hardware floating point then software emulation will be used instead.

If the processor on the target hardware has a floating point unit then the variant or processor HAL will implement the CDL interface `CYGINT_HAL_M68K_VARIANT_FPU`. This allows the architectural HAL and other packages to do the right thing on different hardware.

Saving and restoring hardware floating point context increases interrupt and dispatch latency, code size, and data size. If the application does not actually use floating point then these overheads are unnecessary, and can be suppressed by disabling the configuration option CYGIMP_HAL_M68K_FPU_SAVE. Some applications do use floating point but only in one thread. In that scenario it is also unnecessary to save the floating point context during interrupts and context switches, so the configuration option can be disabled.

The exact behaviour of the hardware floating point unit is determined by the floating point control register %fpcr. By default this is initialized to 0, giving IEE754 standard behaviour, but another initial value can be specified using the configuration option CYGNUM_HAL_M68K_FPU_CR_DEFAULT. For details of the various bits in this control register see appropriate hardware documentation. eCos assumes that the control register does not change on a per-thread basis and hence the register is not saved or restored during interrupt handling or a context switch.

Warning

At the time of writing eCos has not run on an M68K processor with hardware floating point so the support for this is untested.

Other Options

There are configuration options to change the compiler flags used for building this packages. The M68K architectural HAL package does not define any other configuration options that can be manipulated by the user. It does define a number of interfaces such as CYGINT_HAL_M68K_USE_STANDARD_PLATFORM_STUB_SUPPORT which can be used by lower levels of the M68K HAL hierarchy to enable certain functionality within the architectural package. Usually these are of no interest to application developers.

Name

HAL Port — Implementation Details

Description

This documentation explains how the eCos HAL specification has been mapped onto M68K hardware, and should be read in conjunction with that specification. It also describes how variant, processor and platform HALs can modify the default behaviour.

eCos support for any given target will involve either three or four HAL packages: the architectural HAL, the platform HAL, the variant HAL, and optionally a processor HAL. This package, the architectural HAL, provides code and definitions that are applicable to all M68K processors. The platform HAL provides support for one specific board, for example an M5272C3 evaluation board, or possibly for a number of almost-identical boards. The processor HAL, if present, serves mainly to provide details of on-chip peripherals including the interrupt controller. The variant HAL provides functionality that is common to a group of processors, for example all MCFxxxx processors have very similar UARTs and hence can share HAL diagnostic code. There is no fixed specification of what should go into the variant HAL versus the processor HAL. For simplicity the description below only refers to variant HALs, but the work may actually happen in a processor HAL instead.

As a design goal lower-level HALs can always override functionality that is normally provided higher up. For example the architectural HAL will provide the required eCos `HAL_LSBIT_INDEX` and `HAL_MSBIT_INDEX` macros, but these can be provided lower down instead. Many but not all ColdFire processors have the `ff1` and `bitrev` instructions which allow for a more efficient implementation than the default architectural ones. In some areas such as handling context switching the architectural HAL will usually provide the basic functionality but it may be extended by lower HALs, for example to add support for the multiply-accumulate units present in certain ColdFire processors.

The architectural HAL provides header files `cyg/hal/hal_arch.h`, `cyg/hal/hal_intr.h`, `cyg/hal/hal_cache.h`, `cyg/hal/hal_io.h` and `cyg/hal/arch.inc`. These automatically include an equivalent header file from the variant HAL, for example `cyg/hal/var_arch.h`. The variant HAL header will in turn include processor and platform-specific headers. This means that application developers and other packages can simply include the architectural HAL headers without needing to know about variants or platforms. It also allows the variant and platform HALs to override architectural settings.

The port assumes that eCos and application code always runs in supervisor mode, with full access to all hardware and special registers.

Data Types

For eCos purposes all M68K processors are big-endian and 32-bit, so the default data types in `cyg/infra/cyg_type.h` are used. Some variants have external bus widths less than 32-bit, but this does not affect the architectural HAL.

When porting to another variant it is possible to override some or all of the type definitions. The variant HAL needs to implement the CDL interface `CYGINT_HAL_M68K_VARIANT_TYPES` and provide a header file `cyg/hal/var_basetype.h`.

Startup and Exception Vectors

The conventional bootstrap mechanism involves a table of exception vectors at the base of memory. The first two words of this table give the initial program counter and stack pointer. In a typical embedded system the hardware is arranged such that non-volatile flash memory is found at location 0x0 so it is the start of flash that contains the exception vectors and the boot code. The table of exception vectors is used subsequently for interrupt handling and for hardware exceptions such as attempts to execute an illegal instruction. There are a number of common scenarios:

1. On systems with very limited memory flash may remain mapped at location 0 and the table of exception vectors remains mapped there as well. The M68K architecture defines the table to have 256 entries and hence it occupies 1K of memory, but in reality many of the entries are unused so part of the table may get used for code instead. Since the whole exception vector table is in read-only memory parts of the eCos interrupt and exception handling mechanisms have to be statically initialized and macros like `HAL_VSR_SET` are not available.

2. As a minor variation of the previous case, flash remains at location 0 but the table of exception vectors gets remapped elsewhere in the address space, usually RAM. This allows `HAL_VSR_SET` to operate normally but at the cost of increased memory usage. The exception vector table in flash only contains two entries, for the initial program counter and stack pointer. The exception vector table in RAM typically gets initialized at run-time.

3. On systems with more memory it is conventional to rearrange the address map during bootstrap. The flash gets relocated, typically to near the end of the address space, and RAM gets placed at location 0 instead. The exception vector table stays at location 0 but is now in RAM and gets initialized at run-time. The bootstrap exception vector table in flash again only needs two entries. A variation places the RAM elsewhere in the address space and moves the exception vector table there, leaving location 0 unused. This provides some protection against null pointer accesses in errant code.

   As a further complication, larger systems typically support different startup types. The application can be linked against a ROM startup configuration and placed directly in flash, as before. Alternatively there may be a ROM monitor such as RedBoot in the flash, taking care of initial bootstrap. The user's application is linked against a RAM startup configuration, loaded into RAM via the ROM monitor, and debugged using functionality provided by the ROM monitor. Yet another possibility involves a RAM startup application but it gets loaded and run via a hardware debug technology such as BDM, and the ROM monitor is either missing or not used.

The exact hardware details, the various startup types, the steps needed for low-level hardware initialization, and so on are not known to the architectural HAL. Hence although the architectural HAL does provide the basic framework for startup, much of the work is done via macros provided by lower-level HAL packages and those macros are likely to depend on various configuration options. Rather than try to enumerate all the various combinations here it is better to look at the actual code in `vectors.S` and in appropriate variant, processor or platform HALs. `vectors.S` is responsible for any low-level initialization that needs to happen. This includes setting up a standard C environment with the stack pointer set to the startup stack in working RAM, making sure all statically initialized global variables have the correct values, and that all uninitialized global variables are zeroed. Once the C environment has been set up the code jumps to `hal_m68k_c_startup` in file `hal_m68k.c` which completes the initialization and jumps to the application entry point.

Interrupt Handling

The M68K architecture reserves a 1K area of memory for 256 exception vectors. These are used for internal and external interrupts, exceptions, software traps, and special operations such as reset handling. Some of the vectors have well-defined uses. However when it comes to interrupt handling the details will depend on the processor variant and on the platform, and the appropriate package documentation should be consulted for full details. Most platforms will not use the full set of 256 vectors, instead re-using some of this memory for other purposes.

By default the exception vectors are located at location 0, but some variants allow the vectors to be located elsewhere. This is managed by an M68K-specific macro `CYG_HAL_VSR_TABLE`. The default value is 0, but a variant HAL can provide an alternative value.

The standard eCos macros `HAL_VSR_GET` and `HAL_VSR_SET` just manipulate one of the 256 entries in the table of exception vectors. Hence it is usually possible to replace the default handlers for exceptions and traps in addition to interrupt handlers. `hal_intr.h` provides `#define`'s for the more common exception vectors, and additional ones can be provided by the platform or variant. It is the responsibility of the platform or variant HAL to initialize the table, and to provide the `HAL_VSR_SET_TO_ECOS_HANDLER` macro since that requires knowledge of the default table entries.

It should be noted that in some configurations the table of exception vectors may reside in read-only memory so entries cannot be changed. If so then the `HAL_VSR_SET` and `HAL_VSR_SET_TO_ECOS_HANDLER` macros will not be defined. Portable code may need to consider this possibility and test for the existence of these macros before using them.

The architectural HAL provides an entry point `hal_m68k_interrupt_vsr` in the file `hal_arch.S`. When an interrupt occurs the original 68000 pushed the program counter and the status register on to the stack, and then called the VSR via the exception table. On newer variants some additional information is pushed, including details of the interrupt source. `hal_m68k_interrupt_vsr` assumes the latter and can be used directly as the VSR on these newer variants. On older variants a small trampoline is needed which pushes the additional information and then jumps to the generic VSR. Interpreting the additional information is handled via an assembler macro hal_context_extract_isr_vector_shl2 which should be defined by the variant, matching the behaviour of the hardware or the trampoline.

At the architecture level there is no fixed mapping between VSR and ISR vectors. Instead that is left to the variant or platform HAL. The architectural HAL does provide default implementations of HAL_INTERRUPT_ATTACH, HAL_INTERRUPT_DETACH and HAL_INTERRUPT_IN_USE since these just involve updating a static table.

By default the interrupt state control macros HAL_DISABLE_INTERRUPTS, HAL_RESTORE_INTERRUPTS, HAL_ENABLE_INTERRUPTS and HAL_QUERY_INTERRUPTS are implemented by the architectural HAL, and simply involve updating the status register. Disabling interrupts involves setting the three IPL bits to 0x07. Enabling interrupts involves setting those bits to a smaller value, CYGNUM_HAL_INTERRUPT_DEFAULT_IPL_LEVEL, which defaults to 0.

HAL_DISABLE_INTERRUPTS has no effect on non-maskable interrupts. This causes a problem because parts of the system assume that all normal interrupt sources are affected by this macro. If the target hardware can raise non-maskable interrupts then it is the responsibility of application code to install a suitable VSR and handle non-maskable interrupts entirely within the application, bypassing the usual eCos ISR and DSR mechanisms.

The architectural HAL does not provide any support for the interrupt controller management macros like HAL_INTERRUPT_MASK. These can only be implemented on a per-variant, per-processor or per-platform basis.

Exception Handling

Synchronous exception handling is done in much the same way as interrupt handling. The architectural HAL provides a generic entry point hal_m68k_exception_vsr. On some variants this can be used directly as the exception VSR, on others it will be called via a small trampoline.

The details of exception handling vary widely from one variant to the next. Some variants push a great deal of additional information on to the stack for certain exceptions, but not all. The pushed program counter may correspond to the specific instruction that caused the exception, or the next instruction, or there may be only a loose correlation because of buffered writes. The architectural HAL makes no attempt to cope with all these differences, although some variants may provide more advanced support. Otherwise if an exception needs to be handled in a very specific way then it is up to the application to install a suitable VSR and handle the exception directly.

Stacks and Stack Sizes

cyg/hal/hal_arch.h defines values for minimal and recommended thread stack sizes, CYGNUM_HAL_STACK_SIZE_MINIMUM and CYGNUM_HAL_STACK_SIZE_TYPICAL. These values are specific to the current configuration, and are affected mainly by options related to interrupt handling.

By default eCos uses a separate interrupt stack, although this can be disabled through the configuration option CYGIMP_HAL_COMMON_INTERRUPTS_USE_INTERRUPT_STACK. When an interrupt or exception occurs eCos will save the context on the current stack and then switch to the interrupt stack before calling the appropriate ISR interrupt handler. This means that thread stacks can be significantly smaller because there is no need to worry about interrupt handling overheads, just the thread context. However switching the stack does require some extra work and hence increases the interrupt latency. Disabling the interrupt stack removes this processing overhead but requires larger stack sizes. It depends on the application whether or not this is a sensible trade off.

By default eCos does not allow nested interrupts, but this can be controlled via the configuration option CYGSEM_HAL_COMMON_INTERRUPTS_ALLOW_NESTING. Supporting nested interrupts requires larger thread stacks, especially if the separate interrupt stack is also disabled.

Although the M68K has enough registers for typical operation, the calling conventions are memory-oriented. In particular all arguments are pushed on the stack rather than held in registers, and the return address is also pushed rather than ending up in a link register. To allow for this the recommended minimum stack sizes are a little bit larger than for some other architectures. Variant HALs cannot directly affect these stack sizes. However the sizes do depend in part on the size of a thread context, so if for example the processor supports hardware floating point and support for that is enabled then the stack sizes will increase.

Usually the M68K architectural HAL will provide a single block of memory which acts as both the startup and interrupt stack, and there are configuration options to control the size of this block. Alternatively a variant, processor or platform HAL may define either or both of `_HAL_M68K_STARTUP_STACK_` and `_HAL_M68K_INTERRUPT_STACK_BASE_` if for some reason the stacks should not be placed in ordinary RAM.

Thread Contexts and Setjmp/Longjmp

A typical thread context consists of the following:

1. The integer context. This consists of the data registers `%d0` to `%d7` and the address registers `%a0` to `%a6`, The stack pointer register `%a7` does not have to be saved explicitly since it is implicit in the pointer to the saved context.

   The caller-save registers are `%d0`, `%d1`, `%a0`, `%a1`, `%a7` and the status register. The remaining registers are callee-save. Function arguments are always passed on the stack. The result is held in `%d0`.

2. Floating point context, consisting of eight 64-bit floating point registers `%fp0` to `%fp7` and two support registers `%fpsr` and `%fpiar`. Support for this is only relevant if the processor variant has a hardware floating point unit, and even then saving floating point context is optional and can be disabled using a configuration option `CYGIMP_HAL_M68K_FPU_SAVE`. The control register `%fpcr` is not saved as part of the context. It is assumed that a single `%fpcr` value, usually `0`, will be used throughout the application.

   The architectural HAL provides support for the hardware floating point unit. The variant or processor HAL should implement the CDL interface `CYGINT_HAL_M68K_VARIANT_FPU` if this hardware unit is actually present.

3. Some M68K variants have additional hardware units, for example the multiply-accumulate units in certain ColdFire processors. The architectural HAL allows the context to be extended through various macros such as `HAL_CONTEXT_OTHER`.

4. The status register `%sr` and the program counter. These are special because when an interrupt occurs the hardware automatically pushes these onto the stack, but exactly what gets pushed depends on the variant.

`setjmp` and `longjmp` only deal with the integer and fpu contexts. It is assumed that any special hardware units will only be used by application code, not by the compiler. Hence it is the responsibility of application code to define and implement appropriate setjmp semantics for these units. The variant HAL package can override the default implementations if necessary.

When porting to a new M68K variant, if this has a hardware floating point unit then the variant HAL should implement the CDL interface `CYGINT_HAL_M68K_VARIANT_FPU`, thus enabling support provided by the architectural HAL. If the variant has additional hardware units involving state that should be preserved during a context switch or when an interrupt occurs, the variant HAL should define a number of macros. The header file `cyg/hal/var_arch.h` should define `HAL_CONTEXT_OTHER`, `HAL_CONTEXT_OTHER_SIZE`, and `HAL_CONTEXT_OTHER_INIT`, either directly or via `cyg/hal/proc_arch.h`. The assembler header file `cyg/hal/var.inc` should define a number of macros such as `hal_context_other_save_caller`. For details of these macros see the architectural `hal_arch.S` file.

Variants also need to define exactly how the status register and program counter are saved onto the stack when an interrupt or exception occurs. This is handled through C macros `HAL_CONTEXT_PCSR_SIZE`, `HAL_CONTEXT_PCSR_RTE_ADJUST`, and `HAL_CONTEXT_PCSR_INIT`, and a number of assembler macros such as `hal_context_pcsr_save_sr`. Again the architectural files `cyg/hal/hal_arch.h` and `hal_arch.S` provide more details of these.

Bit Indexing

For performance reasons the `HAL_LSBIT_INDEX` and `HAL_MSBIT_INDEX` macros are implemented using assembler functions. A variant HAL can override the default definitions if, for example, the variant has special instructions to perform these operations.

Idle Thread Processing

The default `HAL_IDLE_THREAD_ACTION` implementation is a no-op. A variant HAL may override this, for example to put the processor into sleep mode. Alternative implementations should consider exactly how this macro gets used in eCos kernel code.

Clock Support

The architectural HAL cannot provide the required clock support because it does not know what timer hardware may be available on the target hardware. Instead this is left to either the variant or platform HAL, depending on whether the processor has a suitable on-chip timer or whether an off-chip timer has to be used.

HAL I/O

The M68K architecture does not have a separate I/O bus. Instead all hardware is assumed to be memory-mapped. Further it is assumed that all peripherals on the memory bus are wired appropriately for a big-endian processor and that there is no need for any byte swapping. Hence the various HAL macros for performing I/O simply involve pointers to volatile memory.

The variant, processor and platform equivalents of the `cyg/hal/hal_io.h` header will typically also provide details of some or all of the peripherals, for example register offsets and the meaning of various bits in those registers.

Cache Handling

If the processor has a cache then the variant HAL should implement the CDL interface `CYGINT_HAL_M68K_VARIANT_CACHE`. This causes the architectural header `cyg/hal/hal_cache.h` to pick up appropriate definitions from `cyg/hal/var_cache.h`. The architectural header will provide null defaults for anything not defined by the variant.

Linker Scripts

The architectural HAL will generate the linker script for eCos applications. This involves the architectural file `m68k.ld` and a `.ldi` memory layout file provided lower down, typically by the platform HAL. It is the LDI file which specifies the types and amount of memory available and which places code and data in appropriate places, but most of the hard work is done via macros provided by the architectural `m68k.ld` file.

Diagnostic Support

The architectural HAL does not implement diagnostic support. Instead this is left to the variant or platform HAL, depending on whether suitable peripherals are available on-chip or off-chip.

SMP Support

The M68K port does not have SMP support.

Debug Support

The M68K architectural HAL package provides basic support only for gdb stubs. There is no support for more advanced debug features like hardware watchpoints.

The generic gdb support in the common HAL requires a platform header `<cyg/hal/plf_stub.h`. In practice there is rarely any need for the contents of this file to change between platforms so the architectural HAL can provide a suitable default. It will do so if the CDL interface `CYGINT_HAL_M68K_USE_STANDARD_PLATFORM_STUB_SUPPORT` is implemented.

HAL_DELAY_US Macro

The architectural HAL provides a default implementation of the standard `HAL_DELAY_US` macro using a simply busy loop. To use this support a lower-level HAL should define `_HAL_M68K_DELAY_US_LOOPS_`, typically a small number of about 20 but it will need to be calibrated during the porting process. If the processor has a cache then the lower-level HAL may also define `_HAL_M68K_DELAY_US_LOOPS_UNCACHED_` for the case when a delay loop is triggered while the cache is disabled.

Profiling Support

The M68K architectural HAL implements the `mcount` function, allowing profiling tools like gprof to determine the application's call graph. It does not implement the profiling timer. Instead that functionality needs to be provided by the variant or platform HAL. The implementation of `mcount` requires a dedicated frame pointer register so code should be compiled without the `-fomit-frame-pointer` flag.

Other Functionality

The M68K architectural HAL only implements the functionality provided by the eCos HAL specification and does not export any extra functionality.

Chapter 318. Freescale MCFxxxx Variant Support

Name

CYGPKG_HAL_M68K_MCFxxxx — eCos Support for Freescale MCFxxxx Processors

Description

The Freescale ColdFire family is a range of processors including the MCF5208 and the MCF5282. From a programmer's perspective these processors all share basically the same processor core, albeit with minor differences in the instruction set. They differ in areas like performance, on-chip peripherals and caches. Even when it comes to peripherals there is a lot of commonality. For example many but not all Coldfire processors use the same basic interrupt controller(s) as the MCF5282. Similarly the on-chip UARTs tend to use the same basic design although there are variations in the number of UARTs, the fifo sizes, and in certain details.

The MCFxxxx variant HAL package CYGPKG_HAL_M68K_MCFxxxx provides support for various features that are common to many but not all Coldfire processors. This includes HAL diagnostics via an on-chip UART and interrupt controller management for those processors which have MCF5282-compatible controllers. The variant HAL complements the M68K architectural HAL package. An eCos configuration should also include a processor-specific HAL package such as CYGPKG_HAL_M68K_MCF5272 to support the chip-specific peripherals and cache details, and a platform HAL package such as CYGPKG_HAL_M68K_M5272C3 to support board-level details like external memory chips. The processor or platform HAL can override the functionality provided by the variant HAL.

Configuration

The MCFxxxx variant HAL package should be loaded automatically when eCos is configured for appropriate target hardware. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

On most ColdFire platforms the variant HAL will provide the HAL diagnostics support via one of the UARTs. Some platforms may provide their own HAL diagnostics facility, for example output via an LCD. The variant HAL diagnostics support is active if the processor or platform implements the CYGINT_HAL_M68K_MCFxxxx_DIAGNOSTICS_USE_DEFAULT interface. It is also active only in configurations which do not rely on an underlying rom monitor such as RedBoot: if CYGSEM_HAL_USE_ROM_MONITOR is enabled then the default diagnostics channel will automatically be inherited from RedBoot. The variant HAL then provides a number of configuration options related to diagnostics:

CYGHWR_HAL_M68K_MCFxxxx_DIAGNOSTICS_PORT

> This selects the destination for HAL diagnostics. The number of UARTs available depends on the processor, and on any given board some of the UARTs may not be connected. Hence the variant HAL looks for configuration options CYGHWR_HAL_M68K_MCFxxxx_UART0, CYGHWR_HAL_M68K_MCFxxxx_UART1 and CYGHWR_HAL_M68K_MCFxxxx_UART2 to see which on-chip UARTs are actually available on the processor and target hardware, and uses this information to let the user select a UART.
>
> Instead of using a uart the diagnostics support can be set to discard all output. This is useful when other packages or application code makes unconditional use of HAL diagnostics facilities, for example to report error conditions, but the target hardware does not have a spare UART. Alternatively when debugging via a hardware debug solution such as BDM it is possible to direct the diagnostics output to a gdb hwdebug file I/O channel. By default this will also discard diagnostics output. However if the application is running inside a gdb session and the gdb **set hwdebug** command has been used then the diagnostics will be output via gdb.
>
> When a UART is in use as the HAL diagnostics channel, that UART should not be used for any other purpose. In particular application code should avoid using it for I/O via the serial driver.

CYGNUM_HAL_M68K_MCFxxxx_DIAGNOSTICS_BAUD

> When a UART is selected for HAL diagnostics this option specifies the default baud rate. The most common setting is 38400. That provides a compromise between performance and reliability, especially in electrically noisy environments such as an

industrial environment or a test farm. Some platforms may define CYGNUM_HAL_M68K_MCFxxxx_DIAGNOSTICS_DEFAULT_BAUD to handle scenarios where another default baud rate is preferable, typically for compatibility with existing software.

CYGNUM_HAL_M68K_MCFxxxx_DIAGNOSTICS_ISRPRI

Usually the HAL diagnostics channel is driven in polled mode but in some scenarios interrupts are required. For example, when debugging an application over a serial line on top of the gdb stubs provided by RedBoot, the user should be able to interrupt the application with a control-C. The application will not be polling the HAL diagnostics UART at this point so instead the eCos interrupt management code interacts with the gdb stubs to do the right thing. This configuration option selects the interrupt priority. It should be noted that on some processors with MCF5282-compatible interrupt controllers all priorities for enabled interrupts should be unique, and it is the responsibility of application developers to ensure this condition is satisfied.

The HAL Port

This section describes how the MCFxxxx variant HAL package implements parts of the eCos HAL specification. It should be read in conjunction with similar sections from the architectural and processor HAL documentation.

HAL I/O

The cyg/hal/var_io.h header provides various definitions for on-chip peripherals, where the current processor has peripherals compatible with the MCF5282's or which are available on several different coldfires. This header is automatically included by the architectural cyg/hal/hal_io.h so other packages and application code will usually only include the latter.

It is up to the processor HAL to specify exactly what var_io.h should export. For example the MCF5213's proc_io.h header contains the following:

```
# define HAL_MCFxxxx_HAS_MCF5282_INTC              1
# define HAL_MCFxxxx_INTC0_BASE                    (HAL_MCF521x_IPSBAR + 0x00000C00)
```

This enables support within the variant HAL for a single MCF5282-compatible interrupt controller, and cases var_io.h to export symbols such as:

```
#ifdef HAL_MCFxxxx_HAS_MCF5282_INTC
// Two 32-bit interrupt mask registers
# define HAL_MCFxxxx_INTCx_IMRH                    0x0008
# define HAL_MCFxxxx_INTCx_IMRL                    0x000C
…
# define HAL_MCFxxxx_INTCx_ICRxx_IL_MASK           (0x07 << 3)
# define HAL_MCFxxxx_INTCx_ICRxx_IL_SHIFT          3
```

Symbols such as HAL_MCFxxxx_INTCx_IMRH can be used to access the relevant hardware registers via HAL_READ_UINT32 and HAL_WRITE_UINT32. Symbols like HAL_MCFxxxx_INTCx_ICRxx_IL_MASK can be used to generate or decode the contents of the hardware registers.

The header file does mostly use a naming convention, but is not guaranteed to be totally consistent. There may also be discrepancies with the documentation because the manuals for the various Coldfire processors are not always consistent about their naming schemes. All I/O definitions provided by the variant HAL will start with HAL_MCFxxxx_, followed by the name of the peripheral. If a peripheral is likely to be a singleton, for example an on-chip flash unit, then the name is unadorned. If there may be several instances of the peripheral then the name will be followed by a lower case x. For example:

```
# define HAL_MCFxxxx_CFM_CR                        0x0000
…
# define HAL_MCFxxxx_UARTx_UMR                     0x00
```

Register names will be relative to some base address such as HAL_MCFxxxx_CFM_BASE or HAL_MCFxxxx_UART0_BASE, so code accessing a register would look like:

```
    HAL_READ_UINT32(HAL_MCFxxxx_CFM_BASE + HAL_MCFxxxx_CFM_PROT, reg);
    …
    HAL_WRITE_UINT8(base + HAL_MCFxxxx_UARTx_UTB, '*');
```

Usually the register names are singletons, but in some cases such as the interrupt controller priority registers there may be multiple instances of the register and the names will be suffixed appropriately. For example HAL_MCFxxxx_INTCx_ICRxx_IL_MASK indicates the field IL within one of the ICR registers within one of the interrupt controllers.

As mentioned earlier the processor HAL's proc_io.h will control which definitions are exported by var_io.h. Sometimes the processor HAL will then go on to undefine or redefine some of the symbols, to reflect incompatibilities between the processor's devices and the equivalent devices on the MCF5282. There may also be additional symbols for the devices, and there will be additional definitions for any processor-specific hardware. In particular GPIO pin handling is handled by the processor HAL, not by the variant HAL. Application developers should examine proc_io.h as well as var_io.h and the processor-specific documentation to see exactly what I/O definitions are provided. When porting to a new Coldfire processor it is best to start with an existing processor HAL and copy code as appropriate. A search for _HAS_ in var_io.h will also be informative.

Thread Contexts and Setjmp/Longjmp

All MCFxxxx processors support interrupts and exceptions in a uniform way. When an interrupt or exception occurs the hardware pushes the current program counter, the status register, and an additional 16-bit word containing information about the interrupt source, for a total of 64 bits. Hence the PCSR part of a thread context consists of two 32-bit integers, and the variant HAL provides appropriate C and assembler macros to examine and manipulate these.

Not all MCFxxxx processors have hardware floating point, so support for this is left to the processor HAL package. Some MCFxxxx processors have additional hardware units such as a multiply-accumulator, but these are not currently supported by eCos.

HAL Diagnostics

The various MCFxxxx processors usually have one or more UARTs based on very similar hardware. The variant HAL package can provide HAL diagnostic support using such a UART. There are some minor differences such as fifo sizes, and the UARTs will be accessed at different memory locations. These differences are handled by a small number of macros provided by the processor and platform HAL.

The MCFxxxx variant HAL only provides HAL diagnostic support via a UART if the processor or platform HAL does not provide an alternative implementation. That copes with situations where the on-chip UARTs are not actually accessible on the target board and an alternative communication channel must be used.

If the variant HAL should implement HAL diagnostics then the processor or platform HAL should implement the CDL interface CYGINT_HAL_M68K_MCFxxxx_DIAGNOSTICS_USE_DEFAULT. It should also define one or more of CYGHWR_HAL_M68K_MCFxxxx_UART0, CYGHWR_HAL_M68K_MCFxxxx_UART1 and CYGHWR_HAL_M68K_MCFxxxx_UART2, and ensure that any multi-purpose GPIO pins are set correctly. The variant HAL will take care of the rest.

Cache Handling

MCFxxxx processors support a number of different caching schemes. Partial support for some of is provided by the variant HAL's cyg/hal/var_cache.h, but it is up to the processor HAL to define which caching scheme should be used, as well as parameters such as the cache size.

Exceptions

All MCFxxxx processors support synchronous exceptions in a uniform way, with the hardware pushing sufficient information on to the stack to identify the nature of the exception. This means that the architectural entry point hal_m68k_exception_vsr can be used as the default VSR for all exceptions, with no need for separate trampoline functions.

The variant HAL does not provide any special support for recovering from exceptions.

Interrupts

All MCFxxxx processors supports interrupts in a uniform way. When an interrupt occurs the hardware pushes sufficient information on to the stack to identify the interrupt. Therefore the architectural entry point `hal_m68k_interrupt_vsr` can be used as the default VSR for all interrupts, with the variant just supplying a small number of macros that allow the generic code to extract details of the interrupt source. There is no need for separate trampoline functions for every interrupt source.

On processors which have MCF5282-compatible interrupt and edge port modules the variant HAL can provide the `HAL_INTERRUPT_MASK`, `HAL_INTERRUPT_UNMASK`, `HAL_INTERRUPT_SET_LEVEL`, `HAL_INTERRUPT_ACKNOWLEDGE` and `HAL_INTERRUPT_CONFIGURE` macros. There is support for processors with a single interrupt controller or with two separate interrupt controllers. Otherwise these macros are left to the processor HAL. The allocation of interrupt vectors to the various on-chip devices is also a characteristic of the processor HAL. `proc_intr.h` should be consulted for appropriate definitions, for example `CYGNUM_HAL_ISR_UART0`.

The mask and umask operations are straightforward: if the interrupt controller has the `SIMR` and `CIMR` registers those will be used; otherwise the `IRM` registers will be updated by a read-modify-write cycle. The acknowledge macro is only relevant for external interrupts coming in via the edge port module and will clear the interrupt by writing to the `EPIER` register. There is no simple way to clear interrupts generated by the on-chip peripherals, so that is the responsibility of the various device drivers or of application code. The configure macro is only relevant for external interrupts and involves manipulating the edge port module.

The `HAL_INTERRUPT_SET_LEVEL` macro is used implicitly by higher level code such as `cyg_interrupt_create`. With MCF5282-compatible interrupt controllers the priority level corresponds to the `ICRxx` register. The exact format depends on the processor. Interrupt priorities corresponding to IPL level 7 are non-maskable. Such interrupts cannot be managed safely by the usual eCos ISR and DSR mechanisms. Instead application code will have to install a custom VSR and manage the entire interrupt.

Some MCF5282-compatible interrupt controllers have a major restriction: all interrupt priorities within each controller must be unique. If two interrupts go off at the same time and have exactly the same priority then the controllers' behaviour is undefined. In a typical application some of the interrupts will be handled by eCos device drivers while others will be handled directly by application code. Since eCos cannot know which interrupts may get used, it cannot allocate unique priorities. Instead this has to be left to the application developer. eCos does provide configuration options such as `CYGNUM_KERNEL_COUNTERS_CLOCK_ISR_PRIORITY` and `CYGNUM_DEVS_SERIAL_MCFxxxx_SERIAL0_ISR_PRIORITY` to provide control over the eCos-managed interrupts, and provides default values for these which are unique.

Caution

Non-unique interrupt priorities can lead to very confusing system behaviour. For example on an MCF5282, if the PIT3 system clock (interrupt 0x3a) and ethernet RX frame (interrupt 0x1b) are accidentally given the same priority and go off at the same time, the interrupt controller may actually issue an interrupt 0x3b, the bitwise or of the two interrupt numbers. That interrupt belongs to the on-chip flash module. There may not be an installed handler for that interrupt at all, and even if there is a handler it will only manipulate the flash hardware and not clear the system clock and ethernet interrupts. Hence the system is likely to go into a spin, continually trying to service the wrong interrupt. To track down such problems during debugging it may prove useful to install a breakpoint on the `hal_arch_default_isr` function.

Clock Support

On processors with an MCF5282-compatible programmable interrupt timer module or PIT, the variant HAL can provide the `HAL_CLOCK_INITIALIZE`, `HAL_CLOCK_RESET`, `HAL_CLOCK_READ` and `HAL_CLOCK_LATENCY` macros. These macros are used by the eCos kernel to implement the system clock and may be used for other purposes in non-kernel configurations. When multiple timers are available it is up to the processor or platform HAL to select which one gets used for the system clock. It is also up to the processor or platform HAL to provide various clock-related configuration options such as `CYGNUM_HAL_RTC_PERIOD`. Those options need to take into account the processor clock speed, which is usually a characteristic of the platform and hence not known to the variant HAL.

When porting to a new Coldfire processor, the processor or platform HAL should define the symbols CYGNUM_HAL_INTERRUPT_RTC, _HAL_MCFxxxx_CLOCK_PIT_BASE_, and _HAL_MCFxxxx_CLOCK_PIT_PRE_. Existing ports can be examined for more details.

Reset

On processors with an MCF5282-compatible reset module or RST, the variant HAL can provide the HAL_PLATFORM_RESET macro. That macro is typically used by the gdb stubs support inside RedBoot to reset the hardware between debug sessions, ensuring that each session runs in as close to pristine hardware as possible. The macro uses the SOFTRST bit of the RCR register.

Bit Indexing

By default the variant HAL will provide versions of HAL_LSBIT_INDEX and HAL_MSBIT_INDEX which are more efficient than the default ones in the architectural HAL. The implementation uses the **ff1.l** and **bitrev.l** instructions. If the Coldfire processor does not support these instructions then the processor HAL should define _HAL_M68K_MCFxxxx_NO_FF1_.

Other Issues

The MCFxxxx variant HAL does not affect the implementation of data types, stack size definitions, idle thread processing, linker scripts, SMP support, system startup, or debug support.

Other Functionality

The MCFxxxx variant HAL only implements functionality defined in the eCos HAL specification and does not export any additional functions.

Chapter 319. Freescale MCF5272 Processor Support

Name

CYGPKG_HAL_M68K_MCF5272 — eCos Support for the Freescale MCF5272 Processor

Description

The MCF5272 is one member of the Freescale MCFxxxx ColdFire range of processors. It comes with a number of on-chip peripherals including 2 UARTs, ethernet, and USB slave. The processor HAL package CYGPKG_HAL_M68K_MCF5272 provides support for features that are specific to the MCF5272. It complements the M68K architectural HAL package CYGPKG_HAL_M68K and the variant HAL package CYGPKG_HAL_M68K_MCFxxxx. An eCos configuration should also include a platform HAL package, for example CYGPKG_HAL_M68K_M5272C3 to support board-level details like the external memory chips.

Configuration

The MCF5272 processor HAL package should be loaded automatically when eCos is configured for appropriate target hardware. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

The component CYGPKG_HAL_M68K_MCF5272_HARDWARE contains configuration options for the available hardware. This includes all GPIO pin settings, with defaults provided by the platform HAL. In turn the pin settings are used to determine defaults for other hardware settings, for example which of the two on-chip uarts are usable. Users can override these settings if necessary, subject to any constraints imposed by the platform HAL, but care has to be taken that the resulting configuration still matches the actual hardware.

The option CYGIMP_HAL_M68K_MCF5272_IDLE controls what happens in configurations containing the eCos kernel when the idle thread runs, i.e. when there is nothing for the processor to do until the next interrupt comes in. Usually the processor made to sleep, halting the cpu but leaving all peripherals active.

The package contains a single configuration option CYGFUN_HAL_M68K_MCF5272_PROFILE_TIMER. This controls the support for gprof-based profiling. By default it is active and enabled if the configuration contains the gprof profiling package, otherwise inactive. The relevant code uses hardware timer 2, so that timer is no longer available for application code. If the timer is required but a platform HAL provides an alternative implementation of the profiling support then this option can be disabled.

The HAL Port

This section describes how the MCF5272 processor HAL package implements parts of the eCos HAL specification. It should be read in conjunction with similar sections from the architectural and variant HAL documentation.

HAL I/O

The header file cyg/hal/proc_io.h specifies which generic MCFxxxx devices are present, and provides details of MCF5272-specific devices. This header file is automatically included by the architectural header cyg/hal/hal_io.h, so typically application code and other packages will just include the latter.

It should be noted that the Freescale documentation is occasionally confusing when it comes to numbering devices. For example the four on-chip timers are numbered TMR0 to TMR3, but in the interrupt controller the corresponding interrupts are numbered TMR1 to TMR4. The eCos port consistently starts numbering at 0, so these interrupts have been renamed TMR0 to TMR3.

Interrupt Handling

The header file cyg/hal/proc_intr.h provides VSR and ISR vector numbers for all interrupt sources. The VSR vector number, for example CYGNUM_HAL_VECTOR_TMR0, should be used for calls like cyg_interrupt_get_vsr. It corresponds directly to the M68K exception number. The ISR vector number, for example CYGNUM_HAL_ISR_TMR0, should be used for calls like `cyg_interrupt_create`. This header file is automatically included by the architectural header `cyg/hal/hal_intr.h`, and other packages and application code will normally just include the latter.

The eCos HAL macros `HAL_INTERRUPT_MASK`, `HAL_INTERRUPT_UNMASK`, `HAL_INTERRUPT_SET_LEVEL`, `HAL_INTERRUPT_ACKNOWLEDGE`, and `HAL_INTERRUPT_CONFIGURE` are implemented by the processor HAL. The mask and unmask operations are straightforward, simply manipulating the on-chip interrupt controller. The acknowledge and configure macros are only relevant for external interrupts: internal interrupts generated by on-chip devices do not need to be acknowledged. The set-level operation, used implicitly by higher level code such as `cyg_interrupt_create`, is mapped on to M68K IPL levels so interrupts can be given a priority between 1 and 7. Priority 7 corresponds to non-maskable interrupts and must be used with care: such interrupts cannot be managed safely by the usual eCos ISR and DSR mechanisms; instead application code will have to install a custom VSR and manage the entire interrupt.

Clock Support

The processor HAL provides support for the eCos system clock. This always uses hardware timer 3, which should not be used directly by application code. If gprof-based profiling is in use then that will use hardware timer 2. Timers 0 and 1 are never used by eCos so application code is free to manipulate these as required.

Some of the configuration options related to the system clock, for example `CYGNUM_HAL_RTC_PERIOD`, are actually contained in the platform HAL rather than the processor HAL. These options need to take into account the processor clock speed, a characteristic of the platform rather than the processor.

Cache Handling

The MCF5272 has a small instruction cache of 1024 bytes. This is fully supported by the processor HAL. There is no data cache.

Idle Thread Support

The configuration option `CYGIMP_HAL_M68K_MCF5272_IDLE` controls what happens when the kernel idle thread runs. The default behaviour is to put the processor to sleep until the next interrupt.

Profiling Support

The MCF5272 processor HAL provides a profiling timer for use with the gprof profiling package. This uses hardware timer 2, so application code should not manipulate this timer if profiling is enabled. The M68K architectural HAL implements the `mcount` function so profiling is fully supported on all MCF5272-based platforms.

Other Issues

The MCF5272 processor HAL does not affect the implementation of data types, stack size definitions, linker scripts, SMP support, system startup, or debug support. The architectural HAL's bit index instructions are used rather than the MCFxxxx variant HAL's versions since the MCF5272 does not implement the ff1 and bitrev instructions.

Other Functionality

The MCF5272 processor HAL only implements functionality defined in the eCos HAL specification and does not export any additional functions.

Chapter 320. Freescale M5272C3 Board Support

Name

eCos Support for the Freescale M5272C3 Board — Overview

Description

The Freescale M5272C3 board has an MCF5272 ColdFire processor, 4MB of external SDRAM, 2MB of external flash memory, and connectors plus required support chips for all the on-chip peripherals. By default the board comes with its own dBUG ROM monitor, located in the bottom half of the flash.

For typical eCos development a RedBoot image is programmed into the top half of the flash memory, and the board is made to boot this image rather than the existing dBUG monitor. RedBoot provides gdb stub functionality so it is then possible to download and debug eCos applications via the gdb debugger. This can happen over either a serial line or over ethernet.

Supported Hardware

In a typical setup the bottom half of the flash memory is reserved for the dBUG ROM monitor and is not accessible to eCos. That leaves four flash blocks of 256K each. Of these one is used for the RedBoot image and another is used for managing the flash and holding RedBoot fconfig values. The remaining two blocks at 0xFFF40000 and 0xFFF80000 can be used by application code.

By default eCos will only support the four megabytes of external SDRAM present on the initial versions of the board, accessible at location 0x00000000. Later versions come with 16MB. If all 16MB of memory are required then the ACR0 register needs to be changed. The default value is controlled by the configuration option `CYGNUM_HAL_M68K_M5272C3_ACR0`, but this option is only used during ROM startup so in a typical setup it would be necessary to rebuild and update RedBoot. Alternatively the register can be updated by application code, preferably using a high priority static constructor to ensure that the extra memory is visible before any code tries to use that memory. It will also be necessary to change the memory layout so that the linker knows about the additional memory.

By default the 4K of internal SRAM is mapped to location 0x20000000 using the RAMBAR register. This is not used by eCos or by RedBoot so can be used by application code. The M68K architectural HAL has an `iram1.c` testcase to illustrate the linker script support for this. The internal 16K of ROM is left disabled by default because its contents are of no use to most applications. The on-chip peripherals are mapped at 0x10000000 via the MBAR register.

There is a serial driver `CYGPKG_DEVS_SERIAL_MCFxxxx` which supports both on-chip UARTs. One of the UARTs, usually uart0, can be used by RedBoot for communication with the host. If this UART is needed by the application, either directly or via the serial driver, then it cannot also be used for RedBoot communication. Another communication channel such as ethernet should be used instead. The serial driver package is loaded automatically when configuring for the M5272C3 target.

There is an ethernet driver `CYGPKG_DEVS_ETH_MCFxxxx` for the on-chip ethernet device. This driver is also loaded automatically when configuring for the M5272C3 target. The M5272C3 board does not have a unique MAC address, so a suitable address has to be programmed into flash via RedBoot's **fconfig** command.

eCos manages the on-chip interrupt controller. Timer 3 is used to implement the eCos system clock, but timers 0, 1 and 2 are unused and left for the application. The GPIO pins are manipulated only as needed to get the UARTs and ethernet working. eCos will reset the remaining on-chip peripherals (DMA, USB, PLCI, QSPI and PWM) during system startup or soft reset but will not otherwise manipulate them.

Tools

The M5272C3 port is intended to work with GNU tools configured for an m68k-elf target. The original port was done using m68k-elf-gcc version 3.2.1, m68k-elf-gdb version 5.3, and binutils version 2.13.1.

By default eCos is built using the compiler flag `-fomit-frame-pointer`. Omitting the frame pointer eliminates some work on every function call and makes another register available, so the code should be smaller and faster. However without a frame pointer m68k-elf-gdb is not always able to identify stack frames, so it may be unable to provide accurate backtrace information. Removing this compiler flag from the configuration option `CYGBLD_GLOBAL_CFLAGS` avoids such debug problems.

Name

Setup — Preparing the M5272C3 board for eCos Development

Overview

In a typical development environment the M5272C3 board boots from flash into the RedBoot ROM monitor. eCos applications are configured for a RAM startup, and then downloaded and run on the board via the debugger m68k-elf-gdb. Preparing the board therefore involves programming a suitable RedBoot image into flash memory.

The following RedBoot configurations are supported:

Configuration	Description	Use	File
ROM	RedBoot running from the board's flash	`redboot_ROM.ecm`	`redboot_rom.bin`
dBUG	Used for initial setup	`redboot_DBUG.ecm`	`redboot_dbug.srec`
RAM	Used for upgrading ROM version	`redboot_RAM.ecm`	`redboot_ram.bin`
ROMFFE	RedBoot running from the board's flash at 0xFFE00000	`redboot_ROMFFE.ecm`	`redboot_romffe.bin`

For serial communications all versions run with 8 bits, no parity, and 1 stop bit. The dBUG version runs at 19200 baud. The ROM and RAM versions run at 38400 baud. These baud rates can be changed via the configuration option `CYGNUM_HAL_M68K_MCFxxxx_DIAGNOSTICS_BAUD` and rebuilding RedBoot. By default RedBoot will use the board's terminal port, corresponding to uart0, but this can also be changed via the configuration option `CYGHWR_HAL_M68K_MCFxxxx_DIAGNOSTICS_PORT`. On an M5272C3 platform RedBoot also supports ethernet communication and flash management.

Initial Installation

This process assumes that the board still has its original dBUG ROM monitor and does not require any special debug hardware. It leaves the existing ROM monitor in place, allowing the setup process to be repeated just in case that should ever prove necessary.

Programming the RedBoot rom monitor into flash memory requires an application that can manage flash blocks. RedBoot itself has this capability. Rather than have a separate application that is used only for flash management during the initial installation, a special RAM-resident version of RedBoot is loaded into memory and run. This version can then be used to load the normal flash-resident version of RedBoot and program it into the flash.

The first step is to connect an RS232 cable between the M5272C3 terminal port and the host PC. A suitable cable is supplied with the board. Next start a terminal emulation application such as HyperTerminal or minicom on the host PC and set the serial communication parameters to 19200 baud, 8 data bits, no parity, 1 stop bit (8N1) and no flow control (handshaking). Make sure that the jumper next to the flash chip is set for bootstrap from the bottom of flash, location 0xFFE00000. The details of this jumper depend on the revision of the board, so the supplied board documentation should be consulted for more details. Apply power to the board and you should see a `dBUG>` prompt.

Once dBUG is up and running the RAM-resident version of RedBoot can be downloaded:

```
dBUG> dl
Escape to local host and send S-records now...
```

The required S-records file is `redboot_dbug.srec`, which is normally supplied with the eCos release in the `loaders` directory. If it needs to be rebuilt then instructions for this are supplied below. The file should be sent to the target as raw text using the terminal emulator:

```
S-record download successful!
dBUG>
```

It is now possible to run the RAM-resident version of RedBoot:

```
dBUG> go 0x20000
+FLASH configuration checksum error or invalid key
Ethernet eth0: MAC address 00:00:00:00:00:03
Can't get BOOTP info for device!

RedBoot(tm) bootstrap and debug environment [DBUG]
Non-certified release, version v2_0_1 - built 09:55:34, Jun 24 2003

Platform: M5272C3 (Freescale MCF5272)
Copyright (C) 2000, 2001, 2002, Free Software Foundation, Inc.

RAM: 0x00000000-0x00400000, 0x0003f478-0x003bd000 available
FLASH: 0xffe00000 - 0x00000000, 8 blocks of 0x00040000 bytes each.
RedBoot>
```

At this stage the RedBoot flash management initialization has not yet happened so the warning about the configuration checksum error is expected. To perform this initialization use the **fis init -f** command:

```
RedBoot> fis init -f
About to initialize [format] FLASH image system - continue (y/n)? y
*** Initialize FLASH Image System
... Erase from 0xfff40000-0xfffc0000: ..
... Erase from 0x00000000-0x00000000:
... Erase from 0xfffc0000-0xffffffff: .
... Program from 0x003bf000-0x003ff000 at 0xfffc0000: .
RedBoot>
```

The flash chip on the M5272C3 board is slow at erasing flash blocks so this operation can take some time. At the end the block of flash at location 0xFFFC0000 holds information about the various flash blocks, allowing other flash management operations to be performed. The next step is to set up RedBoot's non-volatile configuration values:

```
RedBoot> fconfig -i
Initialize non-volatile configuration - continue (y/n)? y
Run script at boot: false
Use BOOTP for network configuration: true
DNS server IP address:
GDB connection port: 9000
Force console for special debug messages: false
Network hardware address [MAC]: 0x00:0x00:0x00:0x00:0x00:0x03
Network debug at boot time: false
Update RedBoot non-volatile configuration - continue (y/n)? y
... Erase from 0xfffc0000-0xffffffff: .
... Program from 0x003bf000-0x003ff000 at 0xfffc0000: .
RedBoot>
```

For most of these configuration variables the default value is correct. If there is no suitable BOOTP service running on the local network then BOOTP should be disabled, and instead RedBoot will prompt for a fixed IP address, netmask, and addresses for the local gateway and DNS server. The other exception is the network hardware address, also known as MAC address. All boards should be given a unique MAC address, not the one in the above example. If there are two boards on the same network trying to use the same MAC address then the resulting behaviour is undefined.

It is now possible to load the flash-resident version of RedBoot. Because of the way that flash chips work it is better to first load it into RAM and then program it into flash.

```
RedBoot> load -r -m ymodem -b %{freememlo}
```

The file `redboot_rom.bin` should now be uploaded using the terminal emulator. The file is a raw binary and should be transferred using the Y-modem protocol.

```
Raw file loaded 0x0003f800-0x000545a3, assumed entry at 0x0003f800
xyzModem - CRC mode, 2(SOH)/84(STX)/0(CAN) packets, 5 retries
RedBoot>
```

Once RedBoot has been loaded into RAM it can be programmed into flash:

```
RedBoot> fis create RedBoot -b %{freememlo}
An image named 'RedBoot' exists - continue (y/n)? y
... Erase from 0xfff00000-0xfff40000: .
... Program from 0x0003f800-0x0007f800 at 0xfff00000: .
... Erase from 0xfffc0000-0xffffffff: .
... Program from 0x003bf000-0x003ff000 at 0xfffc0000: .
RedBoot>
```

The flash-resident version of RedBoot has now programmed at location 0xFFF00000, and the flash info block at 0xFFFC0000 has been updated. The initial setup is now complete. Power off the board and set the flash jumper to boot from location 0xFFF00000 instead of 0xFFE00000. Also set the terminal emulator to run at 38400 baud (the usual baud rate for RedBoot), and power up the board again.

```
+Ethernet eth0: MAC address 00:00:00:00:00:03
Can't get BOOTP info for device!

RedBoot(tm) bootstrap and debug environment [ROM]
Non-certified release, version v2_0_1 - built 09:57:50, Jun 24 2003

Platform: M5272C3 (Freescale MCF5272)
Copyright (C) 2000, 2001, 2002, Free Software Foundation, Inc.

RAM: 0x00000000-0x00400000, 0x0000b400-0x003bd000 available
FLASH: 0xffe00000 - 0x00000000, 8 blocks of 0x00040000 bytes each.
RedBoot>
```

When RedBoot issues its prompt it is also ready to accept connections from m68k-elf-gdb, allowing eCos applications to be downloaded and debugged.

Occasionally it may prove necessary to update the installed RedBoot image. This can be done simply by repeating the above process, using dBUG to load the dBUG version of RedBoot `redboot_dbug.srec`. Alternatively the existing RedBoot install can be used to load a RAM-resident version, `redboot_ram.bin`.

The ROMFFE version of RedBoot can be installed at location 0xFFE00000, replacing dBUG. This may be useful if the system needs more flash blocks than are available with the usual ROM RedBoot. Installing this RedBoot image will typically involve a BDM-based utility.

Rebuilding RedBoot

Should it prove necessary to rebuild a RedBoot binary, this is done most conveniently at the command line. The steps needed to rebuild the dBUG version of RedBoot are:

```
$ mkdir redboot_dbug
$ cd redboot_dbug
$ ecosconfig new m5272c3 redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/m68k/mcf52xx/mcf5272/m5272c3/v2_0_1/misc/redboot_DBUG.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

At the end of the build the `install/bin` subdirectory should contain the required file `redboot_dbug.srec`.

Rebuilding the RAM and ROM versions involves basically the same process. The RAM version uses the file `redboot_RAM.ecm` and generates a file `redboot_ram.bin`. The ROM version uses the file `redboot_ROM.ecm` and generates a file `redboot_rom.bin`.

BDM

An alternative to debugging an application on top of Redboot is to use a BDM hardware debug solution. On the eCos side this requires building the configuration for RAM startup and with CYGSEM_HAL_USE_ROM_MONITOR disabled. Note that a RAM build of RedBoot automatically has the latter configuration option disabled, so it is possible to run a RAM RedBoot via BDM and bypass the dBUG stages of the installation process.

On the host-side the details depend on exactly which BDM solution is in use. The recommended BDM debug solution is the Ronetix PEEDI. Other solutions such as the P&E USBMultilink device have proved unreliable, so if a PEEDI is not available then it is recommended that application developers should debug their applications on top of RedBoot's gdb stubs.

The PEEDI requires a configuration file peedi.cfg which can be found in the platform HAL's misc directory. The configuration file will initialize the hardware in the same way as standard eCos applications, so applications can be loaded into RAM and run as normal. The configuration file will need minor edits, for example to specify the correct license keys. For full details see the Ronetix documentation. Once the PEEDI is correctly set up m68k-elf-gdb can then connect to it in the usual way:

```
$ m68k-elf-gdb install/tests/kernel/current/tests/tm_basic
GNU gdb 6.4.50.20060226-cvs (eCosCentric)
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "--host=i686-pc-linux-gnu --target=m68k-elf"...
(gdb) target remote peedi:9000
0x400008f6 in ?? ()
(gdb) load
Loading section .m68k_start, size 0x98 lma 0x40010000
Loading section .text, size 0xa918 lma 0x40010098
Loading section .rodata, size 0x114a lma 0x4001a9b0
Loading section .data, size 0x18c lma 0x4001bafc
Start address 0x40010000, load size 48262
Transfer rate: 260172 bits/sec, 3217 bytes/write.
(gdb) break cyg_test_exit
Breakpoint 1 at 0x40016172: file /home/bartv/ecos/ecospro-common/infra/current/src/tcdiag.cxx, line 310.
void cyg_test_exit(void);
(gdb) continue
Continuing.

Breakpoint 1, cyg_test_exit () at /home/bartv/ecos/ecospro-common/infra/current/src/tcdiag.cxx:310
310         if (code_checksum != cyg_crc16(_stext, _etext - _stext)) {
(gdb) quit
The program is running.  Exit anyway? (y or n) y
$
```

Unlike the PEEDI, some BDM solutions will not automatically initialize the hardware. Instead this can be achieved using a set of example gdb macros which can be found in the bdm.gdb file in the platform HAL's misc subdirectory. The macros need to be called either through a configuration file or directly to initialize the hardware prior to downloading the eCos application.

Name

Configuration — Platform-specific Configuration Options

Overview

The M5272C3 platform HAL package is loaded automatically when eCos is configured for an M5272C3 target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The M5272C3 platform HAL package supports four separate startup types:

RAM
: This is the startup type which is normally used during application development. The board has RedBoot programmed into flash at location 0xFFF00000 and boots from that location. m68k-elf-gdb is then used to load a RAM startup application into memory and debug it. It is assumed that the hardware has already been initialized by RedBoot. By default the application will use eCos' virtual vectors mechanism to obtain certain services from RedBoot, including diagnostic output.

ROM
: This startup type can be used for finished applications which will be programmed into flash at location 0xFFF00000. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization.

ROMFFE
: This is a variant of the ROM startup type which can be used if the application will be programmed into flash at location 0xFFE00000, overwriting the board's dBUG ROM monitor.

DBUG
: This is a variant of the RAM startup which allows applications to be loaded via the board's dBUG ROM monitor rather than via RedBoot. It exists mainly to support the dBUG version of RedBoot which is needed during hardware setup. Once the application has started it will take over all the hardware, and it will not depend on any services provided by dBUG. This startup type does not provide gdb debug facilities.

RedBoot and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building RedBoot.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then one of the serial ports will be claimed for HAL diagnostics. By default eCos will use the terminal port, corresponding to uart0. The auxiliary port, uart1, can be selected instead via the configuration option `CYGHWR_HAL_M68K_MCFxxxx_DIAGNOSTICS_PORT`. The baud rate for the selected port is controlled by `CYGNUM_HAL_M68K_MCFxxxx_DIAGNOSTICS_BAUD`.

Flash Driver

The platform HAL package contains flash driver support. By default this is inactive, and it can be made active by loading the generic flash package `CYGPKG_IO_FLASH`.

Special Registers

The MCF5272 processor has a number of special registers controlling the cache, on-chip RAM and ROM, and so on. The platform HAL provides a number of configuration options for setting these, for example `CYGNUM_HAL_M68K_M5272C3_RAMBAR` controls the initial value of the RAMBAR register. These options are only used during a ROM or ROMFFE startup. For a RAM startup it will be RedBoot that initializes these registers, so if the default values are not appropriate for the target application then it will be necessary to rebuild RedBoot with new settings for these options. Alternatively it should be possible to reprogram some or all of the registers early on during startup, for example by using a high-priority static constructor.

One of the special registers, MBAR, cannot be controlled via a configuration option. Changing the value of this register could have drastic effects on the system, for example moving the on-chip peripherals to a different location in memory, and it would be very easy to end up with inconsistencies between RedBoot and the eCos application. Instead the on-chip peripherals are always mapped to location 0x10000000.

System Clock

By default the system clock interrupts once every 10ms, corresponding to a 100Hz clock. This can be changed by the configuration option `CYGNUM_HAL_RTC_PERIOD`, the number of microseconds between clock ticks. Other clock-related settings are recalculated automatically if the period is changed.

Compiler Flags

The platform HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos. There are three flags specific to this port:

`-mcpu=5272`
: The m68k-elf-gcc compiler supports many variants of the M68K architecture, from the original 68000 onwards. For an MCF5272 processor `-mcpu=5272` should be used.

`-malign-int`
: This option forces m68k-elf-gcc to align integer and floating point data to a 32-bit boundary rather than a 16-bit boundary. It should improve performance. However the resulting code is incompatible with most published application binary interface specifications for M68K processors, so it is possible that this option causes problems with existing third-party object code.

`-fomit-frame-pointer`
: Traditionally the `%A6` register was used as a dedicated frame pointer, and the compiler was expected to generate link and unlink instructions on procedure entry and exit. These days the compiler is perfectly capable of generating working code without a frame pointer, so omitting the frame pointer often saves some work during procedure entry and exit and makes another register available for optimization. However without a frame pointer register the m68k-elf-gdb debugger is not always able to interpret a thread stack, so it cannot reliably give a backtrace. Removing `-fomit-frame-pointer` from the default flags will make debugging easier, but the generated code may be worse.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the M5272C3 hardware, and shold be read in conjunction with that specification. The M5272C3 platform HAL package complements the M68K architectural HAL, the MCFxxxx variant HAL, and the MCF5272 processor HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset the HAL will initialize or reinitialize most of the on-chip peripherals. There is an exception for RAM startup applications which depend on a ROM monitor for certain services: the UARTs and the ethernet device will not be reinitialized because they may be in use by RedBoot for communication with the host.

For a ROM or ROMFFE startup the HAL will perform additional initialization, setting up the external DRAM and programming the various internal registers. The values used for most of these registers are configurable. Full details can be found in the exported headers `cyg/hal/plf.inc` and `cyg/hal/proc.inc`.

Linker Scripts and Memory Maps

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

external SDRAM

This is mapped to location 0x00000000. The first 384 bytes are used for hardware exception vectors. The next 256 bytes are normally used for the eCos virtual vectors, allowing RAM-based applications to use services provided by the ROM monitor. For ROM and ROMFFE startup all remaining SDRAM is available. For RAM and DBUG startup available SDRAM starts at location 0x00020000, with the bottom 128K reserved for use by either the RedBoot or dBUG ROM monitors.

on-chip peripherals

These are accessible at location 0x10000000 onwards, as per the defined symbol `HAL_MCFxxxx_MBAR`. This address cannot easily be changed during development because both the ROM monitor and the application must use the same address. The `%mbar` system register is initialized appropriately during a ROM or ROMFFE startup.

on-chip SRAM

The 4K of internal SRAM are normally mapped at location 0x20000000. The `%rambar` register is initialized during a ROM startup using the value of the configuration option `CYGNUM_HAL_M68K_M5272C3_RAMBAR`. Neither eCos nor RedBoot use the internal SRAM so all of it is available to application code.

on-chip ROM

Usually this is left disabled since its contents are of no interest to most applications. If it is enabled then it is usually mapped at location 0x21000000. The `%rombar` register is initialized during a ROM startup using the value of the configuration option `CYGNUM_HAL_M68K_M5272C3_ROMBAR`.

off-chip Flash

This is located at the top of memory, location 0xFFE00000 onwards. For ROM and RAM startups it is assumed that a jumper is used to disable the bottom half of the flash, so location 0xFFE00000 is actually a mirror of 0xFFF00000. For ROMFFE and DBUG startups all of the flash is visible. By default the flash block at location 0xFFF00000 is used to hold RedBoot or another ROM startup application, and the block at location 0xFFFC00000 is used to hold flash management data and the RedBoot **fconfig** variables. The blocks at 0xFFF400000 and 0xFFF80000 can be used by application code.

Clock Support

The platform HAL provides configuration options for the eCos system clock. This always uses the hardware timer 3, which should not be used directly by application code. The gprof-based profiling code uses timer 2, so that is only available when not profiling. Timers 0 and 1 are never used by eCos so application code is free to manipulate these as required. The actual HAL macros for managing the clock are provided by the MCF5272 processor HAL. The specific numbers used are a characteristic of the platform because they depend on the processor speed.

Other Issues

The M5272C3 platform HAL does not affect the implementation of other parts of the eCos HAL specification. The MCF5272 processor HAL, the MCFxxxx variant HAL, and the M68K architectural HAL documentation should be consulted for further details.

Other Functionality

The platform HAL package also provides a flash driver for the off-chip AMD AM29PL160C flash chip. This driver is inactive by default, and only becomes active if the configuration includes the generic flash support `CYGPKG_IO_FLASH`.

When building RedBoot with the Robust Boot Loader package `CYGPKG_RBL`, the platform HAL provides a macro for the **rbl condboot** command. If the INT6 button is pressed when **rbl condboot** executes then the boot will be aborted, otherwise it will proceed normally.

Chapter 321. Freescale MCF5275 Processor Support

Name

CYGPKG_HAL_M68K_MCF5275 — eCos Support for Freescale MCF5275 Processors

Description

The Freescale MCF5275 microcontroller family covers the MCF5274, MCF5274L, MCF5275 and MCF5275L ColdFire processors. These differ slightly in the set of peripherals available. The L parts are limited to a single ethernet controller and 2 UARTs, with the non-L parts having two ethernet controllers and 3 UARTs. The MCF5275L and MCF5275 parts have a hardware cryptography accelerator, which is not currently used by eCos.

Note

The eCos MCF52xx ethernet driver currently only supports a single ethernet device (FEC0), so the second ethernet available on the MCF5274 and MCF5275 parts is not currently used by eCos.

The processor HAL package CYGPKG_HAL_M68K_MCF5275 provides support for all MCF5275 processors, although at the time of writing it has only been tested on an MCF5274. It complements the M68K architectural HAL package CYGPKG_HAL_M68K and the variant HAL package CYGPKG_HAL_M68K_MCFxxxx. An eCos configuration should also include a platform HAL package to support board-level details. e.g. how the on-chip peripherals are connected to the outside world.

Configuration

The MCF5275 processor HAL package should be loaded automatically when eCos is configured for appropriate target hardware. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

The package's CDL script provides default interrupt priorities for some of the mcf5275's on-chip devices. This makes it easier for the various device driver packages to install unique interrupt priorities, as required by the hardware.

Most of the package's configuration options relate to hardware. The settings are generally determined by the platform HAL and there is little need for application developers to change them. The first hardware option is CYGHWR_HAL_M68K_MCF5275_CPU, identifying the specific MCF5275 processor being used. Legal values are MCF5274L, MCF5275L, MCF5274 and MCF5275. Typically the platform HAL will set this option via a CDL constraint.

Component CYGHWR_HAL_M68K_MCF5275_GPIO contains various options related to pin-connectivity. This gives full control over the PAR pin assignment registers, and for those pins configured as GPIO it is also possible to control the pin direction and data settings. These options are used to initialize the processor's GPIO module early on during system initialization, but applications may change settings later on as necessary. The platform HAL can define CYGHWR_HAL_M68K_MCF5275_BOARD_PINS to specify the default I/O pin mapping to be used.

The GPIO settings are used to determine default settings for on-chip peripherals, including the three UARTs, the I²C bus and the QSPI bus. For example if none of the relevant GPIO pins are assigned to UART2 then component CYGHWR_HAL_M68K_MCFxxxx_UART2 will be disabled by default, and that UART cannot be used for HAL diagnostics nor accessed via the serial device driver. It is possible to override these settings if desired, for example if a UART is connected but will be manipulated directly by application code instead of via a device driver.

1. For each of the three on-chip UARTs there will be a component, e.g. CYGHWR_HAL_M68K_MCFxxxx_UART0, determining whether or not the UART is usable on the target hardware. There are additional options CYGHWR_HAL_M68K_MCFxxxx_UART0_RTS and CYGHWR_HAL_M68K_MCFxxxx_UART0_CTS indicating whether or not the hardware handshake lines are connected, and CYGHWR_HAL_M68K_MCFxxxx_UART0_RS485_RTS to indicate that the RTS line controls an RS485 transceiver.
2. Component CYGHWR_HAL_M68K_MCF5275_I2C determines whether or not the processor HAL will instantiate an I²C bus device hal_mcfxxxx_i2c_bus. There are also options to control the interrupt priority and to set the FDR register which

controls the bus speed. The default bus speed will be the standard I²C bus speed of 100KHz, or as close as can be achieved given hardware limitations.

3. Component CYGHWR_HAL_M68K_MCF5275_SPI determines whether or not the processor HAL will instantiate an SPI bus device hal_mcfxxxx_qspi_bus. It contains an additional configuration option for the interrupt priority.

For configurations which include the eCos kernel, CYGIMP_HAL_M68K_MCF5275_IDLE determines what happens when the idle thread runs.

The option CYGPKG_HAL_M68K_MCF5275_ISR_PRIORITIES provides support for configuring the interrupt priorities as detailed in the interrupt priorities section.

The HAL Port

This section describes how the MCF5275 processor HAL package implements parts of the eCos HAL specification. It should be read in conjunction with similar sections from the architectural and variant HAL documentation.

HAL I/O

The header file cyg/hal/proc_io.h provides definitions of MCF5275-specific on-chip peripherals. Many of the on-chip peripherals are compatible with those on the MCF5282 or other ColdFire processors, and for those peripherals it is the var_io.h header provided by the MCFxxxx variant HAL which provides the appropriate definitions. Both headers are automatically included by the architectural header cyg/hal/hal_io.h, so typically application code and other packages will just include the latter.

The MCF5275 reserves a 1GB area of memory for the internal peripheral space. Usually this is mapped between 0x40000000 and 0x7FFFFFFF. Most of this space is not used, but accessing it can cause problems including apparently locking up the processor such that either a hard reset or a watchdog timeout is needed. The target-side gdb stubs code can trap most accesses initiated by host-side gdb, but cannot protect against errant accesses by application code.

Interrupt Handling

The header file cyg/hal/proc_intr.h provides VSR and ISR vector numbers for all interrupt sources. The VSR vector number, for example CYGNUM_HAL_VECTOR_TMR0, should be used for calls like cyg_interrupt_get_vsr. It corresponds directly to the M68K exception number. The ISR vector number, for example CYGNUM_HAL_ISR_TMR0, should be used for calls like cyg_interrupt_create. This header file is automatically included by the architectural header cyg/hal/hal_intr.h, and other packages and application code will normally just include the latter.

The eCos HAL macros HAL_INTERRUPT_MASK, HAL_INTERRUPT_UNMASK, HAL_INTERRUPT_ACKNOWLEDGE, and HAL_INTERRUPT_CONFIGURE are implemented by the processor HAL. The mask and unmask operations are straightforward, simply manipulating the IMR registers in the on-chip interrupt controllers. The acknowledge macro is only relevant for external interrupts coming in via the edge port module and will clear the interrupt by writing to the EPIER register. There is no simple way to clear interrupts generated by other sources. Instead each such interrupt has to be cleared in a device-specific way, and that is the responsibility of the appropriate device driver. The configure macro is only relevant for external interrupts and involves manipulating the edge port module.

The HAL_INTERRUPT_SET_LEVEL macro, used implicitly by higher level code such as cyg_interrupt_create, is also implemented by the processor HAL. In the MCF5275 processor interrupt priorities have to be managed very carefully. Interrupts are managed via two interrupt controllers, INTC0 and INTC1. Each controller contains ICRxx control registers for each interrupt to manage that interrupt's priority. The HAL_INTERRUPT_SET_LEVEL macro simply fills in the ICRxx register.

An ICRxx value is a six-bit number. The top three bits correspond to the standard M68K IPL interrupt level. The bottom three bits give a finer-grained priority within that IPL level. For example, if the priority argument to cyg_interrupt_create is 42 then that corresponds to an IPL level of 5 and a finer-grained priority of 2. If the system has been configured to support nested interrupts and a level 42 interrupt goes off, the processor's IPL level will be set to 5 so all interrupts with priorities < 48 will remain blocked. The finer-grained priority controls what happens when two interrupts with the same IPL level go off at the same time.

Interrupt priorities between 0 and 7 would correspond to an IPL level of 0. The interrupt controller can only raise an interrupt if the IPL level is at least 1, so the smallest valid interrupt priority is 8. Interrupt priorities between 56 and 63 correspond to IPL level 7, and such interrupts are non-maskable and must be used with care. These interrupts cannot be managed safely by the usual eCos ISR and DSR mechanisms, instead application code will have to install a custom VSR and manage the entire interrupt. This means that interrupt priorities should normally be in the range 8 to 55.

As a special case, external interrupts coming in via the edge port module have hard-wired priorities which do not clash with the programmable ones. For these the priority argument to `HAL_INTERRUPT_SET_LEVEL` and higher-level code is ignored.

The MCF5275 interrupt controllers have a major restriction: all interrupt priorities within each controller must be unique. If two interrupts go off at the same time and have exactly the same priority then the controllers' behaviour is undefined. In a typical application some of the interrupts will be handled by eCos device drivers while others will be handled directly by application code. Since eCos cannot know which interrupts may get used, it cannot allocate unique priorities. Instead this has to be left to the application developer. eCos does provide configuration options such as `CYGNUM_KERNEL_COUNTERS_CLOCK_ISR_PRIORITY` and `CYGNUM_DEVS_SERIAL_MCFxxxx_SERIAL0_ISR_PRIORITY` to provide control over the eCos-managed interrupts, and provides default values for these which are unique.

To ensure that the configured interrupt priorities are unique the processor HAL comes with a test case `intrpri`. The source code for this can be found in the package's `tests` subdirectory. The test examines the ICRxx registers in both interrupt controllers. It will report all priorities that are in use, and report a failure if a non-unique priority is detected. This code may prove useful for application developers trying to allocate interrupt priorities.

Caution

Non-unique interrupt priorities can lead to very confusing system behaviour. For example if the PIT3 system clock (interrupt 0x27) and UART2 (interrupt 0x0F) are accidentally given the same priority and go off at the same time, the interrupt controller may actually issue an interrupt 0x2F, the bitwise-or of the two interrupt numbers. That interrupt belongs to the on-chip USB module. There may not be an installed handler for that interrupt at all, and even if there is a handler it will only manipulate the USB hardware and not clear the system clock and UART interrupts. Hence the system is likely to go into a spin, continually trying to service the wrong interrupt. To track down such problems during debugging it may prove useful to install a breakpoint on the `hal_arch_default_isr` function.

Clock Support

The processor HAL provides support for the eCos system clock. This always uses hardware timer PIT3, which should not be manipulated directly by application code. If gprof-based profiling is enabled then that will use hardware timer PIT2. PIT timers 0 and 1 are never used by eCos so application code is free to manipulate these as required.

Some of the configuration options related to the system clock, for example `CYGNUM_HAL_RTC_PERIOD`, are actually contained in the platform HAL rather than the processor HAL. These options need to take into account the processor clock speed, a characteristic of the platform rather than the processor.

Cache Handling

The MCF5275 has 16K of cache. Usually this will be set up as a split cache, 8K for instructions and 8K for data, which should give the best performance for typical applications. The standard HAL cache macros are supported.

On some platforms it may be better to organize the cache differently. For example if the platform involves running code only out of internal SRAM which may access external data, it may be possible to improve performance by using a 16K data cache instead of a split cache. This is controlled by `CYGIMP_HAL_M68K_MCF5275_CACHE_MODE`, which may be either a fixed `#define` or a configuration option depending on the platform. For more details see the header file `proc_cache.h`.

The HAL also defines a macro `HAL_MEMORY_BARRIER()` which acts to synchronize the pipeline, delaying execution until all previous operations including all pending writes are complete. This will usually be necessary when interacting with devices that access memory directly.

Other Issues

The MCF5275 processor HAL does not affect the implementation of data types, stack size definitions, SMP support, system startup, or debug support. The MCFxxxx variant HAL versions of `HAL_LSBIT_INDEX` and `HAL_MSBIT_INDEX` are used since the processor supports the `ff1.l` and `bitrev.l` instructions. `HAL_DELAY_US` is implemented as a simple counting loop. `HAL_IDLE_THREAD_ACTION` may be defined depending on configuration option `CYGIMP_HAL_M68K_MCF5275_IDLE`.

Other Functionality

The processor HAL will instantiate a cyg_i2c_bus structure `hal_mcfxxxx_i2c_bus` when the configuration option `CYGHWR_HAL_M68K_MCFxxxx_I2C` is enabled. That option is enabled by default if various GPIO pins are configured appropriately. The implementation is provided by the `CYGPKG_DEVS_I2C_MCFxxxx` device driver. The processor HAL does not know what I²C devices may be attached to the bus so that is left to the platform HAL.

The processor HAL will instantiate a cyg_spi_bus structure `hal_mcfxxxx_qspi_bus` when the configuration option `CYGHWR_HAL_M68K_MCFxxxx_SPI` is enabled. That option is enabled by default if various GPIO pins are configured appropriately. The implementation is provided by the `CYGPKG_DEVS_SPI_MCFxxxx_QSPI` device driver. The processor HAL does not know what SPI devices may be attached to the bus so that is left to the platform HAL. All SPI device structures should be placed in the table `mcfxxxx_qspi`.

Chapter 322. Freescale MCF5282 Processor Support

Name

CYGPKG_HAL_M68K_MCF5282 — eCos Support for the Freescale MCF5282 Processor

Description

The MCF5282 is one member of the Freescale MCFxxxx ColdFire range of processors. It comes with a number of on-chip peripherals including 3 UARTs and ethernet. The processor HAL package CYGPKG_HAL_M68K_MCF5282 provides support for features that are specific to the MCF5282. It complements the M68K architectural HAL package CYGPKG_HAL_M68K and the variant HAL package CYGPKG_HAL_M68K_MCFxxxx. An eCos configuration should also include a platform HAL package, for example CYGPKG_HAL_M68K_M5282EVB to support board-level details like the external memory chips.

The MCF5282 processor HAL supports the MCF5280. The only difference between these two processors is the presence of on-chip flash. The platform HAL and the ecos.db **target** entry will handle this difference.

Configuration

The MCF5282 processor HAL package should be loaded automatically when eCos is configured for appropriate target hardware. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

The package's CDL script provides default interrupt priorities for some of the mcf5282's on-chip devices. This makes it easier for the various device driver packages to install unique interrupt priorities, as required by the hardware.

The HAL Port

This section describes how the MCF5282 processor HAL package implements parts of the eCos HAL specification. It should be read in conjunction with similar sections from the architectural and variant HAL documentation.

HAL I/O

The header file cyg/hal/proc_io.h provides definitions of all on-chip peripherals, except for some UART definitions which are provided by the variant HAL instead. This header file is automatically included by the architectural header cyg/hal/hal_io.h, so typically application code and other packages will just include the latter. There are default INIT macros for the various on-chip devices which can be used by platform initialization code, although on any given platform some of the devices may need special attention.

The MCF5282 reserves a 1-gbyte area of memory for the internal peripheral space. Usually this is mapped between 0x40000000 and 0x7fffffff. Most of this space is not used, but accessing it can cause problems including apparently locking up the processor such that either a hard reset or a watchdog timeout is needed. The target-side gdb stubs code can trap most accesses initiated by host-side gdb, but cannot protect against errant accesses by application code.

It should be noted that the Freescale documentation is occasionally confusing when it comes to numbering devices. For example the four on-chip programmable timers are numbered PIT1 to PIT4, but in the interrupt controller the corresponding interrupts are numbered PIT0 to PIT3. The eCos port consistently starts numbering at 0, so the timers have been renamed PIT0 to PIT3.

Interrupt Handling

The header file cyg/hal/proc_intr.h provides VSR and ISR vector numbers for all interrupt sources. The VSR vector number, for example CYGNUM_HAL_VECTOR_TMR0, should be used for calls like cyg_interrupt_get_vsr. It corresponds directly to the M68K exception number. The ISR vector number, for example CYGNUM_HAL_ISR_TMR0, should be used for calls like cyg_interrupt_create. This header file is automatically included by the architectural header cyg/hal/hal_intr.h, and other packages and application code will normally just include the latter.

The eCos HAL macros HAL_INTERRUPT_MASK, HAL_INTERRUPT_UNMASK, HAL_INTERRUPT_ACKNOWLEDGE, and HAL_INTERRUPT_CONFIGURE are implemented by the processor HAL. The mask and unmask operations are straightforward, simply manipulating the IMR registers in the on-chip interrupt controllers. The acknowledge macro is only relevant for external interrupts coming in via the edge port module and will clear the interrupt by writing to the EPIER register. There is no simple way to clear interrupts generated by other sources. Instead each such interrupt has to be cleared in a device-specific way, and that is the responsibility of the appropriate device driver. The configure macro is only relevant for external interrupts and involves manipulating the edge port module.

The HAL_INTERRUPT_SET_LEVEL macro, used implicitly by higher level code such as cyg_interrupt_create, is also implemented by the processor HAL. In the MCF5282 processor interrupt priorities have to be managed very carefully. Interrupts are managed via two interrupt controllers, INTC0 and INTC1. Each controller contains ICRxx control registers for each interrupt to manage that interrupt's priority. The HAL_INTERRUPT_SET_LEVEL macro simply fills in the ICRxx register.

An ICRxx value is a six-bit number. The top three bits correspond to the standard M68K IPL interrupt level. The bottom three bits give a finer-grained priority within that IPL level. For example, if the priority argument to cyg_interrupt_create is 42 then that corresponds to an IPL level of 5 and a finer-grained priority of 2. If the system has been configured to support nested interrupts and a level 42 interrupt goes off, the processor's IPL level will be set to 5 so all interrupts with priorities < 48 will remain blocked. The finer-grained priority controls what happens when two interrupts with the same IPL level go off at the same time.

Interrupt priorities between 0 and 7 would correspond to an IPL level of 0. The interrupt controller can only raise an interrupt if the IPL level is at least 1, so the smallest valid interrupt priority is 8. Interrupt priorities between 56 and 63 correspond to IPL level 7, and such interrupts are non-maskable and must be used with care. These interrupts cannot be managed safely by the usual eCos ISR and DSR mechanisms, instead application code will have to install a custom VSR and manage the entire interrupt. This means that interrupt priorities should normally be in the range 8 to 55.

As a special case, external interrupts coming in via the edge port module have hard-wired priorities which do not clash with the programmable ones. For these the priority argument to HAL_INTERRUPT_SET_LEVEL and higher-level code is ignored.

The MCF5282 interrupt controllers have a major restriction: all interrupt priorities within each controller must be unique. If two interrupts go off at the same time and have exactly the same priority then the controllers' behaviour is undefined. In a typical application some of the interrupts will be handled by eCos device drivers while others will be handled directly by application code. Since eCos cannot know which interrupts may get used, it cannot allocate unique priorities. Instead this has to be left to the application developer. eCos does provide configuration options such as CYGNUM_KERNEL_COUNTERS_CLOCK_ISR_PRIORITY and CYGNUM_DEVS_SERIAL_MCFxxxx_SERIAL0_ISR_PRIORITY to provide control over the eCos-managed interrupts, and provides default values for these which are unique.

To ensure that the configured interrupt priorities are unique the processor HAL comes with a test case intrpri. The source code for this can be found in the package's tests subdirectory. The test examines the ICRxx registers in both interrupt controllers. It will report all priorities that are in use, and report a failure if a non-unique priority is detected. This code may prove useful for application developers trying to allocate interrupt priorities.

Caution

Non-unique interrupt priorities can lead to very confusing system behaviour. For example if the PIT3 system clock (interrupt 0x3a) and ethernet RX frame (interrupt 0x1b) are accidentally given the same priority and go off at the same time, the interrupt controller may actually issue an interrupt 0x3b, the bitwise or of the two interrupt numbers. That interrupt belongs to the on-chip flash module. There may not be an installed handler for that interrupt at all, and even if there is a handler it will only manipulate the flash hardware and not clear the system clock and ethernet interrupts. Hence the system is likely to go into a spin, continually trying to service the wrong interrupt. To track down such problems during debugging it may prove useful to install a breakpoint on the hal_arch_default_isr function.

Clock Support

The processor HAL provides support for the eCos system clock. This always uses hardware timer PIT3, which should not be manipulated directly by application code. If gprof-based profiling is enabled then that will use hardware timer PIT2. PIT timers 0 and 1 are never used by eCos so application code is free to manipulate these as required.

Some of the configuration options related to the system clock, for example `CYGNUM_HAL_RTC_PERIOD`, are actually contained in the platform HAL rather than the processor HAL. These options need to take into account the processor clock speed, a characteristic of the platform rather than the processor.

Cache Handling

The MCF5282 has 2K of cache. Usually this will be set up as a split cache, 1K for instructions and 1K for data, which should give the best performance for typical applications. The standard HAL cache macros are supported.

On some platforms it may be better to organize the cache differently. For example if the platform involves running code only out of internal flash but may access external data, it may be possible to improve performance by using a 2K data cache instead of a split cache. This is controlled by `CYGIMP_HAL_M68K_MCF5282_CACHE_MODE`, which may be either a fixed `#define` or a configuration option depending on the platform. For more details see the header file `proc_cache.h`.

Other Issues

The MCF5282 processor HAL does not affect the implementation of data types, stack size definitions, bit indexing, idle thread processing, linker scripts, SMP support, system startup, or debug support.

Other Functionality

The MCF5282 processor HAL only implements functionality defined in the eCos HAL specification and does not export any additional functions.

Chapter 323. Freescale M5282EVB Board Support

Name

eCos Support for the Freescale M5282EVB Board — Overview

Description

The Freescale M5282EVB board has an MCF5282 ColdFire processor, 16MB of external SDRAM, 2MB of external flash memory, plus required support chips for the on-chip peripherals. By default the board comes with its own dBUG ROM monitor, located in the bottom half of the flash.

For typical eCos development a RedBoot image is programmed into the top half of the flash memory, and the board is made to boot this image rather than the existing dBUG monitor. RedBoot provides gdb stub functionality so it is then possible to download and debug eCos applications via the gdb debugger. This can happen over either a serial line or over ethernet.

Supported Hardware

The memory map used by both eCos and RedBoot is as follows:

Memory	Base	Length
External SDRAM	0x00000000	0x01000000
Internal RAM	0x20000000	0x00010000
On-chip Peripherals	0x40000000	0x40000000
On-chip Flash	0xF0000000	0x00080000
External Flash	0xFFE00000	0x00200000

eCos can be configured for one of four startup types:

RAM
: This is the startup type normally used during application development. RedBoot is programmed into flash and performs the initial bootstrap. m68k-elf-gdb is then used to load a RAM startup application into memory and debug it. By default the application will use eCos' virtual vectors mechanism to obtain certain services from RedBoot, including diagnostic output.

  With a minor change to the eCos configuration this startup type can also be used to debug applications via BDM.

DBUG
: This is a variant of the RAM startup which allows applications to be loaded via the board's dBUG ROM monitor rather than via RedBoot. Once the application has started it will take over all the hardware, and it will not depend on any services provided by dBUG. This startup type does not provide gdb debug facilities. It is used primarily for building a special version of RedBoot, used during hardware setup.

ROM
: This startup type can be used for finished applications which will be programmed into flash at location 0xFFF00000. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization. This startup type is used for building the flash-resident version of RedBoot but can also be used for application code.

ROMFFE
: This is a variant of the ROM startup type which can be used if the application will be programmed into flash at location 0xFFE00000, overwriting the board's dBUG ROM monitor.

For all startup types the external SDRAM is used to hold all data. For RAM and dBUG startup the code also resides in external SDRAM, with the first 64K reserved for use by the ROM monitor. The 64K of internal RAM is not used, either by RedBoot or by eCos, so all of it is available to the application. The 512K of internal flash is not currently used.

In a typical setup the bottom half of the external flash is reserved for the dBUG ROM monitor and is not accessible to eCos. That leaves 16 flash blocks of 64K each. Of these the first two are used for the RedBoot image and another is used for managing the flash and holding RedBoot fconfig values. The remaining 13 blocks from 0xFFF20000 to 0xFFFF0000 can be used by application code.

Alternatively it is possible to build an application for ROM startup and program it into flash at 0xFFF00000, replacing RedBoot. For a larger application ROMFFE startup can be used instead, making all of external flash available to application code.

RedBoot can communicate with the host using either ethernet or one of the UARTs - usually uart0 corresponding to the terminal port on the m5282evb board.

All configurations for the M5282EVB target include an ethernet driver package `CYGPKG_DEVS_ETH_MCFxxxx`. If the application does not actually require ethernet functionality then the package is inactive and the final executable will not suffer any overheads from unused functionality. This is determined by the presence of the generic ethernet I/O package `CYGPKG_IO_ETH_DRIVERS`. Typically the choice of eCos template causes the right thing to happen. For example the default template does not include any TCP/IP stack so `CYGPKG_IO_ETH_DRIVERS` is not included, but both the net and redboot templates do include a TCP/IP stack so will specify that package and hence enable the ethernet driver. The ethernet device can be shared by RedBoot and the application, so it is possible to debug a networked application over ethernet.

The M5282EVB board does not have a serial EPROM or similar hardware providing a unique network MAC address. Instead a suitable address has to be programmed into flash via RedBoot's **fconfig** command.

All configurations for the M5282EVB target include a serial device driver package `CYGPKG_DEVS_SERIAL_MCFxxxx`. The driver as a whole is inactive unless the generic serial support, `CYGPKG_IO_SERIAL_DEVICES` is enabled. Exactly which of the on-chip UARTs are supported is controlled by configuration options within the platform HAL. By default both uart0 and uart1 are supported, corresponding to the terminal and auxiliary ports. If the UART is needed by the application then it cannot also be used by RedBoot for gdb traffic, so another communication channel such as ethernet should be used instead.

All configurations for the M5282EVB target also include a watchdog device driver `CYGPKG_DEVS_WATCHDOG_MCF5282`. This driver is inactive unless the generic watchdog support `CYGPKG_IO_WATCHDOG` is loaded.

The on-chip interrupt controllers and the edge port module are managed by eCos using macros provided by the MCF5282 processor HAL. PIT timer 3 is normally used to implement the eCos system clock. If gprof-based profiling is enabled then that will use PIT timer 2. PIT timers 0 and 1 are unused and can be manipulated by the application. The GPIO pins are manipulated only as needed to get the UART(s) and ethernet working. eCos will reset the remaining on-chip peripherals (DMA, GPT, DMA timers, QSPI, I²C, FlexCAN, and QADC) during system startup or soft reset, but will not otherwise manipulate them.

Tools

The M5282EVB port is intended to work with GNU tools configured for an m68k-elf target. The original port was done using m68k-elf-gcc version 3.2.1, m68k-elf-gdb version 5.3, and binutils version 2.13.1.

By default eCos is built using the compiler flag `-fomit-frame-pointer`. Omitting the frame pointer eliminates some work on every function call and makes another register available, so the code should be smaller and faster. However without a frame pointer m68k-elf-gdb is not always able to identify stack frames, so it may be unable to provide accurate backtrace information. Removing this compiler flag from the configuration option `CYGBLD_GLOBAL_CFLAGS` avoids such debug problems.

A typical setup involves m68k-elf-gdb interacting with RedBoot using either serial or ethernet. Alternatively it is possible to debug via the BDM port. The package's `misc` subdirectory contains a script `bdm.gdb` that contains macros for the low-level hardware initialization normally performed by the ROM startup code. The application should be linked with an eCos configuration using RAM startup, and with the options `CYGSEM_HAL_ROM_MONITOR` and `CYGSEM_HAL_USE_ROM_MONITOR` disabled to stop eCos accessing any services provided by RedBoot. Diagnostic output will be sent out of uart0.

Name

Setup — Preparing the M5282EVB board for eCos Development

Overview

In a typical development environment the M5282EVB board boots from flash into the RedBoot ROM monitor. eCos applications are configured for a RAM startup, and then downloaded and run on the board via the debugger m68k-elf-gdb. Preparing the board therefore involves programming a suitable RedBoot image into flash memory.

The following RedBoot configurations are supported:

Configuration	Description	Use	File
ROM	RedBoot running from the board's flash	`redboot_ROM.ecm`	`redboot_rom.bin`
dBUG	Used for initial setup	`redboot_DBUG.ecm`	`redboot_dbug.srec`
RAM	Used for upgrading ROM version	`redboot_RAM.ecm`	`redboot_ram.bin`
ROMFFE	RedBoot running from the board's flash at 0xFFE00000	`redboot_ROMFFE.ecm`	`redboot_romffe.bin`

For serial communications all versions run with 8 bits, no parity, and 1 stop bit. The dBUG version runs at 19200 baud. The ROM and RAM versions run at 38400 baud. These baud rates can be changed via the configuration option `CYGNUM_HAL_M68K_M5282EVB_DIAG_BAUD`. By default RedBoot will use the board's terminal port, corresponding to uart0, but this can also be changed via the configuration option `CYGHWR_HAL_M68K_M5282EVB_DIAGNOSTICS_PORT`. On an M5282EVB platform RedBoot also supports ethernet communication and flash management.

Initial Installation

This process assumes that the board still has its original dBUG ROM monitor and does not require any special debug hardware. It leaves the existing ROM monitor in place, allowing the setup process to be repeated just in case that should ever prove necessary.

Programming the RedBoot rom monitor into flash memory requires an application that can manage flash blocks. RedBoot itself has this capability. Rather than have a separate application that is used only for flash management during the initial installation, a special RAM-resident version of RedBoot is loaded into memory and run. This version can then be used to load the normal flash-resident version of RedBoot and program it into the flash.

The first step is to connect an RS232 cable between the M5282EVB terminal port and the host PC. A suitable cable is supplied with the board. Next start a terminal emulation application such as HyperTerminal or minicom on the host PC and set the serial communication parameters to 19200 baud, 8 data bits, no parity, 1 stop bit (8N1) and no flow control (handshaking). Make sure that jumper jp16 is set for bootstrap from the bottom of flash, location 0xFFE00000. Apply power to the board and you should see a `dBUG>` prompt.

Once dBUG is up and running the RAM-resident version of RedBoot can be downloaded:

```
dBUG> dl
Escape to local host and send S-records now...
```

The required S-records file is `redboot_dbug.srec`, which is normally supplied with the eCos release in the `loaders` directory. If it needs to be rebuilt then instructions for this are supplied below. The file should be sent to the target as raw text using the terminal emulator:

```
S-record download successful!
dBUG>
```

It is now possible to run the RAM-resident version of RedBoot:

```
dBUG> go 0x10000
+**Warning** FLASH configuration checksum error or invalid key
Use 'fconfig -i' to [re]initialize database
... waiting for BOOTP information
Ethernet eth0: MAC address 00:00:00:00:00:03
Can't get BOOTP info for device!

RedBoot(tm) bootstrap and debug environment [DBUG]
Non-certified release, version UNKNOWN - built 23:36:11, Mar 31 2004

Platform: M5282EVB (Motorola MCF5282)
Copyright (C) 2000, 2001, 2002, Free Software Foundation, Inc.

RAM: 0x00000000-0x01000000, 0x0002f274-0x00fed000 available
FLASH: 0xffe00000 - 0x00000000, 32 blocks of 0x00010000 bytes each.
RedBoot>
```

At this stage, RedBoot flash management initialization has not yet happened so the warning about the configuration checksum error is expected. There will also be a delay while RedBoot tries to contact a local BOOTP server. To perform the flash initialization use the **fis init -f** command:

```
RedBoot> fis init -f
About to initialize [format] FLASH image system - continue (y/n)? y
*** Initialize FLASH Image System
... Erase from 0xfff20000-0xffff0000: ..
... Erase from 0x00000000-0x00000000:
... Erase from 0xffff0000-0xffffffff: .
`... Program from 0x00ff0000-0x01000000 at 0xffff0000: .
RedBoot>
```

At this stage the block of flash at location 0xFFFF0000 holds information about the various flash blocks, allowing other flash management operations to be performed. The next step is to set up RedBoot's non-volatile configuration values:

```
RedBoot> fconfig -i
Initialize non-volatile configuration - continue (y/n)? y
Run script at boot: false
Use BOOTP for network configuration: true
DNS server IP address:
GDB connection port: 9000
Force console for special debug messages: false
Network hardware address [MAC]: 0x00:0x00:0x00:0x00:0x00:0x03
Network debug at boot time: false
Update RedBoot non-volatile configuration - continue (y/n)? y
... Erase from 0xffff0000-0xffffffff: .
... Program from 0x00ff0000-0x01000000 at 0xffff0000: .
RedBoot>
```

For most of these configuration variables the default value is correct. If there is no suitable BOOTP service running on the local network then BOOTP should be disabled, and instead RedBoot will prompt for a fixed IP address, netmask, and addresses for the local gateway and DNS server. The other exception is the network hardware address, also known as the MAC address. All boards should be given a unique MAC address, not the one in the above example. If there are two boards on the same network trying to use the same MAC address then the resulting behaviour is undefined.

It is now possible to load the flash-resident version of RedBoot. Because of the way that flash chips work it is better to first load it into RAM and then program it into flash.

```
RedBoot> load -r -m ymodem -b %{freememlo}
```

The file `redboot_rom.bin` should now be uploaded using the terminal emulator. The file is a raw binary and should be transferred using the Y-modem protocol.

```
Raw file loaded 0x0002f400-0x00045af7, assumed entry at 0x0002f400
```

```
xyzModem - CRC mode, 772(SOH)/0(STX)/0(CAN) packets, 5 retries
RedBoot>
```

Once RedBoot has been loaded into RAM it can be programmed into flash:

```
RedBoot> fis create RedBoot -b %{freememlo}
An image named 'RedBoot' exists - continue (y/n)? y
... Erase from 0xfff00000-0xfff20000: .
... Program from 0x0002f400-0x0004f400 at 0xfff00000: .
... Erase from 0xffff0000-0xffffffff: .
... Program from 0x00ff0000-0x01000000 at 0xffff0000: .
RedBoot>
```

The flash-resident version of RedBoot has now been programmed at location 0xFFF00000, and the flash info block at 0xFFFF0000 has been updated. The initial setup is now complete. Power off the board and set the flash jumper to boot from location 0xFFF00000 instead of 0xFFE00000. Also set the terminal emulator to run at 38400 baud (the usual baud rate for RedBoot), and power up the board again.

```
+Ethernet eth0: MAC address 00:00:00:00:00:03
Can't get BOOTP info for device!

RedBoot(tm) bootstrap and debug environment [ROM]
Non-certified release, version UNKNOWN - built 23:44:10, Mar 31 2004

Platform: M5282EVB (Motorola MCF5282)
Copyright (C) 2000, 2001, 2002, Free Software Foundation, Inc.

RAM: 0x00000000-0x01000000, 0x0000d390-0x00fed000 available
FLASH: 0xffe00000 - 0x00000000, 32 blocks of 0x00010000 bytes each.
RedBoot>
```

When RedBoot issues its prompt it is also ready to accept connections from m68k-elf-gdb, allowing eCos applications to be downloaded and debugged.

Occasionally it may prove necessary to update the installed RedBoot image. This can be done simply by repeating the above process, using dBUG to load the dBUG version of RedBoot `redboot_dbug.srec`. Alternatively the existing RedBoot install can be used to load a RAM-resident version, `redboot_ram.bin`.

The ROMFFE version of RedBoot can be installed at location 0xFFE00000, replacing dBUG. This may be useful if the system needs more flash blocks than are available with the usual ROM RedBoot. Installing this RedBoot image will typically involve a BDM-based utility.

Rebuilding RedBoot

Should it prove necessary to rebuild a RedBoot binary, this is done most conveniently at the command line. The steps needed to rebuild the dBUG version of RedBoot are:

```
$ mkdir redboot_dbug
$ cd redboot_dbug
$ ecosconfig new m5282evb redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/m68k/mcf52xx/mcf5282/m5282evb/current/misc/redboot_DBUG.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

At the end of the build the `install/bin` subdirectory should contain the required file `redboot_dbug.srec`.

Rebuilding the RAM and ROM versions involves basically the same process. The RAM version uses the file `redboot_RAM.ecm` and generates a file `redboot_ram.bin`. The ROM version uses the file `redboot_ROM.ecm` and generates a file `redboot_rom.bin`.

Name

Configuration — Platform-specific Configuration Options

Overview

The M5282EVB platform HAL package is loaded automatically when eCos is configured for an M5282EVB target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The M5282EVB platform HAL package supports four separate startup types, controlled by the configuration option CYG_HAL_STARTUP:

RAM This is the startup type which is normally used during application development. The board has RedBoot programmed into flash at location 0xFFF00000 and boots from that location. m68k-elf-gdb is then used to load a RAM startup application into memory and debug it. It is assumed that the hardware has already been initialized by RedBoot. By default the application will use eCos' virtual vectors mechanism to obtain certain services from RedBoot, including diagnostic output, but that can be disabled via CYGSEM_HAL_USE_ROM_MONITOR.

ROM This startup type can be used for finished applications which boot directly from flash at location 0xFFF00000. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization. The flash-resident version of RedBoot uses this startup type.

ROMFFE This is a variant of the ROM startup type which can be used if the application will be programmed into flash at location 0xFFE00000, overwriting the board's dBUG ROM monitor.

DBUG This is a variant of the RAM startup which allows applications to be loaded via the board's dBUG ROM monitor rather than via RedBoot. It exists mainly to support the dBUG version of RedBoot which is needed during hardware setup. Once the application has started it will take over all the hardware, and it will not depend on any services provided by dBUG. This startup type does not provide gdb debug facilities.

RedBoot and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option CYGSEM_HAL_ROM_MONITOR should be set. Typically this option is set only when building RedBoot.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option CYGSEM_HAL_USE_ROM_MONITOR should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained. That is useful as a testing step before switching to ROM startup. It also allows applications to be run and debugged via BDM.

If the application does not rely on a ROM monitor for diagnostic services then one of the serial ports will be claimed for HAL diagnostics. By default eCos will use the terminal port, corresponding to uart0. The auxiliary port, uart1, can be selected instead via the configuration option CYGHWR_HAL_M68K_M5282EVB_DIAGNOSTICS_PORT. The baud rate for the selected port is controlled by CYGNUM_HAL_M68K_M5282EVB_DIAG_BAUD.

Flash Driver

The platform HAL package contains flash driver support for the external flash. By default this is inactive, and it can be made active by loading the generic flash package CYGPKG_IO_FLASH.

Special Registers

The MCF5282 processor has a number of special registers controlling the cache, on-chip RAM and flash, and so on. The platform HAL provides a number of configuration options for setting these, for example `CYGNUM_HAL_M68K_M5282EVB_RAMBAR` controls the initial value of the RAMBAR register.

System Clock

By default the system clock interrupts once every 10ms, corresponding to a 100Hz clock. This can be changed by the configuration option `CYGNUM_HAL_RTC_PERIOD`, the number of microseconds between clock ticks. Other clock-related settings are recalculated automatically if the period is changed.

Compiler Flags

The platform HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos. There are three flags specific to this port:

`-m5200`
The m68k-elf-gcc compiler supports many variants of the M68K architecture, from the original 68000 onwards. A `-m` option should be used to select the specific variant in use, and with current tools `-m5200` is the closest match for an MCF5282 processor.

`-malign-int`
This option forces m68k-elf-gcc to align integer and floating point data to a 32-bit boundary rather than a 16-bit boundary. It should improve performance. However the resulting code is incompatible with most published application binary interface specifications for M68K processors, so it is possible that this option causes problems with existing third-party object code.

`-fomit-frame-pointer`
Traditionally the `%A6` register was used as a dedicated frame pointer, and the compiler was expected to generate link and unlink instructions on procedure entry and exit. These days the compiler is perfectly capable of generating working code without a frame pointer, so omitting the frame pointer often saves some work during procedure entry and exit and makes another register available for optimization. However, without a frame pointer register, the m68k-elf-gdb debugger is not always able to interpret a thread stack, so it cannot reliably give a backtrace. Removing `-fomit-frame-pointer` from the default flags will make debugging easier, but the generated code may be worse.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the M5282EVB hardware, and should be read in conjunction with that specification. The M5282EVB platform HAL package complements the M68K architectural HAL, the MCFxxxx variant HAL, and the MCF5282 processor HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset the HAL will initialize or reinitialize most of the on-chip peripherals. There is an exception for RAM startup applications which depend on a ROM monitor for certain services: the UARTs and the ethernet device will not be reinitialized because they may be in use by RedBoot for communication with the host. Full details of this initialization can be found in the function `hal_m68k_m5282evb_init` in `platform.c`.

For a ROM or ROMFFE startup the HAL will perform additional initialization, setting up the external DRAM and programming the various internal registers. The values used for some of these registers are configurable. Full details can be found in the exported header `cyg/hal/plf.inc`.

Linker Scripts and Memory Maps

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

External SDRAM

This is mapped to location 0x00000000. Most of the first kilobyte is used for hardware exception vectors. The eCos virtual vectors are also placed here, allowing RAM-based applications to use services provided by the ROM monitor. For ROM and ROMFFE startup all remaining SDRAM is available. For RAM and DBUG startup available SDRAM starts at location 0x00010000, with the bottom 64K reserved for use by either the RedBoot or dBUG ROM monitors.

Internal RAM

The 64K of internal RAM are normally mapped at location 0x20000000. Neither eCos nor RedBoot use the internal RAM so all of it is available to application code.

On-chip Peripherals

These are accessible at location 0x40000000 onwards, as per the defined symbol `HAL_MCF5282_ISPBAR`.

Note

On some other coldfire processors the equivalent register is known as `%mbar`. The symbol `HAL_MCFxxxx_MBAR` is an alias for `HAL_MCF5282_ISPBAR`, making it easier to share device drivers.

On-chip Flash

The 512K of internal flash are normally mapped at location 0xF0000000. Currently this is not used by eCos or RedBoot.

External Flash

This is located at the top of memory, location 0xFFE00000 onwards. For ROM startup it is assumed that a jumper is used to disable the bottom half of the flash, so location 0xFFE00000 is actually a mirror of 0xFFF00000. For ROMFFE and DBUG startups all of the flash is visible. RAM startup will work irrespective of the jumper setting.

In a typical setup the first two 64K flash blocks at location 0xFFF00000 are used to hold RedBoot, and the block at location 0xFFFF0000 is used to hold flash management data and the RedBoot **fconfig** variables. The remaining blocks can be used by application code.

Clock Support

The platform HAL provides configuration options for the eCos system clock. This always uses the hardware timer PIT3, which should not be used directly by application code. The actual HAL macros for managing the clock are provided by the MCF5282 processor HAL. The specific numbers used are a characteristic of the platform because they depend on the processor speed. The gprof-based profiling code uses PIT2. Timers PIT0 and PIT1 are not used by eCos so application code is free to manipulate these as required.

Other Issues

The M5282EVB platform HAL does not affect the implementation of other parts of the eCos HAL specification. The MCF5282 processor HAL, the MCFxxxx variant HAL, and the M68K architectural HAL documentation should be consulted for further details.

Other Functionality

The platform HAL package also provides a flash driver for the off-chip AMD AM29LV160 flash chip. This driver is inactive by default, and only becomes active if the configuration includes the generic flash support `CYGPKG_IO_FLASH`.

The platform HAL provides one additional function to manipulate the on-board LEDs: `void hal_m5282evb_led_set(which, what)`. The `which` argument specifies the LED and should be either 0 or 1. The `what` argument should be non-zero to switch the LED on, zero to switch it off. Note that only two of the four user LEDs can be manipulated in this way: the other two are normally connected to signals needed by one of the uarts.

Chapter 324. Freescale M5282LITE Board Support

Name

eCos Support for the Freescale M5282LITE Board — Overview

Description

The Freescale M5282LITE board has an MCF5282 ColdFire processor, 16MB of external SDRAM, 2MB of external flash memory, and connectors plus required support chips for the on-chip peripherals. By default the board comes with its own dBUG ROM monitor located in the external flash.

For typical eCos development a RedBoot image is programmed into the external flash replacing the existing dBUG monitor. RedBoot provides gdb stub functionality so it is then possible to download and debug eCos applications via the gdb debugger. This can happen over either a serial line or over ethernet.

Supported Hardware

The memory map used by both eCos and RedBoot is as follows:

Memory	**Base**	**Length**
External SDRAM	0x00000000	0x01000000
Internal RAM	0x20000000	0x00010000
On-chip Peripherals	0x40000000	0x40000000
On-chip Flash	0xF0000000	0x00080000
External Flash	0xFFE00000	0x00200000

eCos can be configured for one of three startup types:

RAM
: This is the startup type normally used during application development. RedBoot is programmed into flash and performs the initial bootstrap. m68k-elf-gdb is then used to load a RAM startup application into memory and debug it. By default the application will use eCos' virtual vectors mechanism to obtain certain services from RedBoot, including diagnostic output.

With a minor change to the eCos configuration this startup type can also be used to debug applications via BDM.

DBUG
: This is a variant of the RAM startup which allows applications to be loaded via the board's dBUG ROM monitor rather than via RedBoot. Once the application has started it will take over all the hardware, and it will not depend on any services provided by dBUG. This startup type does not provide gdb debug facilities. It is used primarily for building a special version of RedBoot, used during hardware setup.

ROM
: This startup type can be used for finished applications which will be programmed into flash at location 0xFFE00000. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization. This startup type is used for building the flash-resident version of RedBoot but can also be used for application code.

For all startup types the external SDRAM is used to hold all data. For RAM and dBUG startup the code also resides in external SDRAM, with the first 64K reserved for use by the ROM monitor. The 64K of internal RAM is not used, either by RedBoot or by eCos, so all of it is available to the application. The 512K of internal flash is not currently used.

In a typical setup the first two 64K flash blocks are used for holding the RedBoot image, and the last block is used for managing the flash and holding the RedBoot fconfig values. The remaining 29 blocks from 0xFFE20000 to 0xFFFEFFFF can be used by application code.

RedBoot can communicate with the host using either ethernet or one of the UARTs - usually uart0 corresponding to the existing serial connector on the M5282LITE board.

All configurations for the M5282LITE target include an ethernet driver package CYGPKG_DEVS_ETH_MCFxxxx. If the application does not actually require ethernet functionality then the package is inactive and the final executable will not suffer any overheads from unused functionality. This is determined by the presence of the generic ethernet I/O package CYGPKG_IO_ETH_DRIVERS. Typically the choice of eCos template causes the right thing to happen. For example the default template does not include any TCP/IP stack so CYGPKG_IO_ETH_DRIVERS is not included, but both the net and redboot templates do include a TCP/IP stack so will specify that package and hence enable the ethernet driver. The ethernet device can be shared by RedBoot and the application, so it is possible to debug a networked application over ethernet.

The M5282LITE board does not have a serial EPROM or similar hardware providing a unique network MAC address. Instead a suitable address has to be programmed into flash via RedBoot's **fconfig** command.

All configurations for the M5282LITE target include a serial device driver package CYGPKG_DEVS_SERIAL_MCFxxxx. The driver as a whole is inactive unless the generic serial support, CYGPKG_IO_SERIAL_DEVICES is enabled. Exactly which of the on-chip UARTs are supported is controlled by configuration options within the platform HAL. By default only uart0 is supported since on the standard board that is the only one with a connector. If the UART is needed by the application then it cannot also be used by RedBoot for gdb traffic, so another communication channel such as ethernet should be used instead.

All configurations for the M5282LITE target also include a watchdog device driver CYGPKG_DEVS_WATCHDOG_MCF5282. This driver is inactive unless the generic watchdog support CYGPKG_IO_WATCHDOG is loaded.

The on-chip interrupt controllers and the edge port module are managed by eCos using macros provided by the MCF5282 processor HAL. PIT timer 3 is normally used to implement the eCos system clock. If gprof-based profiling is enabled then that will use PIT timer 2. PIT timers 0 and 1 are unused and can be manipulated by the application. The GPIO pins are manipulated only as needed to get the UART(s) and ethernet working. eCos will reset the remaining on-chip peripherals (DMA, GPT, DMA timers, QSPI, I²C, FlexCAN, and QADC) during system startup or soft reset, but will not otherwise manipulate them.

Tools

The M5282LITE port is intended to work with GNU tools configured for an m68k-elf target. The original port was done using m68k-elf-gcc version 3.2.1, m68k-elf-gdb version 5.3, and binutils version 2.13.1.

By default eCos is built using the compiler flag -fomit-frame-pointer. Omitting the frame pointer eliminates some work on every function call and makes another register available, so the code should be smaller and faster. However without a frame pointer m68k-elf-gdb is not always able to identify stack frames, so it may be unable to provide accurate backtrace information. Removing this compiler flag from the configuration option CYGBLD_GLOBAL_CFLAGS avoids such debug problems.

A typical setup involves m68k-elf-gdb interacting with RedBoot using either serial or ethernet. Alternatively it is possible to debug via the BDM port. The package's misc subdirectory contains a script bdm.gdb that contains macros for the low-level hardware initialization normally performed by the ROM startup code. The application should be linked with an eCos configuration using RAM startup, and with the options CYGSEM_HAL_ROM_MONITOR and CYGSEM_HAL_USE_ROM_MONITOR disabled to stop eCos accessing any services provided by RedBoot. Diagnostic output will be sent out of uart0.

Name

Setup — Preparing the M5282LITE board for eCos Development

Overview

In a typical development environment the M5282LITE board boots from flash into the RedBoot ROM monitor. eCos applications are configured for a RAM startup, and then downloaded and run on the board via the debugger m68k-elf-gdb. Preparing the board therefore involves programming a suitable RedBoot image into flash memory, replacing the existing dBUG monitor.

The following RedBoot configurations are supported:

Configuration	Description	Use	File
ROM	RedBoot running from the board's flash	`redboot_ROM.ecm`	`redboot_rom.bin`
dBUG	Used for initial setup	`redboot_DBUG.ecm`	`redboot_dbug.srec`
RAM	Used for upgrading ROM version	`redboot_RAM.ecm`	`redboot_ram.bin`

For serial communications all versions run with 8 bits, no parity, and 1 stop bit. The dBUG version runs at 19200 baud. The ROM and RAM versions run at 38400 baud. These baud rates can be changed via the configuration option `CYGNUM_HAL_M68K_M5282LITE_DIAG_BAUD` and rebuilding RedBoot. On the standard board only UART0 has a serial connector so by default RedBoot will use that. If the board has been extended to provide connectors for the other on-chip uarts then the configuration option `CYGHWR_HAL_M68K_MCFxxxx_DIAGNOSTICS_PORT` can be used to make RedBoot use one of thoe. On an M5282LITE platform RedBoot also supports ethernet communication and flash management.

Initial Installation

This process assumes that the board still has its original dBUG ROM monitor and does not require any special debug hardware. Programming the RedBoot rom monitor into flash memory requires an application that can manage flash blocks. RedBoot itself has this capability. Rather than have a separate application that is used only for flash management during the initial installation, a special RAM-resident version of RedBoot is loaded into memory and run. This version can then be used to load the normal flash-resident version of RedBoot and program it into the flash.

The first step is to connect an RS232 cable between the M5282LITE serial port and the host PC. A suitable cable is supplied with the board. Next start a terminal emulation application such as HyperTerminal or minicom on the host PC and set the serial communication parameters to 19200 baud, 8 data bits, no parity, 1 stop bit (8N1) and no flow control (handshaking). Apply power to the board and you should see a `dBUG>` prompt.

Once dBUG is up and running the RAM-resident version of RedBoot can be downloaded:

```
dBUG> dl
Escape to local host and send S-records now...
```

The required S-records file is `redboot_dbug.srec`, which is normally supplied with the eCos release in the `loaders` directory. If it needs to be rebuilt then instructions for this are supplied below. The file should be sent to the target as raw text using the terminal emulator:

```
S-record download successful!
dBUG>
```

It is now possible to run the RAM-resident version of RedBoot:

```
dBUG> go 0x10000
FLASH configuration checksum error or invalid key
```

```
... waiting for BOOTP information
Ethernet eth0: MAC address 00:00:00:00:00:03
Can't get BOOTP info for device!

RedBoot(tm) bootstrap and debug environment [DBUG]
Non-certified release, version UNKNOWN - built 11:26:11, Jul 24 2004

Platform: M5282LITE (Motorola MCF5282)
Copyright (C) 2000, 2001, 2002, Free Software Foundation, Inc.
Copyright (C) 2003, 2004, eCosCentric Limited

RAM: 0x00000000-0x01000000, 0x0002d0f0-0x00fed000 available
FLASH: 0xffe00000 - 0x00000000, 32 blocks of 0x00010000 bytes each.
RedBoot>
```

At this stage the RedBoot flash management initialization has not yet happened so the warning about the configuration checksum error is expected. There will also be a delay while RedBoot tries to contact a local BOOTP server. To perform the flash initialization use the **fis init -f** command:

```
RedBoot> fis init -f
About to initialize [format] FLASH image system - continue (y/n)? y
*** Initialize FLASH Image System
... Erase from 0xffff0000-0xffffffff: .
`... Program from 0x00ffef00-0x00fff000 at 0xffff0000: .
RedBoot>
```

At this stage the block of flash at location 0xFFFF0000 holds information about the various flash blocks, allowing other flash management operations to be performed. The next step is to set up RedBoot's non-volatile configuration values:

```
RedBoot> fconfig -i
Initialize non-volatile configuration - continue (y/n)? y
Run script at boot: false
Use BOOTP for network configuration: true
DNS server IP address:
GDB connection port: 9000
Force console for special debug messages: false
Network hardware address [MAC]: 0x00:0xff:0x12:0x34:0x01:0x08
Network debug at boot time: false
Update RedBoot non-volatile configuration - continue (y/n)? y
... Erase from 0xffff0000-0xffffffff: .
... Program from 0x00fef000-0x00fff000 at 0xffff0000: .
RedBoot>
```

For most of these configuration variables the default value is correct. If there is no suitable BOOTP service running on the local network then BOOTP should be disabled, and instead RedBoot will prompt for a fixed IP address, netmask, and addresses for the local gateway and DNS server. The other exception is the network hardware address, also known as MAC address. All boards should be given a unique MAC address, not the one in the above example. If there are two boards on the same network trying to use the same MAC address then the resulting behaviour is undefined.

It is now possible to load the flash-resident version of RedBoot. Because of the way that flash chips work it is better to first load it into RAM and then program it into flash.

```
RedBoot> load -r -m ymodem -b %{freememlo}
```

The file `redboot_rom.bin` should now be uploaded using the terminal emulator. The file is a raw binary and should be transferred using the Y-modem protocol.

```
Raw file loaded 0x0002d400-0x00042c2b, assumed entry at 0x0002d400
xyzModem - CRC mode, 691(SOH)/0(STX)/0(CAN) packets, 5 retries
RedBoot>
```

Once RedBoot has been loaded into RAM it can be programmed into flash:

```
RedBoot> fis create RedBoot -b %{freememlo}
```

```
An image named 'RedBoot' exists - continue (y/n)? y
... Erase from 0xffe00000-0xffe20000: .
... Program from 0x0002d400-0x0004d400 at 0xffe00000: .
... Erase from 0xffff0000-0xffffffff: .
... Program from 0x00fef000-0x00fff000 at 0xffff0000: .
RedBoot>
```

The flash-resident version of RedBoot has now programmed at location 0xFFE00000, and the flash info block at 0xFFFF0000 has been updated. The initial setup is now complete. Power off the board, set the terminal emulator to run at 38400 baud (the usual baud rate for RedBoot), and power up the board again.

```
+Ethernet eth0: MAC address 00:ff:12:34:01:08
IP: 10.1.1.71/255.255.255.0, Gateway: 10.1.1.241
Default server: 10.1.1.1, DNS server IP: 10.1.1.240


RedBoot(tm) bootstrap and debug environment [ROM]
Non-certified release, version UNKNOWN - built 15:41:51, Jul 24 2004

Platform: M5282LITE (Motorola MCF5282)
Copyright (C) 2000, 2001, 2002, Free Software Foundation, Inc.
Copyright (C) 2003, 2004, eCosCentric Limited

RAM: 0x00000000-0x01000000, 0x0000bef8-0x00fed000 available
FLASH: 0xffe00000 - 0x00000000, 32 blocks of 0x00010000 bytes each.
RedBoot>
```

When RedBoot issues its prompt it is also ready to accept connections from m68k-elf-gdb, allowing eCos applications to be downloaded and debugged.

Occasionally it may prove necessary to update the installed RedBoot image. This can be done by loading a RAM-resident version of RedBoot, `redboot_ram.bin`, rather than the dBUG version of RedBoot used above. The ROM version can then be loaded into memory using RedBoot's **load** and the flash version version can be updated using **fis create RedBoot**.

Rebuilding RedBoot

Should it prove necessary to rebuild a RedBoot binary, this is done most conveniently at the command line. The steps needed to rebuild the dBUG version of RedBoot are:

```
$ mkdir redboot_dbug
$ cd redboot_dbug
$ ecosconfig new m5282lite redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/m68k/mcf52xx/mcf5282/m5282lite/current/misc/redboot_DBUG.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

At the end of the build the `install/bin` subdirectory should contain the required file `redboot_dbug.srec`.

Rebuilding the RAM and ROM versions involves basically the same process. The RAM version uses the file `redboot_RAM.ecm` and generates a file `redboot_ram.bin`. The ROM version uses the file `redboot_ROM.ecm` and generates a file `redboot_rom.bin`.

Name

Configuration — Platform-specific Configuration Options

Overview

The M5282LITE platform HAL package is loaded automatically when eCos is configured for an M5282LITE target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The M5282LITE platform HAL package supports three separate startup types: RAM, DBUG and ROM. The configuration option `CYG_HAL_STARTUP`: controls which startup type is being used. For typical application development RAM startup should be used, and the application will be run via m68k-elf-gdb interacting with RedBoot using either serial or ethernet. It is assumed that the low-level hardware initialization, including setting up the memory map, has already been performed by RedBoot. By default the application will use certain services provided by RedBoot via the virtual vector mechanism, including diagnostic output, but that can be disabled via `CYGSEM_HAL_USE_ROM_MONITOR`.

ROM startup can be used for applications which boot directly from flash. All the hardware will be initialized, and the application is self-contained. This startup type is used by the flash-resident version of RedBoot, and can also be used for finished applications.

DBUG startup can be used for applications which will be loaded via the DBUG ROM monitor rather than RedBoot. As with RAM startup it is assumed that the memory map has already been set up, but the application will not use any services provided by the ROM monitor.

RedBoot and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building RedBoot.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained. That is useful as a testing step before switching to ROM startup. It also allows applications to be run and debugged via BDM.

If the application does not rely on a ROM monitor for diagnostic services then one of the serial ports will be claimed for HAL diagnostics. By default eCos will use UART0 since on the standard board that is the only uart with a suitable connector. If the board has been extended with additional transceiver chips and connectors for UART1 or UART2 then one of those can be selected via the `CYGHWR_HAL_M68K_MCFxxxx_DIAGNOSTICS_PORT`. The baud rate for the selected port is controlled by `CYGNUM_HAL_M68K_M5282LITE_DIAG_BAUD`.

Optional Hardware

The M5282LITE board can be customized in a number of ways, primarily by connecting additional hardware to the MCU port. There are a number of configuration options which allow the platform HAL to adapt to minor changes to the hardware:

`CYGHWR_HAL_M68K_M5282LITE_UART0_RTS`

> On the default hardware UART0 RTS and CTS are not connected. If the uart will only be used for interacting with RedBoot and debugging, that should be fine. If the uart will be used for another purpose, for example PPP, then it will usually be desirable to support RTS/CTS hardware handshaking. The board has an option pad to allow the serial connector's RTS pin to be wired to the processor's DTOUT3 pin. Alternatively any of the DTIN3, DTOUT1 or DTIN1 signals on the MCU port could be wired instead. This configuration option can be used to specify how the board has been wired. Note that CTS would have to be wired as well as RTS.

CYGHWR_HAL_M68K_M5282LITE_UART0_CTS

As with RTS, CTS is not wired on the default board. Although there is an option pad this is not usable: the option pad would wire the serial connector's CTS pin to the processor's DTIN3 pin, but DTIN3 cannot be configured to carry the uart CTS signal. Instead one of the DTOUT2, DTIN2, DTOUT0, or DTIN0 signals on the MCU port should be used.

CYGHWR_HAL_M68K_M5282LITE_UART1_CONNECTED
CYGHWR_HAL_M68K_M5282LITE_UART1_RTS
CYGHWR_HAL_M68K_M5282LITE_UART1_RTS_RS485
CYGHWR_HAL_M68K_M5282LITE_UART1_CTS

The default board has no connector for the on-chip UART1. However all the signals are accessible on the MCU port so it is possible to wire up a suitable transceiver chip and connector. Enabling CYGHWR_HAL_M68K_M5282LITE_UART1_CONNECTED specifies that the RX and TX signals are connected. RTS is optional and can come from any of DTOUT3, DTIN3, DTOUT1 or DTIN1, although obviously the same signal cannot be used for both UART0 and UART1. RTS may be used either to tristate an RS485 transceiver in which case CTS should be left disconnected, or it can be used for RS232 hardware handshaking in which case CTS must also be connected. The CTS signal can come from any of DTOUT2, DTIN2, DTOUT0 or DTIN0.

CYGHWR_HAL_M68K_M5282LITE_UART2_CONNECTED

The on-chip UART2 is not connected on the standard board. However if this uart is needed then it can be accessed via the MCU port, using either the A1A0 SCL/SDA signals normally used for I²C, or the A3A2 CANTX/CANRX signals normally used for CAN communication. This UART does not support RTS or CTS.

Flash Driver

The platform HAL package contains flash driver support for the external flash device. By default this is inactive, and it can be made active by loading the generic flash package CYGPKG_IO_FLASH.

Special Registers

The MCF5282 processor has a number of special registers controlling the cache, on-chip RAM and flash, and so on. The platform HAL provides a number of configuration options for setting these, for example CYGNUM_HAL_M68K_M5282LITE_RAMBAR controls the initial value of the RAMBAR register.

System Clock

By default the system clock interrupts once every 10ms, corresponding to a 100Hz clock. This can be changed by the configuration option CYGNUM_HAL_RTC_PERIOD, the number of microseconds between clock ticks. Other clock-related settings are recalculated automatically if the period is changed.

Compiler Flags

The platform HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos. There are three flags specific to this port:

-m5200

The m68k-elf-gcc compiler supports many variants of the M68K architecture, from the original 68000 onwards. A -m option should be used to select the specific variant in use, and with current tools -m5200 is the closest match for an MCF5282 processor.

-malign-int

This option forces m68k-elf-gcc to align integer and floating point data to a 32-bit boundary rather than a 16-bit boundary. It should improve performance. However the resulting code is incompatible with most published application binary interface specifications for M68K processors, so it is possible that this option causes problems with existing third-party object code.

`-fomit-frame-pointer`
: Traditionally the `%A6` register was used as a dedicated frame pointer, and the compiler was expected to generate link and unlink instructions on procedure entry and exit. These days the compiler is perfectly capable of generating working code without a frame pointer, so omitting the frame pointer often saves some work during procedure entry and exit and makes another register available for optimization. However without a frame pointer register the m68k-elf-gdb debugger is not always able to interpret a thread stack, so it cannot reliably give a backtrace. Removing `-fomit-frame-pointer` from the default flags will make debugging easier, but the generated code may be worse.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the M5282LITE hardware, and should be read in conjunction with that specification. The M5282LITE platform HAL package complements the M68K architectural HAL, the MCFxxxx variant HAL, and the MCF5282 processor HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset the HAL will initialize or reinitialize most of the on-chip peripherals. There is an exception for RAM startup applications which depend on a ROM monitor for certain services: the UARTs and the ethernet device will not be reinitialized because they may be in use by RedBoot for communication with the host. Full details of this initialization can be found in the function `hal_m68k_m5282lite_init` in `platform.c`.

For a ROM startup the HAL will perform additional initialization, setting up the external DRAM and chip selects. Full details can be found in the exported header `cyg/hal/plf.inc`.

Linker Scripts and Memory Maps

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

External SDRAM

This is mapped to location 0x00000000. Most of the first kilobyte is used for hardware exception vectors. The eCos virtual vectors are also placed here, allowing RAM-based applications to use services provided by the ROM monitor. For ROM startup all remaining SDRAM is available. For RAM and DBUG startup available SDRAM starts at location 0x00010000, with the bottom 64K reserved for use by either the RedBoot or dBUG ROM monitors.

Internal RAM

The 64K of internal RAM are normally mapped at location 0x20000000. Neither eCos nor RedBoot use the internal RAM so all of it is available to application code.

On-chip Peripherals

These are accessible at location 0x40000000 onwards, as per the defined symbol `HAL_MCF5282_ISPBAR`.

Note

On some other coldfire processors the equivalent register is known as `%mbar`. The symbol `HAL_MCFxxxx_MBAR` is an alias for `HAL_MCF5282_ISPBAR`, making it easier to share device drivers.

On-chip Flash

The 512K of internal flash are normally mapped at location 0xF0000000. Currently this is not used by eCos or RedBoot.

External Flash

This is located at the top of memory, location 0xFFE00000 onwards. In a typical setup the first two 64K flash blocks at location 0xFFE00000 are used to hold RedBoot, and the last flash block at location 0xFFFF0000 is used to hold flash management data and the RedBoot **fconfig** variables. The remaining blocks can be used by application code.

Clock Support

The platform HAL provides configuration options for the eCos system clock. This always uses the hardware timer PIT3, which should not be used directly by application code. The actual HAL macros for managing the clock are provided by the MCF5282 processor HAL. The specific numbers used are a characteristic of the platform because they depend on the processor speed. The gprof-based profiling code uses PIT2. Timers PIT0 and PIT1 are not used by eCos so application code is free to manipulate these as required.

Other Issues

The M5282LITE platform HAL does not affect the implementation of other parts of the eCos HAL specification. The MCF5282 processor HAL, the MCFxxxx variant HAL, and the M68K architectural HAL documentation should be consulted for further details.

Other Functionality

The platform HAL package also provides a flash driver for the off-chip ST M29W160EB flash chip or compatible. This driver is inactive by default, and only becomes active if the configuration includes the generic flash support `CYGPKG_IO_FLASH`.

Chapter 325. SSV DNP/5280 Board Support

Name

eCos Support for the SSV DNP/5280 and DNP/5282 Modules — Overview

Description

The SSV DNP/5280 module has an MCF5280 ColdFire processor, 16MB of external SDRAM, 8MB of external flash memory, an ethernet phy chip, and in later revisions a DS1306 real-time clock. The module needs to be plugged into a suitable carrier board, typically an SSV DNP/EVA2_SV6 or a DNP/EVA6, but custom boards may be used. The carrier board provides power as well as connectors for the ethernet and some of the on-chip devices. The DNP/5282 module is similar but has a smaller footprint. The modules are supplied with some firmware already programmed into the external flash. This firmware can be either RedBoot, a ROM monitor based on eCos, or dBUG.

This package `CYGPKG_HAL_M68K_DNP5280` provides a port to both modules. Specifically it supports the following targets:

`dnp5280`
: A DNP/5280 module plugged into an EVA2_SV6 carrier board.

`dnp5280_v12`
: A V1.2 DNP/5280 plugged into an EVA2_SV6 carrier board. This adds support for the DS1306 clock device on this revision of the module.

`dnp5282`
: A DNP/5282 plugged into an EVA6 carrier board.

eCos configuration options can be used to change the settings. `CYGHWR_HAL_M68K_MCF528x_HARDWARE_DNP528x_BOARD` determines the carrier board. This in turn affects the default GPIO pin settings and hence which of the on-chip peripherals are accessible. These pin settings are also controlled by configuration options. Hence custom carrier boards can be supported by creating an initial configuration for one of the standard targets and then changing the pin settings to reflect the I/O capabilities of the actual carrier board.

Typical eCos development involves using RedBoot as the board's firmware, replacing dBUG if necessary. RedBoot provides gdb stub functionality so it is then possible to download and debug eCos applications via the gdb debugger. This can happen over either a serial line or over ethernet.

The eCos port can be configured for one of three startup types:

RAM
: This is the startup type normally used during application development. RedBoot is programmed into flash and performs the initial bootstrap. m68k-elf-gdb is then used to load a RAM startup application into memory and debug it. By default the application will use eCos' virtual vectors mechanism to obtain certain services from RedBoot, including diagnostic output. The RAM startup type can also be used for finished applications: RedBoot can be made to load and run such applications automatically following bootstrap.

  With a minor change to the eCos configuration this startup type can also be used to debug applications via BDM. eCos will no longer assume the presence of RedBoot and hence will not make any virtual vector calls to obtain RedBoot services.

DBUG
: This is another variant of RAM startup, used only when initializing a board. It can be used to run a special RAM-resident version of RedBoot on top of the dBUG ROM monitor, allowing a ROM startup version of RedBoot to be programmed into flash.

ROM
: This should be used for applications which will boot directly from flash at location 0xFF800000, replacing any ROM monitor. The application will be self-contained. eCos startup code will perform all necessary hardware initialization. ROM startup is used for building the flash-resident version of RedBoot, but can also be used for finished applications.

Supported Hardware

The memory map used by both eCos and RedBoot is as follows:

Memory	Base	Length
External SDRAM	0x00000000	0x01000000
Internal RAM	0x20000000	0x00010000
On-chip Peripherals	0x40000000	0x40000000
External Flash	0xFF800000	0x00800000

DNP/5282 modules also have on-chip flash at 0xF0000000. For all startup types the external SDRAM is used to hold all data. For RAM and DBUG startup the code also resides in external SDRAM, with the first 64K reserved for use by the ROM monitor. In a typical setup RedBoot will occupy the first two external flash blocks, and it will also use the last flash block for storing **fconfig** run-time configuration settings and the **fis** directory.

Code and data can be placed in the internal RAM using the linker script section ".iram_text" for code, and ".iram_data" and ".iram_bss" for initialized and unitialized data respectively. The M68K architectural HAL contains a testcase `iram1.c` which demonstrates how to use these linker sections.

RedBoot can communicate with the host using either ethernet or one of the UARTs - usually uart0 because the DNP/EVA2_SV6 carrier board only provides a connector for that board.

All configurations for the DNP/5280 and DNP/5282 targets include an ethernet driver package `CYGPKG_DEVS_ETH_MCFxxxx`. If the application does not actually require ethernet functionality then the package is inactive and the final executable will not suffer any overheads from unused functionality. This is determined by the presence of the generic ethernet I/O package `CYGPKG_IO_ETH_DRIVERS`. Typically the choice of eCos template causes the right thing to happen. For example the default template does not include any TCP/IP stack so `CYGPKG_IO_ETH_DRIVERS` is not included, but both the net and redboot templates do include a TCP/IP stack so will specify that package and hence enable the ethernet driver. The ethernet device can be shared by RedBoot and the application, so it is possible to debug a networked application over ethernet.

The DNP/5280 board does not have a serial EPROM or similar hardware providing a unique network MAC address. Instead a suitable address has to be programmed into flash via RedBoot's **fconfig** command.

The on-chip uarts are supported via the serial device driver package `CYGPKG_DEVS_SERIAL_MCFxxxx`. The driver as a whole is inactive unless the generic serial support, `CYGPKG_IO_SERIAL_DEVICES` is enabled. Only those uarts for which GPIO pins are configured appropriately are available. By default this is only uart0 when using an EVA2_SV6 carrier board, and both uart0 and uart1 when using an EVA6 carrier board. One of the uarts, typically uart0, may also be used for HAL diagnostics. If so it should not be accessed via the serial driver.

All configurations for the DNP/5280 target also include a watchdog device driver `CYGPKG_DEVS_WATCHDOG_MCF5282`. This driver is inactive unless the generic watchdog support `CYGPKG_IO_WATCHDOG` is loaded.

An I²C bus device driver is also included, but will not be built by default. If the DNP/5280 is plugged into a DNP/EVA2-SV6 carrier board the relevant pins are already in use, connected to the dip switch. Even if a different carrier board is being used the same pins may still be needed for GPIO or for uart2. If the I²C bus can be safely enabled on the target hardware then GPIO pin configuration options should be adjusted to connect the SCL and SDA signals. The MCF5282 processor HAL will then instantiate a cyg_i2c_bus structure `hal_mcfxxxx_i2c_bus`, allowing appropriate `CYG_I2C_DEVICE` structures to be defined.

Similarly an SPI bus device driver is included but will not be built by default. If the GPIO pins are adjusted to connect the QSPI DOUT, DIN and CLK signals then the MCF5282 processor HAL will instantiate an SPI bus device `hal_mcfxxxx_qspi_bus`, allowing appropriate `CYG_MCFxxxx_QSPI_DEVICE` structures to be defined.

Some releases may come with a driver `CYGPKG_DEVS_CAN_FLEXCAN` for the on-chip CAN device. This will be inactive unless the generic CAN support `CYGPKG_IO_CAN` is added to the configuration. In addition it will be necessary to adjust the GPIO pins to connect the CANTX and CANRX signals.

When configured for a `dnp5280_v12` target the configuration will include a wallclock driver `CYGPKG_DEVICES_WALLCLOCK_DALLAS_DS1306`. This will be used automatically by the C library's time-related functions, for example `time` and `asctime`, and can be changed by an eCos-specific function `cyg_libc_time_settime`.

All configurations will include a flash device driver `CYGPKG_DEVS_FLASH_AMD_AM29XXXXX_V2` for the external flash. In addition when configured for a `dnp5282` target the configuration will include a flash driver `CYGPKG_DEVS_FLASH_M68K_MCFxxxx_CFM`. Both drivers will be inactive unless the generic flash support `CYGPKG_IO_FLASH` is added to the configuration.

On an EVA2_SV6 carrier board the platform HAL provides utility routines for accessing the LEDs and dip switch. The on-chip interrupt controllers and the edge port module are managed by eCos using macros provided by the MCF5282 processor HAL. PIT timer 3 is normally used to implement the eCos system clock. If gprof-based profiling is enabled then that will use PIT timer 2. PIT timers 0 and 1 are unused and can be manipulated by the application. The remaining on-chip peripherals are not used by eCos.

Tools

The DNP/5280 port is intended to work with GNU tools configured for an m68k-elf target. The original port was done using m68k-elf-gcc version 3.2.1, m68k-elf-gdb version 5.3, and binutils version 2.13.1.

By default eCos is built using the compiler flag `-fomit-frame-pointer`. Omitting the frame pointer eliminates some work on every function call and makes another register available, so the code should be smaller and faster. However without a frame pointer m68k-elf-gdb is not always able to identify stack frames, so it may be unable to provide accurate backtrace information. Removing this compiler flag from the configuration option `CYGBLD_GLOBAL_CFLAGS` avoids such debug problems.

A typical setup involves m68k-elf-gdb interacting with RedBoot using either serial or ethernet. Alternatively it is possible to debug via the BDM port. The package's `misc` subdirectory contains a script `bdm.gdb` that contains macros for the low-level hardware initialization normally performed by the ROM startup code. The application should be linked with an eCos configuration using RAM startup, and with the options `CYGSEM_HAL_ROM_MONITOR` and `CYGSEM_HAL_USE_ROM_MONITOR` disabled to stop eCos accessing any services provided by RedBoot. Diagnostic output will be sent out of uart0.

Name

Setup — Preparing the DNP/5280 board for eCos Development

Overview

In a typical development environment the DNP/5280 board boots from flash into the RedBoot ROM monitor. eCos applications are configured for a RAM startup, and then downloaded and run on the board via the debugger m68k-elf-gdb. Boards may be shipped with one of two ROM monitors in the flash, either RedBoot or dBUG. If the latter, dBUG must be replaced so preparing the board involves programming a suitable RedBoot image into flash memory.

The following RedBoot configurations are supported:

Configuration	Description	Use	File
ROM	RedBoot running from the board's flash	`redboot_ROM.ecm`	`redboot_rom.bin`
dBUG	Used for initial setup	`redboot_DBUG.ecm`	`redboot_dbug.srec`
RAM	Used for upgrading ROM version	`redboot_RAM.ecm`	`redboot_ram.bin`

For serial communications all versions run with 8 bits, no parity, and 1 stop bit. The dBUG version runs at 115200 baud. The ROM and RAM versions usually run at 38400 baud, but this can be changed via a RedBoot **fconfig** option or by manipulating the configuration option `CYGNUM_HAL_M68K_MCFxxxx_DIAGNOSTICS_BAUD` and then rebuilding RedBoot. By default RedBoot will use the board's terminal port, corresponding to uart0, but this can also be changed via the configuration option `CYGHWR_HAL_M68K_MCFxxxx_DIAGNOSTICS_PORT`. On the DNP/5280 and DNP/5282 platforms RedBoot also supports ethernet communication and flash management.

Initial Installation

This process assumes that the board currently has the dBUG ROM monitor in flash and does not require any special debug hardware. Programming the RedBoot rom monitor into flash memory requires an application that can manage flash blocks. RedBoot itself has this capability. Rather than have a separate application that is used only for flash management during the initial installation, a special RAM-resident version of RedBoot is loaded into memory and run. This version can then be used to load the normal flash-resident version of RedBoot and program it into the flash.

The first step is to connect an RS232 cable between the DNP/5280 terminal port and the host PC. A suitable cable is supplied with the board. Next start a terminal emulation application such as HyperTerminal or minicom on the host PC and set the serial communication parameters to 115200 baud, 8 data bits, no parity, 1 stop bit (8N1) and no flow control (handshaking). Make sure that the RCM jumper is set to boot into dBUG rather than directly into uCLinux. Apply power to the board and you should see a `dBUG>` prompt.

Once dBUG is up and running the RAM-resident version of RedBoot can be downloaded:

```
dBUG> dl
Escape to local host and send S-records now...
```

The required S-records file is `redboot_dbug.srec`, which is normally supplied with the eCos release in the `loaders` directory. If it needs to be rebuilt then instructions for this are supplied below. The file should be sent to the target as raw text using the terminal emulator:

```
S-record download successful!
dBUG>
```

It is now possible to run the RAM-resident version of RedBoot:

```
dBUG> go 0x10000
+**Warning** FLASH configuration checksum error or invalid key
Use 'fconfig -i' to [re]initialize database
... waiting for BOOTP information
Ethernet eth0: MAC address 00:00:00:00:00:03
Can't get BOOTP info for device!

RedBoot(tm) bootstrap and debug environment [DBUG]
Non-certified release, version UNKNOWN - built 21:15:10, Mar 16 2004

Platform: DNP/5280 (Freescale MCF5280)
Copyright (C) 2000, 2001, 2002, Free Software Foundation, Inc.
Copyright (C) 2003, 2004, eCosCentric Limited

RAM: 0x00000000-0x01000000, 0x0002f4c8-0x00fed000 available
FLASH: 0xff800000 - 0x00000000, 128 blocks of 0x00010000 bytes each.
RedBoot>
```

At this stage the RedBoot flash management initialization has not yet happened so the warning about the configuration checksum error is expected. There will also be a delay while RedBoot tries to contact a local BOOTP server. To perform the flash initialization use the **fis init -f** command:

```
RedBoot> fis init -f
About to initialize [format] FLASH image system - continue (y/n)? y
*** Initialize FLASH Image System
... Erase from 0xffff0000-0xffffffff: .
... Program from 0x00ff0000-0x00100000 at 0xffff0000: .
RedBoot>
```

At this stage the block of flash at location 0xFFFF0000 holds information about the various flash blocks, allowing other flash management operations to be performed. The next step is to set up RedBoot's non-volatile configuration values:

```
RedBoot> fconfig -i
Initialize non-volatile configuration - continue (y/n)? y
Run script at boot: false
Use BOOTP for network configuration: true
Console baud rate: 115200
DNS server IP address:
GDB connection port: 9000
Force console for special debug messages: false
Network hardware address [MAC]: 0x00:0x00:0x00:0x00:0x00:0x03
Network debug at boot time: false
Update RedBoot non-volatile configuration - continue (y/n)? y
... Erase from 0xffff0000-0xffffffff: .
... Program from 0x00ff0000-0x01000000 at 0xffff0000: .
RedBoot>
```

For most of these configuration variables the default value is correct. If there is no suitable BOOTP service running on the local network then BOOTP should be disabled, and instead RedBoot will prompt for a fixed IP address, netmask, and addresses for the local gateway and DNS server. The other exception is the network hardware address, also known as MAC address. All boards should be given a unique MAC address, not the one in the above example. If there are two boards on the same network trying to use the same MAC address then the resulting behaviour is undefined.

It is now possible to load the flash-resident version of RedBoot. Because of the way that flash chips work it is better to first load it into RAM and then program it into flash.

```
RedBoot> load -r -m ymodem -b %{freememlo}
```

The file `redboot_rom.bin` should now be uploaded using the terminal emulator. The file is a raw binary and should be transferred using the Y-modem protocol.

```
Raw file loaded 0x0002f800-0x0004613f, assumed entry at 0x0002f800
xyzModem - CRC mode, 726(SOH)/1(STX)/0(CAN) packets, 6 retries
RedBoot>
```

Once RedBoot has been loaded into RAM it can be programmed into flash:

```
RedBoot> fis create RedBoot -b %{freememlo}
An image named 'RedBoot' exists - continue (y/n)? y
... Erase from 0xff800000-0xff820000: .
... Program from 0x0002f800-0x0004f800 at 0xff800000: .
... Erase from 0xffff0000-0xffffffff: .
... Program from 0x00ff0000-0x01000000 at 0xffff0000: .
RedBoot>
```

The flash-resident version of RedBoot has now programmed at location 0xFF800000, and the flash info block at 0xFFFF0000 has been updated. The initial setup is now complete. Reset the board:

```
+Ethernet eth0: MAC address 00:00:00:00:00:03
Can't get BOOTP info for device!

RedBoot(tm) bootstrap and debug environment [ROM]
Non-certified release, version UNKNOWN - built 20:52:02, Mar 16 2004

Platform: DNP/5280 (Freescale MCF5280)
Copyright (C) 2000, 2001, 2002, Free Software Foundation, Inc.
Copyright (C) 2003, 2004, eCosCentric Limited

RAM: 0x00000000-0x01000000, [0x0000d3c8-0x00fed000 available
FLASH: 0xff800000 - 0x00000000, 128 blocks of 0x00010000 bytes each.
RedBoot>
```

When RedBoot issues its prompt it is also ready to accept connections from m68k-elf-gdb, allowing eCos applications to be downloaded and debugged.

Occasionally it may prove necessary to update the installed RedBoot image. This can be done by loading a RAM-resident version of RedBoot, `redboot_ram.bin`, rather than the DBUG version of RedBoot used above. The ROM version can then be loaded into memory using RedBoot's **load** command, and the flash version can be updated using **fis create RedBoot**.

Rebuilding RedBoot

Should it prove necessary to rebuild a RedBoot binary, this is done most conveniently at the command line. The steps needed to rebuild the dBUG version of RedBoot are:

```
$ mkdir redboot_dbug
$ cd redboot_dbug
$ ecosconfig new dnp5280 redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/m68k/mcf52xx/mcf5282/dnp5280/current/misc/redboot_DBUG.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

At the end of the build the `install/bin` subdirectory should contain the required file `redboot_dbug.srec`.

Rebuilding the RAM and ROM versions involves basically the same process. The RAM version uses the file `redboot_RAM.ecm` and generates a file `redboot_ram.bin`. The ROM version uses the file `redboot_ROM.ecm` and generates a file `redboot_rom.bin`.

Name

Configuration — Platform-specific Configuration Options

Overview

The DNP/5280 platform HAL package is loaded automatically when eCos is configured for a dnp5280, dnp5280_v12 or dnp5282 target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The DNP/5280 platform HAL package supports three separate startup types: RAM, DBUG, and ROM. The configuration option `CYG_HAL_STARTUP` controls which startup type is being used. For typical application development RAM startup should be used, and the application will be run via m68k-elf-gdb interacting with RedBoot using either serial or ethernet. It is assumed that the low-level hardware initialization, including setting up the memory map, has already been performed by RedBoot. By default the application will use certain services provided by RedBoot via the virtual vector mechanism, including diagnostic output, but that can be disabled via `CYGSEM_HAL_USE_ROM_MONITOR`.

ROM startup can be used for applications which boot directly from flash. All the hardware will be initialized, and the application is self-contained. This startup type is used by the flash-resident version of RedBoot, and can also be used for finished applications.

DBUG startup can be used for applications which will be loaded via the DBUG ROM monitor rather than RedBoot. As with RAM startup it is assumed that the memory map has already been set up, but the application will not use any services provided by the ROM monitor.

RedBoot and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building RedBoot.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained. That is useful as a testing step before switching to ROM startup. It also allows applications to be run and debugged via BDM.

If the application does not rely on a ROM monitor for diagnostic services then one of the serial ports will be claimed for HAL diagnostics. By default eCos will use uart0. If the actual hardware has connectors for uart1 or uart2, one of these can be selected instead via the configuration option `CYGHWR_HAL_M68K_MCFxxxx_DIAGNOSTICS_PORT`. The baud rate for the selected port is controlled by `CYGNUM_HAL_M68K_MCFxxxx_DIAGNOSTICS_BAUD`.

Optional Hardware

The platform HAL assumes that a DNP/5280 module is plugged into a standard DNP/EVA2_SV6 carrier board, and that a DNP/5282 module is plugged into a DNP/EVA6. This can be changed via the configuration option `CYGHWR_HAL_M68K_MCF528x_HARDWARE_DNP528x_BOARD`. The choice of carrier board determines the default settings of the various GPIO pins, in other words which pins are connected to on-chip peripherals. For example an EVA2_SV6 board only has a single uart transceiver and connector so only uart0's signals are connected to the appropriate pins. An EVA6 has two transceivers and connectors so both uart0 and uart1 are connected. When using a non-standard carrier board it is possible to define the pin connectivity via configuration options in the CDL component `CYGHWR_HAL_M68K_MCF528x_GPIO`. The configuration should adjust accordingly, enabling or disabling devices as appropriate. Mostly this happens in the MCF5282 processor HAL and the MCFxxxx variant HAL.

Special Registers

The MCF5282 processor has a number of special registers controlling the cache, on-chip RAM, and so on. The platform HAL provides a number of configuration options for setting these, for example `CYGNUM_HAL_M68K_DNP5280_RAMBAR` controls the initial value of the RAMBAR register.

System Clock

By default the system clock interrupts once every 10ms, corresponding to a 100Hz clock. This can be changed by the configuration option `CYGNUM_HAL_RTC_PERIOD`, the number of microseconds between clock ticks. Other clock-related settings are recalculated automatically if the period is changed.

Compiler Flags

The platform HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos. There are three flags specific to this port:

`-m5200`

The m68k-elf-gcc compiler supports many variants of the M68K architecture, from the original 68000 onwards. A `-m` option should be used to select the specific variant in use, and with current tools `-m5200` is the closest match for an MCF5282 processor.

`-malign-int`

This option forces m68k-elf-gcc to align integer and floating point data to a 32-bit boundary rather than a 16-bit boundary. It should improve performance. However the resulting code is incompatible with most published application binary interface specifications for M68K processors, so it is possible that this option causes problems with existing third-party object code.

`-fomit-frame-pointer`

Traditionally the `%A6` register was used as a dedicated frame pointer, and the compiler was expected to generate link and unlink instructions on procedure entry and exit. These days the compiler is perfectly capable of generating working code without a frame pointer, so omitting the frame pointer often saves some work during procedure entry and exit and makes another register available for optimization. However without a frame pointer register the m68k-elf-gdb debugger is not always able to interpret a thread stack, so it cannot reliably give a backtrace. Removing `-fomit-frame-pointer` from the default flags will make debugging easier, but the generated code may be worse.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the DNP/5280 hardware, and should be read in conjunction with that specification. The DNP/5280 platform HAL package complements the M68K architectural HAL, the MCFxxxx variant HAL, and the MCF5282 processor HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset the HAL will initialize those devices for which there are active device drivers. There is an exception for RAM startup applications which depend on a ROM monitor for certain services: the UARTs and the ethernet device will not be reinitialized because they may be in use by RedBoot for communication with the host. Full details of this initialization can be found in the function `hal_m68k_dnp5280_init` in `platform.c`.

For a ROM startup the HAL will perform additional initialization, setting up the external DRAM and chip selects. Full details can be found in the exported header `cyg/hal/plf.inc`.

Linker Scripts and Memory Maps

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

External SDRAM

This is mapped to location 0x00000000. Most of the first kilobyte is used for hardware exception vectors. The eCos virtual vectors are also placed here, allowing RAM-based applications to use services provided by the ROM monitor. For ROM startup all remaining SDRAM is available. For RAM and DBUG startup available SDRAM starts at location 0x00010000, with the bottom 64K reserved for use by either the RedBoot or dBUG ROM monitors.

Internal RAM

The 64K of internal RAM are normally mapped at location 0x20000000. Neither eCos nor RedBoot use the internal RAM so all of it is available to application code.

On-chip Peripherals

These are accessible at location 0x40000000 onwards, as per the defined symbol `HAL_MCF5282_ISPBAR`.

Note

On some other ColdFire processors the equivalent register is known as `%mbar`. The symbol `HAL_MCFxxxx_MBAR` is an alias for `HAL_MCF5282_ISPBAR`, making it easier to share device drivers.

On-chip Flash

When configured for a DNP/5282 the on-chip flash will be located at 0xF0000000. This flash is not used by either eCos or RedBoot so it is all available for use by application code.

External Flash

This is located at the top of memory, location 0xFF800000 onwards. In a typical setup the first two 64K flash blocks at location 0xFF800000 are used to hold RedBoot, and the block at location 0xFFFF0000 is used to hold flash management data and the RedBoot **fconfig** variables. The remaining blocks can be used by application code.

Clock Support

The platform HAL provides configuration options for the eCos system clock. This always uses the hardware timer PIT3, which should not be used directly by application code. The actual HAL macros for managing the clock are provided by the MCF5282 processor HAL. The specific numbers used are a characteristic of the platform because they depend on the processor speed. The gprof-based profiling code uses PIT2. Timers PIT0 and PIT1 are not used by eCos so application code is free to manipulate these as required.

Other Issues

The DNP/5280 platform HAL does not affect the implementation of other parts of the eCos HAL specification. The MCF5282 processor HAL, the MCFxxxx variant HAL, and the M68K architectural HAL documentation should be consulted for further details.

Other Functionality

The platform HAL package provides a flash driver for the off-chip AMD AM29LV640 flash chip. This driver is inactive by default, and only becomes active if the configuration includes the generic flash support `CYGPKG_IO_FLASH`.

The platform HAL provides some additional functions for manipulating the LEDs and dipswitch on the DNP/EVA2_SV6 carrier board:

`void hal_dnp5280_led_set(which, what)`

> This can be used to switch one of the LEDs on or off. The `which` argument specifies the LED and should be a number between 0 and 7. The `what` argument should be non-zero to switch the LED on, zero to switch it off. This function must not be called if the processor's QA and QB pins are not actually connected to the LEDs.

`int hal_dnp5280_dipswitch_read(which)`

> This allows application code to query the state of one of the dip switches. The `which` argument should be a number between 1 and 8.

`int hal_dnp5280_dipswitch_read_all(void)`

> This allows application code to query the state of all the dip switches in one go. The result is an 8-bit number with bit 0 corresponding to dipswitch 1.

Chapter 326. Motorola MCF521x Processor Support

Name

CYGPKG_HAL_M68K_MCF521x — eCos Support for Freescale MCF521x Processors

Description

The Freescale MCF521x group of processors is part of the larger family of Coldfire processors. The MCF521x group has several members including the MCF5211, MCF5212 and MCF5213. They differ from other Coldfire processors in that there is no external memory bus, instead all memory is on-chip. For example the MCF5213 has 256KB on-chip flash and 32K of on-chip SRAM. All MCF521x processors have basically the same set of peripherals (CAN is not available on the MCF5211 or MCF5212) but differ in the amount of on-chip memory.

The processor HAL package CYGPKG_HAL_M68K_MCF521x provides support for all MCF521x processors, although at the time of writing it has only been tested on an MCF5213. It complements the M68K architectural HAL package CYGPKG_HAL_M68K and the variant HAL package CYGPKG_HAL_M68K_MCFxxxx. An eCos configuration should also include a platform HAL package, for example CYGPKG_HAL_M68K_M5213EVB to support board-level details like how the on-chip peripherals are connected to the outside world. It should be noted that compared with other eCos Coldfire ports rather more work is done by the MCF521x processor HAL and rather less by the platform HAL. This is possible because of the lack of an external memory bus.

Configuration

The MCF521x processor HAL package should be loaded automatically when eCos is configured for appropriate target hardware. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

The configuration option CYGHWR_HAL_M68K_MCF521x_PROCESSOR defines the exact processor type, for example MCF5213. Usually it will be set by the platform HAL and should not be manipulated by the user.

The option CYG_HAL_STARTUP determines whether the application will reside in RAM or in ROM (flash). With just 32K of RAM on the MCF5213 only fairly simple applications will fit into RAM so the default is ROM startup. This can make debugging significantly more difficult because it requires the use of hardware breakpoints, and the processor only supports a small number of these. Debugging RAM startup applications can be rather easier because the debugger can insert breakpoints by modifying the code. Hence it may sometimes be convenient to debug a cut-down version of an application linked against a RAM startup eCos configuration. Alternatively with ROM startup individual functions can be placed in a “.2ram” section which means they will be copied from flash to RAM during initialization and execute from RAM. Again this may make debugging rather easier, assuming sufficient RAM is available.

For ROM startup the table of M68K exception vectors can reside either in ROM or in RAM. This table occupies 512 bytes (a full M68K exception vector table is 1K but on an MCF521x running eCos only half of this is needed). Keeping it in ROM saves a significant portion of the scarce RAM, but means that some eCos functionality (such as the HAL_VSR_SET macros) is not available. Few applications require that functionality so by default the vectors are kept in ROM.

The processor HAL gives application developers full control over how the GPIO pins should be initialized, using configuration options such as CYGHWR_HAL_M68K_MCF521x_GPIO_PORTQS_QS0. That pin is normally used for QSPI but can also be used for CAN, UART0, or as a general-purpose input or output. For a GPIO output it is also possible to specify whether the pins should be initialized high or low. The default settings for each pin are determined by the platform HAL. However there may be various jumpers or an expansion connector on the board, in which case the platform HAL may not know exactly how the various pins should be set up. Hence the processor HAL allows application developers to override the default settings for every pin. There are also configuration options for controlling the pin slew rates and drive strengths.

The GPIO pin settings are used to determine default values for a variety of other hardware-related configuration options. For example CYGHWR_HAL_M68K_MCFxxxx_UART0 will be enabled if either the TX or RX lines are connected, and that option is used elsewhere in the system when deciding which UARTs are potentially usable for HAL diagnostics or should have serial device driver support. The user may be able to override some of these settings, to handle scenarios where a pin should come up as a GPIO output but may later get switched to e.g. a UART tx line. The platform HAL may impose some restrictions.

If the SPI bus should be enabled then there is a configuration option `CYGNUM_HAL_M68K_MCF521x_QSPI_ISRPRI` to control the interrupt priority of the QSPI bus device. MCF521x processors use the same interrupt controller as the MCF5282, and this has a limitation that all interrupt priorities should be unique. The processor HAL provides non-conflicting defaults for the various on-chip devices but when changing interrupt priorities it is the application developer's responsibility to maintain unique priorities.

If the I^{2}C bus should be enabled then again there is a configuration option `CYGNUM_HAL_M68K_MCF521x_I2C_ISRPRI` to control the interrupt priority of the I^{2}C device. There is also a configuration option `CYGNUM_HAL_M68K_MCF521x_I2C_FDR` to set the `FDR` register which controls the I^{2}C bus speed. The default speed is the I^{2}C standard 100KHz, or as close to that as the hardware allows, but if all attached I^{2}C devices can operate at a faster speed then this option may be adjusted accordingly.

The processor HAL provides a configuration option `CYGNUM_HAL_RTC_PERIOD` to control the system clock speed. The default setting is 10 milliseconds between clock interrupts or as close to that as the hardware allows, giving a 100Hz system clock.

In kernel configurations the behaviour of the idle thread can be controlled using `CYGIMP_HAL_M68K_MCF521x_IDLE`. The default behaviour is `wait` where the cpu, flash and SRAM enter a low power mode but all peripherals continue operating normally. Any interrupt will bring the processor out of low power mode.

The processor HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos. There are three flags specific to the MCF521x port:

`-m528x`
: The m68k-elf-gcc compiler supports many variants of the M68K architecture, from the original 68000 onwards. A `-m` option should be used to select the specific variant in use, and as far as the compiler is concerned the MCF521x has the same cpu core as the MCF5282.

`-malign-int`
: This option forces m68k-elf-gcc to align integer and floating point data to a 32-bit boundary rather than a 16-bit boundary. It should improve performance. However the resulting code is incompatible with most published application binary interface specifications for M68K processors, so it is possible that this option causes problems with existing third-party object code.

`-fomit-frame-pointer`
: Traditionally the `%A6` register was used as a dedicated frame pointer, and the compiler was expected to generate link and unlink instructions on procedure entry and exit. These days the compiler is perfectly capable of generating working code without a frame pointer, so omitting the frame pointer often saves some work during procedure entry and exit and makes another register available for optimization. However without a frame pointer register the m68k-elf-gdb debugger is not always able to interpret a thread stack, so it cannot reliably give a backtrace. Removing `-fomit-frame-pointer` from the default flags will make debugging easier, but the generated code may be worse.

The HAL Port

This section describes how the MCF521x processor HAL package implements parts of the eCos HAL specification. It should be read in conjunction with similar sections from the architectural and variant HAL documentation.

Memory Map and Linker Script

On most platforms the memory map is determined by the platform HAL. However, on an MCF521x processor, all memory is on-chip and the memory map can be determined by the processor HAL, so less work has to be done to port to different MCF521x platforms. For ROM startup, the memory map is as follows:

0x00000000
: The base of on-chip flash, and the table of exception vectors. Usually the first 512 bytes are used for exception vectors, but if `CYGIMP_HAL_STARTUP_VECTORS_IN_RAM` is enabled then only 8 bytes of flash are needed and the exception vectors are relocated to RAM at 0x20000000.

The M68K architecture reserves 1K for the table of exception vectors but on an MCF521x only half of this is needed. Memory between the end of the flash exception vectors and 0x00000400 is used for certain eCos functions such as startup and the interrupt VSR.

0x00000400

The MCF521x on-chip flash requires a 24-byte hal_mcfxxxx_cfm_security_settings data structure at this location. This structure controls the initial locked status of flash blocks and related security settings. eCos provides default settings which leave everything disabled, but the relevant structure `hal_68k_mcf5213_security_settings` is declared weak so applications can override this if desired.

0x00000418 onwards

Code and constant data are placed in on-chip flash immediately after the CFM security settings. If the application does not fill the whole of flash then any remaining flash blocks may be used for storing persistent data using the standard eCos flash API functions such as `cyg_flash_program`.

0x20000000-0x200001FF

This is the base address for the on-chip SRAM. If the run-time exception vectors are placed in RAM via the configuration option `CYGIMP_HAL_STARTUP_VECTORS_IN_RAM` then the first 512 bytes are used for the table of exception vectors. Otherwise this SRAM is available for application data.

0x20000000 onwards or
0x20000200 onwards

On-chip SRAM is used for holding application data, both initialized data held in the `.data` region and uninitialized data in the `.bss` region. Any SRAM left over at the end can be used for a standard C heap accessed via `malloc` and `free`.

The on-chip SRAM can also be used to hold code placed in `.2ram` section. This mechanism is used by the CFM flash driver for certain low-level functions which cannot normally execute from flash. It can also be used for application functions. For code held in flash the debugger must use hardware breakpoints and the processor supports only a limited number of these. For code held in RAM the debugger can use software breakpoints, modifying the instructions at run-time, so there is no limit on the number of breakpoints. Hence placing certain functions in `.2ram` sections and hence in RAM may facilitate debugging. Obviously on-chip RAM is a scarce resource so this technique will not always be applicable.

0x40000000 onwards

The on-chip peripherals are mapped into memory starting at this addres.

For RAM startup the on-chip flash is ignored (although the CFM security settings will still affect the flash device driver). Instead all code and data gets placed into RAM and must somehow fit into the limited amount of available on-chip RAM. This can be useful for specialized applications and for debugging certain problems.

Since all MCF521x memory is on-chip the processor does not have a cache and the default empty cache macros provided by the architectural HAL will be used.

The processor HAL provides the `.ldi` file which, in conjunction with the architectural `m68k.ld` file, is used to generate the linker script.

HAL I/O

The header file `cyg/hal/proc_io.h` provides definitions of MCF521x-specific on-chip peripherals. Many of the on-chip peripherals are compatible with those on the MCF5282, and for those peripherals it is the `var_io.h` header provided by the MCFxxxx variant HAL which provides the appropriate definitions. Both headers are automatically included by the architectural header `cyg/hal/hal_io.h`, so typically application code and other packages will just include the latter.

Interrupt Handling

MCF521x processors implement standard Coldfire interrupt and exception handling, and comes with a single MCF5282-compatible interrupt controller and edge port module. Therefore all interrupt and exception handling is left to the architectural and MCFxxxx variant HAL. The processor's cyg/hal/proc_intr.h serves mainly to define symbols such as CYGNUM_HAL_ISR_UART0, mapping the MCF521x on-chip interrupt sources to the interrupt vectors.

In the default configuration, an MCF521x boots from flash and the exception vectors are held in flash. Hence the HAL_VSR_SET macro and associated functionality are not available. This behaviour can be changed via the configuration option CYGIMP_HAL_STARTUP_VECTORS_IN_RAM.

Clock Support

MCF521x processors come with two MCF5282-compatible programmable interrupt timers. PIT1 is used for the system clock using functionality provided by the variant HAL. PIT0 is available for use by the application. Timer-based profiling is not available since there is not enough RAM available for the arrays needed to hold profiling information.

The MCF521x processor HAL depends on the platform HAL to provide a configuration option CYGHWR_HAL_SYSTEM_CLOCK_HZ, corresponding to the default processor clock speed. The input clock can be provided in various ways and it is only the platform HAL which knows how the clock has been implemented on a given board.

Other Issues

The MCF521x processor HAL does not affect the implementation of data types, stack size definitions, SMP support, system startup, or debug support. The MCFxxxx variant HAL versions of HAL_LSBIT_INDEX and HAL_MSBIT_INDEX are used since the processor supports the ff1.l and bitrev.l instructions.

Other Functionality

The MCF521x processor HAL can instantiate a single flash device structure using the functionality provided by the CYGPKG_DEVS_FLASH_M68K_MCFxxxx_CFM device driver. This will only happen if the generic flash support CYGPKG_IO_FLASH is part of the configuration. Not all applications require flash driver support so to avoid unnecessary code and data overheads the generic flash support is not included in the standard eCos templates. Instead it will have to be added explicitly to the configuration.

The processor HAL will instantiate a cyg_i2c_bus structure hal_mcf521x_i2c_bus when the configuration option CYGHWR_HAL_M68K_MCF521x_I2C is enabled. That option is enabled by default if various GPIO pins are configured appropriately. The implementation is provided by the CYGPKG_DEVS_I2C_MCFxxxx device driver. The processor HAL does not know what I²C devices may be attached to the bus so that is left to the platform HAL.

The processor HAL will instantiate a cyg_spi_bus structure hal_mcf521x_qspi_bus when the configuration option CYGHWR_HAL_M68K_MCF521x_SPI is enabled. That option is enabled by default if various GPIO pins are configured appropriately. The implementation is provided by the CYGPKG_DEVS_SPI_MCFxxxx_QSPI device driver. The processor HAL does not know what SPI devices may be attached to the bus so that is left to the platform HAL. All SPI device structures should be placed in the table mcf521x_qspi.

Chapter 327. Motorola M5213EVB Board Support

Name

CYGPKG_HAL_M68K_M5213EVB — eCos Platform HAL

Description

The Motorola M5212EVB board has an MCF5213 ColdFire processor, support chips for the on-chip peripherals such as the UARTs and CAN bus, power and clock circuitry, a Zigbee chip attached to the MCF5213's SPI serial bus, some LEDs and switches, and an expansion connector for application-specific devices. The MCF5213 does not have an external memory bus, instead applications have to fit into the 256KB of on-chip flash and 32KB of on-chip SRAM.

The platform HAL package CYGPKG_HAL_M68K_M5213EVB provides the platform-specific support needed to configure and build eCos. It complements the M68K architectural HAL, the MCFxxxx variant HAL, and the MCF521x processor HAL. On the M5213EVB the role of the platform HAL is small compared with many other eCos platforms. The absence of an external memory bus means that the differences between one MCF5213 platform and the next will be comparatively small, and much of the work that would normally be done by the platform HAL can instead be done by the processor HAL. The main responsibilities of the platform HAL are to control the hardware clock and to define the default GPIO pin settings.

For typical application development eCos will be configured for a ROM startup. An application linked against eCos will be programmed into the base of flash, location 0x0, using a BDM debug solution. The application will run as soon as the board is reset, or it can be debugged over a BDM device such as the Ronetix PEEDI using the **m68k-elf-gdb** debugger. Alternatively it is possible to configure eCos for a RAM startup, load the application into RAM via m68k-elf-gdb and BDM, and execute it from RAM. However with only 32KB of RAM to work with this will only be possible for comparatively simple applications. Even for ROM startup application complexity will be limited by the 32KB of SRAM for data and the 256KB of flash for code.

All configurations for the M5213EVB include serial, watchdog, flash, I²C and SPI device drivers. However these are not always active:

1. The serial device CYGPKG_DEVS_SERIAL_MCFxxxx is inactive unless the generic serial support CYGPKG_IO_SERIAL is loaded and the configuration option CYGPKG_IO_SERIAL_DEVICES is enabled. Exactly which of the on-chip UARTs are supported depends on options in the processor HAL such as CYGHWR_HAL_M68K_MCFxxxx_UART0, which in turn depend on the GPIO pin settings. With the default settings all three UARTs can be accessed via the serial device driver, using standard I/O facilities and names such as /dev/ser0. However by default UART0 will also be used as the HAL diagnostics channel and if so it should not be accessed via the serial driver. The MCFxxxx variant HAL and the serial device driver provide relevant configuration options.

2. The watchdog device driver CYGPKG_DEVS_WATCHDOG_MCFxxxx is inactive unless the generic watchdog support CYGPKG_IO_WATCHDOG is loaded into the configuration. The latter provides functions such as watchdog_reset for manipulating the watchdog.

3. The flash device driver CYGPKG_DEVS_FLASH_M68K_MCFxxxx_CFM is inactive unless the generic flash support CYGPKG_IO_FLASH is loaded into the configuration. The latter provides functions such as cyg_flash_program for manipulating the on-chip flash.

4. The I²C bus driver CYGPKG_DEVS_I2C_MCFxxxx is automatically available to the application and can be accessed through functions such as cyg_i2c_tx provided by the generic I²C package CYGPKG_IO_I2C. The M5213EVB board does not have any I²C devices on board but the I²C SDA and SCL signals can be accessed via the expansion connector so devices can be attached that way. It would then be up to the application to instantiate appropriate cyg_i2c_device structures. If the application does not use any I²C functionality then it will all be eliminated at link-time and the application will not suffer any unnecessary overheads. The I²C and FlexCAN support are mutually exclusive.

5. The SPI bus driver CYGPKG_DEVS_SPI_MCFxxxx_QSPI is automatically available to the application and can be accessed through functions such as cyg_spi_transfer provided by the generic SPI package CYGPKG_IO_SPI. The M5213EVB has a single on-board SPI device, the Zigbee chip, and the platform HAL provides a cyg_spi_device structure cyg_spi_zigbee_mc13191. The SPI signals are available on the expansion connector so additional devices can be attached that way. It

would then be up to the application to instantiate appropriate cyg_spi_device structures. If the application does not use any SPI functionality then it will all be eliminated at link-time and the application will not suffer any unnecessary overheads

6. The FlexCAN device driver `CYGPKG_DEVS_CAN_FLEXCAN` is inactive unless the generic CAN support `CYGPKG_IO_CAN` is loaded into the configuration. It is also necessary to set the appropriate jumpers, specifically the CAN_EN jumpers must be closed and the COM_SEL jumpers must be set to CAN. The corresponding configuration option `CYGHWR_HAL_M68K_M5213EVB_CAN_EN` should then enabled. The driver provides a single channel, by default “can0”. The FlexCAN and I^{2}C support are mutually exclusive.

The on-chip interrupt controller and the edge port module are managed by eCos using macros provided by the MCFxxxx variant HAL. PIT timer 1 is normally used to implement the eCos system clock. PIT timer 0 is unused and can be manipulated by the application. The remaining peripherals (DMA, GPT, DTIM, ADC, PWM, FLEXCAN) are not used by eCos.

Some standard eCos functionality is not available on the M5213EVB board. On most platforms in a development environment the RedBoot ROM monitor is programmed into flash, allowing applications to be loaded into RAM and debugged over a serial line or ethernet. This is not possible on an M5213EVB. Even if RedBoot's data requirements could be squeezed into the 32KB of available SRAM there would not be enough left for applications. Instead on an M5213EVB debugging involves a hardware debug solution such as BDM and the application is programmed directly into flash. Other eCos functionality such as the Robust Boot Loader (RBL) package or the common HAL's virtual vector mechanism are only relevant in systems containing RedBoot, so will not work on an M5213EVB. The 32KB RAM limitation also means that some of the more advanced eCos functionality such as TCP/IP networking, the JFFS2 flash file system, and gprof-based profiling, will not fit into an M5213EVB.

Setup and eCos Configuration

Both eCos and applications should be built using the GNU tools m68k-elf-gcc, m68k-elf-g++, and so on. The original port of eCos to the M5213EVB was done using m68k-elf-gcc version 3.4.4 (eCosCentric). The recommended BDM debug solution is the Ronetix PEEDI. This requires a configuration file peedi.cfg which can be found in the platform HAL's misc directory. The configuration file will initialize the hardware in the same way as the ROM startup code. It will need minor edits, for example to specify the correct license keys and to select the CORE_FLASH for the M5213EVB. For full details see the Ronetix documentation.

Note

Application development using eCosPro releases prior to version 3.1 operated via the processor's BDM port, the on-chip debug module and a proprietary stub program **m68k-elf-cfpe-stub**. This is no longer supported.

Once the PEEDI is set up applications can be linked against an eCos configuration built with either RAM or ROM startup mode. Applications can then be debugged via **m68k-elf-gdb** either used directly at the command line or via an integrated development environment such as Eclipse.

The M5213EVB board comes with a large number of jumpers. This means that the platform HAL does not automatically know which on-chip peripherals should be connected to the appropriate pins and which pins should be left for general purpose I/O. For example usually processor pin 7 will be used for the UART0 RX line but by removing a jumper it can be disconnected from the RS232 transceiver. Instead some other device can be hooked up to this pin via the expansion connector and the pin should be set up as a GPIO output to drive that device. Jumpers also control which input clock should be used, and that affects how eCos should initialize the clock hardware.

The default eCos configuration will assume the default jumper settings as follows:

CLKSEL 1 and 2 in, 3 to 6 out

Use the external 8MHz reference crystal oscillator Y1 in PLL mode. This will be multiplied internally to give an 80MHz system clock.

UART0_EN all in
UART1_EN all in
UART2_EN all in

All on-chip UARTs have all of TX, RX, RTS and CTS connected to the RS232 transceivers.

COM_SEL all set tot UART

The DB9 socket for UART2_CAN carries the UART2 signals.

LED_EN in	The processor can drive the LEDs.
BDM_EN in	The debug port is used for BDM and not JTAG.

The platform HAL does not concern itself with other jumpers such as those controlling the I^{2}C pull-ups, although obviously these will have to be set appropriately if the corresponding on-chip peripherals are to function correctly.

If any of these jumper settings are changed then configuration options within the platform HAL will need to be changed accordingly. If some other clock input is enabled or if the system clock should run at some speed other than 80MHz then `CYGHWR_HAL_M68K_M5213EVB_SYNCR` and `CYGHWR_HAL_SYSTEM_CLOCK_HZ` should be edited. The first of these determines what gets programmed into the SYNCR register and the processor reference manual should be consulted for more information. The second informs eCos what the actual system clock speed will be. This depends on the hardware as well as the SYNCR setting so cannot be calculated.

If the LED_EN jumper is disconnected then the configuration option `CYGHWR_HAL_M68K_M5213EVB_LED_EN` should be disabled. Similary if any of the jumper blocks UART0_EN, UART1_EN or UART2_EN are disconnected then the corresponding configuration options `CYGHWR_HAL_M68K_M5213EVB_UART0_EN` etc. should be disabled. This assumes all of the jumpers in a jumper block are either in or out. If instead say UART1 still has its TX line connected but not RX, RTS or CTS then the processor HAL provides finer-grained configuration options such as `CYGHWR_HAL_M68K_MCF521x_GPIO_PORTUB_UB1` which allow the application developer to specify exactly which pins should be used for what purpose.

The coarse-grained platform HAL options such as `CYGHWR_HAL_M68K_M5213EVB_LED_EN` are used to determine the default values of various processor HAL options such as `CYGHWR_HAL_M68K_MCF521x_GPIO_PORTTC`. Application developers can always edit the latter options rather than the platform HAL ones to gain full control over each pin. The processor HAL will initialize all the pins as per its GPIO options, and the information is also used by the configuration system to determine whether, for example, the serial device driver should allow access to UART2.

Assuming default jumper settings there is no need to change any of the configuration options. eCos can be built and applications linked as normal. Because only very simple applications will fit into the 32KB of on-chip SRAM the default startup mode is ROM, requiring that the application be programmed into flash at location 0. Any suitable flash programming utility can be used for this as well as the Ronetix PEEDI. Assumed that all the GNU tools are already installed, the examples below assume that a PEEDI JTAG/BDM debugger is attached to the M5213EVB and listening for connections on TCP/IP port 9000.

Programming ROM images with a Ronetix PEEDI

This section describes how to program ROM images using a Ronetix PEEDI debugger.

The PEEDI must be configured to allow communication with your local network, and configured with the parameters for interfacing with the target board. It must then be used to download and program the ROM image into the internal flash. The following steps give a typical outline for doing this. Consult the PEEDI documentation for alternative approaches, such as using FTP or HTTP instead of TFTP.

Preparing the Ronetix PEEDI JTAG/BDM debugger

1. Prepare a PC to act as a host and start a TFTP server on it.
2. Connect the PEEDI JTAG/BDM debugger via both serial and ethernet to the host PC and power it on. Use the serial cable supplied with the PEEDI (straight through, not null modem).
3. Verify the PEEDI is using up-to-date firmware, of version 11.10.1 or later. If the firmware is not recent enough, follow the PEEDI User Manual's instructions which describe how to update the PEEDI firmware.
4. Locate the PEEDI configuration file `peedi.cfg` within the eCos platform HAL package in the source repository. This will be in the directory `packages/hal/m68k/mcf52xx/mcf521x/m5213evb/`*`VERSION`*`/misc` relative to the root of your eCos installation.
5. Place the PEEDI configuration file in a location on the PC accessible to the TFTP server. Later you will configure the PEEDI to load this file via TFTP as its configuration file.

6. Open `peedi.cfg` in an editor such as emacs or notepad and insert your own license information in the `[LICENSE]` section.

7. Install and configure the PEEDI in line with the PEEDI Quick Start Guide or User's Manual, especially configuring PEEDI's RedBoot with the network information. Configure it to use the `peedi.cfg` target configuration file on the TFTP server at the appropriate point of the **config** process, for example with a path such as: `tftp://192.168.7.9/peedi.cfg`

8. Reset the PEEDI.

9. Connect to the PEEDI's CLI interface via TCP/IP on the standard telnet port 23. The **telnet** application is suitable for this. You should see output similar to the following:

```
$ telnet peedi-0
Trying 172.16.19.140...
Connected to peedi-0.
Escape character is '^]'.

PEEDI - Powerful Embedded Ethernet Debug Interface
Copyright (c) 2005-2011 www.ronetix.at - All rights reserved
Hw:1.2, L:BDM v1.1 Fw:11.10.1, SN: PD-XXXX-XXXX-XXXX
------------------------------------------------------------

m5213evb>
```

Preparing the M5213EVB for programming with PEEDI

Follow the steps in this section in order to allow communication between the board and the host PC, and between the board and the BDM device.

If programming a GDB stub ROM or an application which uses serial output, you should first:

1. Connect an adaptor from the serial pins on the board to an RS232 DB9 serial connector or cable, then connect from there to a serial port on the host computer with a null modem DB9 RS232 serial cable.

2. Start a suitable terminal emulator on the host computer such as **minicom** on Linux or PuTTY on Windows. Set the communication parameters to 38400 baud, 8 data bits, no parity bit and 1 stop bit with no flow control.

For all applications, you must:

1. Connect the board to the PEEDI using an appropriate cable from the BDM interface connector to the Target port on the PEEDI.

2. Power up the M5213EVB.

3. Connect to the PEEDI's telnet CLI on port 23 as before.

4. Confirm correct connection with the PEEDI with the **reset reset** command as follows:

```
m5213evb> reset reset
++ info: RESET and BKPT asserted
++ info: RESET released
++ info: BKPT released
++ info: core 0: initialized

m5213evb>
```

Installation into Flash using the Ronetix PEEDI

The following describes the procedure for installing a ROM application into on-chip Flash using the Ronetix PEEDI, using the tm_basic test as an example of such an application.

1. Use **m68k-elf-objcopy** to convert the linked application, in ELF format, into binary format. For example:

```
$ m68k-elf-objcopy -O binary programname programname.bin
```

2. Copy the binary file (.bin file) into a location on the host computer accessible to its TFTP server.

3. Connect to the PEEDI's telnet interface, and program the image into Flash with the following command, replacing *`TFTP_SERVER`* with the address of the TFTP server and *`/BINPATH`* with the location of the .bin file relative to the TFTP server root directory. For example for the tm_basic test:

   ```
   m5213evb> flash program tftp://TFTP_SERVER/BINPATH/tm_basic.bin bin 0x0 erase
   ++ info: Programming image file: tftp://TFTP_SERVER/BINPATH/tm_basic.bin
   ++ info: Programming directly
   ++ info: At absolute address:    0x00000000
   erasing     at 0x00000000 (page #0)
   erasing     at 0x00000800 (page #1)
   programming at 0x00000000
   erasing     at 0x00001000 (page #2)
   erasing     at 0x00001800 (page #3)
   programming at 0x00001000
   erasing     at 0x00002000 (page #4)
   erasing     at 0x00002800 (page #5)
   programming at 0x00002000
   erasing     at 0x00003000 (page #6)
   erasing     at 0x00003800 (page #7)
   programming at 0x00003000
   erasing     at 0x00004000 (page #8)
   erasing     at 0x00004800 (page #9)
   programming at 0x00004000
   erasing     at 0x00005000 (page #10)
   erasing     at 0x00005800 (page #11)
   programming at 0x00005000
   erasing     at 0x00006000 (page #12)
   erasing     at 0x00006800 (page #13)
   programming at 0x00006000
   erasing     at 0x00007000 (page #14)
   erasing     at 0x00007800 (page #15)
   programming at 0x00007000
   erasing     at 0x00008000 (page #16)
   erasing     at 0x00008800 (page #17)
   programming at 0x00008000
   erasing     at 0x00009000 (page #18)
   erasing     at 0x00009800 (page #19)
   programming at 0x00009000
   erasing     at 0x0000A000 (page #20)
   erasing     at 0x0000A800 (page #21)
   programming at 0x0000A000
   erasing     at 0x0000B000 (page #22)
   erasing     at 0x0000B800 (page #23)
   programming at 0x0000B000

   ++ info: successfully programmed 48.00 KB in 1.45 sec

   m5213evb>
   ```

Once programmed into flash the application can be run simply by resetting the board. This may be acheived through the PEEDI telnet session by running the command **reset reset** and issuing the **go** command. For example:

```
m5213evb> reset reset
++ info: user reset
m5213evb>
++ info: RESET and BKPT asserted
++ info: RESET released
++ info: BKPT released
++ info: core 0: initialized

m5213evb> go
```

UART0 will be used for the HAL diagnostics channel so any output generated by the application will appear there. The default communication parameters are 8 bits, no parity, 1 stop bit, and 38400 baud. HAL diagnostics are managed by the MCFxxxx variant HAL and there are two main configuration options: `CYGHWR_HAL_M68K_MCFxxxx_DIAGNOSTICS_PORT` can be used to change the UART used or to cause diagnostics to be discarded completely; `CYGNUM_HAL_M68K_MCFxxxx_DIAGNOSTICS_BAUD` can be used to change the baud rate. In the case of the tm_basic image, output similar to the following should be visible:

Example 327.1. m5213evb Real-time characterization

```
INFO:<code from 0x00000000 -> 0x0000a900, CRC e654>
            Startup, main stack : stack used   356 size  2152
             Startup : Idlethread stack used    76 size  1280

eCos Kernel Timings
Notes: all times are in microseconds (.000001) unless otherwise stated

Reading the hardware clock takes 0 'ticks' overhead
... this value will be factored out of all other measurements
Clock interrupt took   14.28 microseconds (14 raw clock ticks)

Testing parameters:
   Clock samples:            32
   Threads:                   2
   Thread switches:         128
   Mutexes:                  32
   Mailboxes:                21
   Semaphores:               32
   Scheduler operations:    128
   Counters:                 32
   Flags:                    32
   Alarms:                   32


                                 Confidence
     Ave     Min     Max     Var  Ave  Min  Function
  ======  ======  ======  ====== ========== ========
   13.00   13.00   13.00    0.00  100% 100% Create thread
    2.50    2.00    3.00    0.50  100%  50% Yield thread [all suspended]
    3.00    3.00    3.00    0.00  100% 100% Suspend [suspended] thread
    3.00    3.00    3.00    0.00  100% 100% Resume thread
    4.00    4.00    4.00    0.00  100% 100% Set priority
    0.50    0.00    1.00    0.50  100%  50% Get priority
    7.50    7.00    8.00    0.50  100%  50% Kill [suspended] thread
    2.50    2.00    3.00    0.50  100%  50% Yield [no other] thread
    4.00    4.00    4.00    0.00  100% 100% Resume [suspended low prio] thread
    2.00    2.00    2.00    0.00  100% 100% Resume [runnable low prio] thread
    4.00    4.00    4.00    0.00  100% 100% Suspend [runnable] thread
    3.00    3.00    3.00    0.00  100% 100% Yield [only low prio] thread
    3.00    3.00    3.00    0.00  100% 100% Suspend [runnable->not runnable]
    8.00    8.00    8.00    0.00  100% 100% Kill [runnable] thread
    6.00    6.00    6.00    0.00  100% 100% Destroy [dead] thread
   12.00   12.00   12.00    0.00  100% 100% Destroy [runnable] thread
   16.00   15.00   17.00    1.00  100%  50% Resume [high priority] thread
    5.27    5.00    7.00    0.39   74%  74% Thread switch

    0.81    0.00    1.00    0.31   81%  18% Scheduler lock
    2.13    2.00    3.00    0.22   87%  87% Scheduler unlock [0 threads]
    2.13    2.00    3.00    0.22   87%  87% Scheduler unlock [1 suspended]
    2.13    2.00    3.00    0.22   87%  87% Scheduler unlock [many suspended]
    2.13    2.00    3.00    0.22   87%  87% Scheduler unlock [many low prio]

    1.16    1.00    2.00    0.26   84%  84% Init mutex
    3.56    3.00    4.00    0.49   56%  43% Lock [unlocked] mutex
    3.75    3.00    4.00    0.38   75%  25% Unlock [locked] mutex
    2.84    2.00    3.00    0.26   84%  15% Trylock [unlocked] mutex
    2.75    2.00    3.00    0.38   75%  25% Trylock [locked] mutex
    1.38    1.00    2.00    0.47   62%  62% Destroy mutex
   17.28   17.00   18.00    0.40   71%  71% Unlock/Lock mutex
```

```
    1.57    1.00    2.00    0.49   57%  42% Create mbox
    0.81    0.00    1.00    0.31   80%  19% Peek [empty] mbox
    3.52    3.00    4.00    0.50   52%  47% Put [first] mbox
    0.81    0.00    1.00    0.31   80%  19% Peek [1 msg] mbox
    3.57    3.00    4.00    0.49   57%  42% Put [second] mbox
    0.86    0.00    1.00    0.25   85%  14% Peek [2 msgs] mbox
    3.19    3.00    4.00    0.31   80%  80% Get [first] mbox
    3.48    3.00    4.00    0.50   52%  52% Get [second] mbox
    2.71    2.00    3.00    0.41   71%  28% Tryput [first] mbox
    2.76    2.00    3.00    0.36   76%  23% Peek item [non-empty] mbox
    3.00    3.00    3.00    0.00  100% 100% Tryget [non-empty] mbox
    2.57    2.00    3.00    0.49   57%  42% Peek item [empty] mbox
    2.57    2.00    3.00    0.49   57%  42% Tryget [empty] mbox
    0.95    0.00    1.00    0.09   95%   4% Waiting to get mbox
    0.95    0.00    1.00    0.09   95%   4% Waiting to put mbox
    1.57    1.00    2.00    0.49   57%  42% Delete mbox
   11.43   11.00   12.00    0.49   57%  57% Put/Get mbox

    1.09    1.00    2.00    0.17   90%  90% Init semaphore
    2.50    2.00    3.00    0.50  100%  50% Post [0] semaphore
    3.13    3.00    4.00    0.22   87%  87% Wait [1] semaphore
    2.63    2.00    3.00    0.47   62%  37% Trywait [0] semaphore
    2.50    2.00    3.00    0.50  100%  50% Trywait [1] semaphore
    1.25    1.00    2.00    0.38   75%  75% Peek semaphore
    1.25    1.00    2.00    0.38   75%  75% Destroy semaphore
    9.97    9.00   10.00    0.06   96%   3% Post/Wait semaphore

    1.66    1.00    2.00    0.45   65%  34% Create counter
    1.00    1.00    1.00    0.00  100% 100% Get counter value
    1.06    1.00    2.00    0.12   93%  93% Set counter value
    3.63    3.00    4.00    0.47   62%  37% Tick counter
    1.19    1.00    2.00    0.30   81%  81% Delete counter

    1.25    1.00    2.00    0.38   75%  75% Init flag
    3.00    3.00    3.00    0.00  100% 100% Destroy flag
    2.44    2.00    3.00    0.49   56%  56% Mask bits in flag
    2.91    2.00    3.00    0.17   90%   9% Set bits in flag [no waiters]
    4.78    4.00    5.00    0.34   78%  21% Wait for flag [AND]
    4.50    4.00    5.00    0.50  100%  50% Wait for flag [OR]
    4.81    4.00    5.00    0.31   81%  18% Wait for flag [AND/CLR]
    4.50    4.00    5.00    0.50  100%  50% Wait for flag [OR/CLR]
    0.88    0.00    1.00    0.22   87%  12% Peek on flag

    2.28    2.00    3.00    0.40   71%  71% Create alarm
    4.00    4.00    4.00    0.00  100% 100% Initialize alarm
    2.38    2.00    3.00    0.47   62%  62% Disable alarm
    4.88    4.00    5.00    0.22   87%  12% Enable alarm
    2.81    2.00    3.00    0.31   81%  18% Delete alarm
    3.91    3.00    4.00    0.17   90%   9% Tick counter [1 alarm]
   22.31   22.00   23.00    0.43   68%  68% Tick counter [many alarms]
    6.47    6.00    7.00    0.50   53%  53% Tick & fire counter [1 alarm]
  112.81  112.00  113.00    0.31   81%  18% Tick & fire counters [>> together]
   25.16   25.00   26.00    0.26   84%  84% Tick & fire counters [>> separately]
   13.00   13.00   13.00    0.00  100% 100% Alarm latency [0 threads]
   13.66   13.00   16.00    0.88   83%  67% Alarm latency [2 threads]
   13.68   13.00   16.00    0.90   82%  66% Alarm latency [many threads]
   21.02   21.00   23.00    0.03   99%  99% Alarm -> thread resume latency

    1.20    1.00    2.00    0.00            Clock/interrupt latency

    4.65    4.00    8.00    0.00            Clock DSR latency

  210    208     212  (main stack:   845)  Thread stack used (712 total)
           All done, main stack : stack used   845 size  2152
            All done : Idlethread stack used   188 size  1280

Timing complete - 37600 ms total
```

```
PASS:<Basic timing OK>
EXIT:<done>
```

Programming the application with ecoflash

eCos comes with its own utility **ecoflash** that is an expect script. Your host must therefore also have expect installed in order to run **ecoflash**. The expect program is not distributed along with eCosPro but is available for most Linux distributions and from ActiveState for Windows. Assuming both **ecoflash** and expect have been correctly installed, to program an executable `tm_basic` into flash would involve a command such as:

```
$ ecoflash -b m5213evb -t 'remote peedi-0:9000' program tm_basic
Erasing 0x00000000 - 0x0000b6c7
Writing 0x00000000 - 0x00003fff (16384 bytes) from file "/tmp/tm_basic.1358785576", offset 0
Writing 0x00004000 - 0x00007fff (16384 bytes) from file "/tmp/tm_basic.1358785576", offset 16384
Writing 0x00008000 - 0x0000b6c7 (14024 bytes) from file "/tmp/tm_basic.1358785576", offset 32768
```

The `-b` argument identifies the target hardware and the `-t` option informs **ecoflash** how to access the board in the current development environment. Here it is told to interact with a Ronetix JTAG/BDM PEEDI with the hostname `peedi-0` listening on TCP/IP port 9000. For repeated use **ecoflash** supports various environment variables:

```
$ export ECOFLASH_BOARD=m5213evb
$ export ECOFLASH_TARGET='remote peedi-0:9000'
$ ecoflash program tm_basic
Erasing 0x00000000 - 0x0000b6c7
Writing 0x00000000 - 0x00003fff (16384 bytes) from file "/tmp/tm_basic.1358785576", offset 0
Writing 0x00004000 - 0x00007fff (16384 bytes) from file "/tmp/tm_basic.1358785576", offset 16384
Writing 0x00008000 - 0x0000b6c7 (14024 bytes) from file "/tmp/tm_basic.1358785576", offset 32768
```

For the Windows **CMD** environment replace **export** with **SET** above. The separate **ecoflash** documentation in the section ecoflash Flash Programming Utility should be consulted for further details.

Debugging the application with gdb

More commonly some debugging will be necessary and **m68k-elf-gdb** can be used for this. It can be invoked directly for command line use, or it may be used indirectly as the backend for an integrated development environment such as Eclipse. Typical command-line usage for a ROM startup application would involve:

```
$ m68k-elf-gdb --quiet hello
(gdb) shell ecoflash -b m5213evb -t 'remote peedi-0:9000' program hello
Erasing 0x00000000 - 0x00004ad7
Writing 0x00000000 - 0x00003fff (16384 bytes) from file "/tmp/hello.1358786211", offset 0
Writing 0x00004000 - 0x00004ad7 (2776 bytes) from file "/tmp/hello.1358786211", offset 16384
(gdb) set remote memory-write-packet-size 128
(gdb) set remote memory-read-packet-size 128
(gdb) target remote peedi-0:9000
Remote debugging using peedi-0:9000
0x0000045c in main (argc=9146, argv=0x0) at hello.c:53
(gdb) set $pc=hal_m68k_exception_reset
Current language:  auto; currently asm
(gdb) hbreak main
Hardware assisted breakpoint 1 at 0x420: file hello.c, line 48.
(gdb) c
Continuing.

Breakpoint 1, main (argc=9146, argv=0x0) at hello.c:48
48          CYG_TEST_INIT();
Current language:  auto; currently c
```

In typical usage most of these commands will be automated via a macro in the user's `.gdbinit` file. The first two commands limit the size of data transfers between gdb and the PEEDI. This may or may not be necessary, but problems in this area have been observed with some versions of the tools. The shell command invokes **ecoflash** to program the `hello` program into flash. The target command connects gdb to a Ronetix PEEDI listening on TCP/IP port 9000. The entry point for an M5213EVB eCos application is `hal_m68k_exception_reset`. Normally the entry point is set automatically by gdb when the application is loaded into memory, but when debugging an application in flash there is no load phase so the program counter has to be set explicitly.

Normally gdb uses software breakpoints which means that it will modify the instructions in RAM to trigger a breakpoint exception. That is not possible for code in flash. Instead hardware breakpoints must be specified via the **hbreak** command instead of the more usual **break** command. The MCF5213 only supports four hardware breakpoints, one of which may be needed by gdb to implement functionality such as single-stepping at the C level, so debugging a flash-based application can prove more difficult than a RAM-based application. It should also be noted that m68k-elf-gdb has no built-in awareness of eCos data structures so there is no support for thread-aware debugging. On other platforms it is the target-side gdb stubs embedded into RedBoot that provide such support.

For very simple applications where both code and data will fit into the 32KB of available SRAM it is possible to set the configuration option `CYG_HAL_STARTUP` to RAM and then build eCos and the application for that startup. Running such an application via command-line gdb involves:

```
% m68k-elf-gdb --quiet hello
(gdb) set remote memory-write-packet-size 128
(gdb) set remote memory-read-packet-size 128
(gdb) target remote peedi-0:9000
Remote debugging using peedi-0:9000
(gdb) load
Loading section .ram_vectors, size 0x200 lma 0x20000000
Loading section .m68k_start, size 0x48 lma 0x20000200
Loading section .text, size 0x46ac lma 0x20000248
Loading section .rodata, size 0x585 lma 0x200048f4
Loading section .data, size 0x17c lma 0x20004e7c
Start address 0x20000200, load size 20469
Transfer rate: 50385 bits/sec, 108 bytes/write.
(gdb) break main
Breakpoint 1 at 0x2000038e: file hello.c:48
(gdb) c
Continuing.

Breakpoint 1, main (argc=9146, argv=0x0) at hello.c:48
48          CYG_TEST_INIT();
(gdb)
```

The same commands are used to connect to the PEEDI and to set the communication parameters, and again a gdb macro would normally be used for this. The application is then loaded into RAM, which automatically sets the program counter. Ordinary software breakpoints can now be used.

HAL Port Implementation Details

The M5213EVB platform HAL has very limited functionality compared with more typical eCos platform HALs. Because there is no external memory bus much of the work normally done by the platform HAL can instead be done higher up in the MCF521x processor HAL, and hence the code can be shared with other MCF521x-based platforms.

The M5213EVB platform HAL does not override the behaviour of any of the higher-level HALs. The M68K architectural HAL handles the bulk of system startup, interrupt and exception handling, thread context management, and the main linker script. The MCFxxxx variant HAL provides the diagnostics support, interrupt controller management, the system clock, reset, microsecond delay, and the LSBIT and MSBIT utility macros. The MCF521x processor HAL defines the memory map and provides processor-specific startup code, idle thread support. It also initializes the GPIO pins, and instantiates flash, SPI and I²C bus devices. The platform HAL mainly provides the default settings for the GPIO pins and the system clock.

When the configuration option `CYGHWR_HAL_M68K_M5213EVB_LED_EN` is enabled, indicating that the four LEDs are connected to the appropriate GPIO pins, the platform HAL package provides a utility function for manipulating the LEDs:

```
#include <cyg/hal/hal_arch.h>

externC void hal_m5213evb_led_set(int, int);
```

The first argument identifies the LED and should be a number between 0 and 3. The second argument should be 1 to switch the LED on, 0 to switch it off.

When SPI support is enabled in the processor HAL via the configuration option `CYGHWR_HAL_M68K_MCF521x_SPI` the platform HAL will instantiate a cyg_spi_device structure `cyg_spi_zigbee_mc13192` for the on-board Zigbee chip. The platform HAL does not attempt to initialize or otherwise interact with this chip.

The platform HAL provides the necessary support for the ecoflash flash programming utility. The `m5213evb.ecf` configuration file and the `m5213evb_flash.elf` target-side executable may have been installed automatically when you installed eCos. If not, the `misc` subdirectory contains the configuration file and a configuration template `ecoflash.ecm` which can be used to rebuild the target-side executable.

Chapter 328. Freescale M5208EVBe Platform HAL

Name

eCos Support for the Freescale M5208EVBe and M5208EVB (Intec Automation) Boards — Overview

Description

This package `CYGPKG_HAL_M68K_M5208EVBE` provides a port to the Freescale M5208EVBe board. The older M5208EVB is also supported via CDL configuration. The port supports RedBoot programmed into the external flash. This can be used for application bootstrap. It also provides gdb stub functionality, allowing developers to download and debug eCos applications via the gdb debugger. This can happen over either a serial line or over ethernet. Alternatively a BDM hardware debug module can be used.

The eCos port can be configured for one of three startup types:

RAM
: This is the startup type normally used during application development. RedBoot is programmed into flash and performs the initial bootstrap. m68k-elf-gdb is then used to load a RAM startup application into memory and debug it. By default the application will use eCos' virtual vectors mechanism to obtain certain services from RedBoot, including diagnostic output. The RAM startup type can also be used for finished applications: RedBoot can be made to load and run such applications automatically following bootstrap.

RAMBDM
: This is a variant of RAM startup which can be used for debugging using a BDM hardware debug module, if for some reason an application cannot be run on top of RedBoot. The main difference between RAMBDM and RAM startup is that the former does not assume the presence of RedBoot and hence will not make any virtual vector calls to obtain RedBoot services.

ROM
: This startup type can be used for finished applications which will be programmed into external flash at location 0x00000000, and which will execute as soon as the processor starts running. The application will be self-contained with no dependencies on services provided by other software. This startup type is used for building the flash-resident version of RedBoot but can also be used for application code.

Hardware

The memory map used by both eCos and RedBoot for the M5208EVBe is as follows:

Memory	**Base**	**Length**	**Write-protected**
External Flash	0x00000000	0x00800000	Yes
External SDRAM	0x40000000	0x02000000	No
Internal RAM	0x80000000	0x00004000	No
On-chip Peripherals	0xF0000000	0x10000000	No

For the older M5208EVB platform the eCos and RedBoot memory map is:

Memory	**Base**	**Length**	**Write-protected**
External Flash	0x00000000	0x00200000	Yes
External SDRAM	0x40000000	0x02000000	No
Internal RAM	0x80000000	0x00004000	No
On-chip Peripherals	0xF0000000	0x10000000	No

By default caching is enabled for the external flash and SDRAM. There is no need to cache the internal RAM, and caching the peripherals would break all device drivers. As a debugging aid the flash is set to write-protected, which should catch some null pointer indirections. The flash driver will temporarily set this part of the address space to read-write when modifying the flash.

For all startup types the M68K exception vectors, the eCos virtual vector table, and a small amount of additional data is placed at the base of SDRAM. For ROM startup the application's data starts immediately afterwards. For RAM and RAMBDM startup application code starts at 0x40010000, with just under 64K reserved for use by RedBoot, and data follows after the code.

Typically the first 128K of flash is used for RedBoot, and the last 64K of flash is used for RedBoot's FIS and fconfig data. The remainder of the flash is available for use by the application, and is supported via the V2 AMD flash driver CYGPKG_DEVS_FLASH_AMD_AM29XXXXX_V2. That driver is inactive unless the generic flash support CYGPKG_IO_FLASH has been included in the configuration. The amount of flash available is the main difference between the newer M5208EVBe board and the older M5208EVB. The default M5208EVBe platform HAL uses CFI to determine the actual flash chips present, with the M5208EVB being restricted to the known/fixed 2MB flash due to hardware mapping of the flash chip. The generic flash support provides an API that ensures an application need not worry about the details of which flash chips are present.

Code and data can be placed in the internal RAM using the linker script section “.iram_text” for code, and “.iram_data” and “.iram_bss” for initialized and unitialized data respectively. The M68K architectural HAL contains a testcase iram1.c which demonstrates how to use these linker sections.

Of the three on-chip UARTs, UART0 and UART1 have on-board RS232 transceivers and UART2 is available via GPIO configuration on the CN1 I/O header. Normally UART0 will be used as the default diagnostics channel for RedBoot and for stand-alone applications, and may also be inherited as the diagnostics/debug channel when debugging a RAM startup application over serial.

The on-chip ethernet device is supported via the device driver CYGPKG_DEVS_ETH_MCFxxxx. This driver will be inactive unless the generic ethernet support CYGPKG_IO_ETH_DRIVERS is included in the configuration. Typically that will happen automatically when the configuration is created using the net template. For RedBoot or applications run on top of RedBoot the ethernet MAC address will typically be supplied by an **fconfig** setting. Otherwise the address will be set by a configuration option in the ethernet driver. Care should be taken that no two boards on the same network segment accidentally use the same MAC address.

The I²C bus is supported by the generic package CYGPKG_IO_I2C and the device driver CYGPKG_DEVS_I2C_MCFxxxx. Both of these will be included automatically in any configuration for the M5208EVBe, but will be eliminated at link-time if the application does not use any I²C functionality. The I²C bus instance is called hal_mcfxxxx_i2c_bus.

The QSPI bus is supported by the generic package CYGPKG_IO_SPI and the device driver CYGPKG_DEVS_SPI_MCFxxxx_QSPI. Both of these will be included automatically in any configuration for the M5208EVBe, but will be eliminated at link-time if the application does not use any SPI functionality. The SPI bus instance is called hal_mcfxxxx_qspi_bus, and the platform HAL also instantiates an SPI device object hal_m5208evbe_m13192.

All eCos configurations for the M5208EVBe also include a watchdog device driver CYGPKG_DEVS_WATCHDOG_MCFxxxx_SCM. That driver is inactive unless the generic watchdog support CYGPKG_IO_WATCHDOG has been added to the configuration, and should be accessed via the API provided by that package.

eCos also manages the interrupt controllers, the FlexBus settings, and the SDRAM controller. The Crossbar switch is set to favour I/O rather than the cpu, avoiding problems with DMA underruns. Timer PIT0 is normally used for the eCos system clock, and when using the profiling package CYGPKG_PROFILE_GPROF PIT1 will be used for the profiling timer. The remaining hardware is available for use by the application.

Tools

The M5208EVBe port is intended to work with GNU tools configured for an m68k-elf target. The original port was done using m68k-elf-gcc version 4.4.5c m68k-elf-gdb version 7.2, and binutils version 2.20.1.

By default eCos is built using the compiler flag -fomit-frame-pointer. Omitting the frame pointer eliminates some work on every function call and makes another register available, so the code should be smaller and faster. However without a frame pointer m68k-elf-gdb is not always able to identify stack frames, so it may be unable to provide accurate backtrace information. Removing this compiler flag from the configuration option CYGBLD_GLOBAL_CFLAGS avoids such debug problems.

Name

Setup — Preparing the M5208EVBe board for eCos Development

Overview

In a typical development environment the M5208EVBe board boots from flash into the RedBoot ROM monitor. eCos applications are configured for a RAM startup, and then downloaded and run via either ethernet or serial using the m68k-elf-gdb debugger. Preparing the board therefore involves programming a suitable RedBoot image into flash memory. Alternatively it is possible to use a BDM hardware debug module to load and run a RAMBDM startup application, with no need to install RedBoot into flash first. Some functionality provided by RedBoot via virtual vectors will not be available in that scenario. In particular the ethernet driver will not be able to access an fconfig setting for the MAC address, and instead it will use a fixed address controlled by a configuration option.

Setting Up BDM

The recommended BDM debug solution is the Ronetix PEEDI. This requires a configuration file `peedi.m5208evbe.cfg` which can be found in the platform HAL's `misc` directory. The configuration file will initialize the hardware in the same way as the ROM startup code. It will need minor edits, for example to specify the correct license keys and to select the CORE_FLASH respectively for the M5208EVBe or M5208EVB. For full details see the Ronetix documentation.

Once the PEEDI is set up applications can be linked against an eCos configuration built with RAMBDM startup mode. That is largely equivalent to RAM startup but disables the use of virtual vectors, so it does not assume that RedBoot is present. Applications can then be run via m68k-elf-gdb:

```
% m68k-elf-gdb install/tests/kernel/current/tests/tm_basic
…
(gdb) target remote peedi:9000
Remote debugging using peedi:9000
0x00015b6c in ?? ()
(gdb) load
Loading section .m68k_start, size 0xa4 lma 0x40010000
Loading section .text, size 0xaadc lma 0x400100a4
Loading section .rodata, size 0xfc1 lma 0x4001ab80
Loading section .data, size 0x14c lma 0x4001bb44
Start address 0x40010000, load size 48269
Transfer rate: 250 KB/sec, 8044 bytes/write.
(gdb) break cyg_test_exit
Breakpoint 1 at 0x40016670: file /work/ecos/ecospro-common/packages/infra/current/src/tcdiag.cxx, line 316.
(gdb) continue
Continuing.

Breakpoint 1, cyg_test_exit () at /work/ecos/ecospro-common/packages/infra/current/src/tcdiag.cxx:316
316   cyg_test_exit(void)
$
```

The application output will go via the configured HAL diagnostics channel, as per `CYGHWR_HAL_M68K_MCFxxxx_DIAGNOSTICS_PORT`. The default settings will send the output via UART0 at 115200 baud, 8 bits, no parity, and 1 stop bit. The baud rates can be changed via the configuration option `CYGNUM_HAL_M68K_MCFxxxx_DIAGNOSTICS_BAUD`. It is also possible to set the diagnostics port to discard all output or to use the gdb hwdebug fileio channel. The latter requires running an extra gdb command after loading the application:

```
…
(gdb) load
Loading section .m68k_start, size 0xa4 lma 0x40010000
Loading section .text, size 0xaa9c lma 0x400100a4
Loading section .rodata, size 0xfc1 lma 0x4001ab40
Loading section .data, size 0x150 lma 0x4001bb04
Start address 0x40010000, load size 48209
Transfer rate: 250 KB/sec, 8034 bytes/write.
(gdb) set hwdebug
```

```
(gdb) break cyg_test_exit
Breakpoint 1 at 0x40016670: file /work/ecos/ecospro-common/packages/infra/current/src/tcdiag.cxx, line 316.
(gdb) continue
…
```

The application output will now be sent to gdb, which will display it.

Installing RedBoot

The following RedBoot configurations are supported for the M5208EVBe and M5208EVB boards:

Startup	Description	Use	Files
ROM	Runs from the board's flash	`redboot_ROM.ecm`	`redboot.elf`, `redboot.bin`
RAM	Used for upgrading ROM version	`redboot_RAM.ecm`	`redboot.elf`, `redboot.bin`

Note that the RAM and RAMBDM startups are equivalent as far as RedBoot is concerned. The RAM startup version of RedBoot can be run either via BDM or on top of an already installed ROM RedBoot. eCosPro releases come with prebuilt RedBoot images renamed to incorporate the startup type, for example `redboot_ROM.bin`. It is the ROM RedBoot binary that needs to be programmed into flash at location 0x00000000. This may involve a third party flash programming utility. Alternatively it is possible to first run up the RAM startup version via BDM and then use that to program the ROM binary into flash. The eCosPro installation's "loaders/m5208evbe" directory contains the prebuilt RedBoot images for the M5208EVBe target and the "loaders/m5208evb" directory the prebuilt RedBoot images for the M5208EVB. The following example assumes you are using a Ronetix PEEDI BDM debugger to load the appropriate RAM RedBoot prebuilt for your target.

```
% m68k-elf-gdb <path>/redboot.elf
…
(gdb) target remote peedi:9000
Remote debugging using peedi:9000
fis_lookup (name=0x4001bc80 "@\001\ufffd8", num=0x40016a7a) at /work/ecos/ecospro-common/packages/redboot/current/src/
267   {
(gdb) load
Loading section .m68k_start, size 0x98 lma 0x40010000
Loading section .text, size 0x16dd8 lma 0x40010098
Loading section .rodata, size 0x3afc lma 0x40026e70
Loading section .data, size 0xb08 lma 0x4002a96c
Start address 0x40010000, load size 111732
Transfer rate: 274 KB/sec, 12414 bytes/write.
(gdb) continue
```

The RAM RedBoot will now output its banner and prompt via UART0, again at 115200 baud 8 bits no parity 1 stopbit, and will accept commands:

```
+**Warning** FLASH configuration checksum error or invalid key
Use 'fconfig -i' to [re]initialize database
Ethernet eth0: MAC address 00:ff:12:34:56:78
IP: 10.1.1.181/255.255.255.0, Gateway: 10.1.1.241
Default server: 0.0.0.0
DNS server IP: 10.1.1.240, DNS domain name: <null>

RedBoot(tm) bootstrap and debug environment [RAM]
…
Platform: M5208EVBe (Freescale MCF5208)
RAM: 0x40000000-0x41000000 [0x40036424-0x40fed000 available]
FLASH: 0x00000000-0x007fffff, 64 x 0x10000 blocks, 64 x 0x10000 blocks
RedBoot>
```

At this stage the RedBoot flash management initialization has not yet happened so the warning about the configuration checksum error is expected. In this example RedBoot is using a default MAC address and has contacted a local bootp server to get the IP address information. To perform the flash initialization use the **fis init** command.

```
RedBoot> fis init
About to initialize [format] FLASH image system - continue (y/n)? y
** Initialize FLASH Image System
... Erase from 0x007f0000-0x007fffff: .
... Program from 0x40ff0000-0x41000000 to 0x007f0000: .
RedBoot>
```

Now the block of flash at location 0x007F0000 holds information about the various flash blocks, allowing other flash management operations to be performed. The next step is to set up RedBoot's non-volatile configuration values:

```
RedBoot> fconfig -i
Initialize non-volatile configuration - continue (y/n)? y
Run script at boot: false
Use BOOTP for network configuration: true
Default server IP address:
DNS server IP address:
GDB connection port: 9000
Force console for special debug messages: false
Network hardware address [MAC]: 0x00:0xff:0x12:0x34:0x01:0x15
Network debug at boot time: false
Update RedBoot non-volatile configuration - continue (y/n)? y
... Erase from 0x007f0000-0x007fffff: .
... Program from 0x40ff0000-0x41000000 to 0x007f0000: .
RedBoot>
```

For most of these configuration variables the default value will be acceptable, at least initially. If there is no suitable BOOTP service running on the local network then BOOTP should be disabled, and instead RedBoot will prompt for a fixed IP address, netmask, and addresses for the local gateway and DNS server. The other exception is the network hardware address, also known as MAC address. All boards should be given a unique MAC address, not the one in the above example. If there are two boards on the same network trying to use the same MAC address then the resulting behaviour is undefined.

It is now possible to load the flash-resident version of RedBoot. Because of the way that flash chips work this involves first loading it into RAM and then programming it into flash.

```
RedBoot> load -r -m ymodem -b %{freememlo}
```

The ROM startup build of `redboot.bin` should now be uploaded using the terminal emulator. The file is a raw binary and should be transferred using the Y-modem protocol.

```
CRaw file loaded 0x40036800-0x40051dfb, assumed entry at 0x40036800
xyzModem - CRC mode, 878(SOH)/0(STX)/0(CAN) packets, 3 retries
RedBoot>
```

Once RedBoot has been loaded into RAM it can be programmed into flash:

```
RedBoot> fis create RedBoot -b %{freememlo}
An image named 'RedBoot' exists - continue (y/n)? y
... Erase from 0x00000000-0x0001ffff: ..
... Program from 0x40036800-0x40051dfc to 0x00000000: ..
... Erase from 0x007f0000-0x007fffff: .
... Program from 0x40ff0000-0x41000000 to 0x007f0000: .
RedBoot>
```

The flash-resident version of RedBoot has now been programmed at location 0x00000000, and the flash info block at 0x007F0000 has been updated. The initial setup is now complete. The BDM gdb session is no longer required so can be ctrl-C'd and exited. BDM can be disconnected if desired. When the board is powercycled the ROM version of RedBoot should now start up:

```
+... waiting for BOOTP information
Ethernet eth0: MAC address 00:ff:12:34:01:15
IP: 10.1.1.182/255.255.255.0, Gateway: 10.1.1.241
Default server: 0.0.0.0, DNS server IP: 10.1.1.240

RedBoot(tm) bootstrap and debug environment [ROM]
…
```

```
Platform: M5208EVBe (Freescale MCF5208)
RAM: 0x40000000-0x41000000 [0x4000be10-0x40fed000 available]
FLASH: 0x00000000-0x007fffff, 64 x 0x10000 blocks, 64 x 0x10000 blocks
RedBoot>
```

When RedBoot issues its prompt it is also ready to accept connections from m68k-elf-gdb, allowing eCos applications to be downloaded and debugged.

Occasionally it may prove necessary to update the installed RedBoot image. This can be done via BDM, as above, or by loading a RAM RedBoot binary via the ROM RedBoot's **load -r -m ymodem -b 0x40010000** and started with the **go** command. The ROM version can then be loaded into memory and programmed into flash as before.

Rebuilding RedBoot

Should it prove necessary to rebuild a RedBoot binary, this is done most conveniently at the command line. The steps needed to rebuild the ROM version of RedBoot are:

```
$ mkdir redboot_rom
$ cd redboot_rom
$ ecosconfig new m5208evbe redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/m68k/mcf52xx/mcf520x/m5208evbe/<vsn>/misc/redboot_ROM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

At the end of the build the `install/bin` subdirectory should contain the required file `redboot.bin`, as well as the ELF executable `redboot.elf`. Rebuilding the RAM involves basically the same process but using the file `redboot_RAM.ecm`. For the older M5208EVB board the "ecosconfig new m5208evbe redboot" command in the above example should replaced by "ecosconfig new m5208evb redboot".

Name

Configuration — Platform-specific Configuration Options

Overview

The M5208EVBe platform HAL package is loaded automatically when eCos is configured for a m5208evbe target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware. The platform HAL is complemented by the M68K architectural HAL `CYGPKG_HAL_M68K`, the MCFxxxx variant HAL `CYGPKG_HAL_M68K_MCFxxxx`, and the MCF520x processor HAL `CYGPKG_HAL_M68K_MCF520x`.

Startup

The M5208EVBe platform HAL package supports three separate startup types: RAM, RAMBDM and ROM. The configuration option `CYG_HAL_STARTUP`: controls which startup type is being used. For typical application development RAM startup should be used, and the application will be run via m68k-elf-gdb interacting with RedBoot using either serial or ethernet. It is assumed that the low-level hardware initialization, including setting up the memory map, has already been performed by RedBoot. By default the application will use certain services provided by RedBoot via the virtual vector mechanism, including diagnostic output, but that can be disabled via `CYGSEM_HAL_USE_ROM_MONITOR`. RAMBDM startup is a variant of RAM startup which has `CYGSEM_HAL_USE_ROM_MONITOR` disabled by default, and is intended for debugging over BDM instead of via RedBoot.

ROM startup can be used for applications which boot directly from flash. All the hardware will be initialized, and the application is self-contained. This startup type is used by the flash-resident version of RedBoot, and can also be used for finished applications.

M5208EVBe or M5208EVB

By default selecting the m5208evbe target will build code for the newer M5208EVBe board, with its CFI compliant flash interface. Normally the M5208EVBe board has 8MB of on-board flash available.

If support is needed for the older M5208EVB board then the CDL configuration option `CYGPKG_HAL_M68K_M5208EVBE_FLASH_2MB` can be enabled in the configuration. This provides support for the non-CFI flash interface needed to access to the 2MB of on-board flash on the M5208EVB.

RedBoot and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building RedBoot.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained. That is useful as a testing step before switching to ROM startup. It also allows applications to be run and debugged via BDM.

Diagnostics

Diagnostics support is provided by the MCFxxxx variant HAL. For RAM startup the application will inherit its diagnostics channel from RedBoot: when debugging over ethernet diagnostics will travel over the TCP connection between **m68k-elf-gdb** and RedBoot; if RedBoot is set to discard its diagnostics then application diagnostics will be discarded as well; otherwise diagnostics will be sent via UART0.

For other startup types the default diagnostics channel can be set to either of the UARTs, or eCos can be configured to discard all diagnostics. The relevant configuration option is `CYGHWR_HAL_M68K_MCFxxxx_DIAGNOSTICS_PORT`.

Optional Hardware

The MCF520x processor HAL provides configuration options for the GPIO pin assignment registers, effectively controlling which of the on-chip peripherals are connected to the outside board. These settings are used in term to set the defaults for various devices, for example which UARTs are available and whether or not the hardware handshake lines are connected. The platform HAL provides default setttings for all these registers appropriate for a M5208EVBe board, but application developers can override these settings as required. Alternatively application can manipulate the processor's GPIO unit directly.

Cache and On-Chip RAM

The platform HAL contains configuration options for the values of the RAMBAR, CACR, ACR0 and ACR1 control registers. The first of these determines the location of on-chip RAM and does not usually need to be changed by application developers. The other three registers determine the caching behaviour. The default settings enable caching for the external flash and SDRAM only.

System Clock

The board is set to operate at 166.67/83.33 MHz, corresponding to the cpu and peripheral clocks respectively. However PLL dithering is enabled, slightly reducing the effective clock frequency. Programmable interrupt timer PIT0 is used to implement the eCos system clock. By default this is set to tick approximately once every 10ms, corresponding to a 100Hz clock. The frequency can be changed via the configuration option `CYGNUM_HAL_RTC_PERIOD`, which is used to program the timer's PIT Modulus Register. Other clock-related settings are recalculated automatically if the period is changed.

It should be noted that the effects of clock dithering are not precise, and hence the resulting system clock will not run at exactly 100Hz. A consequence of this is that conversions between seconds/nanoseconds and system clock ticks will not be precise and may suffer from rounding errors. If application code requires very precise timings then it may be necessary to do some experimenting and fine tuning.

Compiler Flags

The platform HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos. There are three flags specific to this port:

`-m528x`
: The m68k-elf-gcc compiler supports many variants of the M68K architecture, from the original 68000 onwards. A `-m` option should be used to select the specific variant in use, and with current tools `-m528x` is appropriate for an MCF520x processor.

`-malign-int`
: This option forces m68k-elf-gcc to align integer and floating point data to a 32-bit boundary rather than a 16-bit boundary. It should improve performance. However the resulting code is incompatible with most published application binary interface specifications for M68K processors, so it is possible that this option causes problems with existing third-party object code.

`-fomit-frame-pointer`
: Traditionally the `%A6` register was used as a dedicated frame pointer, and the compiler was expected to generate link and unlink instructions on procedure entry and exit. These days the compiler is perfectly capable of generating working code without a frame pointer, so omitting the frame pointer often saves some work during procedure entry and exit and makes another register available for optimization. However without a frame pointer register the m68k-elf-gdb debugger is not always able to interpret a thread stack, so it cannot reliably give a backtrace. Removing `-fomit-frame-pointer` from the default flags will make debugging easier, but the generated code may be worse.

Name

Test Programs — Details

Test Programs

The M5208EVBe platform HAL contains a simple test program which allows various aspects of the board to be tested.

m5208evbe (SPI and LED) Test

The `m5208evbe` program tests the SPI driver by checking for presence of the on-board M13192 ZigBee transceiver. It currently does no more than verify presence of the device. Also, depending on the setting of the JP16:13 jumpers the device will display some patterns on the on-board LEDs. If the jumpers are not present or the GPIO configuration for the platform does not have the TIMER pins T3:0 configured for GPIO the LED changes will not be seen.

Chapter 329. Motorola MCF532x Processor Support

Name

CYGPKG_HAL_M68K_MCF532x — eCos Support for Freescale MCF532x Processors

Description

The Freescale MCF532x group of processors is part of the larger family of ColdFire processors. The MCF532x group has several members including the MCF5327, MCF5328, MCF53281 and MCF5329. These differ in the set of peripherals available, for example the MCF5327 lacks on-chip ethernet.

The processor HAL package CYGPKG_HAL_M68K_MCF532x provides support for all MCF532x processors, although at the time of writing it has only been tested on an MCF5329. It complements the M68K architectural HAL package CYGPKG_HAL_M68K and the variant HAL package CYGPKG_HAL_M68K_MCFxxxx. An eCos configuration should also include a platform HAL package, for example CYGPKG_HAL_M68K_COBRA5329, to support board-level details like how the on-chip peripherals are connected to the outside world.

The package contains very little code. Instead it consists mainly of definitions, enabling appropriate code in the MCFxxxx variant HAL and in the various device drivers.

Configuration

The MCF532x processor HAL package should be loaded automatically when eCos is configured for appropriate target hardware. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Most of the package's configuration options relate to hardware. The settings are generally determined by the platform HAL and there is little need for application developers to change them. The first hardware option is CYGHWR_HAL_MCF532x_PROCESSOR, identifying the specific MCF532x processor being used. Legal values are MCF5327, MCF5328, MCF53281 and MCF5329. Typically the platform HAL will set this option via a CDL constraint.

Component CYGHWR_HAL_M68K_MCF532x_GPIO contains various options related to pin-connectivity. This gives full control over the PAR pin assignment registers, and for those pins configured as GPIO it is also possible to control the pin direction and data settings. These options are used to initialize the processor's GPIO module early on during system initialization, but applications may change settings later on as necessary.

The GPIO settings are used to determine default settings for the three on-chip uarts, the I²C bus and the QSPI bus. For example if none of the relevant GPIO pins are assigned to uart2 then component CYGHWR_HAL_M68K_MCFxxxx_UART2 will be disabled by default, and that uart cannot be used for HAL diagnostics nor accessed via the serial device driver. It is possible to override these settings if desired, for example if a uart is connected but will be manipulated directly by application code instead of via a device driver.

1. For each of the three on-chip uarts there will be a component, e.g. CYGHWR_HAL_M68K_MCFxxxx_UART0, determining whether or not the uart is usable on the target hardware. There are additional options CYGHWR_HAL_M68K_MCFxxxx_UART0_RTS and CYGHWR_HAL_M68K_MCFxxxx_UART0_CTS indicating whether or not the hardware handshake lines are connected, and CYGHWR_HAL_M68K_MCFxxxx_UART0_RS485_RTS to indicate that the RTS line controls an RS485 transceiver.

2. Component CYGHWR_HAL_M68K_MCF532x_I2C determines whether or not the processor HAL will instantiate an I²C bus device hal_mcfxxxx_i2c_bus. There are also options to control the interrupt priority and to set the FDR register which controls the bus speed. The default bus speed will be the standard I²C bus speed of 100KHz, or as close as can be achieved given hardware limitations.

3. Component CYGHWR_HAL_M68K_MCF532x_SPI determines whether or not the processor HAL will instantiate an SPI bus device hal_mcfxxxx_qspi_bus. It contains an additional configuration option for the interrupt priority.

For configurations which include the eCos kernel, CYGIMP_HAL_M68K_MCF532x_IDLE determines what happens when the idle thread runs.

The HAL Port

This section describes how the MCF532x processor HAL package implements parts of the eCos HAL specification. It should be read in conjunction with similar sections from the architectural and variant HAL documentation.

HAL I/O

The header file cyg/hal/proc_io.h provides definitions of MCF532x-specific on-chip peripherals. Many of the on-chip peripherals are compatible with those on the MCF5282 or other ColdFire processors, and for those peripherals it is the var_io.h header provided by the MCFxxxx variant HAL which provides the appropriate definitions. Both headers are automatically included by the architectural header cyg/hal/hal_io.h, so typically application code and other packages will just include the latter.

Interrupt Handling

MCF532x processors implement standard ColdFire interrupt and exception handling, and come with two MCF5282-compatible interrupt controllers and an edge port module. Therefore all interrupt and exception handling is left to the architectural and MCFxxxx variant HAL. The interrupt controllers are slightly enhanced relative to the MCF5282, with extra registers to facilitate masking and unmasking interrupts. These enhancements are supported. Unlike the MCF5282 interrupt priorities do not have to be unique, so valid interrupt priorities are in the range 1 to 6 corresponding to M68K IPL levels.

The processor's cyg/hal/proc_intr.h serves mainly to define symbols such as CYGNUM_HAL_ISR_UART0, mapping the MCF532x on-chip interrupt sources to the interrupt vectors.

Clock

Typically hardware timer PIT3 will be used for the eCos system clock, and that timer should not be manipulated directly by application code. If gprof-based profiling is enabled then that will use hardware timer PIT2. PIT timers 0 and 1 are not used by eCos so application code is free to manipulate these as required. Some of the configuration options related to the system clock, for example CYGNUM_HAL_RTC_PERIOD, are actually contained in the platform HAL rather than the processor HAL. These options need to take into account the processor clock speed, a characteristic of the platform rather than the processor.

Caching

The processor HAL provides full support for the 16K of unified cache in copyback mode. If desired the cache can also operate in write-through mode, the cache macros will still function correctly. However for a processor running at typically 240MHz write-through cache mode is likely to slow down execution significantly, especially if other devices such as ethernet or the LCD controller need concurrent access to main memory.

The HAL also defines a macro HAL_MEMORY_BARRIER() which acts to synchronize the pipeline, delaying execution until all previous operations including all pending writes are complete. This will usually be necessary when interacting with devices that access memory directly.

Other Issues

The MCF532x processor HAL does not affect the implementation of data types, stack size definitions, SMP support, system startup, or debug support. The MCFxxxx variant HAL versions of HAL_LSBIT_INDEX and HAL_MSBIT_INDEX are used since the processor supports the ff1.l and bitrev.l instructions. HAL_DELAY_US is implemented as a simple counting loop. HAL_IDLE_THREAD_ACTION may be defined depending on configuration option CYGIMP_HAL_M68K_MCF532x_IDLE.

Other Functionality

The processor HAL will instantiate a cyg_i2c_bus structure hal_mcfxxxx_i2c_bus when the configuration option CYGHWR_HAL_M68K_MCFxxxx_I2C is enabled. That option is enabled by default if various GPIO pins are configured appropriately. The implementation is provided by the CYGPKG_DEVS_I2C_MCFxxxx device driver. The processor HAL does not know what I^{2}C devices may be attached to the bus so that is left to the platform HAL.

The processor HAL will instantiate a cyg_spi_bus structure hal_mcfxxxx_qspi_bus when the configuration option CYGHWR_HAL_M68K_MCFxxxx_SPI is enabled. That option is enabled by default if various GPIO pins are configured appropriately. The implementation is provided by the CYGPKG_DEVS_SPI_MCFxxxx_QSPI device driver. The processor HAL does not know what SPI devices may be attached to the bus so that is left to the platform HAL. All SPI device structures should be placed in the table mcfxxxx_qspi.

Chapter 330. senTec Cobra5329 Board Support

Name

eCos Support for the senTec Cobra5329 Board — Overview

Description

The senTec Cobra5329 platform consists of a small module which plugs into a carrier board. The module contains: an MCF5329 ColdFire processor; 16MB of external SDRAM; 16MB of external flash memory; a configuration dip switch; a wallclock device on the I²C bus; a temperature sensor, also on the I²C bus; a multiplexer for the SPI chip select signals; and an ethernet phy. By default the module comes with the dBUG ROM monitor programmed in the external flash. The carrier board adds: power circuitry; transceivers for two RS232 ports; a CAN transceiver; four LEDs; an MMC socket connected to the SPI bus and multiplexed chip select 0; a touch screen controller, also attached to the SPI bus and chip select 1; and connectors for the various peripherals.

This package `CYGPKG_HAL_M68K_COBRA5329` provides a port to the Cobra5329 platform. The port assumes that the cpu module is plugged into a standard carrier board. If instead it is plugged into a custom carrier board with different I/O capabilities then it may still be possible to use the port by adjusting the eCos configuration. However the port has only been tested on a standard board.

For typical eCos development a RedBoot image is programmed into the external flash replacing the existing dBUG monitor. RedBoot provides gdb stub functionality so it is then possible to download and debug eCos applications via the gdb debugger. This can happen over either a serial line or over ethernet. Alternatively a BDM hardware debug module can be used.

The eCos port can be configured for one of four startup types:

RAM
: This is the startup type normally used during application development. RedBoot is programmed into flash and performs the initial bootstrap. m68k-elf-gdb is then used to load a RAM startup application into memory and debug it. By default the application will use eCos' virtual vectors mechanism to obtain certain services from RedBoot, including diagnostic output. The RAM startup type can also be used for finished applications: RedBoot can be made to load and run such applications automatically following bootstrap.

RAMBDM
: This is a variant of RAM startup which can be used for debugging using a BDM hardware debug module, if for some reason an application cannot be run on top of RedBoot. The main difference between RAMBDM and RAM startup is that the former does not assume the presence of RedBoot and hence will not make any virtual vector calls to obtain RedBoot services.

DBUG
: This is another variant of RAM startup, used only when initializing a board. It can be used to run a special RAM-resident version of RedBoot on top of the dBUG ROM monitor, allowing a ROM startup version of RedBoot to be programmed into flash.

ROM
: This startup type can be used for finished applications which will be programmed into external flash at location 0x00000000, and which will execute as soon as the processor starts running. The application will be self-contained with no dependencies on services provided by other software. This startup type is used for building the flash-resident version of RedBoot but can also be used for application code.

Hardware

The memory map used by both eCos and RedBoot is as follows:

Memory	Base	Length	Cached	Write-protected
External Flash	0x00000000	0x01000000	Write-through	Yes
Flash Shadow	0x01000000	0x3F000000	Write-through	Yes
External SDRAM	0x40000000	0x01000000	Copyback	No

Memory	Base	Length	Cached	Write-protected
Internal RAM	0x80000000	0x00008000	Uncached	No
Dummy	0xC0000000	0x20000000	Uncached	Yes
On-chip Peripherals	0xE0000000	0x20000000	Uncached	No

There is a potential problem with external memory accesses on the MCF5329 (device erratum 6): spurious accesses to unpopulated parts of the address space can hang the processor. To avoid this problem the external flash is replicated throughout the bottom of the address space, and a dummy region is created at address 0xC0000000. As a debugging aid the flash is set to write-protected, which should catch some null pointer indirections. The flash driver will temporarily set this part of the address space to read-write when modifying the flash.

For all startup types the M68K exception vectors, the eCos virtual vector table, and a small amount of additional data is placed at the base of SDRAM. For ROM startup the application's data starts immediately afterwards. For RAM and RAMBDM startup application code starts at 0x40010000, with just under 64K reserved for use by RedBoot, and data follows after the code. For DBUG startup application code starts at 0x40020000, with 128K reserved for use by dBUG.

Typically the first 128K of flash is used for RedBoot, and the last 64K of flash @ 0x00FF0000 is used for RedBoot's FIS and fconfig data. The remainder of the flash is available for use by the application, and is supported via the V2 AMD flash driver `CYGPKG_DEVS_FLASH_AMD_AM29XXXXX_V2`. That driver is inactive unless the generic flash support `CYGPKG_IO_FLASH` has been included in the configuration.

Code and data can be placed in the internal RAM using the linker script section “.iram_text” for code, and “.iram_data” and “.iram_bss” for initialized and unitialized data respectively. The M68K architectural HAL contains a testcase `iram1.c` which demonstrates how to use these linker sections.

Of the three on-chip uarts only uart0 and uart1 have external transceivers and connectors. Uart0 is normally used as the default diagnostics channel for RedBoot and for stand-alone applications, and may also be inherited as the diagnostics/debug channel when debugging a RAM startup application over serial. The default settings are 38400 baud 8n1, except when running on top of dBUG when the baud rate is set to 115200 instead. If uart0 is not used for diagnostics then it can be accessed via the serial driver `CYGPKG_DEVS_SERIAL_MCFxxxx` using the device name “/dev/ser0”. Similarly uart1 can be accessed via the serial device driver using the name “/dev/ser1”. The device driver is inactive unless the generic serial support option `CYGPKG_IO_SERIAL_DEVICES` is enabled.

The on-chip ethernet device is supported via the device driver `CYGPKG_DEVS_ETH_MCFxxxx`. This driver will be inactive unless the generic ethernet support `CYGPKG_IO_ETH_DRIVERS` is included in the configuration. Typically that will happen automatically when the configuration is created using the `net` template. The board does not have a serial EEPROM or equivalent to hold a unique ethernet MAC address, so that has to be provided in software instead. For RedBoot or applications run on top of RedBoot the MAC address will typically be supplied by an **fconfig** setting. Otherwise the address will be set by a configuration option in the ethernet driver. Care should be taken that no two boards on the same network segment accidentally use the same MAC address.

The I²C bus is supported by the generic package `CYGPKG_IO_I2C` and the device driver `CYGPKG_DEVS_I2C_MCFxxxx`. Both of these will be included automatically in any configuration for the Cobra5329, but will be eliminated at link-time if the application does not use any I²C functionality. The I²C bus instance is called `hal_mcfxxxx_i2c_bus`, and the platform HAL also instantiates I²C device objects `hal_cobra5329_lm73`, `hal_cobra5329_max3353`, `cyg_i2c_wallclock_isl12028` and `cyg_i2c_wallclock_isl12028_eeprom` for the various devices attached to the I²C bus. These devices are not used by eCos, but example code for how to access the LM73 can be found in the platform testcase `cobra5329.c`.

The QSPI bus is supported by the generic package `CYGPKG_IO_SPI` and the device driver `CYGPKG_DEVS_SPI_MCFxxxx_QSPI`. Both of these will be included automatically in any configuration for the Cobra5329, but will be eliminated at link-time if the application does not use any SPI functionality. The SPI bus instance is called `hal_mcfxxxx_qspi_bus`, and the platform HAL also instances SPI device objects `cyg_spi_mmc_dev0` and `hal_cobra5329_tsc2200` for the two devices attached to the bus. The MMC device can be used with the eCos MMC disk driver. The TSC2200 is not used by eCos, but example code for how to access it can be found in the platform testcase `cobra5329.c`.

Support for USB peripheral mode on the OTG controller is supported by the EHCI peripheral controller driver (CYGPKG_DEVS_USB_PCD_EHCI). A configuration package (CYGPKG_DEVS_USB_COBRA) enables the charge pump. Support is also present for a CDC/ACM USB serial interface.

All eCos configurations for the Cobra5329 also include a watchdog device driver CYGPKG_DEVS_WATCHDOG_MCF532x. That driver is inactive unless the generic watchdog support CYGPKG_IO_WATCHDOG has been added to the configuration, and should be accessed via the API provided by that package.

All eCos configurations for the Cobra5329 also include a wallclock device driver CYGPKG_DEVICES_WALLCLOCK_INTERSIL_ISL12028. This will be used automatically by the C library's time-related functions, for example time and asctime, and can be changed by an eCos-specific function cyg_libc_time_settime. In addition when using RedBoot the **data** command can be used to examine and change the current clock setting.

The platform HAL contains a utility function for manipulating the four LEDs:

```
void hal_cobra5329_led_set(which, on);
```

which should be a number between 1 and 4, and on should be 1 or 0. Example code for driving the LEDs can be found in the platform testcase cobra5329.c.

When using the Robust Bootloader package CYGPKG_RBL, switch 8 on the configuration dipswitch controls the **rbl condboot** functionality.

eCos also manages the interrupt controllers, the FlexBus settings for chip selects 0 and 5, and the SDRAM controller. The Crossbar switch is set to favour I/O rather than the cpu, avoiding problems with DMA underruns. Timer PIT3 is normally used for the eCos system clock, and when using the profiling package CYGPKG_PROFILE_GPROF PIT2 will be used for the profiling timer. Bit 7 of the edge port module is set up for use with the board's IRQ7 button. The remaining hardware is available for use by the application.

Tools

The Cobra5329 port is intended to work with GNU tools configured for an m68k-elf target. The original port was done using m68k-elf-gcc version 3.4.4, m68k-elf-gdb version 6.4, and binutils version 2.16.

By default eCos is built using the compiler flag -fomit-frame-pointer. Omitting the frame pointer eliminates some work on every function call and makes another register available, so the code should be smaller and faster. However without a frame pointer m68k-elf-gdb is not always able to identify stack frames, so it may be unable to provide accurate backtrace information. Removing this compiler flag from the configuration option CYGBLD_GLOBAL_CFLAGS avoids such debug problems.

Name

Setup — Preparing the Cobra5329 board for eCos Development

Overview

In a typical development environment the Cobra5329 board boots from flash into the RedBoot ROM monitor. eCos applications are configured for a RAM startup, and then downloaded and run via either ethernet or serial using the m68k-elf-gdb debugger. Preparing the board therefore involves programming a suitable RedBoot image into flash memory, replacing the existing dBUG monitor. Four different approaches can be used for this:

1. The board can be booted into its existing dBUG ROM monitor, which is then used to load and execute a dBUG startup build of RedBoot into RAM. The RAM-resident version of RedBoot can then be used to initialize the hardware, including uploading a ROM startup version of RedBoot and programming it into flash. If anything goes wrong during this procedure then the dBUG ROM monitor may have been wiped already, leaving the board unusable unless a hardware debug solution is available.
2. A BDM hardware debug module can be used in conjunction with a suitable debugger to load and run a RAM startup build of RedBoot. Again this can then be used to initialize the hardware, including programming a ROM startup version of RedBoot into flash.
3. The ecoflash flash programming utility can be used. Again this depends on a BDM hardware debug module and a suitable debugger. ecoflash can only be used to program a ROM startup version of RedBoot into flash, it cannot perform other tasks such as initializing the **fconfig** board settings. However those other tasks can be performed from inside RedBoot.
4. A third party flash programming utility may be available.

The following RedBoot configurations are supported:

Startup	Description	Use	Files
ROM	Runs from the board's flash	`redboot_ROM.ecm`	`redboot.elf`, `redboot.bin`
dBUG	Used for initial setup	`redboot_DBUG.ecm`	`redboot.elf`, `redboot.bin`, `redboot.srec`
RAM	Used for upgrading ROM version	`redboot_RAM.ecm`	`redboot.elf`, `redboot.bin`

The RAM startup version of RedBoot can be run either via BDM or on top of an already installed ROM RedBoot. Full eCos releases may come with prebuilt RedBoot images renamed to incorporate the startup type, for example `redboot_ROM.bin`.

For serial communications all versions run with 8 bits, no parity, and 1 stop bit. The dBUG version runs at 115200 baud. The ROM and RAM versions run at 38400 baud. These baud rates can be changed via the configuration option `CYGNUM_HAL_M68K_MCFxxxx_DIAGNOSTICS_BAUD` and rebuilding RedBoot. By default uart0 will be used for the diagnostics/debug channel.

Setting Up BDM

The recommended BDM debug solution is the Ronetix PEEDI. Other solutions such as the P&E USBMultilink device have proved unreliable on this board, so if a PEEDI is not available then it is recommended that application developers should debug their applications on top of RedBoot's gdb stubs.

The PEEDI requires a configuration file `peedi.cfg` which can be found in the platform HAL's `misc` directory. The configuration file will initialize the hardware in the same way as standard eCos applications, so applications can be loaded into RAM and run as normal. The configuration file will need minor edits, for example to specify the correct license keys. For full details see the Ronetix documentation. Once the PEEDI is correctly set up m68k-elf-gdb can then connect to it in the usual way:

```
$ m68k-elf-gdb install/tests/kernel/current/tests/tm_basic
GNU gdb 6.4.50.20060226-cvs (eCosCentric)
```

```
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "--host=i686-pc-linux-gnu --target=m68k-elf"...
(gdb) target remote peedi:9000
0x400008f6 in ?? ()
(gdb) load
Loading section .m68k_start, size 0x98 lma 0x40010000
Loading section .text, size 0xa918 lma 0x40010098
Loading section .rodata, size 0x114a lma 0x4001a9b0
Loading section .data, size 0x18c lma 0x4001bafc
Start address 0x40010000, load size 48262
Transfer rate: 260172 bits/sec, 3217 bytes/write.
(gdb) break cyg_test_exit
Breakpoint 1 at 0x40016172: file /home/bartv/ecos/ecospro-common/infra/current/src/tcdiag.cxx, line 310.
void cyg_test_exit(void);
(gdb) continue
Continuing.

Breakpoint 1, cyg_test_exit () at /home/bartv/ecos/ecospro-common/infra/current/src/tcdiag.cxx:310
310         if (code_checksum != cyg_crc16(_stext, _etext - _stext)) {
(gdb) quit
The program is running.  Exit anyway? (y or n) y
$
```

A P&E USBMultilink BDM device can be used in conjunction with CodeSourcery's m68k-elf-cfpe-stub server. Inside the device is a jumper which should be set to the `NO CLK` position. The server requires a configuration file `cobra5329.cfg` which can be found in the platform HAL's `misc` directory. It is recommended that the m68k-elf-cfpe-stub be restarted for every debug session to ensure that the board is properly reset. Power cycling the boards between debug sessions may also help to improve reliability.

```
$ m68k-elf-cfpe-stub -d USBMultilink -l 9000 -t <path>/cobra5329.cfg
```

It is also possible to use a P&E parallel port module or a senTec CobraConnect module with m68k-elf-cfpe-stub. The command line invocation for the later changes to:

```
$ m68k-elf-cfpe-stub -d ParallelPortCable -l 9000 -t <path>/cobra5329.cfg
```

For Linux hosts there is also an alternative to using m68k-elf-cfpe-stub. Cobra5329 boards come with a CD, including a toolchain build in the `install/linux` subdirectory. One of the tools is m68k-bdm-elf-gdb, a variant of m68k-elf-gdb which can access parallel port BDM modules directly. Unlike the PEEDI or m68k-elf-cfpe-stub, m68k-bdm-elf-gdb will not automatically initialize the hardware. Instead this can be achieved using a set of gdb macros which can be found in the `bdm.gdb` file in the platform HAL's `misc` subdirectory. A typical debug session would look like:

```
$ m68k-bdm-elf-gdb install/tests/kernel/current/tests/tm_basic
GNU gdb 6.3
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "--host=i686-pc-linux-gnu --target=m68k-bdm-elf"...
(gdb) target bdm /dev/bdmcf0
GDB target bdm connected to /dev/bdmcf0
 Coldfire debug module version is 9 (5307/5407(e))
(gdb) source /home/bartv/ecos/m68k/hal/m68k/mcf52xx/mcf532x/cobra5329/current/misc/bdm.gdb
(gdb) bdm_preload
(gdb) load
Loading section .m68k_start, size 0x98 lma 0x40010000
Loading section .text, size 0xa918 lma 0x40010098
Loading section .rodata, size 0x114a lma 0x4001a9b0
Loading section .data, size 0x18c lma 0x4001bafc
Start address 0x40010000, load size 48262
Transfer rate: 128698 bits/sec, 502 bytes/write.
```

```
(gdb) bdm_postload
Current language:  auto; currently asm
(gdb) break cyg_test_exit
Breakpoint 1 at 0x40016172: file /home/bartv/ecos/ecospro-common/infra/current/src/tcdiag.cxx, line 310.
(gdb) c
Continuing.
```

Initial Installation Using dBUG

This process assumes that the board still has its original dBUG ROM monitor and does not require any special debug hardware. Programming the RedBoot rom monitor into flash memory requires an application that can manage flash blocks. RedBoot itself has this capability. Rather than have a separate application that is used only for flash management during the initial installation, a special RAM-resident version of RedBoot is loaded into memory and run. This version can then be used to load the normal flash-resident version of RedBoot and program it into the flash.

It should be noted that some Cobra5329 boards have been shipped with a broken version of dBUG. This section describes two different ways of loading RedBoot into RAM, over a serial line or over the network.

The first step is to connect an RS232 cable between the Cobra5329 uart0 serial port and the host PC. Next start a terminal emulation application such as HyperTerminal or minicom on the host PC and set the serial communication parameters to 115200 baud, 8 data bits, no parity, 1 stop bit (8N1) and no flow control (handshaking). Apply power to the board and you should see a `dBUG>` prompt. Once dBUG is up and running the RAM-resident version of RedBoot can be downloaded:

```
dBUG> dl
Escape to local host and send S-records now...
```

The required S-records file is the dBUG startup build of `redboot.srec`, which is normally supplied with the eCos release in the `loaders` directory. If it needs to be rebuilt then instructions for this are supplied below. The file should be sent to the target as raw text using the terminal emulator:

```
S-record download successful!
dBUG>
```

If instead dBUG complained about the S-record addresses during the download then your board has the broken version of dBUG. Instead it will be necessary to run a tftp server on the local network and place the dBUG startup build of `redboot.bin` in the tftp server's directory. The details of this will depend on the tftp server being used. Once this has been accomplished, dBUG's network settings must be set appropriately using the **set** command. The dBUG documentation should be consulted for more information on this. Once networking is functional the RedBoot image can be downloaded:

```
dBUG> dn -i redboot.bin
Address:  0x40020000
Downloading Image 'redboot.bin' from 10.1.1.251
TFTP transfer completed
Read 93972 bytes (184 blocks)
dBUG>
```

When RedBoot has been loaded into RAM, either over serial or via the network, it can be run with the **go** command:

```
dBUG> go 0x40020000
+**Warning** FLASH configuration checksum error or invalid key
Use 'fconfig -i' to [re]initialize database
Ethernet eth0: MAC address 40:01:3b:e0:00:00
IP: 10.1.1.181/255.255.255.0, Gateway: 10.1.1.241
Default server: 0.0.0.0, DNS server IP: 10.1.1.240

RedBoot(tm) bootstrap and debug environment [DBUG]
Non-certified release, version UNKNOWN - built 21:15:30, Mar  5 2008

Platform: Cobra5329 (Freescale MCF5329)
Copyright (C) 2000, 2001, 2002, 2003, 2004 Free Software Foundation, Inc.
Copyright (C) 2003, 2004, 2005, 2006, 2007, 2008 eCosCentric Limited
```

```
RAM: 0x40000000-0x41000000, [0x4004204c-0x40fed000] available
FLASH: 0x00000000-0x00ffffff, 256 x 0x10000 blocks
RedBoot>
```

At this stage the RedBoot flash management initialization has not yet happended so the warning about the configuration checksum error is expected. In this example RedBoot is using a default MAC address and has contacted a local bootp server to get the address information. To perform the flash initialization use the **fis init** command.

```
RedBoot> fis init
About to initialize [format] FLASH image system - continue (y/n)? y
*** Initialize FLASH Image System
... Erase from 0x00ff0000-0x00ffffff: .
... Program from 0x40ff0000-0x41000000 to 0x00ff0000: .
RedBoot>
```

At this stage the block of flash at location 0x00FF0000 holds information about the various flash blocks, allowing other flash management operations to be performed. The next step is to set up RedBoot's non-volatile configuration values:

```
RedBoot> fconfig -i
Initialize non-volatile configuration - continue (y/n)? y
Run script at boot: false
Use BOOTP for network configuration: true
Default server IP address:
DNS server IP address:
GDB connection port: 9000
Force console for special debug messages: false
Network hardware address [MAC]: 0x00:0xff:0x12:0x34:0x01:0x13
Network debug at boot time: false
Update RedBoot non-volatile configuration - continue (y/n)? y
... Erase from 0x00ff0000-0x00ffffff: .
... Program from 0x40ff0000-0x41000000 to 0x00ff0000: .
RedBoot>
```

For most of these configuration variables the default value will be acceptable, at least initially. If there is no suitable BOOTP service running on the local network then BOOTP should be disabled, and instead RedBoot will prompt for a fixed IP address, netmask, and addresses for the local gateway and DNS server. The other exception is the network hardware address, also known as MAC address. All boards should be given a unique MAC address, not the one in the above example. If there are two boards on the same network trying to use the same MAC address then the resulting behaviour is undefined.

It is now possible to load the flash-resident version of RedBoot. Because of the way that flash chips work it is better to first load it into RAM and then program it into flash.

```
RedBoot> load -r -m ymodem -b %{freememlo}
```

The ROM startup build of `redboot.bin` should now be uploaded using the terminal emulator. The file is a raw binary and should be transferred using the Y-modem protocol.

```
Raw file loaded 0x40042400-0x4005936b, assumed entry at 0x40042400
xyzModem - CRC mode, 737(SOH)/0(STX)/0(CAN) packets, 3 retries
RedBoot>
```

Once RedBoot has been loaded into RAM it can be programmed into flash:

```
RedBoot> fis create RedBoot -b %{freememlo}
An image named 'RedBoot' exists - continue (y/n)? y
... Erase from 0x00000000-0x0001ffff: ..
... Program from 0x40042400-0x4005936c to 0x00000000: ..
... Erase from 0x00ff0000-0x00ffffff: .
... Program from 0x40ff0000-0x41000000 to 0x00ff0000: .
RedBoot>
```

The flash-resident version of RedBoot has now been programmed at location 0x00000000, and the flash info block at 0x00FF0000 has been updated. The initial setup is now complete. Power off the board, set the terminal emulator to run at 38400 baud (the usual baud rate for RedBoot), and power up the board again.

```
+... waiting for BOOTP information
Ethernet eth0: MAC address 00:ff:12:34:01:13
IP: 10.1.1.182/255.255.255.0, Gateway: 10.1.1.241
Default server: 0.0.0.0, DNS server IP: 10.1.1.240

RedBoot(tm) bootstrap and debug environment [ROM]
Non-certified release, version UNKNOWN - built 17:42:36, Mar 18 2008

Platform: Cobra5329 (Freescale MCF5329)
Copyright (C) 2000, 2001, 2002, 2003, 2004 Free Software Foundation, Inc.
Copyright (C) 2003, 2004, 2005, 2006, 2007, 2008 eCosCentric Limited

RAM: 0x40000000-0x41000000, [0x4000c030-0x40fed000] available
FLASH: 0x00000000-0x00ffffff, 256 x 0x10000 blocks
RedBoot>
```

When RedBoot issues its prompt it is also ready to accept connections from m68k-elf-gdb, allowing eCos applications to be downloaded and debugged.

Occasionally it may prove necessary to update the installed RedBoot image. This can be done using the RAM startup build of `redboot.bin` rather than the dBUG version of RedBoot used above. It should be loaded using the ROM RedBoot's **load -r -m ymodem -b 0x40010000** and started with the **go** command. The ROM version can then be loaded into memory and programmed into flash as before.

Initial Installation Using BDM

Given a functional BDM setup, it is possible to run a RAM-resident version of RedBoot directly. This involves either m68k-elf-gdb or m68k-bdm-elf-gdb, together with the RAM build of `redboot.elf`, for example:

```
$ m68k-elf-gdb install/bin/redboot.elf
GNU gdb 6.4.50.20060226-cvs (eCosCentric)
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "--host=i686-pc-linux-gnu --target=m68k-elf"...
(gdb) target remote peedi:9000
0x00012086 in ?? ()
(gdb) load
Loading section .m68k_start, size 0x98 lma 0x40010000
Loading section .text, size 0x12a74 lma 0x40010098
Loading section .rodata, size 0x38e9 lma 0x40022b0c
Loading section .data, size 0xad4 lma 0x400263f8
Start address 0x40010000, load size 93897
Transfer rate: 280080 bits/sec, 3611 bytes/write.
(gdb) continue
Continuing.
```

RedBoot's output will be sent out of uart0 at 38400 baud, so an RS232 cable should be connected and a suitable terminal emulation application should be run. Once at the RedBoot prompt the procedure is the same as for the dBUG setup: initializing the flash, loading the ROM build of `redboot.bin` into memory, and then writing it to flash using **fis create**.

Initial Installation Using ecoflash

Given a functional BDM setup, the ecoflash utility can be used to program a RedBoot image directly into flash. Full information on ecoflash can be found elsewhere. It uses environment variables to determine the target hardware and how to access it. When using a PEEDI:

```
$ export ECOFLASH_BOARD=cobra5329
$ export ECOFLASH_TARGET='remote peedi:9000'
```

For m68k-elf-cfpe-stub a different remote address will need to be specified. When using m68k-bdm-elf-gdb:

```
$ export ECOFLASH_BOARD=cobra5329
$ export ECOFLASH_TARGET='bdm /dev/bdmcf0'
$ export ECOFLASH_GDB=m68k-bdm-elf-gdb
```

The **ecoflash info** can be used to verify that everything is working:

```
$ ecoflash info
Target board is cobra5329.
  gdb is "m68k-elf-gdb", gdb target is "remote localhost:9000".
  Target-side executable is version 1.
  Detected 1 bank of flash.
    Start 0x00000000, end 0x00ffffff -> 16384K.
      256 blocks of 64K.
  Flash block locking is not supported.
  Default program location for executables is 0x00000000.
  Target-side buffer for read and write operation is 64K.
```

If anything goes wrong then **ecoflash -v info** will run the same command in verbose mode, possibly providing additional information as to what is going wrong. Once everything is working a ROM startup build of RedBoot can be programmed into flash:

```
$ ecoflash program <path>/redboot.elf
Erasing 0x00000000 - 0x00016f6b
Writing 0x00000000 - 0x0000ffff (65536 bytes) from file "/tmp/redboot.elf.719", offset 0
Writing 0x00010000 - 0x00016f6b (28524 bytes) from file "/tmp/redboot.elf.719", offset 65536
```

When the board is next reset it should boot into RedBoot, with the output sent out of uart 0 at 38400 baud. A terminal emulator can then be used to connect to RedBoot and perform the **fis init** and **fconfig -i** initialization steps.

Rebuilding RedBoot

Should it prove necessary to rebuild a RedBoot binary, this is done most conveniently at the command line. The steps needed to rebuild the dBUG version of RedBoot are:

```
$ mkdir redboot_dbug
$ cd redboot_dbug
$ ecosconfig new cobra5329 redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/m68k/mcf52xx/mcf532x/cobra5329/<vsn>/misc/redboot_DBUG.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

At the end of the build the `install/bin` subdirectory should contain the required files `redboot.srec` and `redboot.bin`, as well as the ELF executable `redboot.elf`.

Rebuilding the RAM and ROM versions involves basically the same process. The RAM version uses the file `redboot_RAM.ecm` and generates a file `redboot_ram.bin`. The ROM version uses the file `redboot_ROM.ecm` and generates a file `redboot_rom.bin`.

Name

Configuration — Platform-specific Configuration Options

Overview

The Cobra5329 platform HAL package is loaded automatically when eCos is configured for a Cobra5329 target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware. The platform HAL is complemented by the M68K architectural HAL `CYGPKG_HAL_M68K`, the MCFxxxx variant HAL `CYGPKG_HAL_M68K_MCFxxxx`, and the MCF532x processor HAL `CYGPKG_HAL_M68K_MCF532x`.

Startup

The Cobra5329 platform HAL package supports three separate startup types: RAM, RAMBDM, DBUG and ROM. The configuration option `CYG_HAL_STARTUP`: controls which startup type is being used. For typical application development RAM startup should be used, and the application will be run via m68k-elf-gdb interacting with RedBoot using either serial or ethernet. It is assumed that the low-level hardware initialization, including setting up the memory map, has already been performed by RedBoot. By default the application will use certain services provided by RedBoot via the virtual vector mechanism, including diagnostic output, but that can be disabled via `CYGSEM_HAL_USE_ROM_MONITOR`. RAMBDM startup is a variant of RAM startup which has `CYGSEM_HAL_USE_ROM_MONITOR` disabled by default, and may be useful when debugging over BDM.

ROM startup can be used for applications which boot directly from flash. All the hardware will be initialized, and the application is self-contained. This startup type is used by the flash-resident version of RedBoot, and can also be used for finished applications.

DBUG startup can be used for applications which will be loaded via the dBUG ROM monitor rather than RedBoot. As with RAM startup it is assumed that the memory map has already been set up, but the application will not use any services provided by the ROM monitor. Typically this startup type is only used when setting up a board, as part of the process of replacing dBUG with RedBoot.

RedBoot and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building RedBoot.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained. That is useful as a testing step before switching to ROM startup. It also allows applications to be run and debugged via BDM.

Diagnostics

Diagnostics support is provided by the MCFxxxx variant HAL. For RAM startup the application will inherit its diagnostics channel from RedBoot: when debugging over ethernet diagnostics will travel over the TCP connection between **m68k-elf-gdb** and RedBoot; if RedBoot is set to discard its diagnostics then application diagnostics will be discarded as well; otherwise diagnostics will be sent over one of the uarts, typically uart0.

For other startup types the default diagnostics channel can be set to either of the uarts, or eCos can be configured to discard all diagnostics. The relevant configuration option is `CYGHWR_HAL_M68K_MCFxxxx_DIAGNOSTICS_PORT`.

Optional Hardware

The MCF532x processor HAL provides configuration options for the GPIO pin assignment registers, effectively controlling which of the on-chip peripherals are connected to the outside board. These settings are used in term to set the defaults for various devices, for example which uarts are available and whether or not the hardware handshake lines are connected. The platform HAL provides default setttings for all these registers appropriate for a Cobra5329 cpu module plugged into a standard carrier board, with no additional hardware. If the cpu module is plugged into a different carrier board or if extra hardware is hooked up to the various expansion sockets, the pin assignments can be changed as appropriate. Alternatively application can manipulate the processor's GPIO unit directly.

Cache and On-Chip RAM

The platform HAL contains configuration options for the values of the RAMBAR, CACR, ACR0 and ACR1 control registers. The first of these determines the location of on-chip RAM and does not usually need to be changed by application developers. The other three registers determine the caching behaviour. The default settings have the external flash cached in write-through mode, the external SDRAM cached in copyback mode, and everything else uncached. Normally it should be necessary to change these settings, and for a processor that can run at 240MHz it is important to make the best possible use of the cache. However, when debugging over BDM using hardware other than a Ronetix PEEDI it has been found that debug reliability could be improved somewhat when the cache was disabled, i.e. when the top bit of the CACR register was cleared. Obviously this is at the cost of greatly reduced performance.

System Clock

By default the platform HAL assumes a system clock running at 240/80MHz, 240MHz for the CPU and 80MHz for the peripherals. `CYGHWR_HAL_SYSTEM_CLOCK_HZ` corresponds to the peripheral clock speed. It is possible to run the board at a slower 180/60MHz by changing one of the switches in the configuration dip switch. If so then the configuration option must be changed for both RedBoot and the application, since it affects various I/O settings such as uart baud rates.

The port uses programmable interrupt timer PIT3 to implement the eCos system clock. By default this is set to tick once every 10ms, corresponding to a 100Hz clock. The frequency can be changed via the configuration option `CYGNUM_HAL_RTC_PERIOD`, which is used to program the timer's PIT Modulus Register (PMR3). Other clock-related settings are recalculated automatically if the period is changed.

Compiler Flags

The platform HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos. There are three flags specific to this port:

`-m528x`

The m68k-elf-gcc compiler supports many variants of the M68K architecture, from the original 68000 onwards. A `-m` option should be used to select the specific variant in use, and with current tools `-m528x` is the closest match for an MCF532x processor.

`-malign-int`

This option forces m68k-elf-gcc to align integer and floating point data to a 32-bit boundary rather than a 16-bit boundary. It should improve performance. However the resulting code is incompatible with most published application binary interface specifications for M68K processors, so it is possible that this option causes problems with existing third-party object code.

`-fomit-frame-pointer`

Traditionally the `%A6` register was used as a dedicated frame pointer, and the compiler was expected to generate link and unlink instructions on procedure entry and exit. These days the compiler is perfectly capable of generating working code without a frame pointer, so omitting the frame pointer often saves some work during procedure entry and exit and makes another register available for optimization. However without a frame pointer register the m68k-elf-gdb debugger is not always able to interpret a thread stack, so it cannot reliably give a backtrace. Removing `-fomit-frame-pointer` from the default flags will make debugging easier, but the generated code may be worse.

Chapter 331. Motorola MCF520x Processor Support

Name

CYGPKG_HAL_M68K_MCF520x — eCos Support for Freescale MCF520x Processors

Description

The Freescale MCF5207 and MCF5208 processors are part of the larger family of ColdFire processors. The two processors differ in the peripheral support: the MCF5208 has on-chip ethernet, the MCF5207 does not. The processor HAL package CYGPKG_HAL_M68K_MCF520x provides support for both processors, although at the time of writing it has only been tested on an MCF5208. It complements the M68K architectural HAL package CYGPKG_HAL_M68K and the variant HAL package CYGPKG_HAL_M68K_MCFxxxx. An eCos configuration should also include a platform HAL package to support board-level details like how the on-chip peripherals are connected to the outside world.

The package contains very little code. Instead it consists mainly of definitions, enabling appropriate code in the MCFxxxx variant HAL and in the various device drivers.

Configuration

The MCF520x processor HAL package should be loaded automatically when eCos is configured for appropriate target hardware. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Most of the package's configuration options relate to hardware. The settings are generally determined by the platform HAL and there is little need for application developers to change them. The first hardware option is CYGHWR_HAL_MCF520x_PROCESSOR, identifying the specific MCF520x processor being used. Legal values are MCF5207 and MCF5208. Typically the platform HAL will set this option via a CDL constraint.

Component CYGHWR_HAL_M68K_MCF520x_GPIO contains various options related to pin-connectivity. This gives full control over the PAR pin assignment registers, and for those pins configured as GPIO it is also possible to control the pin direction and data settings. These options are used to initialize the processor's GPIO module early on during system initialization, but applications may change settings later on as necessary.

The GPIO settings are used to determine default settings for the three on-chip uarts, the I²C bus and the QSPI bus. For example if none of the relevant GPIO pins are assigned to uart2 then component CYGHWR_HAL_M68K_MCFxxxx_UART2 will be disabled by default, and that uart cannot be used for HAL diagnostics nor accessed via the serial device driver. It is possible to override these settings if desired, for example if a uart is connected but will be manipulated directly by application code instead of via a device driver.

1. For each of the three on-chip uarts there will be a component, e.g. CYGHWR_HAL_M68K_MCFxxxx_UART0, determining whether or not the uart is usable on the target hardware. There are additional options CYGHWR_HAL_M68K_MCFxxxx_UART0_RTS and CYGHWR_HAL_M68K_MCFxxxx_UART0_CTS indicating whether or not the hardware handshake lines are connected, and CYGHWR_HAL_M68K_MCFxxxx_UART0_RS485_RTS to indicate that the RTS line controls an RS485 transceiver.

2. Component CYGHWR_HAL_M68K_MCF520x_I2C determines whether or not the processor HAL will instantiate an I²C bus device hal_mcfxxxx_i2c_bus. There are also options to control the interrupt priority and to set the FDR register which controls the bus speed. The default bus speed will be the standard I²C bus speed of 100KHz, or as close as can be achieved given hardware limitations.

3. Component CYGHWR_HAL_M68K_MCF520x_SPI determines whether or not the processor HAL will instantiate an SPI bus device hal_mcfxxxx_qspi_bus. It contains an additional configuration option for the interrupt priority.

For configurations which include the eCos kernel, CYGIMP_HAL_M68K_MCF520x_IDLE determines what happens when the idle thread runs. It should be noted that there are hardware errata associated with this functionality, and when systems were configured to use WAIT or DOZE mode problems were observed where the processor appeared to fail to wake up even though there were pending interrupts.

The HAL Port

This section describes how the MCF520x processor HAL package implements parts of the eCos HAL specification. It should be read in conjunction with similar sections from the architectural and variant HAL documentation.

HAL I/O

The header file `cyg/hal/proc_io.h` provides definitions of MCF520x-specific on-chip peripherals. Many of the on-chip peripherals are compatible with those on the MCF5282 or other ColdFire processors, and for those peripherals it is the `var_io.h` header provided by the MCFxxxx variant HAL which provides the appropriate definitions. Both headers are automatically included by the architectural header `cyg/hal/hal_io.h`, so typically application code and other packages will just include the latter.

Interrupt Handling

MCF520x processors implement standard ColdFire interrupt and exception handling, and come with one MCF5282-compatible interrupt controller and an edge port module. Therefore all interrupt and exception handling is left to the architectural and MCFxxxx variant HAL. The interrupt controllers are slightly enhanced relative to the MCF5282, with extra registers to facilitate masking and unmasking interrupts. These enhancements are supported. Unlike the MCF5282 interrupt priorities do not have to be unique, so valid interrupt priorities are in the range 1 to 6 corresponding to M68K IPL levels.

The processor's `cyg/hal/proc_intr.h` serves mainly to define symbols such as `CYGNUM_HAL_ISR_UART0`, mapping the MCF520x on-chip interrupt sources to the interrupt vectors.

Clock

Typically hardware timer PIT0 will be used for the eCos system clock, and that timer should not be manipulated directly by application code. If gprof-based profiling is enabled then that will use hardware timer PIT1. Some of the configuration options related to the system clock, for example `CYGNUM_HAL_RTC_PERIOD`, are actually contained in the platform HAL rather than the processor HAL. These options need to take into account the processor clock speed, a characteristic of the platform rather than the processor.

Caching

Support for the cache is provided via the generic HAL macros in the MCFxxxx variant HAL cache. The HAL also defines a macro `HAL_MEMORY_BARRIER()` which acts to synchronize the pipeline, delaying execution until all previous operations including all pending writes are complete. This will usually be necessary when interacting with devices that access memory directly.

Other Issues

The MCF520x processor HAL does not affect the implementation of data types, stack size definitions, SMP support, system startup, or debug support. The MCFxxxx variant HAL versions of `HAL_LSBIT_INDEX` and `HAL_MSBIT_INDEX` are used since the processor supports the `ff1.l` and `bitrev.l` instructions. `HAL_DELAY_US` is implemented as a simple counting loop. `HAL_IDLE_THREAD_ACTION` may be defined depending on configuration option `CYGIMP_HAL_M68K_MCF520x_IDLE`.

Other Functionality

The processor HAL will instantiate a cyg_i2c_bus structure `hal_mcfxxxx_i2c_bus` when the configuration option `CYGHWR_HAL_M68K_MCFxxxx_I2C` is enabled. That option is enabled by default if various GPIO pins are configured appropriately. The implementation is provided by the `CYGPKG_DEVS_I2C_MCFxxxx` device driver. The processor HAL does not know what I²C devices may be attached to the bus so that is left to the platform HAL.

The processor HAL will instantiate a cyg_spi_bus structure `hal_mcfxxxx_qspi_bus` when the configuration option `CYGHWR_HAL_M68K_MCFxxxx_SPI` is enabled. That option is enabled by default if various GPIO pins are configured appropriately. The implementation is provided by the `CYGPKG_DEVS_SPI_MCFxxxx_QSPI` device driver. The processor HAL does not know what SPI devices may be attached to the bus so that is left to the platform HAL. All SPI device structures should be placed in the table `mcfxxxx_qspi`.

Part LXXXII. MIPS Architecture

Table of Contents

Chapter 332. MIPS Architectural HAL

Name

`CYGPKG_HAL_MIPS` — eCos Support for the MIPS Architecture

Description

The MIPS architecture HAL provides support for all MIPS base processors. This includes both legacy devices, MIPS32 based devices, and Release 2 devices. Support is included for MIP16 and microMIPS instruction sets in addition to the MIPS32 instruction set.

This HAL contains support for CPU initialization, exception and interrupt entry and exit, thread context switching, interrupt masking, timer management, cache management and debugging.

Name

Options — Configuring the MIPS Architectural HAL Package

Description

The MIPS architectural HAL is included in all ecos.db entries for MIPS targets, so the package will be loaded automatically when creating a configuration. It should never be necessary to load the package explicitly or to unload it.

The MIPS architectural HAL contains a number of configuration points. Few of these should be altered by the user, they are mainly present for the variant and platform HALs to select different architectural features.

`CYGINT_HAL_MIPS_VARIANT`

This interface is implemented by each variant HAL and acts as a check on the number of variants included in the configuration.

`CYGPKG_HAL_MIPS_MIPS16_SUPPORT`

This option is usually set with a requires statement by the variant or platform HAL to enable support for MIPS16 applications. The eCos operating system and library remains in the MIPS32 instruction set.

`CYGINT_HAL_MIPS_MICROMIPS_SUPPORT`

This interface is implemented by the variant or platform HAL to enable support for the microMIPS instruction set.

`CYGPKG_HAL_MIPS_MICROMIPS_SUPPORT`

This option is usually enabled implicitly by the variant or platform HAL implementing `CYGINT_HAL_MIPS_MICROMIPS_SUPPORT`. Its value controls the level of support for the microMIPS instruction set. If set to `APP` then only application code may be compiled in the microMIPS instruction set, eCos remains in the MIPS32 instruction set. If set to `ECOS` then the whole of eCos is also compiled to the microMIPS instruction set, although those parts of the HAL involved in startup and exception handling remain in MIPS32 instructions. If set to `FULL` then all code is compiled or assembled to microMIPS. This latter option is only applicable to CPU variants that only execute microMIPS instructions and is not currently supported.

`CYGNUM_HAL_MIPS_RELEASE`

This option defines the architecture release that the processor supports. It is usually set by the variant or platform HAL to enable various configuration changes in the architecture HAL.

`CYGINT_HAL_MIPS_DEBUG_MODE`

This interface is implemented by the variant or platform HAL to enable support for Debug Mode. Debug Mode is only available on Release 2 devices that have EJTAG support. It is used primarily to support single step and breakpoints.

`CYGHWR_HAL_MIPS_CPU_FREQ`

This option contains the frequency of the CPU in MegaHertz. This will usually be set from the variant or platform HAL, which may define additional clock related options. This may affect thing like serial device, interval clock and memory access speed settings.

`CYGSEM_HAL_MIPS_EMULATE_UNIMPLEMENTED_FPU_OPS`

Enabling this option will include a hook in the exception processing so that Unimplemented Operation FPU exceptions may be handled. This option has no effect if there is no hardware floating-point unit. Note that not all situations in which an exception is raised may be handled. If not, the exception will be passed on as normal through the standard exception delivery mechanism.

`CYGINT_HAL_MIPS_STUB_REPRESENT_32BIT_AS_64BIT`

This interface may be implemented by MIPS variant or platform HALs to instruct the MIPS stub to interwork correctly with GDB which expects 64-bit register values, even in application code which has been compiled as 32-bit. Do not use this for real 64-bit code.

`CYGINT_HAL_MIPS_INTERRUPT_RETURN_KEEP_SR_IM`

On some MIPS variants, the status register (SR) contains a number of interrupt mask bits (IM\[0..7\]). Default behaviour is to restore the whole SR over an interrupt. This means that if the ISR modifies those bits, the change is lost when the interrupt returns. If this interface is implemented, changes made to the SR IM bits by an ISR will instead be preserved. Variants whose HAL_INTERRUPT_MASK() routines (et al) modify the IM bits in the SR should implement this interface to get the necessary preserving behaviour.

Redefinable Macros

In addition to the CDL configuration points above, there are a number of assembler macros that may be redefined. The assembler header `cyg/hal/arch.inc` contains default implementations of most of these. Variant or platform HALs may supply alternative definitions of these and define the matching preprocessor macro. For example, the default implementation of the `hal_intc_init` macro sets up the default interrupt mechanism using the Status register. By defining `CYGPKG_HAL_MIPS_INTC_INIT_DEFINED`, this macro may be redefined. Defining `CYGPKG_HAL_MIPS_INTC_DEFINED` allows all the `hal_intc_*` macros to be redefined. This same approach is applicable to most other macros or macro groups in `arch.inc`.

The same approach is also applicable to C-level macros for controlling interrupts, the timer, caches, bit indexing. A `..._DEFINED` macro is tested to determine whether a macro, or group of macros, have been defined in the variant or platform and if not then a default implementation is defined.

Compiler Flags

It is normally the responsibility of the platform HAL to define the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos.

Linker Scripts

The linker script, supplied by either the variant or platform HALs, must define some symbols that the architecture HAL depends on:

hal_vsr_table

This defines the location of the VSR table. First level interrupt and exception trampolines use the value of the Cause register ExcCode field to index this table and vector to a VSR routine. Generally this table should be placed in RAM close to the vector trampolines. This table should be 64 entries in length, although not all will be used.

hal_virtual_vector_table

This defines the location of the virtual vector table used to communicate between a ROM monitor and an eCos application. This table needs to be word aligned. It is usually placed in internal SRAM just after the VSR table, perhaps aligned to a convenient boundary. This table should be 64 entries in length.

Name

HAL Port — Implementation Details

Description

This documentation explains how the eCos HAL specification has been mapped onto the MIPS hardware and should be read in conjunction with the manuals for the processor in use. It should be noted that the architectural HAL is usually complemented by a variant HAL and a platform HAL, and those may affect or redefine some parts of the implementation.

Exports

The architectural HAL provides header files `cyg/hal/hal_arch.h`, `cyg/hal/hal_intr.h` and `cyg/hal/hal_io.h`. These header files export the functionality provided by all the MIPS HALs for a given target, automatically including headers from the lower-level HALs as appropriate. For example the platform HAL may provide a header `cyg/hal/plf_io.h` containing additional I/O functionality, but that header will be automatically included by `cyg/hal/hal_io.h` so there is no need to include it directly.

Additionally, the architecture HAL provides the `cyg/hal/basetype.h` header, which defines the basic properties of the architecture, including endianness, data type sizes and alignment constraints.

Startup

The conventional bootstrap mechanism involves the CPU starting execution at 0xBFC00000. Normally ROM or flash will be mapped here and a ROM startup RedBoot or application will be linked to start at this address. Some variants have an on-board boot ROM that runs at this address, and RedBoot or applications must be placed elsewhere in memory. In either case, execution must normally start at the `reset_vector` location in `vectors.S`.

The architectural HAL provides a default implementation of the low-level startup code which will be appropriate in nearly all scenarios. For a ROM startup this includes copying initialized data from flash to RAM. For all startup types it will involve zeroing bss regions and setting up the general C environment. It may also set up the exception trampolines in low memory, initializing CP0 registers, the memory controller, interrupt controller caches, timers, MMU and FPU, mostly by invoking variant or platform HAL defined macros. Depending on the variant and platform, some of these things are initialized in assembly code during startup, others may be initialized in later C code.

In addition to the setup it does itself, the initialization code calls out to the variant and platform HALs to perform their own initialization. Full initialization is handled by `hal_variant_init` and `hal_platform_init`. The former should handle any further initialization of the CPU variant and on-chip devices. The platform initialization routine will complete any initialization needed for devices external to the microprocessor.

The architectural HAL also initializes the VSR and virtual vector tables, sets up HAL diagnostics, and invokes C++ static constructors, prior to calling the first application entry point `cyg_start`. This code resides in `src/hal_misc.c`.

Interrupts and Exceptions

The eCos interrupt and exception architecture is built around a table of pointers to Vector Service Routines that translate hardware exceptions and interrupts into the function calls expected by eCos.

The vector table is either constructed at runtime or is part of the initialized data of the executable. For ROM, ROMRAM and JTAG startup all entries are initialized. For RAM startup only the interrupt VSR table entry is (re-)initialized to point to the VSR in the loaded code, the exception vectors are left pointing to the VSRs of the loading software, usually RedBoot or GDB stubs.

When an exception occurs it is delivered to a shared trampoline routine, `other_vector` which reads the Cause register, isolates the ExcCode field and uses it to index the VSR table and jump to the routine at the given offset. VSR table entries usually point to either `__default_exception_vsr` or `__default_interrupt_vsr`, which are responsible for delivering the exception or interrupt to the kernel.

The exception VSR, `__default_exception_vsr`, saves the CPU state on the thread stack, optionally switches to the interrupt stack and calls `cyg_hal_exception_handler()` to pass the exception on. Depending on the configuration, this routine then partly decodes the exception and passes it on for FPU emulation or exception handling, limited memory access errors, GDB stub exception handling or application level handling. When it finally returns the VSR jumps to code common with the interrupt VSR to restore the interrupted state and resume execution.

The interrupt VSR, `__default_interrupt_vsr`, saves the CPU state in the same way as the exception VSR, increments the scheduler lock and switches to the interrupt stack. It then calls two variant or platform supplied macros, `hal_intc_decode` and `hal_intc_translate` to query the interrupt controller for an interrupt number and then translate the interrupt number into an interrupt table offset. This offset is used to fetch an ISR from the interrupt handler table, and a data pointer from the interrupt data table, and the ISR is called to handle the interrupt. When the ISR returns, the stack pointer is switched back to the thread stack and `interrupt_end()` called. This may result in a thread context switch and the current thread may not resume for some time. When it does, the interrupted CPU state is restored from the thread stack and execution resumed from where it was interrupted.

The architectural HAL provides default implementations of `HAL_DISABLE_INTERRUPTS`, `HAL_RESTORE_INTERRUPTS`, `HAL_ENABLE_INTERRUPTS` and `HAL_QUERY_INTERRUPTS`. These involve manipulation of the Status register IE bit. Similarly there are default implementations of the interrupt controller macros `HAL_INTERRUPT_MASK`, `HAL_INTERRUPT_UNMASK` and `HAL_INTERRUPT_ACKNOWLEDGE` macros that manipulate the Status register IM bits. `HAL_INTERRUPT_SET_LEVEL` and `HAL_INTERRUPT_CONFIGURE` are no-ops at the architectural level. If a variant or platform contains an external interrupt controller, then it should redefine these macros to manipulate it.

Stacks and Stack Sizes

`cyg/hal/hal_arch.h` defines values for minimal and recommended thread stack sizes, `CYGNUM_HAL_STACK_SIZE_MINIMUM` and `CYGNUM_HAL_STACK_SIZE_TYPICAL`. These values depend on a number of configuration options.

The MIPS architecture HAL has the option of either using thread stacks for all exception and interrupt processing or implementing a separate interrupt stack. The default is to use an interrupt stack, since not doing so would require significantly larger per-thread stacks. This can be changed with the configuration option `CYGIMP_HAL_COMMON_INTERRUPTS_USE_INTERRUPT_STACK`.

Thread Contexts and setjmp/longjmp

`cyg/hal/hal_arch.h` defines a thread context data structure, the context-related macros, and the `setjmp`/`longjmp` support. The implementations can be found in `src/context.S`. The context structure is defined as a single structure used for all purposes: thread context, exceptions and interrupts. However, not all fields will be stored in all cases.

Bit Indexing

The architectural HAL provides inline assembler implementations of `HAL_LSBIT_INDEX` and `HAL_MSBIT_INDEX` which use algorithmic methods to extract a bit index in constant time. Variant HALs for later versions of the architecture can replace these with macros that use inline assembly to use CLZ or other instructions.

Idle Thread Processing

The architecture HAL provides a default `HAL_IDLE_THREAD_ACTION` implementation that simply spins. Variant and platform HALs can provide a replacement if required.

Clock Support

The architectural HAL provides a default implementation of the various system clock macros such as `HAL_CLOCK_INITIALIZE`. These macros use the architecture defined CP0 Count and Compare registers to implement the eCos system clock. The variant or platform HAL needs to define `CYGNUM_HAL_RTC_PERIOD` in terms of the frequency supplied to the Count register.

HAL I/O

The MIPS architecture does not have a separate I/O bus. Instead all hardware is assumed to be memory-mapped. Further it is assumed that all peripherals on the memory bus will switch endianness with the processor and that there is no need for any byte swapping. Hence the various HAL macros for performing I/O simply involve pointers to volatile memory.

The variant and platform files included by the `cyg/hal/hal_io.h` header will typically also provide details of some or all of the peripherals, for example register offsets and the meaning of various bits in those registers.

Cache Handling

The architecture HAL provides standard macros for dealing with both data and instruction caches. These macros make use of the CACHE instruction to affect cache contents. The architecture HAL does not provide support for enabling and disabling the caches, since there is no common mechanism for doing this; these must be implemented by the variant HAL.

Linker Scripts

The architecture HAL does not provide the main linker script, this must be supplied by the variant HAL and the makefile rules to generate the final `target.ld` must be included in the variant's CDL file.

Diagnostic Support

The architectural HAL does not implement diagnostic support. Instead this is left to the variant or platform HAL, depending on whether suitable peripherals are available on-chip or off-chip.

SMP Support

The MIPS architectural HAL does not provide any SMP support.

Debug Support

The architectural HAL provides basic support for gdb stubs using the debug monitor exceptions. Breakpoints may be implemented using a fixed-size list of breakpoints, as per the configuration option `CYGNUM_HAL_BREAKPOINT_LIST_SIZE`. When a JTAG device is connected to a MIPS device, it will steal breakpoints and other exceptions from the running code. Therefore debugging from RedBoot or the GDB stubs can only be done after detaching any JTAG debugger and power-cycling the board.

Debug support depends on the exact CPU model. Older parts, pre MIPS32R2, use the BREAK instruction for breakpoints and rely on instruction analysis to plant a breakpoint for single step for both MIPS32 and MIPS16 instruction sets. CPUs with debug mode use the SDBBP instructions for breakpoints and the Debug register SSt bit to implement single step for both MIPS32 and microMIPS instruction sets.

HAL_DELAY_US() Macro

`cyg/hal/hal_intr.h` provides a simple implementation of the `HAL_DELAY_US` macro based around reading the system timer. The timer must therefore be initialized before this macro is used, from either the variant or platform HAL initialization routines.

Profiling Support

The MIPS architectural HAL implements the `mcount` function, allowing profiling tools like gprof to determine the application's call graph. It does not implement the profiling timer. Instead that functionality needs to be provided by the variant or platform HAL.

Chapter 333. MIPS32 Variant HAL

Name

`CYGPKG_HAL_MIPS_MIPS32` — eCos Support for the MIPS32 Architecture Variant

Description

The MIPS32 variant HAL provides support for all MIPS32 base processors. This includes both legacy devices and Release 2 devices. It extends and modifies the generic architecture support provided by the architecture HAL to work with processors that conform to the MIPS32 specification.

Name

Options — Configuring the MIPS32 Variant HAL Package

Description

The MIPS32 variant HAL is included in all ecos.db entries for MIPS32-based targets, so the package will be loaded automatically when creating a configuration. It should never be necessary to load the package explicitly or to unload it.

The MIPS32 variant HAL contains a number of configuration points. Few of these should be altered by the user, they are mainly present for the platform HAL to select different features.

`CYGHWR_HAL_MIPS_MIPS32_CORE`

> This defines the CPU core on the target hardware. It is usually set by the platform HAL and may change the configuration of this HAL and of any changes it makes to the architecture HAL.

`CYGHWR_HAL_MIPS_MIPS32_ENDIAN`

> The MIPS32 core can use either a big or little endian mode. Platforms with a fixed endianness should set this to `Little` or `Big` as appropriate. Bi-endian platforms may allow this to be set by the user, or implicitly by supplying a platform configuration option to control endianness.

Compiler Flags

It is normally the responsibility of the platform HAL to define the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos. The endianness flags, `-EL` and `-EB` are controlled here by the `CYGHWR_HAL_MIPS_MIPS32_ENDIAN` option. The selection of the instruction set for eCos: MIPS32 or microMIPS, is also selected here based on various architecture options.

Name

MIPS32 HAL Port — Implementation Details

Description

This documentation explains how the eCos HAL specification has been mapped onto the MIPS hardware and should be read in conjunction with the manuals for the processor in use. It should be noted that the variant HAL is usually complemented by an architecture HAL and a platform HAL, and those may affect or redefine some parts of the implementation.

Exports

The variant HAL provides header files `cyg/hal/var_arch.h`, `cyg/hal/var_intr.h` and `cyg/hal/var_cache.h`. These header files export the functionality of this HAL to the architecture HAL, and are included automatically by the architecture HAL where needed.

Additionally, the variant HAL provides the `cyg/hal/variant.inc` header, which is included into the architecture assembly code and redefines some macros used there, to configure the architecture to the MIPS32 variant.

Startup

The MIPS32 HAL does no additional initialization beyond the redefined assembly macros mentioned above. As a purely CPU-based variant there are no additional variant-specific devices to be initialized.

Interrupts and Exceptions

The MIPS32 HAL defines some additional exception vectors that are present in MIPS32 variants. Otherwise it takes no part in interrupt or exception delivery.

Bit Indexing

The MIPS32 HAL provides replacement implementations of the `HAL_LSBIT_INDEX` and `HAL_MSBIT_INDEX` macros that use inline assembly CLZ instructions.

Cache Handling

The main contribution that the MIPS32 HAL makes is in cache handling. It provides cache handling macros that extend and modify those supplied in the architecture HAL. It also implements cache enable and disable support via the K0 field of the Config0 register. It also provides default cache dimension declarations, as well as a mechanism for these to be defined by the platform HAL.

Linker Scripts

The MIPS32 HAL provides the main linker script for use by all platforms. The MIPS32 HAL will generate the linker script for eCos applications. This involves the file `src/mips_mips32.ld` and a `.ldi` memory layout file, typically provided by the platform HAL. It is the `.ldi` file which places code and data in the appropriate places for the startup type, but most of the hard work is done via macros in the `mips_mips32.ld` file.

Chapter 334. MIPS SEAD3 Board Support

Name

eCos Support for the MIPS SEAD3 Board — Overview

Description

The MIPS SEAD3 board comes in three variants. The LX50 is fitted with a M14K processor and no SDRAM. The LX110 is fitted with a M14Kc processor and 512MiB of SDRAM. The LX155 is virtually identical to the LX110, but is fitted with a larger FPGA. From the point of view of eCos, it is considered identical to the LX110 and all references in this documentation and in eCos to the LX110 should be considered to include the LX155. All boards have 4MiB of SRAM and 32MiB of NOR flash, a dual 16550 compatible UART, and a SMSC LAN9211 ethernet controller. A two line LCD display and LEDs are also present. For typical eCos development a RedBoot image is programmed into the external flash. RedBoot provides gdb stub functionality so it is then possible to download and debug eCos applications via the gdb debugger. This can happen over either a serial line or over ethernet. It is also possible to debug applications using the JTAG interface.

Supported Hardware

The memory map used by both eCos and RedBoot is as follows:

Memory	Base	Length
SDRAM	0x80000000	0x20000000
SRAM	0x80000000	0x00400000
User Flash	0x9c000000	0x02000000
SRAM Shadow	0x8E000000	0x00400000
Interrupt Controller (if present)	0xBB1C0000	0x00020000
Peripherals	0xBF000000	0x00010000
SMSC LAN9211	0xBF010000	0x00010000
Boot Flash	0xBFC00000	0x00600000

On the LX110/LX155 SDRAM is mapped to physical address 0x00000000 and SRAM is visible only at the shadow location. On the LX50 SRAM is mapped to physical location 0x00000000 as well as the shadow location. SDRAM, SRAM and flash are normally accessed via the kseg0 segment and hence via the cache. The peripherals are normally accessed via kseg1 and hence uncached.

eCos can be configured for one of three startup types:

RAM

> This is the startup type normally used during application development. RedBoot is programmed into flash and performs the initial bootstrap. mip-sde-elf-gdb is then used to load a RAM startup application into memory and debug it. By default the application will use eCos' virtual vectors mechanism to obtain certain services from RedBoot, including diagnostic output. The bottom 1MB of RAM is used for RedBoot code and data so the application will start at 0x80100000.

ROM

> This startup type can be used for finished applications which will be programmed into the start of external flash at location 0xbfc00000. On power-up the processor will automatically execute the contents of flash from 0xbfc00000. The application will initialize the system, copy its data to RAM and zero its BSS. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization. This startup type is used for building the flash-resident version of RedBoot but can also be used for application code.

ROMRAM

This startup type can be used for finished applications which will be programmed into the start of external flash at location 0xbfc00000. On power-up the processor automatically execute the contents of flash from 0xbfc00000. The application will initialize the system, copy itself from flash to RAM, and zero its BSS. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization. This startup type is used for building the flash-resident version of RedBoot but can also be used for application code.

JTAG

This can be used to run applications via JTAG rather than RedBoot. The application will be loaded at location 0x80000000 and it will take over all the hardware. Uart0 will be used for all HAL diagnostics and standard output. A JTAG application build may prove useful for debugging certain problems, especially ones related to interrupts and exceptions. However the JTAG software may not fully cope with the executables and debug information generated by the GNU tools, so the user experience may be poor compared with using the GNU mips-sde-elf-gdb debugger.

In a typical setup RedBoot is programmed into the boot flash, which eCos does not manage. The last 256KiB of User flash is used for managing the flash and holding the RedBoot fconfig values. The remaining blocks from 0x9c000000 to 0x9dfbffff can be used by application code.

RedBoot can communicate with the host using either uart0 or ethernet.

All configurations for the SEAD3 target include an ethernet driver package `CYGPKG_DEVS_ETH_SMSC_LAN9118`. If the application does not actually require ethernet functionality then the package is inactive and the final executable will not suffer any overheads from unused functionality. This is determined by the presence of the generic ethernet I/O package `CYGPKG_IO_ETH_DRIVERS`. Typically the choice of eCos template causes the right thing to happen. For example, the default template does not include any TCP/IP stack so `CYGPKG_IO_ETH_DRIVERS` is not included, but both the net and redboot templates do include a TCP/IP stack so will specify that package and hence enable the ethernet driver. The ethernet device can be shared by RedBoot and the application, so it is possible to debug a networked application over ethernet.

The SEAD3 board has a serial EPROM providing a unique network MAC address.

All configurations for the SEAD3 target include serial device driver packages `CYGPKG_IO_SERIAL_GENERIC_16X5X` and `CYGPKG_IO_SERIAL_MIPS_SEAD3`. The `16X5X` driver provided generic support for 16X5X compatible UARTs while the `SEAD3` package provides configuration to adapt that driver to the SEAD3 board. The driver as a whole is inactive unless the generic serial support, `CYGPKG_IO_SERIAL_DEVICES` is enabled. Both UART0 and UART1 are connected, and both have hardware flow control lines routed to the connector. UART0 is routed to a standard 9-pin RS232 connector and UART1 is connected to an FTDI USB adaptor. If a UART is needed by the application then it cannot also be used by RedBoot for gdb traffic, so care should be exercised in selecting which UART to use for these purposes. Alternatively another communication channel such as ethernet should be used instead.

The GIC interrupt controller is managed by eCos using macros provided by the SEAD3 platform HAL. The architecture COUNTER/COMPARE timer is used to implement the eCos system clock. If gprof-based profiling is enabled then that will use the GIC compare register and its associated interrupt. If the core does not include a GIC, then the configuration option `CYGHWR_HAL_SEAD3_HAS_GIC` must be disabled, and gprof profiling will not be possible. At the moment, without a GIC additional interrupt decoding is performed to indicate which of the two UART devices generated an interrupt, as these are multiplexed in the same interrupt vector (2). However decoding is not performed on the multiplexed peripherals on interrupt vector 0: PIC32 GPIO, SPI or USB as these peripherals are as yet not supported in eCosPro.

Tools

The SEAD3 port is intended to work with GNU tools configured for an mips-sde-elf target. The original port was done using mips-sde-elf-gcc version 4.4 mips-sde-elf-gdb version 6.8, and binutils version 2.19.

Name

Setup — Preparing the SEAD3 board for eCos Development

Overview

In a typical development environment the SEAD3 board boots from flash into the RedBoot ROM monitor. eCos applications are configured for a RAM startup, and then downloaded and run on the board via the debugger mips-sde-elf-gdb. Preparing the board therefore involves programming a suitable RedBoot image into flash memory.

The following RedBoot configurations are supported:

Configuration	Description	Use	File
ROMRAM	RedBoot loaded from boot flash, running in RAM	`redboot_ROMRAM.ecm`	`redboot_romram.bin`
ROM	RedBoot running from boot flash	`redboot_ROM.ecm`	`redboot_rom.bin`
JTAG	RedBoot debug	`redboot_JTAG.ecm`	`redboot.elf`

For serial communications all versions run at 115200 baud with 8 bits, no parity, and 1 stop bit. The baud rates can be changed via the **baud** RedBoot command. RedBoot will support communication on both UARTs. Ethernet communication and flash management are also supported.

Switch Setting

The LX50, LX110 and LX155 boards need the SW1 and SW2 switch banks set in order to execute code properly. These are mostly to overcome problems in the hardware or FPGA firmware.

The LX50 needs to run at 50MHz, and the SDRAM must be mapped to zero, so the switches are set as follows:

Switch	Setting
SW1[1]	ON
SW1[2]	OFF
SW1[3]	OFF
SW1[4]	ON
SW1[5]	OFF
SW1[6]	OFF
SW1[7]	OFF
SW1[8]	OFF
SW2[1]	OFF
SW2[2]	OFF
SW2[3]	ON
SW2[4]	OFF

The LX110 and LX155 also need to run at 50MHz, but the SDRAM does not need to be mapped to zero, so the switches are set as follows:

Switch	Setting
SW1[1]	ON

Switch	Setting
SW1[2]	OFF
SW1[3]	OFF
SW1[4]	ON
SW1[5]	OFF
SW1[6]	OFF
SW1[7]	OFF
SW1[8]	OFF
SW2[1]	OFF
SW2[2]	OFF
SW2[3]	OFF
SW2[4]	OFF

These settings are the defaults for the boards as delivered from MIPS and should generally not be altered.

Initial Installation

RedBoot is installed using the USB download mechanism available on the board. The reader is referred to the SEAD3 documentation for a full description; however, the following steps should suffice to install RedBoot.

Connect a USB cable between your host machine and the SEAD3 board's USB download connector.

If your host system is Windows based open "Printers and Faxes" in the Start Menu. Click on "Add a printer". Then click "Next". Select "Local printer attached to this computer". Uncheck the box that says "Automatically detect and install my Plug and Play printer". Click "Next". Select a printer port. Click on "Use the following port", and select USB001 (virtual printer port for USB). If you have previously installed a USB printer, you may see more than one USBxxx choice. You must choose the one associated with the port connected to your USB cable. If necessary, use trial and error. When you have finished click "Next". Under "Manufacturers", select "Generic". Under "Printers", select "Generic/Text Only". Click "Next".

Locate the file `redboot_ROMRAM.fl` and load into **WordPad**; make sure "Word Wrap" is turned OFF and that "Print page numbers" is unchecked in "Print Setup". Print the document to the printer set up above.

For Linux users the file can be copied to the printer using a command similar to either of the following:

-
  ```
      % cat redboot_ROMRAM.fl >/dev/usblp0
  ```

-
  ```
      % cat redboot_ROMRAM.fl >/dev/usb/lp0
  ```

It may be necessary to execute this as root, depending on the permissions on the printer device.

Disconnect the USB cable and either connect a serial cable between the board and your host or plug the USB cable into the UART1 USB adaptor socket. Run a terminal emulator (HyperTerminal, TeraTerm or minicom) at 115200 baud attached to the serial port. Power cycle the SEAD3 board and you should see output similar to this:

```
+**Warning** FLASH configuration checksum error or invalid key
Use 'fconfig -i' to [re]initialize database
Ethernet eth0: MAC address 00:d0:a0:00:08:1c
IP: 10.0.2.4/255.0.0.0, Gateway: 10.0.0.3
Default server: 0.0.0.0
DNS server IP: 10.0.0.1, DNS domain name: <null>

RedBoot(tm) bootstrap and debug environment [ROMRAM]
Non-certified release, version UNKNOWN - built 14:40:05, Apr 23 2010
```

```
Copyright (C) 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009
Free Software Foundation, Inc.
Copyright (C) 2003, 2004, 2005, 2006, 2007, 2008 eCosCentric Limited
RedBoot is free software, covered by the eCos license, derived from the
GNU General Public License. You are welcome to change it and/or distribute
copies of it under certain conditions. Under the license terms, RedBoot's
source code and full license terms must have been made available to you.
Redboot comes with ABSOLUTELY NO WARRANTY.

Platform: LX110 (M14Kc) LE
RAM: 0x80000000-0x9b000000 [0x8000b8b0-0x9afbd000 available]
FLASH: 0x9c000000-0x9dffffff, 128 x 0x40000 blocks
RedBoot>
```

The exact details may vary slightly, depending on whether or not the ethernet is plugged in yet. If no ethernet cable in plugged in there may be a delay before this output completes. At this stage the RedBoot flash management initialization has not yet happened so the warning about the configuration checksum error is expected. To perform the flash initialization use the **fis init -f** command:

```
RedBoot> fis init
About to initialize [format] FLASH image system - continue (y/n)? y
*** Initialize FLASH Image System
... Unlocking from 0x9dfc0000-0x9dffffff: .
... Erase from 0x9dfc0000-0x9dffffff: .
... Program from 0x9afc0000-0x9b000000 to 0x9dfc0000: .
... Locking from 0x9dfc0000-0x9dffffff: .
RedBoot>
```

At this stage the block of flash at location 0x9dFc0000 holds information about the various flash blocks, allowing other flash management operations to be performed. The next step is to set up RedBoot's non-volatile configuration values:

```
RedBoot> fconfig -i
Initialize non-volatile configuration - continue (y/n)? y
Run script at boot: false
Use BOOTP for network configuration: true
Default server IP address: 10.0.1.1
Console baud rate: 115200
DNS domain name: xxxxxxx.com
DNS server IP address: 10.0.0.1
GDB connection port: 9000
Force console for special debug messages: false
Network debug at boot time: false
Update RedBoot non-volatile configuration - continue (y/n)? y
... Unlocking from 0x9dfc0000-0x9dffffff: .
... Erase from 0x9dfc0000-0x9dffffff: .
... Program from 0x9afc0000-0x9b000000 to 0x9dfc0000: .
... Locking from 0x9dfc0000-0x9dffffff: .
RedBoot>
```

For most of these configuration variables the default value is correct. If there is no suitable BOOTP service running on the local network then BOOTP should be disabled, and instead RedBoot will prompt for a fixed IP address, netmask, and addresses for the local gateway and DNS server.

Initialization is now complete. Press the board reset button and the following output should be seen:

```
+Ethernet eth0: MAC address 00:d0:a0:00:08:1c
IP: 10.0.2.4/255.0.0.0, Gateway: 10.0.0.3
Default server: 10.0.1.1
DNS server IP: 10.0.0.1, DNS domain name: <null>

RedBoot(tm) bootstrap and debug environment [ROMRAM]
Non-certified release, version UNKNOWN - built 14:40:05, Apr 23 2010

Copyright (C) 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009
Free Software Foundation, Inc.
Copyright (C) 2003, 2004, 2005, 2006, 2007, 2008 eCosCentric Limited
```

```
RedBoot is free software, covered by the eCos license, derived from the
GNU General Public License. You are welcome to change it and/or distribute
copies of it under certain conditions. Under the license terms, RedBoot's
source code and full license terms must have been made available to you.
Redboot comes with ABSOLUTELY NO WARRANTY.

Platform: LX110 (M14Kc) LE
RAM: 0x80000000-0x9b000000 [0x8000b8b0-0x9afbd000 available]
FLASH: 0x9c000000-0x9dffffff, 128 x 0x40000 blocks
RedBoot>
```

When RedBoot issues its prompt it is also ready to accept connections from **mips-sde-elf-gdb**, allowing eCos applications to be downloaded and debugged.

Occasionally it may prove necessary to update the installed RedBoot image. This can be done by repeating this process.

Rebuilding RedBoot

Should it prove necessary to rebuild a RedBoot binary, this is done most conveniently at the command line. The steps needed to rebuild the ROMRAM version of RedBoot are:

```
$ mkdir redboot_romram
$ cd redboot_romran
$ ecosconfig new sead3_14kc redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/mips/sead3/current/misc/redboot_ROMRAM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

At the end of the build the `install/bin` subdirectory should contain the required file `redboot_romram.fl`. This builds RedBoot for the LX155/M14Kc or LX110/M14Kc boards; to build for the LX50/M14K board substitute sead3_14k for sead3_14kc in the above.

Rebuilding the ROM and JTAG versions involves basically the same process. The ROM version uses the file `redboot_ROM.ecm` and generates an ELF executable `redboot.elf`, which will be automatically converted to `redboot_romram.fl` for flash programming. The JTAG version uses the file `redboot_JTAG.ecm` and generates an ELF executable `redboot.elf`, which may need to be converted to another format before it can be used with the JTAG software.

Note

The program that creates the `redboot_romram.fl` file is a perl script, `srecconv.pl`, that requires the `perl` command to be in your path. While this is normally available under Linux, Windows users may be required to install perl on their machine to create `redboot_romram.fl`.

Name

Configuration — Platform-specific Configuration Options

Overview

The SEAD3 platform HAL package is loaded automatically when eCos is configured for an SEAD3 target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The SEAD3 platform HAL package supports four separate startup types: RAM, ROM, ROMRAM and JTAG. The configuration option `CYG_HAL_STARTUP` controls which startup type is being used. For typical application development RAM startup should be used, and the application will be run via mips-sde-elf-gdb interacting with RedBoot using either serial or ethernet. It is assumed that the low-level hardware initialization, including setting up the memory map, has already been performed by RedBoot. By default the application will use certain services provided by RedBoot via the virtual vector mechanism, including diagnostic output, but that can be disabled via `CYGSEM_HAL_USE_ROM_MONITOR`.

ROM startup can be used for applications which are programmed into the boot flash at 0xbfc00000. On power up the processor will jump to this location and execute the code that is there. The startup code will copy the applications data segment from ROM to RAM at 0x80000000 and zero the BSS. Code execution will continue from ROM. All the hardware will be initialized, and the application is self-contained. This startup type can be used by the flash-resident version of RedBoot, and can also be used for finished applications that run stand-alone.

ROMRAM startup can be used for applications which are programmed into the boot flash at 0xbfc00000. On power up the processor will jump to this location and execute the code that is there. The startup code will copy the applications taxt and data segments from ROM to RAM at 0x80000000 and zero the BSS. Code execution will continue from RAM. All the hardware will be initialized, and the application is self-contained. This startup type can be used by the flash-resident version of RedBoot, and can also be used for finished applications that run stand-alone.

JTAG startup can be used for applications which will be debugged via JTAG instead of RedBoot. The behaviour is a combination of ROM and RAM startup: the application is loaded at 0x80000000 and initializes all the hardware, with no dependencies on services provided by a ROM monitor.

RedBoot and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building RedBoot.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained. That is useful as a testing step before switching to ROM startup. It also allows applications to be run and debugged via JTAG.

If the application does not rely on a ROM monitor for diagnostic services then UART0 will be used for HAL diagnostics and standard output. The default baud rate is controlled by `CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL_BAUD`. If UART0 is needed by the HAL diagnostics code it cannot be accessed via the serial driver and applications should be loaded via ethernet. Diagnostic output can also be switched to using UART1 by setting `CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL` to 1.

System Clock

The coprocessor 0 COUNTER/COMPARE timer is used for the eCos system clock. The configuration option `CYGNUM_HAL_RTC_PERIOD` controls the value programmed into the compare register. The value of this is calculated from the CPU frequency and the value of `CYGNUM_HAL_RTC_DENOMINATOR`. The calculations are arranged so that

CYGNUM_HAL_RTC_DENOMINATOR effectively defines the clock frequency and it the only configuration option that need to be changed to select a different clock rate.

Board Type Selection

The option CYGHWR_HAL_MIPS_SEAD3 selects the SEAD3 board variant. It may be set to either LX50 or LX110. Normally this will be set automatically when selecting for the sead3_14k or sead3_14kc targets.

Endian Mode Selection

The option CYGHWR_HAL_MIPS_SEAD3_ENDIAN selects the CPU and peripheral endian mode. It may be set to either Little or Big. The default is Little. The BIGEND switch, SW2[1], needs to be set to match and if RAM applications are to be used a matching RedBoot must also be installed. Note also that the RedBoot flash directory and configuration are not stored in an endian-independent manner and would need to be reinitialized.

Instruction Set Selection

The option CYGPKG_HAL_MIPS_MICROMIPS_SUPPORT selects instruction set support. It may be set to either APP, ECOS or FULL. The default is APP which causes eCos and RedBoot to be compiled in the MIPS32 instruction set, but allows user code to be compiled in the microMIPS instruction set. When set to ECOS then all eCos/RedBoot C and C++ code is compiled into microMIPS instructions; however assembly level startup and exception handling code remains in MIPS32 instructions, as it must since the CPU starts in this instruction set, and switches to it for all exceptions. The FULL option is present for future devices that operate entirely in microMIPS mode, and is not currently supported.

Unlike endian mode, the instruction set selection of RedBoot and eCos applications need not match. A MIPS32 RedBoot can load and run microMIPS eCos applications and a microMIPS RedBoot can load and run MIPS32 applications.

To compile applications into microMIPS the standard flags that are used in eCos and exported to the ecos.mak file should be used except that the -mips32r2 flag should be replaced by -mmicromips and the options -mno-jals -minterlink-mips16 added.

Flash Driver

The platform HAL package contains flash driver support for the user flash device. By default this is inactive, and it can be made active by loading the generic flash package CYGPKG_IO_FLASH. The boot flash is not programmable at runtime.

Ethernet Support

The platform HAL provides the platform-specific support for a single SMSC LAN9211 ethernet device, if the generic ethernet support is enabled. The MAC address is stored in an EEPROM connected to the LAN9211 and will be loaded into the MAC automatically on reset.

Compiler Flags

The platform HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos. There are some flags specific to this port:

-mips32r2

> The mips-sde-elf-gcc toolchain defaults to supporting the mips32 release 2 architecture, so this option is not strictly necessary. However, it is good practice to include it.

`-mmicromips`

To compile code in the microMIPS instruction set, this option must be substituted for `-mips32r2`.

`-mno-jals` `-minterlink-mips16`

These options are necessary to link MIPS32 code with microMIPS code. They both restrict the compiler to generating instructions for calls and other control transfers that can be converted to instructions that switch the instruction set. Without these options some instructions either cannot be converted, or will be converted incorrectly.

`-G0`

MIPS calling conventions reserve one register for use as a global pointer register. In theory this allows static variables in one 64K area of memory to be accessed using just one instruction instead of two, and the `-G` option provides some control over this. However due to limitations within the current linker all modules have to be compiled with the same `-G` setting, and the compiler support libraries are built with `-G0`. Therefore all eCos and application modules also have to be built with `-G0` and this optimization is not available.

`-EB, EL`

The eCos port supports both big-endian and little-endian modes.

`-msoft-float`

The M14K family does not have a hardware floating point unit so software floating point has to be used instead.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the SEAD3 hardware, and should be read in conjunction with that specification. The SEAD3 platform HAL package complements the MIPS architectural HAL, the MIPS32 variant HAL It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset the HAL will set up some of the peripherals (UART, GIC, LCD etc.) appropriately for eCos, but other peripherals (USB, PIC32) are left to their default settings. Full details of this initialization can be found in the function `hal_platform_init` in `sead3.c`.

Memory Map

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

SDRAM

On the LX110/LX155 this is 512MiB in size and mapped to location 0x80000000. The first four kilobytes are reserved for exception vectors and for data which needs to be shared between RAM startup applications and the ROM monitor. For JTAG startup, code will be loaded from 0x80001000 onwards. For ROM startup data will be loaded from 0x80001000 onwards. For ROMRAM startup, the text and data of the applications will be loaded from 0x80001000 onwards. For RAM startup, code will start at 0x80040000, reserving the bottom 256KiB for RedBoot's code and data. This memory is not present on the LX50.

SRAM

On the LX50 this is 4MiB in size and is mapped to location 0x80000000. The first four kilobytes are reserved for exception vectors and for data which needs to be shared between RAM startup applications and the ROM monitor. For JTAG startup, code will be loaded from 0x80001000 onwards. For ROM startup data will be loaded from 0x80001000 onwards. For ROMRAM startup, the text and data of the applications will be loaded from 0x80001000 onwards. For RAM startup, code will start at 0x80040000, reserving the bottom 256KiB for RedBoot's code and data. This memory is present on the LX110/LX155, where it is mapped to 0x1E000000 and is unused by eCos.

On-chip Peripherals

These are accessible at location 0xBF000000 onwards.

Flash

The user flash device is located at 0x9C000000 onwards. CFI is used at run-time to query the flash chip and adapt to it. Redboot is not held in this flash device, but in a separate boot flash that is not accessible at run time. The last user flash block at location 0x9DFC0000 is used to hold flash management data and the RedBoot **fconfig** variables. The remaining blocks can be used by application code.

Clock Support

The platform HAL provides configuration options for the eCos system clock. This always uses the architectural COUNTER/COMPARE registers accessed via coprocessor 0. The actual HAL macros for managing the clock are provided by the MIPS architectural HAL. The specific numbers used are a characteristic of the platform because they depend on the processor speed. If the interrupt controller is present, its compare register and interrupt is used for profiling.

Other Issues

The SEAD3 platform HAL does not affect the implementation of other parts of the eCos HAL specification.

Name

JTAG Debugging — Using System Navigator

Overview

The SEAD 3 board can be debugged using the System Navigator JTAG probe. This consists of the probe itself plus the Navigator Console software. The Navigator Console should be installed onto your host system according to the "MIPS Navigator Console Getting Started Guide". To do this you need to obtain a license from MIPS. For Linux users, the Navigator Console version 4.0.16 currently only installs on RedHat Enterprises Linux. Once the console software is installed, connect the System Navigator to the host.

Setup for standard eCosPro GNU tools

To use the System Navigator from GDB, the Navigator Console must be started independently. The Navigator Console must be supplied with a startup script which can be chosen at startup, or supplied on the command line. Choose either `mips_m14k.tcl` for the LX150, or `mips_m14kc.tcl` for the LX110/LX155. When started correctly, the console window will appear and show something similar to the following:

```
Main console display active (Tcl8.5.6 / Tk8.5.6)
mips_m14k initialization successful.
(scripts) 1 %
```

In addition, for Linux you should add the path to the Navigator Console binaries directory to your library load path. For example, add the following line to your `.bashrc`:

```
export LD_LIBRARY_PATH=~/MIPS/NavigatorConsole/bin:$LD_LIBRARY_PATH
```

Download and install the Sourcery CodeBench Lite tools for MIPS ELF. It is sufficient to just download the TAR archive as this download is solely required to obtain the **mips-sde-elf-sprite** tool. Place the "bin" directory for those tools (containing **mips-sde-elf-sprite**) in your shell's PATH environment variable, making sure it comes *after* the PATH component for the eCosPro GNU tools so that the eCosPro tools are still preferred.

Connect to the target from GDB by using the following connection command at the GDB console:

```
(gdb) target remote | mips-sde-elf-sprite -q -a 'mdi:/1/1?rst=5&lib=/home/USER/MIPS/NavigatorConsole/bin/libsysnav_mdi
```

Substitute in the correct location for the `libsysnav_mdi.so` as required.

Consult the Navigator Console documentation `NavConGdbGuide.pdf` for further guidance on use with GDB.

Setup for CodeSourcery tools

While eCosCentric no longer recommend using the CodeSourcery tools (other than to obtain the **mips-sde-elf-sprite** tool), this documentation has been preserved in case some users decide they do wish to use those tools.

To use the System Navigator from GDB, the Navigator Console must be started independently. The Navigator Console must be supplied with a startup script which can be chosen at startup, or supplied on the command line. Choose either `mips_m14k.tcl` or `mips_m14kc.tcl` for the LX50 and LX110 respectively. When started correctly, the console window will appear and show something similar to the following:

```
Main console display active (Tcl8.5.6 / Tk8.5.6)
mips_m14k initialization successful.
(scripts) 1 %
```

To use the System Navigator from GDB, GDB must be supplied with the name of a dynamic library to load, and the location to load it from. This is best done from a `.gdbinit` file. For Window this should contain:

```
set mdi library C:\MIPS\NavigatorConsole\bin\sysnav_mdi.dll
set mdi target 1
set mdi connectreset 7
```

And for Linux:

```
set mdi library libsysnav_mdi.so
set mdi connectreset 7
set mdi target 1
```

In addition, for Linux you should add the path to the Navigator Console binaries directory to your library load path. For example, add the following line to your `.bashrc`:

```
export LD_LIBRARY_PATH=~/MIPS/NavigatorConsole/bin:$LD_LIBRARY_PATH
```

With these files set up it should be possible to start GDB and connect by giving the following command:

```
(gdb) target mdi 1:1
Selected device jtagindex-0 on MIPS mips_m14k
Connected to MDI target
(gdb)
```

GDB should now be ready to download and debug a JTAG startup application.

Chapter 335. MIPS Malta Board Support

Name

eCos Support for the Malta Board — Overview

Description

This document covers the MIPS Malta single board computer based on the MIPS 4Kc, 4KEc and 5Kc processors. Suport for the 5Kc is restricted to RedBoot only, however, all 4Kc configurations of eCos and RedBoot will also function on a 5Kc.

The Malta board contains the processor, 32Mb of RAM, 4MB of flash memory, an AMD Am79C973 PCnet ethernet MAC, connections for two serial channels and the various other peripherals on the board.

For typical eCos development, a RedBoot image is programmed into the flash memory, and the board will boot this image from reset. RedBoot provides gdb stub functionality so it is then possible to download and debug stand-alone and eCos applications via the gdb debugger. This can happen over either a serial line or over ethernet.

Supported Hardware

The flash memory consists of two Intel 28F160 devices in parallel, giving 32 blocks of 128k bytes each. In a typical setup, the first two flash blocks are used for the ROM RedBoot image. The topmost block is used to manage the flash and hold RedBoot **fconfig** values. The remaining 29 blocks between 0xbe040000 and 0xbe3dffff can be used by application code.

There is a serial driver `CYGPKG_IO_SERIAL_GENERIC_16X5X` which supports the two UART serial devices on the Malta board. This is configured for the Malta by the `CYGPKG_IO_SERIAL_MIPS_MALTA` package. These devices can be used by RedBoot for communication with the host. If either of these devices is needed by the application, either directly or via the serial driver, then it cannot also be used for RedBoot communication. Another communication channel such as ethernet should be used instead. The serial driver package is loaded automatically when configuring for the Malta target.

There is an ethernet driver `CYGPKG_DEVS_ETH_AMD_PCNET` for the AMD Am79C973 PCnet ethernet device. A second package `CYGPKG_DEVS_ETH_MIPS_MIPS32_MALTA` is responsible for configuring this generic driver to the Malta hardware. These drivers are also loaded automatically when configuring for the Malta target.

eCos manages the on-chip interrupt controller. The MIPS32 architectural Count and Compare registers are used to implement the eCos system clock and the microsecond delay function. Other devices (Caches, PCI, UARTs, memory and interrupt controllers) are initialized only as far as is necessary for eCos to run. Other devices are not touched.

Tools

The Malta port is intended to work with GNU tools configured for an mipsisa32-elf target. The original port was undertaken using mipsisa32-elf-gcc version 3.2.1, mipsisa32-elf-gdb version 5.3, and binutils version 2.13.1.

Name

Setup — Preparing the Malta board for eCos Development

Overview

In a typical development environment, the Malta board boots from flash into the RedBoot ROM monitor. eCos applications are configured for RAM startup and then downloaded and run on the board via the debugger **mipsisa32-elf-gdb**. Preparing the board therefore usually involves programming a suitable RedBoot image into flash memory.

The following RedBoot configurations are supported:

Configuration	Description	Use	File
RAM	RedBoot running from RAM, usually loaded by another version of RedBoot	`redboot_RAM.ecm`	`redboot_RAM.bin`
ROM	RedBoot running from ROM	`redboot_ROM.ecm`	`redboot_ROM.bin`

For serial communications, all versions run with 8 bits, no parity, and 1 stop bit at 38400 baud. RedBoot also supports ethernet communication and flash management.

Initial Installation

Installing RedBoot is a matter of downloading a new binary image and overwriting the existing Boot monitor ROM image.

RedBoot is installed using the code download facility built into the Malta board. See the Malta User manual for details.

Quick Download Instructions

Here are quick start instructions for downloading the prebuilt RedBoot image.

1. Locate the prebuilt files in the bin directory: `deleteall.fl` and `redboot_ROM.fl`.
2. Make sure switch S5-1 is ON. Reset the board and verify that the LED display reads `Flash DL`.
3. Make sure your parallel port is connected between the 1284 port of the Malta board and the parallel port of your host system.
4. Send the `deleteall.fl` file to the parallel port to erase previous images:

   ```
   $ cat deleteall.fl >/dev/lp0
   ```

   When this is complete, the LED display should read `Deleted`.
5. Send the RedBoot image to the board:

   ```
   $ cat redboot_ROM.fl >/dev/lp0
   ```

   When this is complete, the LED display should show the last address programmed. This will be something like: `1fc17000`.
6. Connect a serial cable between one of the Malta board serial ports and a serial port on your host. Use a terminal emulator to monitor the serial port (HyperTerminal on Windows or minicom on Linux).
7. Change switch S5-1 to OFF and reset the board. The LED display should read `RedBoot` and something similar to the following should be output on the serial port:

   ```
   No devices on IDE controller 0
   ```

```
No devices on IDE controller 1
... waiting for BOOTP information
Ethernet eth0: MAC address 00:d0:a0:00:01:cb
IP: 10.0.0.203/255.255.255.0, Gateway: 10.0.0.3
Default server: 10.0.0.1, DNS server IP: 10.0.0.1

RedBoot(tm) bootstrap and debug environment [ROM]
Non-certified release, version UNKNOWN - built 15:01:37, Oct 20 2004

Platform: Malta (MIPS32 4Kc)
Copyright (C) 2000, 2001, 2002, 2003, 2004 Free Software Foundation, Inc.
Copyright (C) 2003, 2004, eCosCentric Limited

RAM: 0x80000400-0x82000000, [0x8000cc40-0x81ed1000] available
FLASH: 0xbe000000 - 0xbe400000, 32 blocks of 0x00020000 bytes each.
RedBoot>
```

8. Run the RedBoot **fis init** and **fconfig** commands to initialize the flash.

Malta Download Format

In order to download RedBoot to the Malta board, it must be converted to the Malta download format.

The *Atlas/Malta Developer's Kit* CD contains an srecconv.pl utility which requires Perl. This utility is part of the `yamon/yamon-src-02.00.tar.gz` tarball on the Dev Kit CD. The path in the expanded tarball is `yamon/bin/tools`. To use srecconv to convert the S-record file:

```
$ cp redboot_ROM.srec redboot_ROM.rec
$ srecconv.pl -ES L -A 29 redboot_ROM
```

The resulting file is named `redboot_ROM.fl`.

Rebuilding RedBoot

Should it prove necessary to rebuild a RedBoot binary, this is done most conveniently at the command line. The steps needed to rebuild the the ROM version of RedBoot for the Malta are:

```
$ mkdir redboot_malta_rom
$ cd redboot_malta_rom
$ ecosconfig new malta redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/mips/malta/current/misc/redboot_ROM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

At the end of the build the `install/bin` subdirectory should contain the file `redboot.srec`. This can then be converted into the download format by using the srecconv program as described above.

Additional commands

The **exec** command which allows the loading and execution of Linux kernels, is supported for this architecture. The **exec** parameters used for MIPS boards are:

-b *<addr>*	Location to store command line and environment passed to kernel
-w *<time>*	Wait time in seconds before starting kernel
-c *"params"*	Parameters passed to kernel
<addr>	Kernel entry point, defaulting to the entry point of the last image loaded

Linux kernels on MIPS platforms expect the entry point to be called with arguments in the registers equivalent to a C call with prototype:

```
void Linux(int argc, char **argv, char **envp);
```

RedBoot will place the appropriate data at the offset specified by the *-b* parameter, or by default at address 0x80080000, and will set the arguments accordingly when calling into the kernel.

The default entry point, if no image with explicit entry point has been loaded and none is specified, is 0x80000750.

Other Issues

The Malta platform HAL does not affect the implementation of other parts of the eCos HAL specification. The MIPS32 variant HAL, and the MIPS architectural HAL documentation should be consulted for further details.

Name

Configuration — Platform-specific Configuration Options

Overview

The Malta platform HAL package is loaded automatically when eCos is configured for a `malta` target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Target Selection

The exact processor type is determined by the target selected during configuration. The following targets are currently supported:

malta_mips32_4kc	Board fitted with 4Kc core card.
malta_mips32_4kec	Board fitted with 4KEc core card.
malta_mips32_5kc	Board fitted with 5Kc core card. This will only support a RedBoot configuration.

Startup

The Malta platform HAL package supports two startup types:

RAM

> This is the startup type which is normally used during application development. The board has RedBoot programmed into flash and boots into that initially. mipsisa32-elf-gdb is then used to load a RAM startup application into memory and debug it. It is assumed that the hardware has already been initialized by RedBoot. By default the application will use the eCos virtual vectors mechanism to obtain certain services from RedBoot, including diagnostic output.

ROM

> This startup type can be used for finished applications which will be programmed into flash at physical address 0xbe000000. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization.

RedBoot and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building RedBoot.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then the serial port will be claimed for HAL diagnostics.

Flash Driver

The Malta board contains two 16 bit Intel 28F160 flash devices arranged in parallel to form a 32 bit wide interface. The `CYGPKG_DEVS_FLASH_INTEL_28FXXX` package contains all the code necessary to support these parts and the `CYGPKG_DEVS_FLASH_MALTA` package contains definitions that customize the driver to the Malta board.

Ethernet Driver

The Malta board contains an AMD Am79C973 PCnet ethernet MAC. The `CYGPKG_DEVS_ETH_AMD_PCNET` package contains all the code necessary to support this device and the `CYGPKG_DEVS_ETH_MIPS_MIPS32_MALTA` package contains definitions that customize the driver to the Malta board.

System Clock

By default, the system clock interrupts once every 10ms, corresponding to a 100Hz clock. This can be changed by the configuration option `CYGNUM_HAL_RTC_DENOMINATOR` which corresponds to the clock frequency. Other clock-related settings are recalculated automatically if the denominator is changed.

Compiler Flags

The platform HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos.

`-mips32`

The mipsisa32-elf-gcc compiler supports many variants of the MIPS architecture. A `-m` option should be used to select the specific variant in use, and with current tools `-mips32` is the correct option for the 4Kc and 4KEc processors.

`-mips64`

If the board is populated with a 5Kc processor, then RedBoot may be built with 64 bit support. In that case, stand-alone applications may be built with the `-mips64` option. This option is not currently supported by eCos applications.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the Malta hardware, and should be read in conjunction with that specification. The Malta platform HAL package complements the MIPS architectural HAL and the MIPS32 variant HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset the HAL will initialize or reinitialize the peripherals that it uses. There is an exception for RAM startup applications which depend on a ROM monitor for certain services.

For ROM startup, the HAL will perform additional initialization, setting up the memory controller and PCI bridge and programming the various internal registers. This is done in the assembler macros defined in the `arch.inc`, `variant.inc` and `platform.inc` headers and in the `hal_platform_init()` function in `plf_misc.c`.

Linker Scripts and Memory Maps

The platform HAL package provides the memory layout information needed to generate the linker script. The MMU is not enabled for this platform, however, the normal MIPS segment address translations are present. The key memory locations are as follows.

NOTE

The virtual memory maps in this section use a C and B column to indicate whether or not the region is cached (C) or buffered (B).

```
Virtual Address Range   C B  Description
----------------------- - -  -----------
0x80000000 - 0x81ffffff Y Y  SDRAM
0x9e000000 - 0x9e3fffff Y N  System flash (cached)
0x9fc00000 - 0x9fffffff Y N  System flash (mirrored)
0xa8000000 - 0xb7ffffff N N  PCI Memory Space
0xb4000000 - 0xb40fffff N N  Galileo System Controller
0xb8000000 - 0xb80fffff N N  Southbridge / ISA
0xb8100000 - 0xbbdfffff N N  PCI I/O Space
0xbe000000 - 0xbe3fffff N N  System flash (noncached)
0xbf000000 - 0xbfffffff N N  Board logic FPGA
```

MIPS16 Support

The Malta platform HAL enables MIPS16 support in the architecture HAL for those Core boards that contain capable processors. This allows *application* code to be compiled using MIPS16 options and linked against the 32 bit mode eCos library.

To compile for MIPS16 the standard flags that are used in eCos and exported to the `ecos.mak` file should be used except that the `-mips32` flag should be replaced by `-mips16 -fwritable-strings`. The `-mips16` option enables MIPS16 compilation and the `-fwritable-strings` option is a work-around for a bug in the compiler.

Chapter 336. NXP PNX83xx Common Support

Name

`CYGPKG_HAL_MIPS_PNX83xx` — eCos Support for NXP PNX83xx Processors

Description

The NXP PNX83xx family is a range of processors including the PNX8310 and PNX8330. This package provides support for features that are common across the range. For example the PNX8310 and PNX8330 use similar UARTs, so the device definitions are provided by this package rather than duplicated in the two processor HAL packages. Similarly HAL diagnostics support using these UARTs is provided here.

Configuration

The PNX83xx common HAL package should be loaded automatically when eCos is configured for appropriate target hardware. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

The package does not contain any configuration options.

Chapter 337. NXP PNX8310 Processor Support

Name

CYGPKG_HAL_MIPS_PNX8310 — eCos Support for the NXP PNX8310 Processor

Description

The NXP PNX8310 processor is based around a PR1910 MIPS core, complemented by a range of on-chip peripherals. The HAL package CYGPKG_HAL_MIPS_PNX8310 provides the processor-specific support, combining the functionality of an eCos variant HAL and processor HAL. It complements the MIPS architectural HAL package CYGPKG_HAL_MIPS, and the PNX83xx support package CYGPKG_HAL_MIPS_PNX83xx which contains support for features common to several members of the PNX83xx family. An eCos configuration should also include a platform HAL package to support board-level details like the memory chips and off-chip peripherals.

Configuration

The PNX8310 HAL package should be loaded automatically when eCos is configured for appropriate target hardware. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware. The package does not contain any user-settable options.

The HAL Port

This section describes how the PNX8310 HAL package implements parts of the eCos HAL specification. It should be read in conjunction with similar sections from the architectural and variant HAL documentation.

HAL I/O

The header file cyg/hal/var_io.h provides definitions of all on-chip peripherals, except for some UART definitions which are provided by the PNX83xx support HAL instead. This header file is automatically included by the architectural header cyg/hal/hal_io.h, so typically application code and other packages will just include the latter. The register addresses are all in kseg1 so will be accessed uncached.

Interrupt Handling

The header file cyg/hal/var_intr.h provides ISR vector numbers for all interrupt sources, for example CYGNUM_HAL_ISR_I2C0 and CYGNUM_HAL_ISR_PIO_1. These vector numbers should be used for calls like cyg_interrupt_create. The header file is automatically included by the architectural header cyg/hal/hal_intr.h, and other packages and application code will normally just include the latter.

The interrupt vectors come in three groups. There are three vectors for the timers implemented by the PR1910 core's coprocessor 0. There are 20 vectors for the other on-chip peripherals, managed by the priority interrupt controller. One of these, CYGNUM_HAL_ISR_PIO is reserved for use by the HAL's interrupt decoding code to detect PIO interrupts. Finally there are 16 vectors for interrupts on the PIO pins.

The eCos HAL macros HAL_INTERRUPT_MASK, HAL_INTERRUPT_UNMASK, HAL_INTERRUPT_ACKNOWLEDGE, HAL_INTERRUPT_CONFIGURE and HAL_INTERRUPT_SET_LEVEL are implemented by the processor HAL. The implementations depend on the interrupt vector. HAL_INTERRUPT_ACKNOWLEDGE is only needed for PIO interrupts, otherwise it is a no-op. HAL_INTERRUPT_CONFIGURE is also only relevant for PIO interrupts.

Interrupt priorities should be in the range 1 to 14, and correspond to the *int_priority* fields in the pic_int_reg registers. 1 is the lowest priority and 14 the highest. Interrupt priorities are ignored for the three timer interrupts. All PIO interrupt sources operate at the same priority, which is the highest priority assigned to any of the PIO vectors.

Interrupt chaining via the common HAL's configuration option CYGIMP_HAL_COMMON_INTERRUPTS_CHAIN is supported for PIO interrupts only. This makes it possible to connect several external peripherals' interrupt lines to a single PIO pin if desired.

Clock Support

The PR1910 core provides three timers, TMR1, TMR2 and TMR3. TMR1 is used for the eCos system clock. TMR2 is used for gprof-based profiling if enabled, otherwise it can be used by the application. TMR3 is normally used only for the watchdog device driver.

Cache Handling

The PNX8310 has an 8K unified cache, which is automatically initialized and enabled by the eCos startup code. The standard macros `HAL_UCACHE_INVALIDATE_ALL` and `HAL_UCACHE_SYNC` are supported, and both the DCACHE and ICACHE variants are just mapped on to these. Working ENABLE, DISABLE and IS_ENABLED macros are provided as well but these are not generally useful on a MIPS processor.

Profiling Support

The PNX8310 HAL provides a profiling timer for use with the gprof profiling package. This uses the PR1910 coprocessor 0 timer TMR2, so application code should not manipulate this timer if profiling is enabled. The MIPS architectural HAL implements the `mcount` function so profiling is fully supported on all PNX8310-based platforms.

Linker Script

During a build the PNX8310 HAL provides a linker script suitable for use with all C and C++ applications. This also allows parts of the application code and data to be placed in the on-chip deeply embedded memory, using ELF linker sections `.dem_text` for code, `.dem_data` for statically initialized data, and `.dem_bss` for uninitialized data. The `dem1.c` testcase in this package illustrates how to use this functionality.

Other Issues

The `HAL_PLATFORM_RESET` is implemented via the PNX8310's system reset unit, and involves a full reset of the core and all peripherals. Hence, whenever a soft reset is performed by the application or via a gdb command, the system should start up again in a clean state, and there is no need for the system to reinitialize all the peripherals.

The PNX8310 HAL does not affect the implementation of data types, stack size definitions, bit indexing, idle thread processing, SMP support, system startup, or debug support.

Other Functionality

The PNX8310 processor HAL only implements functionality defined in the eCos HAL specification and does not export any additional functions.

Chapter 338. NXP STB200 Board Support

Name

eCos Support for the NXP STB200 Board — Overview

Description

The NXP STB200 board has a PNX8310 processor, 16MB of external SDRAM, 4MB of external flash, an SMSC LAN9118 ethernet chip, and connectors plus required support chips for various on-chip peripherals. For typical eCos development a RedBoot image is programmed into the external flash. RedBoot provides gdb stub functionality so it is then possible to download and debug eCos applications via the gdb debugger. This can happen over either a serial line or over ethernet.

Supported Hardware

The memory map used by both eCos and RedBoot is as follows:

Memory	Base	Length
External SDRAM	0x80000000	0x01000000
External Flash	0x98000000	0x00400000
Internal RAM	0xA4000000	0x00001000
On-chip Peripherals	0xB7000000	0x00200000
SMSC LAN9118	0xBA000000	0x02000000

External SDRAM and flash are normally accessed via the kseg0 segment and hence via the cache. The internal RAM and the peripherals are normally accessed via kseg1 and hence uncached. Accesses to the on-chip RAM are as fast as cache accesses so there is no point in going through the cache for those.

eCos can be configured for one of three startup types:

RAM

> This is the startup type normally used during application development. RedBoot is programmed into flash and performs the initial bootstrap. mipsisa32-elf-gdb is then used to load a RAM startup application into memory and debug it. By default the application will use eCos' virtual vectors mechanism to obtain certain services from RedBoot, including diagnostic output. The bottom 256K of RAM is used for RedBoot code and data so the application will start at 0x80040000.

ROMRAM

> This startup type can be used for finished applications which will be programmed into the start of external flash at location 0xB8000000. On power-up the chip's bootloader will automatically load the application into RAM at location 0x80001000 and start it. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization. This startup type is used for building the flash-resident version of RedBoot but can also be used for application code.

JTAG

> This can be used to run applications via JTAG rather than RedBoot. As with ROMRAM startup, the application will be loaded at location 0x80001000 and it will take over all the hardware. Uart1 will be used for all HAL diagnostics and standard output. A JTAG build of RedBoot can be used during hardware setup to program the ROMRAM version into flash. A JTAG application build may prove useful for debugging certain problems, especially ones related to interrupts and exceptions. However the JTAG software may not fully cope with the executables and debug information generated by the GNU tools, so the user experience may be poor compared with using the GNU mipsisa32-elf-gdb debugger.

In a typical setup the first 128K of flash is used for holding the RedBoot image, and the last 64K is used for managing the flash and holding the RedBoot fconfig values. The remaining blocks from 0x98020000 to 0x983EFFFF can be used by application code.

RedBoot can communicate with the host using either uart1 or ethernet. The PNX8310's uart0 is not connected on this board.

All configurations for the STB200 target include an ethernet driver package `CYGPKG_DEVS_ETH_SMSC_LAN9118`. If the application does not actually require ethernet functionality then the package is inactive and the final executable will not suffer any overheads from unused functionality. This is determined by the presence of the generic ethernet I/O package `CYGPKG_IO_ETH_DRIVERS`. Typically the choice of eCos template causes the right thing to happen. For example, the default template does not include any TCP/IP stack so `CYGPKG_IO_ETH_DRIVERS` is not included, but both the net and redboot templates do include a TCP/IP stack so will specify that package and hence enable the ethernet driver. The ethernet device can be shared by RedBoot and the application, so it is possible to debug a networked application over ethernet.

The STB200 board does not have a serial EPROM or similar hardware providing a unique network MAC address. Instead a suitable address has to be programmed into flash via RedBoot's **fconfig** command.

All configurations for the STB200 target include a serial device driver package `CYGPKG_DEVS_SERIAL_MIPS_PNX8310`. The driver as a whole is inactive unless the generic serial support, `CYGPKG_IO_SERIAL_DEVICES` is enabled. Only uart1 has a suitable connector so that is the only device which can be accessed through the serial driver. The hardware flow control lines are not connected so only software flow control is available. If the UART is needed by the application then it cannot also be used by RedBoot for gdb traffic, so another communication channel such as ethernet should be used instead.

All configurations for the STB200 target include a watchdog device driver package `CYGPKG_DEVS_WATCHDOG_MIPS_PNX8310`. This is inactive unless the generic watchdog support, `CYGPKG_IO_WATCHDOG` is loaded.

The on-chip interrupt controller is managed by eCos using macros provided by the PNX8310 processor HAL. The on-chip timer TMR1 is used to implement the eCos system clock. If gprof-based profiling is enabled then that will use TMR2, otherwise that timer can be used by the application. TMR3 is normally used only by the watchdog device driver. GPIO pins 17 and 18 are used for uart1, and pin 14 is used for ethernet interrupts. The remaining GPIO pins are not used by eCos. Other on-chip peripherals are left to their initial settings and not manipulated by eCos.

Tools

The STB200 port is intended to work with GNU tools configured for an mipsisa32-elf target. The original port was done using mipsisa32-elf-gcc version 3.4.4 mipsisa32-elf-gdb version 6.3, and binutils version 2.16. The PNX8310's PR1910 core does not implement the full mips32 functionality so all application code should be compiled with `-mips2`.

Name

Setup — Preparing the STB200 board for eCos Development

Overview

In a typical development environment the STB200 board boots from flash into the RedBoot ROM monitor. eCos applications are configured for a RAM startup, and then downloaded and run on the board via the debugger mipsisa32-elf-gdb. Preparing the board therefore involves programming a suitable RedBoot image into flash memory.

The following RedBoot configurations are supported:

Configuration	**Description**	**Use**	**File**
ROM	RedBoot running from the board's flash	`redboot_ROMRAM.ecm`	`redboot_romram.bin`
JTAG	Used for initial setup	`redboot_JTAG.ecm`	`redboot.elf`
RAM	For debugging RedBoot	`redboot_RAM.ecm`	`redboot_ram.bin`

For serial communications all versions run at 384000 baud with 8 bits, no parity, and 1 stop bit. The baud rates can be changed via the configuration option `CYGNUM_HAL_MIPS_STB200_DIAG_BAUD` and rebuilding RedBoot. Only uart1 has a serial connector so RedBoot will use that. Ethernet communication and flash management are also supported.

Initial Installation

This process assumes that RedBoot has not yet been installed into flash, so JTAG has to be used to program the `redboot_romram.bin` file into flash. This can be done either via a flash programming utility or by first running a JTAG version of RedBoot and using that to initialize and program the flash. This second approach is described here.

The first step is to set up a suitable JTAG module and associated debug software, as per the instructions supplied with the JTAG kit. Next connect a straight-throught RS232 cable between the STB200's serial port and the host PC, and start a terminal emulation application such as HyperTerminal or minicom on the host PC. The serial communication parameters should be 38400 baud, 8 data bits, no parity, 1 stop bit (8N1) and no flow control (handshaking).

It is now necessary to load and run a JTAG build of RedBoot. A prebuilt image `redboot_JTAG.elf` is supplied with eCos, or can be rebuilt as described below. This file is an ELF executable complete with relocation and debug information, and may need to be converted to another format for use with the JTAG software. For example, when using an Ashling Opella unit, the SymFinder utility sfdwarf should be used to convert the ELF executable to a `.CSO` file. This utility may give numerous warnings which can be ignored. The `.CSO` file can be loaded via the JTAG debugger, and it can be started running at location 0x80001000. At this point RedBoot will output text similar to the following on the serial port:

```
+FLASH configuration checksum error or invalid key
Ethernet eth0: MAC address 00:FF:12:34:56:78
... waiting for BOOTP information
Can't get BOOTP info for device!

RedBoot(tm) bootstrap and debug environment [JTAG]
Non-certified release, version UNKNOWN - built 20:35:11, Sep 29 2005

Platform: STB200 (Philips PNX8310)
Copyright (C) 2000, 2001, 2002, 2003, 2004 Free Software Foundation, Inc.
Copyright (C) 2003, 2004, 2005 eCosCentric Limited

RAM: 0x80000000-0x81000000, [0x80028720-0x80fed000] available
FLASH: 0x98000000 - 0x983fffff 8 x 0x2000 blocks 63 x 0x10000 blocks
RedBoot>
```

The exact details may vary slightly, depending on the flash chip present and whether or not the ethernet is plugged in yet. At this stage the RedBoot flash management initialization has not yet happened so the warning about the configuration checksum error is expected. There will also be a delay while RedBoot tries to contact a local BOOTP server. To perform the flash initialization use the **fis init -f** command:

```
RedBoot> fis init -f
About to initialize [format] FLASH image system - continue (y/n)? y
*** Initialize FLASH Image System
... Erase from 0x983f0000-0x983fffff: .
... Program from 0x80ff0000-0x81000000 to 0x983f0000: .
RedBoot>
```

At this stage the block of flash at location 0x983F0000 holds information about the various flash blocks, allowing other flash management operations to be performed. The next step is to set up RedBoot's non-volatile configuration values:

```
RedBoot> fconfig -i
Initialize non-volatile configuration - continue (y/n)? y
Run script at boot: false
Use BOOTP for network configuration: true
Default server IP address: 10.1.1.1
GDB connection port: 9000
Force console for special debug messages: false
Network hardware address [MAC]: 0x00:0xff:0x12:0x34:0x01:0x0C
Network debug at boot time: false
Update RedBoot non-volatile configuration - continue (y/n)? y
... Erase from 0x983f0000-0x983fffff: .
... Program from 0x80ff0000-0x81000000 to 0x983f0000: .
RedBoot>
```

For most of these configuration variables the default value is correct. If there is no suitable BOOTP service running on the local network then BOOTP should be disabled, and instead RedBoot will prompt for a fixed IP address, netmask, and addresses for the local gateway and DNS server. The other exception is the network hardware address, also known as MAC address. All boards should be given a unique MAC address, not the one in the above example. If there are two boards on the same network trying to use the same MAC address then the resulting behaviour is undefined.

It is now possible to load the flash-resident version of RedBoot. Because of the way that flash chips work it is better to first load it into RAM and then program it into flash.

```
RedBoot> load -r -m ymodem -b %{freememlo}
```

The file `redboot_romram.bin` should now be uploaded using the terminal emulator. The file is a raw binary and should be transferred using the Y-modem protocol.

```
CRaw file loaded 0x80028800-0x8004568f, assumed entry at 0x80028800
xyzModem - CRC mode, 930(SOH)/0(STX)/0(CAN) packets, 5 retries
RedBoot>
```

Once RedBoot has been loaded into RAM it can be programmed into flash:

```
RedBoot> fis create RedBoot -b %{freememlo}
An image named 'RedBoot' exists - continue (y/n)? y
... Erase from 0x98000000-0x9801ffff: .........
... Program from 0x80028800-0x80048800 to 0x98000000: .........
... Erase from 0x983f0000-0x983fffff: .
... Program from 0x80ff0000-0x81000000 to 0x983f0000: .
RedBoot>
```

The flash-resident version of RedBoot has now been programmed at location 0x98000000, and the flash info block at 0x983F0000 has been updated. The initial setup is now complete and the board can now boot from flash. This can be done either by powering the board down and back up, or simply by using RedBoot's **reset** command:

```
RedBoot> reset
... Resetting.+... waiting for BOOTP information
```

```
Ethernet eth0: MAC address 00:ff:12:34:01:0c
IP: 10.1.1.153/255.255.255.0, Gateway: 10.1.1.241
Default server: 10.1.1.1

RedBoot(tm) bootstrap and debug environment [ROMRAM]
Non-certified release, version UNKNOWN - built 12:46:19, Oct  2 2005

Platform: STB200 (Philips PNX8310)
Copyright (C) 2000, 2001, 2002, 2003, 2004 Free Software Foundation, Inc.
Copyright (C) 2003, 2004, 2005 eCosCentric Limited

RAM: 0x80000000-0x81000000, [0x800288e0-0x80fed000] available
FLASH: 0x98000000 - 0x983fffff 8 x 0x2000 blocks 63 x 0x10000 blocks
RedBoot>
```

When RedBoot issues its prompt it is also ready to accept connections from mipsisa32-elf-gdb, allowing eCos applications to be downloaded and debugged.

Occasionally it may prove necessary to update the installed RedBoot image. This can be done at the ROMRAM RedBoot prompt - there is no need to run the JTAG version again unless the version already installed has been corrupted. It involves loading the new image into RAM using RedBoot's **load** command, and then programming it into flash using **fis create RedBoot**

Rebuilding RedBoot

Should it prove necessary to rebuild a RedBoot binary, this is done most conveniently at the command line. The steps needed to rebuild the ROMRAM version of RedBoot are:

```
$ mkdir redboot_romram
$ cd redboot_romram
$ ecosconfig new stb200 redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/mips/pnx83xx/pnx8310/stb200/current/misc/redboot_ROMRAM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

At the end of the build the `install/bin` subdirectory should contain the required file `redboot_romram.bin`.

Rebuilding the JTAG and RAM versions involves basically the same process. The JTAG version uses the file `redboot_JTAG.ecm` and generates an ELF executable `redboot.elf`, which may need to be converted to another format before it can be used with the JTAG software. The RAM version uses the file `redboot_RAM.ecm` and generates a raw binary `redboot.ram.bin` which can be loaded into memory at 0x80040000 and executed from there.

Name

Configuration — Platform-specific Configuration Options

Overview

The STB200 platform HAL package is loaded automatically when eCos is configured for an STB200 target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The STB200 platform HAL package supports three separate startup types: RAM, ROMRAM and JTAG. The configuration option `CYG_HAL_STARTUP` controls which startup type is being used. For typical application development RAM startup should be used, and the application will be run via mipsisa32-elf-gdb interacting with RedBoot using either serial or ethernet. It is assumed that the low-level hardware initialization, including setting up the memory map, has already been performed by RedBoot. By default the application will use certain services provided by RedBoot via the virtual vector mechanism, including diagnostic output, but that can be disabled via `CYGSEM_HAL_USE_ROM_MONITOR`.

ROMRAM startup can be used for applications which are programmed into the base of flash at 0x98000000. On power up the chip's boot loader will load the first 8K of code from flash to RAM at 0x80001000 and branch to that location. The startup code will copy the remainder of the application from flash to RAM, and subsequently the flash will not be used for executing any code. All the hardware will be initialized, and the application is self-contained. This startup type is used by the flash-resident version of RedBoot, and can also be used for finished applications.

JTAG startup can be used for applications which will be debugged via JTAG instead of RedBoot. The behaviour is mostly the same as for ROMRAM startup: the application is loaded at 0x80001000 and initializes all the hardware, with no dependencies on services provided by a ROM monitor. There are some minor differences in the startup code, for example it is not necessary to copy the remainder of the application from flash to RAM.

RedBoot and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building RedBoot.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained. That is useful as a testing step before switching to ROM startup. It also allows applications to be run and debugged via BDM.

If the application does not rely on a ROM monitor for diagnostic services then uart1 will be used for HAL diagnostics and standard output. The default baud rate is controlled by `CYGNUM_HAL_MIPS_STB200_DIAG_BAUD`. The board does not have a connector for uart0 so that cannot be used instead. Since uart1 is needed by the HAL diagnostics code it cannot be accessed via the serial driver.

System Clock

The coprocessor 0 TMR1 timer is used for the eCos system clock. The configuration option `CYGNUM_HAL_RTC_PERIOD` controls the value programmed into the compare register. TMR1 ticks at 120MHz so the default value of 1200000 corresponds to a 100Hz system clock or one tick per 10ms. Other clock-related settings are recalculated automatically if the period is changed.

Flash Driver

The platform HAL package contains flash driver support for the external flash device. By default this is inactive, and it can be made active by loading the generic flash package `CYGPKG_IO_FLASH`. The board may use one of a variety of flash chips. The exact type present is determined at run-time using CFI and the system will adjust accordingly.

Ethernet Support

The platform HAL provides the platform-specific support for a single SMSC LAN9118 ethernet device, if the generic ethernet support is enabled. The configuration `CYGNUM_HAL_MIPS_STB200_ETH_ISR_PRIORITY` provides control over the interrupt priority used for this device. The board does not have a suitable EEPROM so the MAC address is provided via a RedBoot fconfig option.

Compiler Flags

The platform HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos. There are four flags specific to this port:

`-mips2`

The mipsisa32-elf-gcc toolchain defaults to supporting the mips32 architecture, and the PNX8310's PR1910 core does not fully implement this. Specifying `-mips2` restricts the compiler to using a subset of the mips32 instruction set appropriate for the PR1910.

`-G0`

MIPS calling conventions reserve one register for use as a global pointer register. In theory this allows static variables in one 64K area of memory to be accessed using just one instruction instead of two, and the `-G` option provides some control over this. However due to limitations within the current linker all modules have to be compiled with the same `-G` setting, and the compiler support libraries are built with `-G0`. Therefore all eCos and application modules also have to be built with `-G0` and this optimization is not available.

`-EB`

The eCos port only supports big-endian mode so `-EB` must be specified.

`-msoft-float`

The PR1910 does not have a hardware floating point unit so software floating point has to be used instead.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the STB200 hardware, and should be read in conjunction with that specification. The STB200 platform HAL package complements the MIPS architectural HAL, the PNX8310 variant HAL, and the PNX83xx support HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset the HAL will set up some of the on-chip peripherals appropriately for eCos, but most peripherals are left to their default settings. Full details of this initialization can be found in the function `hal_platform_init` in `stb200.c`.

Linker Scripts and Memory Maps

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

External SDRAM
: This is mapped to location 0x80000000. The first four kilobytes are reserved for exception vectors and for some data which needs to be shared between RAM startup applications and the ROM monitor. For ROMRAM or JTAG startup, code will be loaded from 0x80001000 onwards. For RAM startup, code will start at 0x80040000, reserving the bottom 256K for RedBoot's code and data.

Internal RAM
: The 4K of deeply-embedded internal RAM is mapped at location 0xA4000000. Neither eCos nor RedBoot use this so all of it is available to the application. The PNX8310 variant HAL documentation explains how to map code or data into this memory.

On-chip Peripherals
: These are accessible at location 0xB7000000 onwards.

External Flash
: This is located at location 0x98000000 onwards. Different STB200 boards may come with different flash chips so CFI is used at run-time to query the flash chip and adapt. Typically the first 128K of flash at location 0x98000000 is used to hold RedBoot, and the last flash block at location 0x983F0000 is used to hold flash management data and the RedBoot **fconfig** variables. The remaining blocks can be used by application code.

Clock Support

The platform HAL provides configuration options for the eCos system clock. This always uses the hardware timer TMR1, part of the PR1910 core and is accessed via coprocessor 0. The actual HAL macros for managing the clock are provided by the PNX8310 variant HAL. The specific numbers used are a characteristic of the platform because they depend on the processor speed. TMR2 is used by the gprof-based profiling code, or is available for application use when profiling is not enabled. TMR3 is normally used only for the watchdog.

MIPS16 Support

The STB200 platform HAL enables MIPS16 support in the architecture HAL. This allows *application* code to be compiled using MIPS16 options and linked against the 32 bit mode eCos library.

To compile for MIPS16 the standard flags that are used in eCos and exported to the `ecos.mak` file should be used except that the `-mips2` flag should be replaced by `-mips16 -fwritable-strings`. The `-mips16` option enables MIPS16 compilation and the `-fwritable-strings` option is a work-around for a bug in the compiler.

Other Issues

The STB200 platform HAL does not affect the implementation of other parts of the eCos HAL specification.

Chapter 339. NXP PNX8330 Processor Support

Name

CYGPKG_HAL_MIPS_PNX8330 — eCos Support for the NXP PNX8330 Processor

Description

The NXP PNX8330 processor is based around a 4KEc MIPS32 core, complemented by a range of on-chip peripherals. The HAL package CYGPKG_HAL_MIPS_PNX8330 provides the processor-specific support. It complements the MIPS architectural HAL package CYGPKG_HAL_MIPS, the MIPS32 variant package CYGPKG_HAL_MIPS_MIPS32 and the PNX83xx support package CYGPKG_HAL_MIPS_PNX83xx which contains support for features common to several members of the PNX83xx family. An eCos configuration should also include a platform HAL package to support board-level details like the memory chips and off-chip peripherals.

Configuration

The PNX8330 HAL package should be loaded automatically when eCos is configured for appropriate target hardware. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware. The package does not contain any user-settable options.

The HAL Port

This section describes how the PNX8330 HAL package implements parts of the eCos HAL specification. It should be read in conjunction with similar sections from the architectural and variant HAL documentation.

HAL I/O

The header file cyg/hal/pnx8330_io.h provides definitions of the on-chip peripherals used by eCos, except for some UART definitions which are provided by the PNX83xx support HAL instead. This header file is automatically included by the architectural header cyg/hal/hal_io.h, so typically application code and other packages will just include the latter. The register addresses are all in kseg1 so will be accessed uncached.

Interrupt Handling

The header file cyg/hal/pnx8330_intr.h provides ISR vector numbers for all interrupt sources, for example CYGNUM_HAL_ISR_I2C0 and CYGNUM_HAL_ISR_PIO_1. These vector numbers should be used for calls like cyg_interrupt_create. The header file is automatically included by the architectural header cyg/hal/hal_intr.h, and other packages and application code will normally just include the latter.

The interrupt vectors come in four groups. There are six vectors corresponding to the external interrupts available in the CPU SR and CAUSE registers. Of these, interrupt number 2 is attached to the on-chip interrupt controller, and interrupt number 5 is attached to the internal COMPARE register, which is used to supply system time interrupts. There are 37 vectors for the on-chip peripherals, managed by the priority interrupt controller. One of these, CYGNUM_HAL_ISR_PIO is reserved for use by the HAL's interrupt decoding code to detect PIO interrupts and decode them into the next 16 vectors. Another peripheral interrupt, CYGNUM_HAL_ISR_CONFIG is decoded into the final seven interrupt vectors, which correspond to the interrupt sources available from the CONFIG unit.

The eCos HAL macros HAL_INTERRUPT_MASK, HAL_INTERRUPT_UNMASK, HAL_INTERRUPT_ACKNOWLEDGE, HAL_INTERRUPT_CONFIGURE and HAL_INTERRUPT_SET_LEVEL are implemented by the processor HAL. The implementations depend on the interrupt vector. HAL_INTERRUPT_ACKNOWLEDGE is only needed for PIO and CONFIG interrupts, otherwise it is a no-op. HAL_INTERRUPT_CONFIGURE is only relevant for PIO interrupts.

Interrupt priorities should be in the range 1 to 14, and correspond to the *int_priority* fields in the pic_int_reg registers. 1 is the lowest priority and 14 the highest. Interrupt priorities are ignored for the COMPARE interrupt. All PIO and CONFIG interrupt sources operate at the same priority, which is the highest priority assigned to any of the PIO or CONFIG vectors.

Interrupt chaining via the common HAL's configuration option CYGIMP_HAL_COMMON_INTERRUPTS_CHAIN is supported for PIO interrupts only. This makes it possible to connect several external peripherals' interrupt lines to a single PIO pin if desired.

Clock Support

The 4kEc core provides standard COUNTER and COMPARE registers which are used for the eCos system clock. The CONFIG unit timer 0 is used for gprof-based profiling if enabled, otherwise it can be used by the application. The CONFIG unit watchdog timer is supported by a watchdog driver.

Cache Handling

The PNX8330 has an 8K data cache and a 16k instruction cache, which are automatically initialized and enabled by the eCos startup code. All the standard cache control macros are supported through the cache instruction. However, since all memory is always available both cached and uncached as part of the architecture, these are not always necessary.

Profiling Support

The PNX8330 HAL provides a profiling timer for use with the gprof profiling package. This uses the PNX8330 configuration timer 0, so application code should not manipulate this timer if profiling is enabled. The MIPS architectural HAL implements the mcount function so profiling is fully supported on all PNX8330-based platforms.

Other Issues

The macro HAL_PLATFORM_RESET is implemented via the PNX8330's system reset unit, and involves a full reset of the core and all peripherals. Hence, whenever a soft reset is performed by the application or via a gdb command, the system should start up again in a clean state, and there is no need for the system to reinitialize all the peripherals.

The PNX8330 HAL does not affect the implementation of data types, stack size definitions, bit indexing, idle thread processing, SMP support, system startup, or debug support.

Other Functionality

The PNX8330 processor HAL only implements functionality defined in the eCos HAL specification and does not export any additional functions.

Chapter 340. NXP STB220 Board Support

Name

eCos Support for the NXP STB220 Board — Overview

Description

The NXP STB220 board has a PNX8330 processor, 32MB of external SDRAM, 16MB of external flash, an SMSC LAN9118 ethernet chip, and connectors plus required support chips for various on-chip peripherals. For typical eCos development a RedBoot image is programmed into the external flash. RedBoot provides gdb stub functionality so it is then possible to download and debug eCos applications via the gdb debugger. This can happen over either a serial line or over ethernet.

Supported Hardware

The memory map used by both eCos and RedBoot is as follows:

Memory	Base	Length
External SDRAM	0x80000000	0x04000000
External Flash	0x98000000	0x01000000
On-chip Peripherals	0xB7000000	0x00200000

External SDRAM and flash are normally accessed via the kseg0 segment and hence via the cache. The peripherals are normally accessed via kseg1 and hence uncached.

eCos can be configured for one of four startup types:

RAM

> This is the startup type normally used during application development. RedBoot is programmed into flash and performs the initial bootstrap. mipsisa32-elf-gdb is then used to load a RAM startup application into memory and debug it. By default the application will use eCos' virtual vectors mechanism to obtain certain services from RedBoot, including diagnostic output. The bottom 1MB of RAM is used for RedBoot code and data so the application will start at 0x80100000.

ROM

> This startup type can be used for finished applications which will be programmed into the start of external flash at location 0xB8000000. On power-up the chip's bootloader will automatically execute the contents of flash from 0xB8000000. The application will initialize SDRAM, copy its data to RAM and zero its BSS. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization. This startup type is used for building the flash-resident version of RedBoot but can also be used for application code.

ROMRAM

> This startup type can be used for finished applications which will be programmed into the start of external flash at location 0xB8000000. On power-up the chip's bootloader will automatically execute the contents of flash from 0xB8000000. The application will initialize SDRAM, copy itself from flash to RAM, and zero its BSS. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization. This startup type is used for building the flash-resident version of RedBoot but can also be used for application code.

JTAG

> This can be used to run applications via JTAG rather than RedBoot. The application will be loaded at location 0x80000000 and it will take over all the hardware. Uart0 will be used for all HAL diagnostics and standard output. A JTAG build of RedBoot can be used during hardware setup to program the ROM or ROMRAM versions into flash. A JTAG application build may prove useful for debugging certain problems, especially ones related to interrupts and exceptions. However the JTAG software

may not fully cope with the executables and debug information generated by the GNU tools, so the user experience may be poor compared with using the GNU mipsisa32-elf-gdb debugger.

In a typical setup the first 128KB of flash is used for holding the RedBoot image, and the last 128KB is used for managing the flash and holding the RedBoot fconfig values. The remaining blocks from 0x98020000 to 0x98FEFFFF can be used by application code.

RedBoot can communicate with the host using either uart0 or uart1.

All configurations for the STB220 target include an ethernet driver package `CYGPKG_DEVS_ETH_SMSC_LAN9118`. If the application does not actually require ethernet functionality then the package is inactive and the final executable will not suffer any overheads from unused functionality. This is determined by the presence of the generic ethernet I/O package `CYGPKG_IO_ETH_DRIVERS`. Typically the choice of eCos template causes the right thing to happen. For example, the default template does not include any TCP/IP stack so `CYGPKG_IO_ETH_DRIVERS` is not included, but both the net and redboot templates do include a TCP/IP stack so will specify that package and hence enable the ethernet driver. The ethernet device can be shared by RedBoot and the application, so it is possible to debug a networked application over ethernet.

The STB220 board does not have a serial EPROM or similar hardware providing a unique network MAC address. Instead a suitable address has to be programmed into flash via RedBoot's **fconfig** command.

All configurations for the STB220 target include a serial device driver package `CYGPKG_DEVS_SERIAL_MIPS_PNX8310` (this driver is shared with PNX8310 based targets, and for historical reasons it is named for them, however it it applicable to both). The driver as a whole is inactive unless the generic serial support, `CYGPKG_IO_SERIAL_DEVICES` is enabled. Both Uart0 and uart1 are connected, however, only Uart0 has hardware flow control lines routed to the connector. If a UART is needed by the application then it cannot also be used by RedBoot for gdb traffic, so care should be exercised in selecting which UART to use for these purposes. Alternatively another communication channel such as ethernet chould be used instead.

All configurations for the STB220 target include a watchdog device driver package `CYGPKG_DEVS_WATCHDOG_MIPS_PNX8330`. This is inactive unless the generic watchdog support, `CYGPKG_IO_WATCHDOG` is loaded.

The on-chip interrupt controller is managed by eCos using macros provided by the PNX8330 processor HAL. The architecture COUNTER/COMPARE timer is used to implement the eCos system clock. If gprof-based profiling is enabled then that will use CONFIG unit timer 0, otherwise that timer can be used by the application. GPIO pins 4 to 7 are used for UART1 and pins 10 to 13 may be used for ethernet interrupts. The remaining GPIO pins are not used by eCos. Other on-chip peripherals are left to their initial settings and not manipulated by eCos.

Tools

The STB220 port is intended to work with GNU tools configured for an mipsisa32-elf target. The original port was done using mipsisa32-elf-gcc version 3.4.4 mipsisa32-elf-gdb version 6.3, and binutils version 2.16.

Name

Setup — Preparing the STB220 board for eCos Development

Overview

In a typical development environment the STB220 board boots from flash into the RedBoot ROM monitor. eCos applications are configured for a RAM startup, and then downloaded and run on the board via the debugger mipsisa32-elf-gdb. Preparing the board therefore involves programming a suitable RedBoot image into flash memory.

The following RedBoot configurations are supported:

Configuration	Description	Use	File
ROM	RedBoot running from parallel flash	`redboot_ROM.ecm`	`redboot_rom.bin`
ROMRAM	RedBoot copied from flash into RAM	`redboot_ROMRAM.ecm`	`redboot_romram.bin`
JTAG	Used for initial setup	`redboot_JTAG.ecm`	`redboot.elf`
RAM	For debugging RedBoot	`redboot_RAM.ecm`	`redboot_ram.bin`

For serial communications all versions run at 384000 baud with 8 bits, no parity, and 1 stop bit. The baud rates can be changed via the configuration option `CYGNUM_HAL_MIPS_STB220_DIAG_BAUD` and rebuilding RedBoot. RedBoot will support communication on either UART. Flash management is supported. Ethernet communication and flash management are also supported.

For EV8330 boards with DDR RAM there are alternative configuration files for the ROM and ROMRAM configurations. These have the same names as the files given above, with `_DDR` added giving `redboot_ROM_DDR.ecm` and `redboot_ROMRAM_DDR.ecm`.

Initial Installation

This process assumes that RedBoot has not yet been installed into flash, so JTAG has to be used to program the `redboot_romram.bin` file into flash. This can be done either via a flash programming utility or by first running a JTAG version of RedBoot and using that to initialize and program the flash. This second approach is described here.

The first step is to set up a suitable JTAG module and associated debug software, as per the instructions supplied with the JTAG kit. Next connect a straight-throught RS232 cable between the STB220's serial port and the host PC, and start a terminal emulation application such as HyperTerminal or minicom on the host PC. The serial communication parameters should be 38400 baud, 8 data bits, no parity, 1 stop bit (8N1) and no flow control (handshaking).

Ensure that SW801 is set to the CSn0 position, SW802 is set to OFF and that SW803 is set to 16b. It will also be necessary to configure the JTAG software to make both the FLASH and SDRAM accessible.

It is now necessary to load and run a JTAG build of RedBoot. A prebuilt image `redboot_JTAG.elf` is supplied with eCos, or can be rebuilt as described below. This file is an ELF executable complete with relocation and debug information, and may need to be converted to another format for use with the JTAG software. For example, when using an Ashling Opella unit, the SymFinder utility sfdwarf should be used to convert the ELF executable to a `.CSO` file. This utility may give numerous warnings which can be ignored. The `.CSO` file can be loaded via the JTAG debugger, and it can be started running at location 0x80001000. At this point RedBoot will output text similar to the following on the serial port:

```
**Warning** FLASH configuration checksum error or invalid key
Use 'fconfig -i' to [re]initialize database

RedBoot(tm) bootstrap and debug environment [JTAG]
```

```
Non-certified release, version UNKNOWN - built 15:19:46, Dec  1 2005

Platform: STB220 (Philips PNX8330)
Copyright (C) 2000, 2001, 2002, 2003, 2004 Free Software Foundation, Inc.
Copyright (C) 2003, 2004, 2005 eCosCentric Limited

RAM: 0x80000000-0x84000000, [0x8001aec0-0x83fdd000] available
FLASH: 0x98000000 - 0x98ffffff 128 x 0x20000 blocks
RedBoot>
```

The exact details may vary slightly, depending on the flash chip present and whether or not the ethernet is plugged in yet. At this stage the RedBoot flash management initialization has not yet happened so the warning about the configuration checksum error is expected. To perform the flash initialization use the **fis init -f** command:

```
RedBoot> fis init -f
About to initialize [format] FLASH image system - continue (y/n)? y
*** Initialize FLASH Image System
... Erase from 0x98020000-0x98fdffff: ...............................................................................
...............................................
... Unlocking from 0x98fe0000-0x98ffffff: .
... Erase from 0x98fe0000-0x98ffffff: .
... Program from 0x83fe0000-0x84000000 to 0x98fe0000: .
... Locking from 0x98fe0000-0x98ffffff: .
RedBoot>
```

At this stage the block of flash at location 0x98FE0000 holds information about the various flash blocks, allowing other flash management operations to be performed. The next step is to set up RedBoot's non-volatile configuration values:

```
RedBoot> fconfig -i
Initialize non-volatile configuration - continue (y/n)? y
Run script at boot: false
Update RedBoot non-volatile configuration - continue (y/n)? y
... Unlocking from 0x98fe0000-0x98ffffff: .
... Erase from 0x98fe0000-0x98ffffff: .
... Program from 0x83fe0000-0x84000000 to 0x98fe0000: .
... Locking from 0x98fe0000-0x98ffffff: .
RedBoot>
```

For most of these configuration variables the default value is correct.

It is now possible to load the flash-resident version of RedBoot. Because of the way that flash chips work it is better to first load it into RAM and then program it into flash.

```
RedBoot> load -r -m ymodem -b %{freememlo}
```

The file `redboot_romram.bin` should now be uploaded using the terminal emulator. The file is a raw binary and should be transferred using the Y-modem protocol.

```
CRaw file loaded 0x8001b000-0x80031887, assumed entry at 0x8001b000
xyzModem - CRC mode, 724(SOH)/0(STX)/0(CAN) packets, 4 retries
RedBoot>
```

Once RedBoot has been loaded into RAM it can be programmed into flash:

```
RedBoot> fis create RedBoot -b %{freememlo}
An image named 'RedBoot' exists - continue (y/n)? y
... Unlocking from 0x98000000-0x9801ffff: .
... Erase from 0x98000000-0x9801ffff: .
... Program from 0x8001b000-0x8003b000 to 0x98000000: .
... Locking from 0x98000000-0x9801ffff: .
... Unlocking from 0x98fe0000-0x98ffffff: .
... Erase from 0x98fe0000-0x98ffffff: .
... Program from 0x83fe0000-0x84000000 to 0x98fe0000: .
... Locking from 0x98fe0000-0x98ffffff: .
RedBoot>
```

The flash-resident version of RedBoot has now been programmed at location 0x98000000, and the flash info block at 0x98FE0000 has been updated. The initial setup is now complete and the board can now boot from flash. To do this, power the board down, detach the JTAG module and power the board up. It should produce the following output:

```
+
RedBoot(tm) bootstrap and debug environment [ROMRAM]
Non-certified release, version UNKNOWN - built 14:37:14, Dec  1 2005

Platform: STB220 (Philips PNX8330)
Copyright (C) 2000, 2001, 2002, 2003, 2004 Free Software Foundation, Inc.
Copyright (C) 2003, 2004, 2005 eCosCentric Limited

RAM: 0x80000000-0x84000000, [0x8001b350-0x83fdd000] available
FLASH: 0x98000000 - 0x98ffffff 128 x 0x20000 blocks
RedBoot>
```

When RedBoot issues its prompt it is also ready to accept connections from mipsisa32-elf-gdb, allowing eCos applications to be downloaded and debugged.

Occasionally it may prove necessary to update the installed RedBoot image. This can be done at the ROMRAM RedBoot prompt - there is no need to run the JTAG version again unless the version already installed has been corrupted. It involves loading the new image into RAM using RedBoot's **load** command, and then programming it into flash using **fis create RedBoot**

Rebuilding RedBoot

Should it prove necessary to rebuild a RedBoot binary, this is done most conveniently at the command line. The steps needed to rebuild the ROMRAM version of RedBoot are:

```
$ mkdir redboot_romram
$ cd redboot_romram
$ ecosconfig new stb220 redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/mips/pnx83xx/pnx8330/stb220/current/misc/redboot_ROMRAM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

At the end of the build the `install/bin` subdirectory should contain the required file `redboot_romram.bin`.

Rebuilding the JTAG and RAM versions involves basically the same process. The JTAG version uses the file `redboot_JTAG.ecm` and generates an ELF executable `redboot.elf`, which may need to be converted to another format before it can be used with the JTAG software. The RAM version uses the file `redboot_RAM.ecm` and generates a raw binary `redboot.ram.bin` which can be loaded into memory at 0x80100000 and executed from there.

Name

Configuration — Platform-specific Configuration Options

Overview

The STB220 platform HAL package is loaded automatically when eCos is configured for an STB220 target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The STB220 platform HAL package supports four separate startup types: RAM, ROM, ROMRAM and JTAG. The configuration option `CYG_HAL_STARTUP` controls which startup type is being used. For typical application development RAM startup should be used, and the application will be run via mipsisa32-elf-gdb interacting with RedBoot using either serial or ethernet. It is assumed that the low-level hardware initialization, including setting up the memory map, has already been performed by RedBoot. By default the application will use certain services provided by RedBoot via the virtual vector mechanism, including diagnostic output, but that can be disabled via `CYGSEM_HAL_USE_ROM_MONITOR`.

ROM startup can be used for applications which are programmed into the base of a parallel flash device at 0x98000000. On power up the chip's boot loader will jump to this location and execute the code that is there. The startup code will copy the applications data segment from ROM to RAM at 0x80000000 and zero the BSS. Code execution will continue from ROM. All the hardware will be initialized, and the application is self-contained. This startup type can be used by the flash-resident version of RedBoot, and can also be used for finished applications.

ROMRAM startup can be used for applications which are programmed into the base of flash at 0x98000000. On power up the chip's boot loader will jump to this location and execute the code that is there. The startup code will copy the application from flash to RAM at 0x80000000, and subsequently the flash will not be used for executing any code. All the hardware will be initialized, and the application is self-contained. This startup type is used by the flash-resident version of RedBoot, and can also be used for finished applications.

JTAG startup can be used for applications which will be debugged via JTAG instead of RedBoot. The behaviour is mostly the same as for ROMRAM startup: the application is loaded at 0x80000000 and initializes all the hardware, with no dependencies on services provided by a ROM monitor. There are some minor differences in the startup code, for example it is not necessary to copy the remainder of the application from flash to RAM.

RedBoot and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building RedBoot.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained. That is useful as a testing step before switching to ROM startup. It also allows applications to be run and debugged via JTAG.

If the application does not rely on a ROM monitor for diagnostic services then uart0 will be used for HAL diagnostics and standard output. The default baud rate is controlled by `CYGNUM_HAL_MIPS_PNX83XX_DIAG_BAUD`. Since uart0 is needed by the HAL diagnostics code it cannot be accessed via the serial driver and uart1 should be used for this purpose.

System Clock

The coprocessor 0 COUNTER/COMPARE timer is used for the eCos system clock. The configuration option `CYGNUM_HAL_RTC_PERIOD` controls the value programmed into the compare register. The value of this is calculated from the CPU frequency and the value of `CYGNUM_HAL_RTC_DENOMINATOR`. The calculations are arranged so that

`CYGNUM_HAL_RTC_DENOMINATOR` effectively defines the clock frequency and it the only configuration option that need to be changed to select a different clock rate.

DRAM Type Selection

EV8330 boards can be fitted with either DDR RAM or SDRAM. These require different initialization of the the SDRAM interface controller. This is selected by setting `CYGHWR_HAL_PNX8330_DRAM` to either `SDRAM` or `DDR`. The default is to select SDRAM.

Flash Driver

The platform HAL package contains flash driver support for the external flash device. By default this is inactive, and it can be made active by loading the generic flash package `CYGPKG_IO_FLASH`. The board may use one of a variety of flash chips. The exact type present is determined at run-time using CFI and the system will adjust accordingly.

Ethernet Support

The platform HAL provides the platform-specific support for a single SMSC LAN9118 ethernet device, if the generic ethernet support is enabled. The configuration `CYGNUM_HAL_MIPS_STB220_ETH_ISR_PRIORITY` provides control over the interrupt priority used for this device. The option `CYGNUM_HAL_MIPS_STB220_ETH_ISR_PIO` controls which PIO pin the ethernet interrupt is connected to and `CYGNUM_HAL_MIPS_STB220_ETH_CS` controls which chip select is used to access the ethernet device. These options must match the setting of the jumpers on the board. The board does not have a suitable EEPROM so the MAC address is provided via a RedBoot fconfig option.

Compiler Flags

The platform HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos. There are four flags specific to this port:

`-mips32`
: The mipsisa32-elf-gcc toolchain defaults to supporting the mips32 architecture, so this option is not strictly necessary. However, it is good practice to include it.

`-G0`
: MIPS calling conventions reserve one register for use as a global pointer register. In theory this allows static variables in one 64K area of memory to be accessed using just one instruction instead of two, and the `-G` option provides some control over this. However due to limitations within the current linker all modules have to be compiled with the same `-G` setting, and the compiler support libraries are built with `-G0`. Therefore all eCos and application modules also have to be built with `-G0` and this optimization is not available.

`-EB`
: The eCos port only supports big-endian mode so `-EB` must be specified.

`-msoft-float`
: The PNX8330 does not have a hardware floating point unit so software floating point has to be used instead.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the STB220 hardware, and should be read in conjunction with that specification. The STB220 platform HAL package complements the MIPS architectural HAL, the PNX8330 variant HAL, and the PNX83xx support HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset the HAL will set up some of the on-chip peripherals appropriately for eCos, but most peripherals are left to their default settings. Full details of this initialization can be found in the function `hal_platform_init` in `stb220.c`.

Linker Scripts and Memory Maps

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

External SDRAM
: This is mapped to location 0x80000000. The first four kilobytes are reserved for exception vectors and for some data which needs to be shared between RAM startup applications and the ROM monitor. For ROMRAM or JTAG startup, code will be loaded from 0x80001000 onwards. For ROM startup data will be loaded from 0x80001000 onwards. For RAM startup, code will start at 0x80100000, reserving the bottom 1MB for RedBoot's code and data.

On-chip Peripherals
: These are accessible at location 0xB7000000 onwards.

External Flash
: This is located at location 0x98000000 onwards. Different STB220 boards may come with different flash chips so CFI is used at run-time to query the flash chip and adapt. Typically the first 128K of flash at location 0x98000000 is used to hold RedBoot, and the last flash block at location 0x98DE0000 is used to hold flash management data and the RedBoot **fconfig** variables. The remaining blocks can be used by application code.

Clock Support

The platform HAL provides configuration options for the eCos system clock. This always uses the 4kEc COUNTER/COMPARE registers accessed via coprocessor 0. The actual HAL macros for managing the clock are provided by the MIPS architectural HAL. The specific numbers used are a characteristic of the platform because they depend on the processor speed. The CONFIG unit's timer 0 is used for profiling and its watchdog timer is supported by a watchdog driver.

MIPS16 Support

The STB220 platform HAL enables MIPS16 support in the architecture HAL. This allows *application* code to be compiled using MIPS16 options and linked against the 32 bit mode eCos library.

To compile for MIPS16 the standard flags that are used in eCos and exported to the `ecos.mak` file should be used except that the `-mips32` flag should be replaced by `-mips16 -fwritable-strings`. The `-mips16` option enables MIPS16 compilation and the `-fwritable-strings` option is a work-around for a bug in the compiler.

Other Issues

The STB220 platform HAL does not affect the implementation of other parts of the eCos HAL specification.

Part LXXXIII. NIOS2 Architecture

Table of Contents

Chapter 341. Nios II Architectural Support

Name

CYGPKG_HAL_NIOS2 — eCos Support for the Nios II Architecture

Description

The Altera Nios II is a configurable processor. Using the Altera design suite a system designer can build a custom hardware design. This will involve a Nios II cpu, possibly fine-tuned as appropriate for the application with e.g. appropriate amounts of cache and custom instructions. Next those peripherals needed by the application are added. The resulting design is compiled and programmed into a suitably-sized FPGA. This FPGA will be the heart of a larger circuit containing other devices such as external flash and RAM memories and support chips such as RS232 transceivers. A number of reference hardware designs are included with the design suite. The eCos Nios II architectural HAL package together with supporting HAL packages and device drivers allow eCos to be configured and built for a hardware design. Complete applications can then be built with eCos.

The configurable nature of the Nios II architecture means that the eCos port is implemented somewhat differently from ports to other architectures, although the same basic functionality is available. A conventional eCos port involves an architectural HAL package, a variant HAL, possibly a processor HAL, and a platform HAL. Typically the architectural HAL provides a framework for the lower HALs, interrupt and exception processing, context switch support, and gdb stub debug support. The variant HAL deals with cpu issues which are not common to all members of the basic architecture, for example caches and optional FPU and MMU support. The processor HAL deals with the specific set of devices on one specific chip that implements the basic architecture, for example timers, a uart for diagnostics, and interrupt controllers. Base address for the various devices and their interrupt vectors are also defined here. The platform HAL deals with devices that are off-chip. This includes external flash and RAM, as well as specialized I/O chips on an I²C or SPI bus. Typically the external memory will determine how the system boots up so issues like different startup types are largely handled by the platform HAL.

Exactly what work is done where depends very much on the specific processor, for example an interrupt controller may be supported by the variant HAL instead of the processor HAL if it is expected that all processors implementing that variant will use the same interrupt controller. Typically lower HALs are able to extend or override the behaviour of higher ones as appropriate, for example a platform HAL could redefine the interrupt controller support if the platform includes an external interrupt controller as well as an on-chip one.

The Nios II port still involves an architectural HAL and a platform HAL. The architectural HAL tends to do more of the work than other architectural HALs. For example it provides default implementations of system clock support using an Avalon timer and diagnostics using an Avalon uart. The platform HAL still deals with the off-chip devices such as external flash and RAM. For example the devkit platform HAL CYGPKG_HAL_NIOS2_DEVKIT provides the appropriate support for the Stratix II/2s60_RoHS and the Cyclone II/2c35 boards and compatibles.

There is also a new type of HAL, the hardware design HAL. Typically this does not contain any code, only definitions and eCos configuration support. It contains the information needed by the other HALs and by device drivers to adapt themselves to a specific hardware design, for example the presence of certain cpu instructions, the existence and size of cache, and the base addresses and interrupt vectors assigned to the various on-chip and off-chip devices. Essentially this corresponds to the information manipulated within the SOPC Builder component of the Altera Design Suite. An example of a hardware design HAL is CYGPKG_HAL_NIOS2_CYCLONE2_2C35_TSEPLUS, corresponding to the eCosPro TSEplus hardware design for a Cyclone II/2c35 board.

An eCos target, as defined by an entry in the toplevel ecos.db database, consists of the Nios II architectural HAL, a hardware design HAL, a platform HAL, and a set of device drivers. Typically each target will have its own hardware design HAL, although in some cases multiple targets may use the same hardware design HAL to allow for incompatible device drivers. If a platform HAL is fairly generic then it may be shared between several targets, otherwise a target will need its own platform HAL. Porting eCos to a new Nios II design will typically involve creating a new hardware design HAL, usually by cloning an existing one, and either reusing or cloning an existing platform HAL, followed by creating the appropriate entries in ecos.db.

Note

At the time of writing hardware design HALs are manually written or cloned. It is expected that in a future release these HALs will be generated automatically from the output of SOPC Builder, with little or no need for manual intervention. For simple hardware the need for a separate platform HAL may also be eliminated at that time.

Name

Installation — Generic Instructions

Overview

Typical Nios II hardware for use with eCos will include an FPGA to run the hardware design, a bank each of external parallel NOR flash and SDRAM, an rs232 transceiver plus a uart in the hardware design to act as the diagnostics and debug channel, support logic to initialize the hardware following power up, and any other peripherals appropriate to the application. The external flash is needed to hold the hardware design as well as code and persistent data, and external SDRAM is needed because a typical FPGA cannot provide enough RAM for the system's requirements. Relative to the FPGA and the SDRAM the external flash is usually somewhat slow.

In a self-contained production system both the hardware design and all code will reside in the flash, and the FPGA will start executing code from flash shortly after power up once the hardware design has been loaded. This code will be linked against a build of eCos configured to use a ROM startup. The code can be the actual application, which means that the entire application will execute in place from flash and all hardware resources are available to it. However because of flash speed issues the resulting performance may be poor. An alternative approach is to have a two-stage boot process: the code that runs from flash is the RedBoot ROM monitor, linked using a ROM startup; following system initialization RedBoot then loads the application from flash into RAM and starts it running; this load can be done using either the **fis load** and **go** commands in a boot script, or alternatives such as the Robust Boot Loader `CYGPKG_RBL`. The application is linked against a separate eCos build configured for RAM startup, and can access some facilities provided by RedBoot such as the fconfig persistent data database. However some flash and RAM needs to be reserved for RedBoot.

An alternative available on some platforms is a ROMRAM startup. Here the application starts up as per a ROM startup, but then copies itself into RAM and continues executing from there, avoiding the performance problems of executing from flash. There is no RedBoot in the system so no resources need be reserved for it, but on the other hand its facilities are not available.

Another alternative is to use an EPCS serial flash chip instead of or in addition to the parallel flash. The hardware design and the boot code both reside in the EPCS chip, and the FPGA will load the design on power up. The design automatically includes a very simple bootloader provided by Altera which runs, copies the main boot code from the EPCS chip to RAM, and then jumps to the entry point.

Obviously during software development it is undesirable to reprogram the parallel flash or the EPCS chip following every build. Instead the nios2-elf-gdb debugger will load the application into RAM and run it under debugger control, complete with facilities such as breakpoints and single stepping. The debugger can be run either directly from the command line or from inside an integrated development environment, depending on the user's preferences and the tools available.

nios2-elf-gdb can interact with the target in two ways. It can use dedicated debug hardware such as the jtag-based Altera USB Blaster. Alternatively it can communicate with the RedBoot ROM monitor over either an rs232 connection or an ethernet network. Both approaches have advantages and disadvantages. Debugging via jtag can be less intrusive on the application's behaviour, but some functionality such as RedBoot's fconfig data will not be available. If the hardware design incorporates hardware breakpoints then jtag also offers limited capabilities for debugging code that executes from flash. RedBoot has some knowledge of eCos internals so debugging via RedBoot also provides some advanced functionality such as thread-aware debugging. Applications that will be debugged via RedBoot should be linked against an eCos built configured for RAM startup. Note that this means that exactly same application image can later be programmed into flash in a production system. Applications that will be debugged via jtag should be linked against an eCos built configured for RAMJTAG startup.

Preparing a board for debugging via jtag simply involves setting up the hardware and software in accordance with the appropriate Altera documentation. If the gdb hwdebug diagnostics channel is used then it will also be necessary to run the gdb command **set hwdebug** to activate that channel. Debugging via RedBoot is more complicated because it means installing RedBoot and initializing the **fis** and **fconfig** persistent data. This is covered in more detail below. Typically the RedBoot image resides in the first 128K of flash, the last 64K of flash is used for RedBoot's persistent data, and the first 64K of external SDRAM is reserved for use by RedBoot.

Embedded hardware varies widely and eCos is itself a highly configurable operation system, so many variations of the above setup are possible. Hence some platforms will have their own installation instructions and the platform HAL documentation should be consulted first before proceeding with the instructions below. In addition some of the information such as the location of RedBoot depend on the hardware design and possibly the platform HAL, so again the appropriate documentation should be consulted for details.

Installing RedBoot

Typically the hardware design, specifically the `.sof` file that is the final result of compilation, needs to be programmed into flash alongside RedBoot and everything else. This can be done either using the graphical tools provided with Quartus and the Altera IDE, or using command line tools like sof2flash and nios2-flash-programmer. These tools require a jtag connection to the target, and the Altera documentation should be consulted for further details. The location of the hardware design within the flash is determined by the board's reset logic so that information can be found in the hardware design and platform HAL documentation.

The next step is to program a ROM-startup build of RedBoot into flash at the reset location. Typically the external flash is placed at location 0x00000000 within the address map and execution will start from that address, but this may be changed within the hardware design. eCosPro releases for a supported target will come with prebuilts of RedBoot in a variety of file formats (usually the ELF executable, raw binary, and S-records). Alternatively RedBoot can be rebuilt for the target as per the standard RedBoot documentation. This may either generate the various file formats automatically or it may be necessary to use nios2-elf-objcopy to convert from the ELF executable to other formats.

There are two main ways of getting RedBoot into flash. The first is to use the Altera utilities, either the graphical ones or the command-line tools. For example nios2-flash-programmer can be used together with the S-record version of the RedBoot build. When the board is reset afterwards RedBoot will start running, sending output out of the board's first serial port and providing a prompt. Starting up a terminal emulator program on the host will allow the user to see this output and to invoke RedBoot commands. The rs232 parameters will depend on the hardware design, but typically RedBoot will communicate at 115200 baud, 8 bits, no parity, 1 stop bit. At this stage there will be a number of warnings from RedBoot about uninitialized **fis** and **fconfig** settings, which is to be expected since the relevant initialization commands have not been run yet. These commands are **fis init** and **fconfig -i**, and the RedBoot documentation should be consulted for further information.

If the hardware design image is held in flash then it is usually a good idea at this stage to create one or more **fis** entries, marking the relevant part of the flash as in use and preventing RedBoot from overwriting the image by accident. In theory this will also allow RedBoot commands to be used to update the hardware design image if needed, although the jtag flash utilities are generally more convenient for this. For example, on a Stratix II/2s60_RoHS board the current hardware design resides at offset 0x00800000 within the flash and for safety there is also a factory fallback design at 0x00C00000.

```
RedBoot> fis create -f 0x00800000 -l 0x00400000 -n hwdesign_user
RedBoot> fis create -f 0x00C00000 -l 0x003F0000 -n hwdesign_bak
```

Here the `-f` specifies the address, `-l` the length, and `-n` prevents RedBoot from initializing the relevant parts of flash and thus overwriting the hardware designs already programmed.

If the hardware design includes a system id register then during initialization RedBoot will check the current value of that register with the setting defined in the hardware design HAL. A mismatch indicates that the RedBoot build does not correspond to the hardware design being used, so RedBoot may not be fully functional and may even fail before getting as far as providing a prompt. Sometimes this warning is innocuous because, even though a custom hardware design is being used, it is fully compatible with the one RedBoot was built for (same memory map, same interrupt vector assignments, same settings for those peripherals used by RedBoot). If so the warning can either be ignored or RedBoot can be rebuilt with an updated system id value.

The alternative approach to installing RedBoot via the jtag flash programming utilities is to go via a RAM build of RedBoot. This has the advantage of supplying the user with more information as the installation process proceeds. For example if there is a problem with the way the hardware design attempts to access flash then a RAM build of RedBoot may still function to some extent but report errors whenever attempting to access the flash, whereas a ROM build of RedBoot trying to execute from that flash may fail silently.

The file that is needed is an RedBoot ELF executable for a RAM startup build (RedBoot is not a typical eCos application and for it there is no difference between the RAM and RAMJTAG startup types). Again eCosPro releases for a supported target will come with prebuilts, alternatively Redboot can be rebuilt in the usual way. The RedBoot build will be run via nios2-elf-gdb over jtag, so typically this will first involve starting up nios2-gdb-server:

```
$ nios2-gdb-server --tcpport 9000
```

And then at a separate shell prompt:

```
$ nios2-elf-gdb <path>/redboot.elf
(gdb) target remote localhost:9000
(gdb) set $status=0
(gdb) load
(gdb) continue
```

Setting `$status` to 0 has the effect of disabling interrupts, useful if the board was previously used to run an application and that run left interrupts enabled. If an interrupt is pending then as soon as the **continue** command is executed and RedBoot starts running, before it has a chance to run a single instruction let alone initialize the system appropriately for interrupt handling, an interrupt exception will occur. Disabling interrupts before the **continue** avoids any such problems.

Once the RAM RedBoot starts running it should send output out of the board's first serial port and provide a prompt, as before. If this does not happen it indicates a mismatch between the RedBoot build and the hardware design, for example the RedBoot build being used may be for a different target or the hardware design currently running on the board may not be the one that RedBoot was built for. If the discrepancy is minor then the RAM RedBoot may still provide some diagnostics indicating what is wrong, for example it may warn about a system id mismatch.

Once the RAM RedBoot is up and running the **fis init** and **fconfig -i** commands can be run as before, and fis entries for any hardware design(s) in reserved areas of flash can be created. Finally a ROM version of RedBoot can be uploaded:

```
RedBoot> load -r -m ymodem -b %{freememlo}
```

The raw binary version of the RedBoot build, typically `redboot.rom.bin` should now be uploaded via the host's terminal emulator program using a ymodem transfer. Once this has finished the ROM RedBoot can be installed into flash, and a **reset** command will restart the processor from the reset vector which should now be RedBoot code.

```
RedBoot> fis create RedBoot -b %{freememlo}
…
RedBoot> reset
```

Updating RedBoot

Sometimes it may be necessary or desirable to install a new version of RedBoot, either because the hardware design has changed or because of a software update. The new version can be installed using the above instructions, usually skipping the **fis** and **fconfig** initializations because there is no need to repeat these. Alternatively it is also possible to use an existing still-functional ROM RedBoot to run up a RAM RedBoot and then use the latter to install a new ROM RedBoot - obviously it is not possible for a ROM RedBoot to update itself because it would be overwriting its own code. This has the advantage of not requiring jtag unless something goes badly wrong and major recovery is needed, but it will take somewhat longer. At the ROM RedBoot prompt:

```
RedBoot> load -r -m ymodem -b <address>
RedBoot> go
```

Again this involves a ymodem transfer from the host to the target, this time using the raw binary file `redboot.ram.bin`. The address should be the execution address for RAM applications, typically 64K into external RAM but the details will depend on the hardware design and the platform. It is also possible to skip the `-r` option and the address and transfer an ELF executable, but this involves a larger transfer so will take more time. The **go** command transfers control to the RAM RedBoot, and the ROM RedBoot binary can then be uploaded and programmed into flash as before.

Name

Options — Configuring the Nios II Architectural HAL Package

Description

The Nios II architectural HAL is included in all ecos.db entries for Nios II targets, so the package will be loaded automatically when creating a configuration. It should never be necessary to load the package explicitly or to unload it.

The architectural HAL contains a number of configuration options, although relatively few compared with many other architectures. This is because many aspects of the system that would normally be handled at compile-time by eCos configuration options can instead be controlled during the hardware design process. For example, the uart used for the diagnostics and debug channel can be hardwired to a particular baud rate, eliminating the need to control this via an eCos configuration option.

The most important configuration options in the architectural HAL are `CYGBLD_GLOBAL_CFLAGS` and `CYGBLD_GLOBAL_LDFLAGS`, the default compiler flags for building eCos and for linking testcases. The flags are exported in the file `install/include/pkgconf/ecos.mak` in the eCos build tree and are usually used for building application code as well, either as is or after any appropriate changes. Users may wish to change these flags, for example to build eCos with different compiler optimization settings. The default values of these options adapt to information provided by the hardware design HAL about the cpu's capabilities, for example the availability of the various multiply and divide instructions.

`CYGSEM_HAL_COMMON_INTERRUPTS_FIXED_GP` affects the way eCos interacts with the global pointer register or gp. The Nios II ABI uses gp to access certain variables, allowing up to 64K worth of data to be accessed via a single instruction rather than two separate instructions. The gp register is initialized during application startup. The use of a global pointer can cause problems. For example when the system involves a ROM RedBoot and a RAM application, both are built independently and have their own area for small variables and hence their own gp values. Whenever the system switches between code running in the application and code running in RedBoot the gp register must be switched as well. That means when an interrupt or exception occurs the gp register may have the wrong value for the interrupt or exception handler, and hence the gp register must be saved, updated, and restored alongside other parts of the cpu state. This adds to the interrupt latency. If it is known that there will only be one application running in the system, e.g. when using ROM or RAMJTAG startup, then the system can assume a fixed global pointer register and there is no need for the interrupt and exception handlers to do anything with it. This configuration option controls the gp behaviour, and usually its default value determined from other configuration options will be appropriate.

A build of ROM RedBoot normally includes gdb stubs support, including breakpoints. Usually ordinary breakpoints involve inserting a special breakpoint into the image in memory, and hence debugging is generally limited to applications executing from RAM. On most architectures the host-side gdb will implement this via a memory write using the gdb remote protocol and no special stub support is needed for that. The Nios II architecture has two breakpoint instructions, one for use when debugging over jtag and one for use when debugging via a target-side gdb stub like the one in RedBoot. nios2-elf-gdb defaults to using the jtag version of the breakpoint instruction, which would break debugging via RedBoot. Hence the Nios II implementation of the gdb stubs code uses a different mechanism, involving a target-side list of the breakpoints that have been set and inserting the appropriate breakpoint instruction on the target-side rather than via memory writes initiated from the host. The gdb stubs code works without dynamic memory allocation so this list is a fixed size, and hence only a fixed number of breakpoints are supported. By default this limit is set to 25 breakpoints, which should suffice for all but the most complicated debug scenarios. If it should ever be necessary to have more breakpoints then the limit can be increased by changing the configuration option `CYGNUM_HAL_BREAKPOINT_LIST_SIZE` in a ROM Redboot configuration and rebuilding and installing the updated RedBoot.

Name

HAL Port — Implementation Details

Description

This documentation explains how the eCos HAL specification has been mapped onto Nios II hardware and should be read in conjunction with that specification. It should be noted that the architectural HAL is usually complemented by a hardware design HAL and a platform HAL, and those may affect or redefine some parts of the implementation.

Exports

The architectural HAL provides header files `cyg/hal/hal_arch.h`, `cyg/hal/hal_intr.h`, `cyg/hal/hal_cache.h`, `cyg/hal/hal_io.h` and `cyg/hal/arch.inc`. These header files export the functionality provided by all the Nios II HALs for a given target, automatically including headers from the lower-level HALs as appropriate. For example the platform HAL may provide a header `cyg/hal/plf_io.h` containing additional I/O functionality, but that header will be automatically included by `cyg/hal/hal_io.h` so there is no need to include it directly.

One header file is worth a special mention: `pkgconf/nios2_hwconfig.h`. This file is provided by the hardware design HAL and contains definitions such as base addresses and interrupt vectors. It is automatically included and used by the architectural HAL headers, but its contents may prove useful to application developers.

Data Types

The architectural HAL assumes that the Nios II cpu in the hardware design uses 32-bit arithmetic, little-endian byte ordering, and software floating point.

Startup

The architectural HAL provides a default implementation of the low-level startup code which will be appropriate in nearly all scenarios. For a ROM startup this includes clearing the instruction and data caches, if present, and copying initialized data from flash to RAM. For all startup types it will involve zeroing bss regions and setting up the stack and the general C environment. It may also include installing the exception vector code at the desired location as well as copying code and data to on-chip RAM or external SRAM as required. The platform HAL can override or extend this as required. The code assumes that all of flash is directly accessible. Platform HALs may override this as required, for example when booting from a serial flash rather more work is needed to copy data from flash to RAM. More information on the low-level startup code can be found in the source file `src/vectors.S`.

The architectural HAL also implements the next stage of the startup code, including initializing the VSR and virtual vector tables, setting up HAL diagnostics, and invoking C++ static constructors, prior to calling the first application entry point `cyg_start`. This code resides in `src/nios2.c`.

The current code assumes that there is no memory management or MMU and hence will not perform any MMU initialization.

Interrupts and Exceptions

The architectural HAL provides default implementations of `HAL_DISABLE_INTERRUPTS`, `HAL_RESTORE_INTERRUPTS`, `HAL_ENABLE_INTERRUPTS` and `HAL_QUERY_INTERRUPTS`. These just involve simple manipulation of the `status` control register. Similarly there are default implementations of the interrupt controller macros `HAL_INTERRUPT_MASK`, `HAL_INTERRUPT_UNMASK`, and `HAL_QUERY_INTERRUPT_MASKED` macros. These are slightly more complicated to cope with nested interrupt scenarios, involving a shadow mask as well as the `ienable` control register. `HAL_INTERRUPT_ACKNOWLEDGE` is a no-op because the hardware has no need for clearing an interrupt centrally. Instead interrupts must be acknowledged within each device as appropriate, using device-specific code.

`HAL_INTERRUPT_CONFIGURE` is a no-op. This macro is normally only relevant to GPIO interrupts, affecting level versus edge triggering, and on a Nios II an entire GPIO unit generates a single interrupt but the various inputs to that port can be controlled individually. Instead it is up to application code to set the various registers within each GPIO unit appropriately.

HAL_INTERRUPT_SET_LEVEL is also a no-op. However the architectural HAL does support prioritized nested interrupts when CYGSEM_HAL_COMMON_INTERRUPTS_ALLOW_NESTING is enabled. The implementation assumes that interrupt vector 0 has the highest priority, down to interrupt vector 31 as the lowest. In other words, assuming nested interrupt support is enabled, if the cpu is busy processing an interrupt from the device attached to interrupt vector 9 and an interrupt 0 occurs then the latter will be handled immediately and the processing of vector 9 resumes later. Since the assignment of interrupt vectors to devices in SOPC Builder is arbitrary this gives full flexibility without the overheads of managing priorities in software.

Interrupt handlers are managed by a simple table cyg_hal_interrupt_handlers so the implementation of HAL_INTERRUPT_ATTACH, HAL_INTERRUPT_DETACH, and HAL_INTERRUPT_IN_USE is straightforward.

By default interrupt handlers run on a separate interrupt stack. This saves memory because there is no need to allow for interrupt processing overhead on every thread stack. However switching to the interrupt stack requires a number of extra instructions so increases the interrupt latency. If the latter is more important than memory usage then CYGIMP_HAL_COMMON_INTERRUPTS_USE_INTERRUPT_STACK should be disabled.

That leaves VSR management and exceptions. On the Nios II interrupt and exception processing is somewhat simpler than on most other architectures. Typically the cpu indirects through a table in memory, the VSR table, with the table index depending on the interrupt vector or the exception being thrown. The eCos macro HAL_VSR_SET updates an entry in the table, allowing applications to take over completely certain interrupt sources and process them as quickly as possible, bypassing the overheads of the default general-purpose VSR handler used by eCos. For example a custom VSR written in assembler could save only a few registers before manipulating the hardware, whereas the general-purpose VSR handler needs to save much of the cpu state before calling the application's interrupt handler written in C. The net result is reduced interrupt latency for critical interrupts, at the cost of more complicated application code.

The Nios II implementation is very different. When an interrupt or exception occurs the cpu jumps to a fixed location in memory defined in the hardware design, the exception vector. The code at that location needs to determine whether it was invoked as the result of an interrupt or a processor exception. There is no hardware equivalent of the VSR table. eCos needs to provide the implementation of the code at the exception vector and there is no simple way for applications to provide a customized version. That makes it difficult for an application to handle critical interrupts with a minimum latency. It also causes other complications, for example it makes it difficult for a RAM startup eCos application to handle interrupts while the gdb stubs inside RedBoot handle exceptions including breakpoints.

To avoid these problems, the Nios II architectural HAL implements a VSR table in software. The code at the exception vector simply indirects through slot 0 of the VSR table, which involves a three instruction overhead compared with a more conventional implementation. Applications needing very fast handling of critical interrupts can use HAL_VSR_SET to install a custom handler in slot 0. That handler can check whether or not a critical interrupt is pending and process it immediately, Otherwise it can chain to the original VSR handler. The overall effect is to provide a mechanism for very fast handling of certain interrupts, at the cost of an extra three instructions for ordinary interrupts which are handled by interrupt handlers written in C.

The default handler for VSR 0 checks whether or not any interrupts are pending. If so then it saves the current cpu state, or the minimum subset thereof if CYGDBG_HAL_COMMON_INTERRUPTS_SAVE_MINIMUM_CONTEXT is enabled, switches to the interrupt stack if necessary, and invokes the appropriate interrupt handler installed via HAL_INTERRUPT_ATTACH or the higher-level functions like cyg_interrupt_create or cyg_drv_interrupt_create.

If the default handler for VSR 0 determines that no interrupt is pending then it must have been invoked as the result of a processor exception. The cpu provides only minimal support for detecting the nature of the exception. A breakpoint trap can be detected by examining the instruction that caused the exception. Anything else is treated as an illegal instruction. Both exceptions are processed by indirecting through further slots in the VSR table, thus allowing the gdb stubs code inside RedBoot to handle exceptions inside an eCos application. It should be noted that this mechanism is critically dependent on reliable interrupt reporting. If it is possible for an interrupt line to glitch, causing an interrupt but leaving the ipending register clear before the VSR 0 handler reads it, then this will be interpreted as an illegal instruction exception. Conceivably this can also affect device drivers if a write to a hardware register may result in an interrupt being cleared in a couple of cycles just as that interrupt is about to happen.

The full implementation of the interrupt and exception handling code can be found in src/nios2asm.S, and that code can be used as the starting point for custom application VSRs.

Stacks and Stack Sizes

cyg/hal/hal_arch.h defines values for minimal and recommended thread stack sizes, CYGNUM_HAL_STACK_SIZE_MINIMUM and CYGNUM_HAL_STACK_SIZE_TYPICAL. These values depend on a number of configuration options. Specifically if the use of a separate interrupt stack CYGIMP_HAL_COMMON_INTERRUPTS_USE_INTERRUPT_STACK is disabled to reduce interrupt latency then thread stacks have to be rather larger to cope with the interrupt processing overhead. If nested interrupts CYGSEM_HAL_COMMON_INTERRUPTS_ALLOW_NESTING are also enabled then thread stacks must be much larger.

The Nios II architectural HAL always provides a separate stack to run the startup code and for exception processing. This stack will also be used for interrupts if CYGIMP_HAL_COMMON_INTERRUPTS_USE_INTERRUPT_STACK is enabled, and its size is determined by CYGNUM_HAL_COMMON_INTERRUPTS_STACK_SIZE.

Thread Contexts and setjmp/longjmp

cyg/hal/hal_arch.h defines a thread context data structure, the context-related macros, and the setjmp/longjmp support. The implementations can be found in src/nios2asm.S. The context structure is straightforward, containing space for the integer registers and the status, ienable and ipending registers. Any floating point arithmetic is assumed to be implemented in software. A single data structure is used for the thread context in all eCos configurations. However some fields will only be used in certain configurations, for example when CYGDBG_HAL_COMMON_INTERRUPTS_SAVE_MINIMUM_CONTEXT is disabled. The use of a single data structure avoids complications when debugging a RAM startup application on top of a ROM RedBoot because both need to use the same structure.

Bit Indexing

The architectural HAL provides an assembler implementation of HAL_LSBIT_INDEX, optimized for faster context switching to higher priority threads. HAL_MSBIT_INDEX uses a straightforward C implementation since it is not performance-sensitive.

Idle Thread Processing

The Nios II instruction set does not include an idle instruction so the idle thread always simply spins, rather than suspend the cpu until an interrupt occurs.

Clock Support

The architectural HAL provides a default implementation of the various system clock macros such as HAL_CLOCK_INITIALIZE. These macros assume that the hardware design implements the system clock using a simple Avalon timer. A platform HAL can provide an alternative implementation if necessary but that is unlikely ever to be necessary because including an Avalon timer in the design is generally straightforward. The timer may be designed with either a fixed period, typically 10 milliseconds to give a 100Hz system clock, or with a variable period in which case there will a configuration option CYGNUM_HAL_RTC_PERIOD to control the clock frequency. Ideally the system clock should be designed with the readable snapshot option enabled. Otherwise the HAL will not be able to provide the HAL_CLOCK_READ macro and there will be no support for clock timings with a finer granularity than the interval between interrupts.

HAL I/O

The various I/O macros for accessing hardware registers such as HAL_READ_UINT8 are implemented using the **ldbuio** and related instructions. This ensures that all I/O accesses bypass any data cache that may be included the hardware design.

Cache Handling

The hardware design may include instruction and data caches, and there is some control over parameters such as the cache sizes and the line sizes. cyg/hal/hal_cache.h defines those cache macros which are appropriate for the current hardware design, typically the `INVALIDATE`, `INVALIDATE_ALL`, `FLUSH`, `STORE` and `SYNC` ones. The hardware does not allow the caches to be enabled or disabled so the `IS_ENABLED` macro always returns a 1 and the `ENABLE` and `DISABLE` macros are never defined. Note that this may confuse some existing code which assumes that these macros are always available.

In addition `cyg/hal/hal_arch.h` defines macros `CYGARC_CACHED_ADDRESS` and `CYGARC_UNCACHED_ADDRESS` which assume that the cpu only addresses 31 bits worth of address space, and that addresses with the top bit set access the same memory locations as those with the top bit clear, but bypassing the cache. This is the default behaviour for Nios II processors. The header file also provides a `HAL_MEMORY_BARRIER` macro which issues a **sync** instruction to cause pending memory operations to complete.

Linker Scripts

The architectural HAL will generate the linker script for eCos applications. This involves the architectural file `src/nios2.ld` and a `.ldi` memory layout file, typically provided by the platform HAL but using some definitions from the hardware design HAL. It is the `.ldi` file which places code and data in the appropriate places for the startup type, but most of the hard work is done via macros in the `nios2.ld` file. This includes macros for placing code and data in on-chip RAM as well as external flash and SDRAM, using linker sections `.iram_text`, `.iram_data` and `.iram_bss`. Code should only be placed in on-chip RAM if the hardware design includes a connection between the RAM and the cpu's instruction master port.

Diagnostic Support

By default the architectural HAL provides a diagnostics and debug channel using the first uart in the hardware design. If the design does not include any uarts then all diagnostics output will instead be discarded. The configuration option `CYGIMP_HAL_NIOS2_DIAGNOSTICS_PORT` can be used to select discard mode even when a uart is available. If the hardware includes an ethernet device then debugging is still possible over the network. Alternatively when debugging via JTAG it is possible to direct the diagnostics output to a gdb hwdebug file I/O channel. By default this will also discard diagnostics output. However if the application is running inside a gdb session and the gdb **set hwdebug** command has been used then the diagnostics will be output via gdb. Platform HALs may implement alternative diagnostics facilities.

SMP Support

The Nios II architectural HAL does not provide any SMP support.

Debug Support

The architectural HAL provides basic support for gdb stubs. Due to a conflict between jtag and stubs gdb support, breakpoints are always implemented using a fixed-size list of breakpoints, as per the configuration option `CYGNUM_HAL_BREAKPOINT_LIST_SIZE`. A hardware design may include hardware breakpoint support but these are not accessible to the gdb stubs code, only via jtag. Hence if a debug session requires the use of hardware breakpoints, for example when debugging code in flash, a jtag-based debug solution must be used instead of gdb stubs.

HAL_DELAY_US() Macro

`cyg/hal/hal_intr.h` provides a simple implementation of the `HAL_DELAY_US` macro using a busy loop. It requires that the hardware design HAL provides a count value `HAL_NIOS2_DELAY_US_LOOPS` appropriate to the cpu speed and the absence or presence of the instruction cache.

Other Functionality

If the hardware design includes a system id register then all RedBoot builds will include some extra initialization code checking the register's current value against the one specified by the hardware design HAL, reporting any mismatches. This helps to guard against accidentally running the wrong build of RedBoot or the wrong hardware design. Note that if there is a serious incompatibility between the two then system bootstrap may fail long before this check gets to run, or the diagnostics channel may be inoperable preventing the warning from reaching the user.

The architectural HAL provides the support needed by the gprof profiling package. This includes the `mcount` needed for callgraph profiling and `hal_enable_profile_timer` for timer-based profiling. The latter can be implemented in two ways. If the hardware design includes a dedicated Avalon timer labelled `“profiling”` then this will be used. Otherwise the sys_clk timer will be used for profiling as well as for the main system clock. Overloading the system clock in this way is less desirable because it means the profiling sampling is likely to miss any code that runs after clock events.

Otherwise the Nios II architectural HAL only implements the functionality provided by the eCos HAL specification and does not export anything extra.

Chapter 342. Nios II Stratix II/2s60_RoHS and Cyclone II/2c35 Platform HAL

Name

`CYGPKG_HAL_NIOS2_DEVKIT` — eCos Platform HAL Support for the Stratix II/2s60-RoHS and Cyclone II/2c35 Boards

Description

The package `CYGPKG_HAL_NIOS2_DEVKIT` provides platform HAL support for the Altera Stratix II/2s60-RoHS and Cyclone II/2c35 development boards, and may also be usable with various other boards depending on compatibility. It is always used in conjunction with the Nios II architectural HAL `CYGPKG_HAL_NIOS2` and with a hardware design HAL which provides details of the design that is programmed into the FPGA.

The main characteristics of both boards are as follows:

1. An FPGA which gets loaded on power up with a hardware design. The eCos Nios II HALs assume that this hardware design consists of a Nios II processor plus various peripherals. Different FPGAs are used on the different boards.

2. A 16MB AMD flash device or compatible. This is a 16-bit device but is attached to the FPGA via an 8-bit data bus. Part of the flash memory is used to hold the current hardware design and a factory default design. On the Stratix II board these reside at offsets 0x00800000 and 0x00C00000 respectively. On the Cyclone II board the offsets are 0x00C00000 and 0x00E00000. In a typical eCos setup this flash will also hold a RedBoot image and an area for RedBoot's **fconfig** and **fis** persistent data. It may also hold one or more application images which RedBoot can load into RAM, and any other persistent data that the application needs.

3. Reset circuitry which on power up loads the hardware design from the flash device into the FPGA. This happens before any code starts running so eCos does not interact with this circuitry in any way.

4. 32MB of SDRAM. In a typical setup the first 64K of this is reserved for RedBoot data and for special bits of code and data such as the Nios II exception vector, while the remainder holds the application code and data.

5. 2MB of SRAM. Typically eCos does not use any of this so all of it is available to the application, and there is linker script support for placing code and data there. Note that if the hardware design places the exception vector in SRAM then eCos will use a small amount of memory at the start of SRAM. Hardware designs can also include on-chip RAM or IRAM which applications can use in much the same way as SRAM.

6. A transceiver for a single uart. This is normally used by eCos as the diagnostics and debug channel so all hardware designs should include a uart connected to the transceiver.

7. A jtag connector, providing an alternative debug mechanism to RedBoot's gdb stubs. The jtag connector can also be used for programming the flash, for example when installing a new hardware design.

8. A lan91c111 ethernet chip. This can be used by one of the eCos TCP/IP stacks and for network debugging. Alternatively network connectivity can be provided by an ethernet device incorporated into the hardware design.

9. A row of LEDs and a dual seven-segment display.

10. Various other devices such as switches which are not directly supported by eCos, and which instead are left available for application use. The Stratix II and Cyclone II boards also come with expansion connectors so that additional hardware can be connected.

Installation

Generic instructions for setting up a board for use with eCos are provided by the Nios II architectural HAL documentation. The Stratix II and Cyclone II boards have no special requirements so the generic instructions should be followed. The address map depends on the hardware design so the documentation for the hardware design HAL should be consulted for details.

Configuration Options

The devkit platform HAL package should be loaded automatically when eCos is configured for appropriate target hardware. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

The main configuration option in this package is `CYG_HAL_STARTUP`. This can take the following values:

`ROM`

This startup type should be used for code that should execute from flash. The executable will start executing from the Nios II reset vector and will contain all the low-level initialization code needed to bring up the system. Typically this startup type is only used for the RedBoot ROM monitor. On the Stratix II and Cyclone II boards the flash is accessed via an 8-bit data bus so executing code from flash will be slow, even with an instruction cache. Instead application code will normally execute in RAM. It will either be loaded during a gdb debug session, or it can be loaded from flash and started automatically by RedBoot using the **load** and **go** commands.

`RAM`

This startup type should be used for applications that should execute from RAM, either via a gdb debug session interacting with RedBoot's gdb stubs or loaded from flash by RedBoot. eCos will assume that certain services such as diagnostics channel support will be provided by RedBoot and can be accessed via the virtual vector mechanism.

`RAMJTAG`

This startup type is also for applications that should execute from RAM. However it is assumed that the application is not running on top of RedBoot and cannot access any services via the virtual vector mechanisms. Instead the HAL diagnostics channel will be implemented by direct access to the uart, and functionality like RedBoot's **fconfig** settings will not be available at all. This startup type is used primarily when debugging an application via jtag instead of via RedBoot's gdb stubs.

Builds of RedBoot are a special case. For a RAM-resident version of RedBoot there is no difference between the `RAM` and `RAMJTAG` startup types.

If the hardware design includes support for the lan91c111 ethernet chip and if the eCos configuration involves ethernet networking then there are two options related to the lan91c111 device. `CYGDAT_HAL_NIOS2_DEVKIT_ETH_LAN91C111_NAME` specifies the name of the ethernet device, defaulting to `“eth0”`. There should be no need to change this unless the hardware design includes other ethernet devices and their drivers use the same name for the device instances. `CYGDAT_HAL_NIOS2_DEVKIT_ETH_LAN91C111_MAC` specifies the fallback ethernet station address or MAC address. The Stratix II and Cyclone II boards do not have a serial eeprom or similar hardware to provide a unique MAC address, so instead this address has to be provided by software. Usually it will be held in flash as a RedBoot **fconfig** persistent variable. However if the **fconfig** entries have not been initialized, or if they are not accessible because the application is being debugged over jtag and uses the `RAMJTAG` startup type, then the lan91c111 ethernet device will instead use the fallback MAC address specified by this configuration option. No two boards on the same network should ever use the same MAC address so great care must be taken when debugging multiple boards over jtag using the same eCos configuration.

The devkit platform HAL contains a number of other configuration options but these are mainly for internal use by other packages and it should not normally be necessary to edit their values manually.

The HAL Port

The devkit platform HAL does not change the Nios II architectural HAL's implementation of the eCos HAL specification. It does provide the platform-specific linker script support. This takes into account the supported startup types, the locations of the Nios II reset and exception vectors, and the presence of SRAM and any on-chip IRAM.

eCos itself does not use any of the SRAM or the IRAM, except as necessary for exception processing. Instead these memories are available for use by the application. Any code that should end up running from SRAM should be placed in a `.sram_text` section.

Similarly initialized data should be placed in `.sram_data`, and uninitialized data should go in `.sram_bss`. All uninitialized data will be zeroed by the eCos startup code. The equivalent sections for on-chip IRAM are `.iram_text`, `.iram_data` and `.iram_bss`. The platform HAL comes with a testcase `tests/memories.c` which also serves as an example of how to use this functionality.

Other Functionality

If the hardware design includes GPIO units for the row of LEDs and for the dual seven-segment display then the platform HAL provides a number of utility functions for manipulating these.

```
#include <cyg/hal/hal_io.h>

externC void    hal_nios2_led_set(int /* which */, int /* on */);
externC void    hal_nios2_led_set_all(int /* settings */);
externC void    hal_nios2_set_7seg(int /* left */, int /* right */);
```

There is a testcase `tests/lights.c` which also serves as an example of how to use functions.

At the end of its initialization sequence RedBoot will switch all the LEDs off, thus setting them to a known state, and it will set the seven-segment display to `rb`, acting as a visual hint to the user that initialization is complete.

Reusing the Devkit Platform HAL

The devkit platform HAL is intended for use with the Stratix II/2s60_RoHS and Cyclone II/2c35 boards. However it should be usable as is with various other boards, or need only minor changes. The important points to note are as follows:

1. The devkit platform HAL assumes that there is an AMD-compatible flash device on the board, and that it is a 16-bit device attached to the FPGA via an 8-bit bus. The exact flash device and its size do not matter since a run-time CFI query is performed to determine the actual characteristics of the flash device. However if the flash device is not AMD-compatible, not a 16-bit device, or not attached via an 8-bit bus, then the file `src/devkit_flash.c` will have to be changed. With default configuration settings it is also assumed that the Nios II reset vector is at the start of the flash, that the first 128K of flash are available for RedBoot code, and that the last 64K of flash can be used for RedBoot's **fis** and **fconfig** persistent data.

2. The linker script assumes that there is a bank of SDRAM and that this will be used as the main location for application code and data. The size of the SDRAM is obtained from the hardware design HAL. Typically the first 64K is reserved for RedBoot's data area and for special bits of code such as the Nios II exception vector.

3. The SRAM and on-chip IRAM are optional. The devkit platform HAL checks whether or not the hardware design HAL provides base address and size definitions for these.

4. Usually at least one uart is essential to provide the diagnostics and debug channel.

5. The lan91c111 ethernet chip is optional. The devkit platform HAL will only instantiate the ethernet device if the hardware design HAL specifies that this chip is present.

6. Similarly the row of LEDs and the dual seven-segment display are optional and support for these is only provided if the hardware design includes GPIO units labelled “led” and “seven_seg”

Chapter 343. Nios II Cyclone II/2c35 Standard H/W Design HAL

Name

CYGPKG_HAL_NIOS2_CYCLONE2_2C35_STANDARD — eCos Support for the Standard Hardware Design on a Cyclone II/2c35 Board

Description

This package provides the hardware design HAL for the standard hardware design running on a Cyclone II/2c35 board. This design is provided with the Altera Nios II Embedded Design Suite in the directory `nios2eds/examples/vhdl/niosII_cycloneII_2c35/standard/`. It includes the following functionality:

CPU
: A Nios II/s processor running at 85MHz. This has 4K of instruction cache and no data cache. It has level 1 jtag support only with no hardware breakpoints. The reset vector is at address 0x00000000 in external flash and the exception vector is at address 0x02100020 in on-chip IRAM.

Flash
: 16MB of external AMD flash at 0x00000000 attached via an 8-bit data bus.

SDRAM
: 32MB of external SDRAM at 0x04000000.

SRAM
: 2MB of external SRAM at 0x01000000.

IRAM
: 4K of on-chip RAM at 0x02100000.

system clock
: An Avalon timer labelled `sys_clk` used to implement the main eCos system clock. This is hardwired to run at 100Hz.

uart
: An Avalon uart connected to the external transceiver on the board. This is hardwired at 115200 baud, 8 bits, no parity, 1 stop bit, and no RTS/CTS support. This uart is used by eCos for the HAL diagnostics and debug channel.

lan91c111
: An interface to the external lan91c111 ethernet chip on the board. This provides network communications for RedBoot and for eCos applications using one of the available TCP/IP stacks.

GPIO
: GPIO units connected to the row of LEDs, the dual seven-segment display, and the buttons. The devkit platform HAL provides some utility functions for the first two of these.

sysid
: A system id register. This is used by RedBoot to check that the current hardware design matches the RedBoot build.

Other
: A number of other hardware units including a second Avalon timer, a jtag uart, a character LCD controller, and an EPCS serial flash controller. These are not currently used by eCos so can be accessed directly by application code.

Configuration Options

This hardware design HAL package will be loaded automatically when creating an eCos configuration for the `nios2_cyclone2_2c35_standard` target, together with the Nios II architectural HAL and the devkit platform HAL. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware. The package does not contain any user configuration options.

Memory Map

For typical eCos usage the memory map is as follows:

Address	Purpose
0x00000000	16MB of flash

Address	**Purpose**
0x00000000	reset vector
0x00000000	128K for RedBoot code
0x00c00000	current hardware design
0x00e00000	factory hardware design
0x00FF0000	RedBoot fis and fconfig data
0x01000000	2MB of SRAM
0x02100000	4K of on-chip IRAM
0x02100020	exception vector
0x02120000	peripherals
0x04000000	32MB of SDRAM
0x04000000	64K for RedBoot data
0x04010000	application code and data

Chapter 344. Nios II Cyclone II/2c35 TSEplus H/W Configuration HAL

Name

Overview — eCos Support for the TSEplus Hardware Design on a Cyclone II/2c35 Board

Description

This package provides the hardware design HAL for the eCosPro TSEplus hardware design running on a Cyclone II/2c35 board. The design is based on the TSE_SGDMA design provided with the Altera Nios II Embedded Design Suite, with a number of extensions. It includes the following functionality:

CPU
A Nios II/f processor running at 85MHz. This has 4K of instruction cache and 2K of data cache. It has level 2 jtag support with two hardware breakpoints. The reset vector is at address 0x00000000 in external flash and the exception vector is at address 0x04000020 in external SDRAM.

Flash
16MB of external AMD flash at 0x00000000 attached via an 8-bit data bus.

SDRAM
32MB of external SDRAM at 0x04000000.

SRAM
2MB of external SRAM at 0x01200000.

IRAM
8K of on-chip RAM at 0x01400000. Some of this will used by the triple speed ethernet device driver to store DMA descriptors.

system clock
An Avalon timer labelled `sys_clk` used to implement the main eCos system clock. This defaults to 100Hz but can be changed via the `CYGNUM_HAL_RTC_PERIOD` eCos configuration option.

uart
An Avalon uart connected to the external transceiver on the board. This is hardwired to 8 bits, no parity, and 1 stop bit. The default baud rate is 115200 but can be changed at run-time. The RTS and CTS signals are supported. This uart is used by eCos for the HAL diagnostics and debug channel.

TSE
A triple speed ethernet tse_mac and associated rx_sgdma and tx_sgdma scatter-gather DMA controllers. This provides network communications for RedBoot and for eCos applications using one of the available TCP/IP stacks. Note that this device requires an external phy board to be plugged into one of the main board's expansion connectors.

lan91c111
An interface to the external lan91c111 ethernet chip on the board. This can be used for network communication instead of the triple speed ethernet device, useful if the external phy board is not available.

watchdog
An Avalon timer set up to act as a watchdog device with a 10-second timeout.

profiling
An additional Avalon timer used for gprof-based profiling.

GPIO
GPIO units connected to the row of LEDs, the dual seven-segment display, and the buttons. The devkit platform HAL provides some utility functions for the first two of these.

sysid
A system id register. This is used by RedBoot to check that the current hardware design matches the RedBoot build.

Other
A number of other hardware units including a second Avalon timer, a jtag uart, a character LCD controller, and an EPCS serial flash controller. These are not currently used by eCos so can be accessed directly by application code.

Configuration Options

The TSEplus hardware design HAL package is used with two different eCos targets. The `nios2_cyclone2_2c35_tseplus` target includes the TSE ethernet driver but not the lan91c111 ethernet driver, and can be used when the external phy board is present.

The `nios2_cyclone2_2c35_lan91c111` target includes the lan91c111 ethernet driver but no the TSE ethernet driver and can be used in the absence of the external phy board. When using a RAM startup configuration the same eCos target should be used for both RedBoot and the application.

The hardware design HAL package will be loaded automatically when creating an eCos configuration for either target, together with the Nios II architectural HAL and the devkit platform HAL. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware. The package does not contain any user configuration options.

The package contains a single configuration option, `CYGNUM_HAL_RTC_PERIOD`. This determines the period of the system clock. By default this operates at 100Hz but the value can be changed if a faster or slower clock is desired. The value is used to program the *`PERIODH`* and *`PERIODL`* registers of the `sys_clk` Avalon timer.

Memory Map

For typical eCos usage the memory map is as follows:

Address	Purpose
0x00000000	16MB of flash
0x00000000	reset vector
0x00000000	128K for RedBoot code
0x00800000	current hardware design
0x00c00000	factory hardware design
0x00FF0000	RedBoot fis and fconfig data
0x01200000	2MB of SRAM
0x01400000	8K of on-chip IRAM
0x01401E00	TSE DMA descriptors
0x01403000	peripherals
0x04000000	32MB of SDRAM
0x04000020	exception vector
0x04000100	~64K for RedBoot data
0x04010000	application code and data

Chapter 345. Nios II Stratix II/2s60_RoHS Standard H/W Design HAL

Name

`CYGPKG_HAL_NIOS2_STRATIX2_2S60_ROHS_STANDARD` — eCos Support for the Standard Hardware Design on a Stratix II/2s60-RoHS Board

Description

This package provides the hardware design HAL for the standard hardware design running on a Stratix II/2s60-RoHS board. This design is provided with the Altera Nios II Embedded Design Suite in the directory `nios2eds/examples/vhdl/niosII_stratixII_2s60_RoHS/standard/`. It includes the following functionality:

CPU
: A Nios II/s processor running at 100MHz. This has 4K of instruction cache and no data cache. It has level 1 jtag support only with no hardware breakpoints. The reset vector is at address 0x00000000 in external flash and the exception vector is at address 0x02100020 in on-chip IRAM.

Flash
: 16MB of external AMD flash at 0x00000000 attached via an 8-bit data bus.

SDRAM
: 32MB of external SDRAM at 0x04000000.

SRAM
: 2MB of external SRAM at 0x01000000.

IRAM
: 4K of on-chip RAM at 0x02100000.

system clock
: An Avalon timer labelled `sys_clk` used to implement the main eCos system clock. This is hardwired to run at 100Hz.

uart
: An Avalon uart connected to the external transceiver on the board. This is hardwired at 115200 baud, 8 bits, no parity, 1 stop bit, and no RTS/CTS support. This uart is used by eCos for the HAL diagnostics and debug channel.

lan91c111
: An interface to the external lan91c111 ethernet chip on the board. This provides network communications for RedBoot and for eCos applications using one of the available TCP/IP stacks.

GPIO
: GPIO units connected to the row of LEDs, the dual seven-segment display, and the buttons. The devkit platform HAL provides some utility functions for the first two of these.

sysid
: A system id register. This is used by RedBoot to check that the current hardware design matches the RedBoot build.

Other
: A number of other hardware units including a second Avalon timer, a jtag uart, a character LCD controller, and an EPCS serial flash controller. These are not currently used by eCos so can be accessed directly by application code.

Configuration Options

This hardware design HAL package will be loaded automatically when creating an eCos configuration for the `nios2_stratix2_2s60_rohs_standard` target, together with the Nios II architectural HAL and the devkit platform HAL. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware. The package does not contain any user configuration options.

Memory Map

For typical eCos usage the memory map is as follows:

Address	**Purpose**
0x00000000	16MB of flash

Address	**Purpose**
0x00000000	reset vector
0x00000000	128K for RedBoot code
0x00800000	current hardware design
0x00c00000	factory hardware design
0x00FF0000	RedBoot fis and fconfig data
0x01000000	2MB of SRAM
0x02100000	4K of on-chip IRAM
0x02100020	exception vector
0x02120000	peripherals
0x04000000	32MB of SDRAM
0x04000000	64K for RedBoot data
0x04010000	application code and data

Chapter 346. Nios II Stratix II/2s60_RoHS TSEplus H/W Design HAL

Name

— eCos Support for the TSEplus Hardware Design on a Stratix II/2s60-RoHS Board

Description

This package provides the hardware design HAL for the eCosPro TSEplus hardware design running on a Stratix II/2s60-RoHS board. The design is based on the TSE_SGDMA design provided with the Altera Nios II Embedded Design Suite, with a number of extensions. It includes the following functionality:

CPU
A Nios II/f processor running at 85MHz. This has 4K of instruction cache and 2K of data cache. It has level 2 jtag support with two hardware breakpoints. The reset vector is at address 0x00000000 in external flash and the exception vector is at address 0x04000020 in external SDRAM.

Flash
16MB of external AMD flash at 0x00000000 attached via an 8-bit data bus.

SDRAM
32MB of external SDRAM at 0x04000000.

SRAM
2MB of external SRAM at 0x01200000.

IRAM
8K of on-chip RAM at 0x01400000. Some of this will used by the triple speed ethernet device driver to store DMA descriptors.

system clock
An Avalon timer labelled `sys_clk` used to implement the main eCos system clock. This defaults to 100Hz but can be changed via the `CYGNUM_HAL_RTC_PERIOD` eCos configuration option.

uart
An Avalon uart connected to the external transceiver on the board. This is hardwired to 8 bits, no parity, and 1 stop bit. The default baud rate is 115200 but can be changed at run-time. The RTS and CTS signals are supported. This uart is used by eCos for the HAL diagnostics and debug channel.

TSE
A triple speed ethernet tse_mac and associated rx_sgdma and tx_sgdma scatter-gather DMA controllers. This provides network communications for RedBoot and for eCos applications using one of the available TCP/IP stacks. Note that this device requires an external phy board to be plugged into one of the main board's expansion connectors.

lan91c111
An interface to the external lan91c111 ethernet chip on the board. This can be used for network communication instead of the triple speed ethernet device, useful if the external phy board is not available.

watchdog
An Avalon timer set up to act as a watchdog device with a 10-second timeout.

profiling
An additional Avalon timer used for gprof-based profiling.

GPIO
GPIO units connected to the row of LEDs, the dual seven-segment display, and the buttons. The devkit platform HAL provides some utility functions for the first two of these.

sysid
A system id register. This is used by RedBoot to check that the current hardware design matches the RedBoot build.

Other
A number of other hardware units including a second Avalon timer, a jtag uart, a character LCD controller, and an EPCS serial flash controller. These are not currently used by eCos so can be accessed directly by application code.

Configuration Options

The TSEplus hardware design HAL package is used with two different eCos targets. The `nios2_stratix2_2s60_rohs_tseplus` target includes the TSE ethernet driver but not the lan91c111 ethernet driver, and can be used when the external phy board is present. The `nios2_stratix2_2s60_rohs_lan91c111` target includes the lan91c111 ethernet driver but no the TSE ethernet driver and can be used in the absence of the external phy board. When using a RAM startup configuration the same eCos target should be used for both RedBoot and the application.

The hardware design HAL package will be loaded automatically when creating an eCos configuration for either target, together with the Nios II architectural HAL and the devkit platform HAL. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware. The package does not contain any user configuration options.

The package contains a single configuration option, `CYGNUM_HAL_RTC_PERIOD`. This determines the period of the system clock. By default this operates at 100Hz but the value can be changed if a faster or slower clock is desired. The value is used to program the *`PERIODH`* and *`PERIODL`* registers of the `sys_clk` Avalon timer.

Memory Map

For typical eCos usage the memory map is as follows:

Address	**Purpose**
0x00000000	16MB of flash
0x00000000	reset vector
0x00000000	128K for RedBoot code
0x00800000	current hardware design
0x00c00000	factory hardware design
0x00FF0000	RedBoot fis and fconfig data
0x01200000	2MB of SRAM
0x01400000	8K of on-chip IRAM
0x01401E00	TSE DMA descriptors
0x01403000	peripherals
0x04000000	32MB of SDRAM
0x04000020	exception vector
0x04000100	~64K for RedBoot data
0x04010000	application code and data

Chapter 347. Board-level Support for the Nios II Embedded Evaluation Kit, Cyclone III edition

Name

CYGPKG_HAL_NIOS2_NEEK_CYCLONE3_BOARD — eCos Platform HAL Support for the Nios II Embedded Evaluation Kit, Cyclone III Edition

Description

This package provides platform HAL support for the Altera Nios II Embedded Evaluation Kit, Cyclone III Edition, also known as the NEEK board. Since this platform is based around an FPGA it can run a variety of hardware designs, and each design needs its own h/w design HAL to define the details of the hardware configured into the FPGA. The platform HAL contains support for some of the off-chip peripherals on the board, for example the external flash memory and the I²C device used to hold the ethernet MAC address. It also contains some code and configuration options which are likely to be reusable across many h/w designs, to avoid duplicating code unnecessarily.

The h/w design determines which peripherals are available, and that in turn affects how eCos can be used. Hence details of setting up a NEEK board for eCos development can be found in the h/w design HAL documentation. In addition the Nios II architectural HAL documentation contains some generic setup instructions which will be applicable to most h/w designs.

Configuration Options

This platform HAL package should be loaded automatically when eCos is configured for appropriate target hardware. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

The main configuration option provided by this package is `CYG_HAL_STARTUP`. This can take either two or three values, depending on the h/w design:

`ROM`
: This startup type should be used for code that should execute from the external flash. The executable will start running from the Nios II reset vector defined in the h/w design and will contain all the low-level initialization code needed to bring up the system. ROM startup can be used for production systems. It can also be used for building the RedBoot ROM monitor on h/w designs that support RedBoot, and the application can then be loaded into RAM and run from there.

`RAMJTAG`
: This startup type will be used mainly for debugging via JTAG. The nios2-elf-gdb debugger will be used, directly or indirectly, to load the application into RAM and start it running.

`RAM`
: This startup type is only available if the h/w design incorporates a uart. It assumes that a ROM-startup RedBoot has been programmed into flash. JTAG is not used for debugging, only for the initial installation. For debugging, nios2-elf-gdb connects to the RedBoot ROM monitor over serial or ethernet and can then be used to load the application into RAM and run it. For production systems a RedBoot boot script can be used to load the application from flash to RAM and start it running.

For ROM and RAMJTAG startup, if the h/w design incorporates a uart then by default that will be used for the diagnostics channel. This can be changed via the configuration option `CYGIMP_HAL_NIOS2_DIAGNOSTICS_PORT`. The alternative destination is `discard` which just discards all diagnostics output. For RAM startup the application will inherit its diagnostics channel from the RedBoot ROM monitor.

The remaining configuration options in this package are primarily for internal use within the configuration system.

External Flash

The NEEK board comes with a single Strata-compatible external flash chip. Assuming the h/w design incorporates support for this, the Strata V2 flash driver will automatically be loaded when creating a new eCos configuration. However the driver will be inactive by default. To activate the driver the generic flash support `CYGPKG_IO_FLASH` must be added to the configuration. The platform HAL will then instantiate a flash device object.

I2C Buses and Devices

The NEEK board comes with a number of I^{2}C devices on two sets of GPIO lines. Given the right support in the h/w design the platform HAL will instantiate bit-banged I^{2}C bus objects and define I^{2}C device objects. The latter can then be manipulated using the generic eCos I^{2}C API, for example `cyg_i2c_tx`, as defined by the package `CYGPKG_IO_I2C`.

If the h/w design HAL defines symbols `HAL_NIOS2_AVALON_PIO_id_eeprom_dat_BASE` and `HAL_NIOS2_AVALON_PIO_id_eeprom_scl_BASE` then the platform HAL will instantiate an I^{2}C bus `hal_neek_cyclone3_id_eeprom_bus` and an I^{2}C device `hal_neek_cyclone3_24lc02b`, corresponding to an EEPROM chip. This chip is factory-programmed with an ethernet MAC address in bytes 2-7. If the h/w design and the eCos configuration involve an ethernet device then the platform HAL will provide this MAC address to the ethernet driver. The other bytes in the EEPROM are not used by eCos so are available to application code.

If the h/w design HAL defines symbols `HAL_NIOS2_AVALON_PIO_lcd_i2c_scl_BASE` and `HAL_NIOS2_AVALON_PIO_lcd_i2c_sdat_BASE` then the platform HAL will instantiate an I^{2}C bus `hal_neek_cyclone3_lcd_i2c_bus` and two I^{2}C device objects `hal_neek_cyclone3_wm8731` and `hal_neek_cyclone3_adv7180`.

LEDs

The NEEK board comes with four LEDs, although not all h/w designs will allow access to all four. If the h/w design incorporates a suitable GPIO port and the h/w design HAL defines the symbol `HAL_NIOS2_AVALON_PIO_led_BASE` then the platform HAL will provide two additional functions for manipulating the LEDs:

```
#include <cyg/hal/hal_io.h>
extern void hal_nios2_led_set(int which, int on);
extern void hal_nios2_led_set_all(int setting);
```

The first function can be used to switch a single LED on or off. The second can be used to change the setting of all four LEDs, using the bottom four bits of the *`setting`* argument.

Chapter 348. Nios II Embedded Evaluation Kit, Cyclone III Edition, appselector H/W Design HAL

Name

CYGPKG_HAL_NIOS2_NEEK_CYCLONE3_APPSELECTOR — eCos Support for the Appselector Hardware Design on a Nios II Embedded Evaluation Kit, Cyclone III Edition

Description

This package provides the hardware design HAL for the version 8.0 appselector hardware design running on a Nios II Embedded Evaluation Kit, Cyclone III edition, also known as the NEEK board. This is the hardware design programmed into boards as shipped from the factory, and in the examples/application_selector and factory_recovery directories of the software supplied with the board.

Note

There are a number of different and incompatible versions of the appselector hardware design. This package supports only the 8.0 version, a copy of which is supplied with eCosPro releases. If your NEEK board runs a different version of the hardware design then it will be necessary to update the board before eCos can be used. Instructions on how to do this are given below.

The hardware design includes the following functionality:

CPU

A Nios II/s processor running at 60MHz. This has 4K of instruction cache and 2K of data cache. It has level 1 jtag support only with no hardware breakpoints. The reset vector is at address 0x04100000 in external flash and the exception vector is at address 0x05000020 in external SRAM.

SDRAM

32MB of external SDRAM at 0x00000000. If RedBoot is used then the bottom 64K will be reserved for RedBoot. The remainder is used as the default location for application data, and for application code for RAM and RAMJTAG startups.

SRAM

1MB of external SRAM at 0x05000000. Some of this will be used for exception and interrupt handling. The remainder is available for use by the application. Code and data can be placed here by putting it into .sram_text, .sram_data and .sram_bss sections. The platform HAL package CYGPKG_HAL_NIOS2_NEEK_CYCLONE3_BOARD contains a memories.c testcase which can be used as an example.

IRAM

4K of on-chip RAM at 0x08000000. This can be used only for holding data, not code. Some of this is used for holding DMA descriptors for the triple-speed ethernet device. The remainder is available for use by application data, by putting it into .iram_data and .iram_bss sections. The platform HAL package CYGPKG_HAL_NIOS2_NEEK_CYCLONE3_BOARD contains a memories.c testcase which can be used as an example.

Flash

16MB of external Strata flash at 0x04000000 attached via an 16-bit data bus. This is supported via the V2 Strata flash driver CYGPKG_DEVS_FLASH_STRATA_V2. The driver will be inactive unless the generic flash support package CYGPKG_IO_FLASH has been added to the configuration. The first megabyte of the flash is reserved for holding the hardware design. Locations 0x04100000 onwards are used to hold the code for ROM startup applications, or to hold RedBoot if that is used. RedBoot's FIS and fconfig data are held at the end of the flash. The remainder is available for use by the application.

system clock

An Avalon timer labelled sys_clk used to implement the main eCos system clock. By default the system clock will operate at 100Hz, but this can be changed by editing the configuration option CYGNUM_HAL_RTC_PERIOD.

uart

An Avalon uart connected to the external transceiver on the board. This is hardwired at 115200 baud, 8 bits, no parity, 1 stop bit, and no RTS/CTS support. Usually this uart will be used by eCos and/or RedBoot for the HAL diagnostics and debug channel, so it will not be available to the application. If the uart is not used in this way then it can be accessed by the application code via the serial device driver `CYGPKG_DEVS_SERIAL_NIOS2_AVALON_UART`. The configuration will also need to include the generic serial support package `CYGPKG_IO_SERIAL`, and the option `CYGPKG_IO_SERIAL_DEVICES` will need to be enabled.

tse

A triple-speed ethernet device, and associated DMA engines. In eCos configurations which involve networking this device will be supported via the ethernet driver package `CYGPKG_DEVS_ETH_NIOS2_TSE`. The ethernet's MAC address is held in an external EEPROM attached to a bit-banged I^{2}C bus.

I2C

The NEEK board has two I^{2}C buses, implemented by bit-banging GPIO ports: `hal_neek_cyclone3_id_eeprom_bus` and `hal_neek_cyclone3_lcd_i2c_bus`. There are I^{2}C device instances for the devices attached to these buses: `hal_neek_cyclone3_24lc02b` for the EEPROM, `hal_neek_cyclone3_adv7180` for the video decoder and `hal_neek_cyclone3_wm8731` for the sound chip. The generic I^{2}C package `CYGPKG_IO_I2C` provides an API for manipulating such devices. Bytes 2 to 7 of the EEPROM are used to hold the ethernet MAC address so should not be changed by application code. If any of the bus or device instances are not used directly or indirectly then they will be removed by link-time garbage collection.

sysid

A system id register. This is used by RedBoot to check that the current hardware design matches the RedBoot build.

Other

A number of other hardware units including a jtag uart, an SPI bus, and a framebuffer device driving an LCD panel. These are not currently used by eCos so can be accessed directly by application code.

Setting up a Board

The eCos port targets a VHDL hardware design named appselector. This hardware design is provided by Altera, and new NEEK boards usually come with this h/w design preprogrammed into the external flash. However, there are several different and incompatible versions of this design. The eCos port specifically targets the 8.0 version. If your board comes with a different version of the h/w design or if a different h/w design has been programmed into flash then it will first be necessary to program the right design into flash. The required image file restore_cycloneIII_3c25.flash can be found in the nios2/ subdirectory of the installation. The sources for this h/w design are also included with the release.

The most convenient way to install a h/w design is to use Altera's flash programming tools, provided with Quartus and the Nios II Embedded Development Suite (nios2eds). The NEEK board has a built-in jtag interface accessible via a USB port, which can be used for this purpose.

The Quartus tools can be downloaded from Altera. Version 13.1 is the last version released that supports the Cyclone III FPGA that the Neek board is based upon. The free “web” version can be downloaded from http://dl.altera.com/13.1/?edition=web and the subscription version from http://dl.altera.com/13.1/?edition=subscription. Version 13.1, unlike some earlier versions, incorporates the EDS tools within the Quartus download.

Assuming that the Altera software as been correctly installed and that a USB cable has been connected between the host PC and the NEEK board, the following commands will install Altera's appselector application and support data, as well as the hardware design. On a Windows PC these commands need to be issued within the Altera EDS Nios command shell, not the Windows or eCos command shells.

```
$ cd <ecosproinstalldir>/ecos-<version>/nios2
$ nios2-flash-programmer --base=0x04000000 --go restore_cycloneIII_3c25.flash
```

If application development will use just the jtag interface then the board can now be used to run eCos applications configured for RAMJTAG startup. This involves running the Nios II gdb server supplied by Altera:

```
$ nios2-gdb-server --tcpport 9000 --tcppersist
```

eCos applications and tests configured for RAMJTAG startup can now be downloaded and executed via nios2-elf-gdb at another command line prompt. For example:

```
nios2-elf-gdb <executable-file>
(gdb) target remote localhost:9000
(gdb) set $ienable=0
(gdb) load
(gdb) continue
```

When using jtag, by default diagnostic output will be directed to the serial port and may be viewed using a terminal emulator configured for 115200/8N1 with no handshaking.

If instead application development will happen via RedBoot then the next step is to install a RedBoot image, replacing the appselector application. Note you must power cycle the board between the above step and this one. Start by connecting a serial cable between the host PC and the NEEK board, and start up a terminal emulator running at 115200/8N1 with no handshaking. Next use Altera's EDS command shell to issue the following commands to program the RedBoot image into flash via jtag:

```
$ cd <ecosproinstalldir>/ecos-<version>/loaders/nios2_neek_cyclone3_appselector
$ bin2flash --location=0x00100000 \
          --input=redboot_ROM.bin \
          --output=redboot.flash
$ nios2-flash-programmer --base=0x04000000 \
                       --sidp=0x08002f40 \
                       --id=1727563914 \
                       --no-keep-nearby \
                       --go \
                       redboot.flash
```

The --sidp and --id arguments are used to check that the board is running the correct h/w design. The --go argument causes the board to restart, so RedBoot should now be running and should have output its banner and a prompt. The terminal emulator can now be used to execute the following commands at the RedBoot prompt to initialise the Flash contents:

```
RedBoot> fis init
RedBoot> fconfig -i
```

eCos applications and tests configured for RAM startup can now be downloaded and executed via nios2-elf-gdb. For example, to run the prebuilt eCos test over serial, first exit the terminal emulator so that nios2-elf-gdb can access the serial port, and then use the following commands:

```
nios2-elf-gdb <path>/thread_gdb
(gdb) set remotebaud 115200
(gdb) target remote /dev/ttyS0
(gdb) load
(gdb) continue
```

RedBoot binaries are provided under the 'loaders' directory in the release. They may be rebuilt by making RAM and ROM startup RedBoot images in the documented manner, each in a separate working directory. For RAM startup:

```
$ ecosconfig new nios2_neek_cyclone3_appselector redboot
$ ecosconfig import <path>/redboot_RAM.ecm
$ ecosconfig check
$ ecosconfig tree
$ make
```

The `.ecm` import file can be found in the `misc` subdirectory of this package. Similarly for ROM startup:

```
$ ecosconfig new nios2_neek_cyclone3_appselector redboot
$ ecosconfig import <path>/redboot_ROM.ecm
$ ecosconfig check
$ ecosconfig tree
$ make
```

Configuration Options

This hardware design HAL package will be loaded automatically when creating an eCos configuration for the `nios2_neek_cyclone3_appselector` target, together with the Nios II architectural HAL `CYGPKG_HAL_NIOS2` and the platform HAL

CYGPKG_HAL_NIOS2_NEEK_CYCLONE3_BOARD. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Selecting the nios2_neek_cyclone3_appselector target will also load a flash driver, a serial driver, an ethernet driver, and I²C support. The flash driver will be inactive unless the configuration also includes the generic flash support package CYGPKG_IO_FLASH. The serial driver will be inactive unless the configuration includes the generic serial support package CYGPKG_IO_SERIAL and the option CYGPKG_IO_SERIAL_DEVICES is enabled. The ethernet driver will be active only in eCos configurations which involve networking. The I²C package is always active, but any functionality that is not used directly or indirectly will be removed via link-time garbage collection.

The port to the appselector h/w design supports three startup types:

ROM
: This startup type should be used for code that should execute from the external flash. The executable will start running from the Nios II reset vector and will contain all the low-level initialization code needed to bring up the system. ROM startup can be used for production systems. It can also be used for building the RedBoot ROM monitor on h/w designs that support RedBoot, and the application can then be loaded into RAM and run from there.

RAMJTAG
: This startup type will be used mainly for debugging via JTAG. The nios2-elf-gdb debugger will be used, directly or indirectly, to load the application into RAM and start it running.

RAM
: This startup type assumes that a ROM-startup RedBoot has been programmed into flash. JTAG is not used for debugging, only for the initial installation. For debugging, nios2-elf-gdb connects to the RedBoot ROM monitor over serial or ethernet and can then be used to load the application into RAM and run it. For production systems a RedBoot boot script can be used to load the application from flash to RAM and start it running.

For ROM and RAMJTAG startup, by default the uart will be used for the diagnostics channel. This can be changed via the configuration option CYGIMP_HAL_NIOS2_DIAGNOSTICS_PORT. The alternative destination is discard which just discards all diagnostics output. For RAM startup the application will inherit its diagnostics channel from the RedBoot ROM monitor.

The configuration option CYGNUM_HAL_RTC_PERIOD can be used to change the system clock frequency. The default value gives a 100Hz clock.

Part LXXXIV. PowerPC Architecture

Table of Contents

Chapter 349. A&M Adder Board Support

Name

eCos Support for the Adder Board — Overview

Description

This document covers two Analogue & Micro boards, the Adder I and the Adder II. The Adder I board contains an MPC850 processor, 8Mb of RAM, 4MB of flash memory, and external connections for two serial channels and ethernet. The Adder II is identical except that it is built around an MPC852T processor, which is largely compatible with the MPC850. Everything in this document applies to both boards unless otherwise stated.

For typical eCos development, a RedBoot image is programmed into the flash memory, and the board will boot this image from reset. RedBoot provides gdb stub functionality so it is then possible to download and debug stand-alone and eCos applications via the gdb debugger. This can happen over either a serial line or over ethernet.

Supported Hardware

The flash memory consists of 64 blocks of 64k bytes each. In a typical setup, the first three flash blocks are used for the ROMRAM RedBoot image The topmost block is used to manage the flash and hold RedBoot fconfig values. The remaining 60 blocks between 0xFE030000 and 0xFE3EFFFF can be used by application code.

There is a serial driver `CYGPKG_IO_SERIAL_POWERPC_QUICC_SMC` which supports both the SMC2 (AdderII: SMC1) and SCC3 based on-chip serial devices. These devices can be used by RedBoot for communication with the host. If either of these devices is needed by the application, either directly or via the serial driver, then it cannot also be used for RedBoot communication. Another communication channel such as ethernet should be used instead. The serial driver package is loaded automatically when configuring for the Adder target.

There is an ethernet driver `CYGPKG_DEVS_ETH_POWERPC_ADDER` for the on-chip ethernet device (`CYGPKG_DEVS_ETH_POWERPC_ADDERII` for the AdderII). This driver is also loaded automatically when configuring for the Adder target.

eCos manages the on-chip interrupt controller. The architecture-defined decrementer is used to implement the eCos system clock and the microsecond delay function. Other on-chip devices (Caches, PIO, UARTs, FEC) are initialized only as far as is necessary for eCos to run. Other devices (SPI,I2C, PCMCIA) are not touched.

Tools

The Adder port is intended to work with GNU tools configured for a powerpc-eabi target. The original port was undertaken using powerpc-eabi-gcc version 3.2.1, powerpc-eabi-gdb version 5.3, and binutils version 2.13.1.

Name

Setup — Preparing the Adder board for eCos Development

Overview

In a typical development environment, the Adder board boots from flash into the RedBoot ROM monitor. eCos applications are configured for RAM startup and then downloaded and run on the board via the debugger **powerpc-eabi-gdb**. Preparing the board therefore usually involves programming a suitable RedBoot image into flash memory.

The following RedBoot configurations are supported:

Configuration	Description	Use	File
ROMRAM	RedBoot running from RAM, but contained in the board's flash boot sector.	adder[II]_redboot_ROMRAM.ecm	adder[II]_redboot_ROMRAM.bin

For serial communications, all versions run with 8 bits, no parity, and 1 stop bit at 38400 baud. The baud rate can be changed using the flash configuration console baud rate option. RedBoot also supports ethernet communication and flash management.

Initial Installation

Flash Installation

The Adder boards are shipped from A&M with a version of RedBoot already installed. If the software distribution you are using provides a more recent certified version of RedBoot then you should install that in place of the existing version.

Updating RedBoot is a simple matter of downloading a new binary image and overwriting the existing ROM image. Connect a serial cable between the Adder board and a host computer and start a terminal emulator such as HyperTerminal or minicom. When RedBoot starts up you will see something similar to this:

```
+... waiting for BOOTP information
Ethernet eth0: MAC address 00:02:b3:46:01:00
IP: 10.0.0.202/255.255.255.0, Gateway: 10.0.0.1
Default server: 10.0.0.102, DNS server IP: 10.0.0.1

RedBoot(tm) bootstrap and debug environment [ROMRAM]
Non-certified release, version UNKNOWN - built 17:00:15, Jan  5 2004

Platform: A&M Adder II (PowerPC 852T)
Copyright (C) 2000, 2001, 2002, Free Software Foundation, Inc.

RAM: 0x00000000-0x00800000, [0x0003afc0-0x007ed000] available
FLASH: 0xfe000000 - 0xfe400000, 64 blocks of 0x00010000 bytes each.
RedBoot>
```

To download the new RedBoot image to the board via ethernet, ensure that the ethernet cable is connected (if not, connect it and reset RedBoot). Then copy the new RedBoot image (`adderII_redboot_ROMRAM.bin`) to the TFTP server directory (`/tftpboot` on Linux). You can now download the new RedBoot image with the following command:

```
RedBoot> load -r -b %{FREEMEMLO} -h 10.0.0.100 adderII_redboot_ROMRAM.bin
```

If the TFTP server is running on the machine with the IP address shown against "Default server" at startup, then the `-h <host>` option may be omitted.

Alternatively, if no TFTP server is available, the file may be downloaded more slowly over the serial line with the following command:

```
RedBoot> load -r -b %{FREEMEMLO} -m ymodem
```

Use the terminal emulator's Y-Modem file transfer option to send the file `adderII_redboot_ROMRAM.bin`. Once the file has been uploaded, you can check that it has been transferred correctly using the **cksum** command. On the host (Linux or Cygwin) run the **cksum** program on the binary file:

```
$ cksum adderII_redboot_ROMRAM.bin
1574308703 150312 adderII_redboot_ROMRAM.bin
```

In RedBoot, run the **cksum** command on the data that has just been loaded:

```
RedBoot> cksum -b %{FREEMEMLO} -l 150312
POSIX cksum = 1574308703 150312 (0x5dd60b5f 0x00024b28)
```

The second number in the output of the host **cksum** program is the file size, which should be used as the argument to the `-l` option in the RedBoot **cksum** command. The first numbers in each instance are the checksums, which should be equal.

If the program has downloaded successfully, then it can be programmed into the flash using the following command:

```
RedBoot> fis create -b %{FREEMEMLO} RedBoot
An image named 'RedBoot' exists - continue (y/n)? y
... Erase from 0xfe000000-0xfe030000: ...
... Program from 0x0003b000-0x0006b000 at 0xfe000000: ...
... Erase from 0xfe3f0000-0xfe400000: .
... Program from 0x007f0000-0x00800000 at 0xfe3f0000: .
RedBoot>
```

The Adder board may now be reset either by cycling the power, or with the **reset** command. It should then display the startup screen for the new version of RedBoot.

This description has used the Adder II as an example. The process for the Adder I is identical except that the new image file is called `adder_redboot_ROMRAM.bin`. Make sure you load the correct file if both are present.

Rebuilding RedBoot

Should it prove necessary to rebuild a RedBoot binary, this is done most conveniently at the command line. The steps needed to rebuild the the ROMRAM version of RedBoot for the Adder I are:

```
$ mkdir redboot_adder_romram
$ cd redboot_adder_romram
$ ecosconfig new adder redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/powerpc/adder/v2_0_9/misc/adder_redboot_ROMRAM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

To rebuild the ROMRAM version of RedBoot for the Adder II:

```
$ mkdir redboot_adderII_romram
$ cd redboot_adderII_romram
$ ecosconfig new adderII redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/powerpc/adder/v2_0_9/misc/adderII_redboot_ROMRAM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

At the end of the build the `install/bin` subdirectory should contain the file `redboot.bin`.

Name

Configuration — Platform-specific Configuration Options

Overview

The Adder platform HAL package is loaded automatically when eCos is configured for an `adder` or `adderII` target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The Adder platform HAL package supports three separate startup types:

RAM
: This is the startup type which is normally used during application development. The board has RedBoot programmed into flash at location 0xFE000000 and boots from that location. powerpc-eabi-gdb is then used to load a RAM startup application into memory and debug it. It is assumed that the hardware has already been initialized by RedBoot. By default the application will use the eCos virtual vectors mechanism to obtain certain services from RedBoot, including diagnostic output.

ROM
: This startup type can be used for finished applications which will be programmed into flash at location 0xFE000000. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization.

ROM-RAM
: This startup type can be used for finished applications which will be programmed into flash at location 0xFE000000. However, when it starts up the application will first copy itself to RAM at 0x00000000 and then run from there. RAM is generally faster than flash memory, so the program will run more quickly than a ROM-startup application. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization.

RedBoot and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building RedBoot.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then the serial port will be claimed for HAL diagnostics.

Flash Driver

The Adder board contains a 4Mb AMD AM29XXXXX series flash device The `CYGPKG_DEVS_FLASH_AMD_AM29XXXXX` package contains all the code necessary to support these parts and the `CYGPKG_DEVS_FLASH_POWERPC_ADDER` package contains definitions that customize the driver to the Adder board.

Ethernet Driver

The Adder I board uses SCC2 configured to be a 10Mb/s ethernet interface. The `CYGPKG_DEVS_ETH_POWERPC_QUICC` package contains all the code necessary to support this device and the `CYGPKG_DEVS_ETH_POWERPC_ADDER` package contains definitions that customize the driver to the Adder board.

The Adder II board uses the separate FEC (Fast Ethernet Controller) device. The `CYGPKG_DEVS_ETH_POWERPC_FEC` package contains all the code necessary to support this device and the `CYGPKG_DEVS_ETH_POWERPC_ADDERII` package contains definitions that customize the driver to the Adder II board.

System Clock

By default, the system clock interrupts once every 10ms, corresponding to a 100Hz clock. This can be changed by the configuration option `CYGNUM_HAL_RTC_DENOMINATOR` which corresponds to the clock frequency. Other clock-related settings are recalculated automatically if the denominator is changed.

Compiler Flags

The platform HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos. There are two flags specific to this port:

`-mcpu=860`

The powerpc-eabi-gcc compiler supports many variants of the PowerPC architecture. A `-m` option should be used to select the specific variant in use, and with current tools `-mcpu-860` is the correct option for both Adder boards.

`-msoft-float`

The PowerPC processor used in the Adder boards does not have a floating point unit. Therefore it is necessary to translate any floating point operations into software emulation. This option tells the compiler to do that.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the Adder hardware, and should be read in conjunction with that specification. The Adder platform HAL package complements the PowerPC architectural HAL and the MPC8XX variant HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset the HAL will initialize or reinitialize most of the on-chip peripherals. There is an exception for RAM startup applications which depend on a ROM monitor for certain services.

For ROM or ROMRAM startup, the HAL will perform additional initialization, setting up the external RAM and programming the various internal registers. This is all done in the `hal_hardware_init` function in the assembler source file `adder.S`.

Linker Scripts and Memory Maps

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

Flash
: This is located at address 0xFE000000 of the physical memory space.

SDRAM
: This is located at address 0x00000000 of the physical memory space. The first 12k bytes are used for hardware exception vectors. The next 512 bytes are used for the VSR table and the next 256 bytes are normally used for the eCos virtual vectors, allowing RAM-based applications to use services provided by the ROM monitor. For ROM/ROMRAM startup, all remaining SDRAM is available. For RAM startup, available RAM starts at location 0x00060000, with the bottom 384kB reserved for use by RedBoot.

on-chip peripherals
: These are accessible via the CPU IMMR register, which is normally set to 0xFA200000. However, applications should not rely on this. See the documentation for the MPC850 or MPC852T for information on the on-chip peripherals.

off-chip peripherals
: The Adder II has an MPC180 Encryption Processor on-board. However, this is not used by eCos.

Other Issues

The Adder platform HAL does not affect the implementation of other parts of the eCos HAL specification. The MPC8XX variant HAL, and the PowerPC architectural HAL documentation should be consulted for further details.

Chapter 350. ADS512101 Board Support

Name

eCos Support for the ADS512101 Board — Overview

Description

This document covers the ADS512101 board. The board contains an MPC5121e microprocessor, 128MiB of RAM, 64MiB of Flash and 128KiB of internal SRAM. There are external connections for a single UART and the Fast Ethernet Controller.

For typical eCos development, a RedBoot image is programmed into the on-chip flash memory, and the board will boot this image from reset. RedBoot provides gdb stub functionality so it is then possible to download and debug stand-alone and eCos applications via the gdb debugger using the serial line.

Supported Hardware

The Flash memory consists of 256 blocks each of 256KiB, occupying 64MiB. The Flash is actually composed of two 32Mib devices operating in parallel. In a typical setup, RedBoot is programmed into flash at 0xFFF00000 and occupies the next 768KiB. The topmost block is used to manage the flash and holds RedBoot fconfig values. The first 255MiB may be used by application code.

There is a serial driver `CYGPKG_IO_SERIAL_POWERPC_PSC` which supports both the PSC based on-chip serial devices. Only PSC3 is actually brought out to a usable external connector. This device can be used by RedBoot for communication with the host. The serial driver package is loaded automatically when configuring for the ADS512101 target.

The FEC driver, `CYGPKG_DEVS_ETH_POWERPC_FEC` is used to control the FEC. The package `CYGPKG_DEVS_ETH_POWERPC_ADS512101` is used to configure the generic driver for the MPC5121e and this board.

eCos manages the on-chip interrupt controller. The architecture-defined decrementer is used to implement the eCos system clock and the microsecond delay function. A GPT is used to implement a profiling timer. Other on-chip devices (Caches, GPIO, UARTs) are initialized only as far as is necessary for eCos to run. The remaining devices (PCI, PATA, SATA etc.) are not touched.

Tools

The ADS512101 port is intended to work with GNU tools configured for a powerpc-eabi target. The original port was undertaken using powerpc-eabi-gcc version 4.4.5, powerpc-eabi-gdb version 7.2, and binutils version 2.20.1.

Name

Setup — Preparing the ADS512101 board for eCos Development

Overview

In a typical development environment, the ADS512101 board boots from flash into the RedBoot ROM monitor. eCos applications are configured for RAM startup and then downloaded and run on the board via the debugger **powerpc-eabi-gdb**. Preparing the board therefore usually involves programming a suitable RedBoot image into flash memory.

The following RedBoot configurations are supported:

Configuration	Description	Use	File
ROMRAM	RedBoot loaded from ROM into RAM	`redboot_ROMRAM.ecm`	`redboot_ROMRAM.bin`
ROM	RedBoot running directly from flash	`redboot_ROM.ecm`	`redboot_ROM.bin`
RAM	RedBoot running in external RAM	`redboot_RAM.ecm`	`redboot_RAM.bin`
JTAG	RedBoot running in external RAM, loaded by JTAG	`redboot_JTAG.ecm`	`redboot_JTAG.bin`

Under normal circumstances the ROMRAM RedBoot is used. The JTAG RedBoot is used to install the ROMRAM RedBoot on the board.

For serial communications, all versions run with 8 bits, no parity, and 1 stop bit at 115200 baud. The baud rate can be changed using the flash configuration console baud rate option. RedBoot also supports flash management.

Initial Installation

The simplest approach to installing RedBoot is to make use of a JTAG device to run a version of RedBoot on the board and use that to download and install RedBoot. The following is a simple step-by-step guide to installing RedBoot on the board using a Ronetix PEEDI JTAG emulator:

1. Set up the PEEDI as described in the Ronetix documentation. The `peedi.mpc5121ads.cfg` file should be used to setup and configure the hardware.

2. Connect a null modem serial cable between the ADS512101 board and a suitable host. Run a serial terminal emulator (Hyperterm or minicom) on the host, connecting to the serial device at 115200 baud with no flow control.

3. Connect an ethernet cable between the board and your local network.

4. From the `loaders/ads512101` sub-directory of your eCosPro installation, copy `redboot_JTAG.srec` and `redboot_ROMRAM.bin` to the data area of a TFTP server the PEEDI can access.

5. Connect a telnet session to the PEEDI and issue a reset command to the PEEDI to put the device into a known state:

   ```
   mpc5121> reset stop
   ++ info: user reset
   mpc5121>
   ++ info: HRESET, SRESET and TRST asserted
   ++ info: TRST released
   ++ info: BYPASS check passed
   ++ info: 1 TAP controller(s) detected
   ```

```
++ info: TAP : IDCODE = 0x1540A01D, Freescale MPC5121
++ info: HRESET and SRESET released
++ info: CPU PVR is 0x80862010 (e300c4)
++ info: CPU SVR is 0x80180020
++ info: setting breakpoint at 0xFFF00100
++ info: core 0: initialized
```

6. Now issue the following command, substituting your own TFTP server address:

```
mpc5121>> mem load tftp://10.0.1.1/redboot_JTAG.srec srec
** warning: default file for this core not specified
** warning: use CORE_FILE parameter to specify default file
++ info: Loading image file:  tftp://10.0.1.1/redboot_JTAG.srec
++ info: At absolute address: 0x00000000
loading at 0x0
loading at 0x3300
loading at 0xB300
loading at 0x13300
loading at 0x1B300
loading at 0x22080

Successfully loaded 154KB (158408 bytes) in 1.2s
mpc5121>>
```

7. Now issue the go command:

```
mpc5121> go 0x100
```

You should see something similar to the following output on the board serial line.

```
+**Warning** FLASH configuration checksum error or invalid key
Use 'fconfig -i' to [re]initialize database
Ethernet eth0: MAC address 08:00:3e:28:7a:ba
IP: 10.0.2.5/255.0.0.0, Gateway: 10.0.0.3
Default server: 0.0.0.0
DNS server IP: 10.0.1.1, DNS domain name: <null>

RedBoot(tm) bootstrap and debug environment [JTAG]
Non-certified release, version UNKNOWN - built 15:05:36, Jun 20 2011

Copyright (C) 2000-2009 Free Software Foundation, Inc.
Copyright (C) 2003-2011 eCosCentric Limited
RedBoot is free software, covered by the eCos license, derived from the
GNU General Public License. You are welcome to change it and/or distribute
copies of it under certain conditions. Under the license terms, RedBoot's
source code and full license terms must have been made available to you.
Redboot comes with ABSOLUTELY NO WARRANTY.

Platform: ADS512101 (PowerPC MPC5121e)
RAM: 0x00000000-0x08000000 [0x0003e400-0x07fb1000 available]
FLASH: 0xfc000000-0xffffffff, 256 x 0x40000 blocks
RedBoot>
```

8. RedBoot's flash management and configuration should be initialized as follows:

```
RedBoot> fis init
About to initialize [format] FLASH image system - continue (y/n)? y
*** Initialize FLASH Image System
... Unlocking from 0xfffc0000-0xffffffff: .
... Erase from 0xfffc0000-0xffffffff: .
... Program from 0x07fc0000-0x08000000 to 0xfffc0000: .
... Locking from 0xfffc0000-0xffffffff: .
RedBoot> fconfig -i
Initialize non-volatile configuration - continue (y/n)? y
Run script at boot: false
Use BOOTP for network configuration: true
Default server IP address: 10.0.1.1
```

```
Console baud rate: 115200
DNS domain name: example.com
DNS server IP address: 10.0.1.1
Network hardware address [MAC]: 0x08:0x00:0x3E:0x28:0x7A:0xBA
GDB connection port: 9000
Force console for special debug messages: false
Network debug at boot time: false
Update RedBoot non-volatile configuration - continue (y/n)? y
... Unlocking from 0xfffc0000-0xffffffff: .
... Erase from 0xfffc0000-0xffffffff: .
... Program from 0x07fc0000-0x08000000 to 0xfffc0000: .
... Locking from 0xfffc0000-0xffffffff: .
RedBoot>
```

For the "Default server IP address", enter the IP address of the TFTP server on which the `redboot_ROMRAM.bin` is to be found.

9. Now we need to download and program a ROMRAM version of RedBoot. From RedBoot, issue the following command:

```
RedBoot> load -r -b %{freememlo} redboot_ROMRAM.bin
Using default protocol (TFTP)
Raw file loaded 0x0003e400-0x000682a7, assumed entry at 0x0003e400
RedBoot>
```

10. Program the RedBoot into the board:

```
RedBoot> fis cre RedBoot
An image named 'RedBoot' exists - continue (y/n)? y
... Unlocking from 0xfff00000-0xfffbffff: ...
... Erase from 0xfff00000-0xfffbffff: ...
... Program from 0x0003e400-0x000682a8 to 0xfff00000: ...
... Locking from 0xfff00000-0xfffbffff: ...
RedBoot>
```

11. RedBoot is now programmed into the board. Detach the PEEDI and reset the board and you should see the following output:

```
+Ethernet eth0: MAC address 08:00:3e:28:7a:ba
IP: 10.0.2.5/255.0.0.0, Gateway: 10.0.0.3
Default server: 0.0.0.0
DNS server IP: 10.0.1.1, DNS domain name: <null>

RedBoot(tm) bootstrap and debug environment [ROMRAM]
Non-certified release, version UNKNOWN - built 15:12:36, Jun 20 2011

Copyright (C) 2000-2009 Free Software Foundation, Inc.
Copyright (C) 2003-2011 eCosCentric Limited
RedBoot is free software, covered by the eCos license, derived from the
GNU General Public License. You are welcome to change it and/or distribute
copies of it under certain conditions. Under the license terms, RedBoot's
source code and full license terms must have been made available to you.
Redboot comes with ABSOLUTELY NO WARRANTY.

Platform: ADS512101 (PowerPC MPC5121e)
RAM: 0x00000000-0x08000000 [0x0003e400-0x07fb1000 available]
FLASH: 0xfc000000-0xffffffff, 256 x 0x40000 blocks
RedBoot>
```

To reinstall RedBoot, a new binary file can be installed and programmed into flash from the installed ROMRAM RedBoot, from step 9 above. It is not necessary to use JTAG for this unless the board is rendered unusable.

Rebuilding RedBoot

Should it prove necessary to rebuild a RedBoot binary, this is done most conveniently at the command line. The steps needed to rebuild the the ROM version of RedBoot for the ADS512101 are:

```
    $ mkdir redboot_ads512101_romram
    $ cd redboot_ads512101_romram
    $ ecosconfig new ads512101 redboot
    $ ecosconfig import $ECOS_REPOSITORY/hal/powerpc/ads512101/current/misc/ads512101_redboot_ROMRAM.ecm
    $ ecosconfig resolve
    $ ecosconfig tree
    $ make
```

At the end of the build the `install/bin` subdirectory should contain the file `redboot.bin`.

Name

Configuration — Platform-specific Configuration Options

Overview

The ADS512101 platform HAL package is loaded automatically when eCos is configured for an `ads512101` target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The ADS512101 platform HAL package supports four separate startup types:

RAM
: This is the startup type which is normally used during application development. The board has ROMRAM RedBoot running from 0x00000000 and applications will be loaded from 0x00100000. powerpc-eabi-gdb is then used to load a RAM startup application into memory and debug it. It is assumed that the hardware has already been initialized by RedBoot. By default the application will use the eCos virtual vectors mechanism to obtain certain services from RedBoot, including diagnostic output.

ROM
: This startup type can be used for finished applications which will be programmed into flash at location 0xFFF00000. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization.

ROMRAM
: This startup type can be used for finished applications which will be programmed into flash at location 0xFFF00000. The first thing the application does is to relocate itself to RAM at location 0x00000000. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization.

JTAG
: This startup type can be used for finished applications which will be loaded into external RAM via a JTAG debugger. The application will be self-contained with no dependencies on services provided by other software. The JTAG debugger should initialize the hardware enough to load the code into RAM, eCos startup code will perform any further hardware initialization.

RedBoot and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building RedBoot.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then the serial port will be claimed for HAL diagnostics.

Flash Driver

The ADS512101 board contains 64MiB of flash memory. The `CYGPKG_DEVS_FLASH_AMD_AM29XXXXX_V2` package contains all the code necessary to support access to the flash. The ADS512101 platform HAL package contains definitions that customize the driver to the ADS512101 board.

System Clock

By default, the system clock interrupts once every 10ms, corresponding to a 100Hz clock. This can be changed by the configuration option `CYGNUM_HAL_RTC_DENOMINATOR` which corresponds to the clock frequency. Other clock-related settings are recalculated automatically if the denominator is changed.

Ethernet Driver

The ADS512101 board uses the MPC5121e's internal FEC ethernet device attached to an external PHY. The `CYGPKG_DEVS_ETH_POWERPC_FEC` package contains all the code necessary to support this device. The `CYGPKG_DEVS_ETH_POWERPC_ADS512101` package contains definitions that customize the driver to the ADS512101 board. This driver is not active until the generic Ethernet support package, `CYGPKG_IO_ETH_DRIVERS`, is included in the configuration.

RTC Driver

The ADS512101 board uses an ST M41T662 I²C Real Time Clock. The `CYGPKG_DEVICES_WALLCLOCK_ST_M41TXX` package contains all the code necessary to support this device. This device also needs the `CYGPKG_IO_I2C` package to be loaded. This driver is not active until the generic wallclock device support package, `CYGPKG_IO_WALLCLOCK`, is included in the configuration.

Watchdog Driver

The ADS512101 board uses the MPC5121e's internal watchdog support. The `CYGPKG_DEVICES_WATCHDOG_MPC512X` package contains all the code necessary to support this device. Within that package the `CYGNUM_DEVICES_WATCHDOG_POWERPC_MPC512X_DESIRED_TIMEOUT_MS` configuration option controls the watchdog timeout, and by default will force a reset of the board upon timeout. This driver is not active until the generic watchdog device support package, `CYGPKG_IO_WATCHDOG`, is included in the configuration.

UART Serial Driver

The ADS512101 board uses the MPC5121e's Programmable Serial Controllers (PSC's) configured for UART mode. Two serial UART adaptors are available on the board. However, only PSC3 is attached to a (pin D-Sub connector. PSC4 is connected to header P8.

I2C Driver

The MPC512X HAL contains a driver for the I²C busses on the board. There are several devices attached to the busses, of which, only the RTC on Bus0 is actually used bt eCos.

CAN Driver

The MPC512X contains four Freescale MSCAN devices, although only one of these is brought out to an external connector. The package `CYGPKG_DEVS_CAN_MSCAN` is a general driver for the MSCAN, and the package `CYGPKG_DEVS_CAN_ADS512101` configures this for the ADS512101 platform.

Compiler Flags

The platform HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos. There is one flag specific to this port:

`-mcpu=e300c3`

The powerpc-eabi-gcc compiler supports many variants of the PowerPC architecture. A `-m` option should be used to select the specific variant in use. The MPC5121e is a e300c4 processor, and the current tools do not have an option to select this processor directly so instead we select a processor that is identical as far as the compiler is concerned, the e300c3.

Name

JTAG support — Usage

Use of JTAG for debugging

JTAG can be used to single-step and debug applications loaded in RAM, or even applications resident in ROM.

The MPC512x core only supports two hardware breakpoints, and so they should be used sparingly. If using a GDB front-end such as Eclipse, check it has not set unnecessary extra breakpoints. Some JTAG devices give the option of whether to set hardware or software breakpoints by default. Be sure to configure your device appropriately.

Ronetix PEEDI notes

On the Ronetix PEEDI, the `peedi.mpc5121ads.cfg` file should be used to setup and configure the hardware to an appropriate state to load programs. This includes setting up the clocks, chip selects and SDRAM controller.

The `peedi.mpc5121ads.cfg` file also contains an option to define whether hardware or software breakpoints are used by default, using the `CORE0_BREAKMODE` directive in the `[PLATFORM_MPC8300]` section. Edit this file if you wish to use hardware break points, and remember to restart the PEEDI to make the changes take effect.

On the PEEDI, debugging can be performed either via the telnet interface or using **powerpc-eabi-gdb** and the GDB interface. In the case of the latter, **powerpc-eabi-gdb** needs to connect to TCP port 2000 on the PEEDI's IP address. For example:

```
    (gdb) target remote 111.222.333.444:2000
```

By default when the PEEDI is powered up, the target will always run the initialization section of the `peedi.mpc5121ads.cfg` file, and halts the target. This behaviour is repeated whenever the board is reset.

If the board is reset (by pressing the reset button) and the '**go**' command is then given, then the board will boot as normal. If a RedBoot is resident in flash, it will be run.

Consult the PEEDI documentation for information on other features.

Configuration of JTAG applications

If the JTAG device has initialized the processor, such as by using the `peedi.mpc5121ads.cfg` configuration on the PEEDI, applications can be loaded directly into RAM without requiring a ROM monitor. This loading can be done directly through the JTAG device, or where supported by the JTAG device, through GDB.

In order to configure the application to support this mode, some configuration settings are required. Firstly `CYGSEM_HAL_USE_ROM_MONITOR` must be disabled. Secondly the `CYGDBG_HAL_DIAG_TO_DEBUG_CHAN` option should be disabled in order to prevent HAL diagnostic output being encoded into GDB ($O) packets. Selecting the JTAG startup type in the configuration tool sets these options automatically.

Running RAM applications

Once loaded and running via JTAG, HAL diagnostic output will appear by default on PSC3.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the ADS512101 hardware, and should be read in conjunction with that specification. The ADS512101 platform HAL package complements the PowerPC architectural HAL and the MPC51XX variant HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset the HAL will initialize or reinitialize the on-chip peripherals that eCos uses. There is an exception for RAM startup applications which depend on a ROM monitor for certain services.

For ROM startup, the HAL will perform additional initialization, setting up the external RAM and programming the various internal registers. This is all done in the `hal_hardware_init` function in the assembler source file `ads512101.S`.

Linker Scripts and Memory Maps

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

Flash
: This is located at address 0xFC000000 of the physical memory space. It is mapped by the BAR registers 1-1 to virtual address 0xFC000000 with caching enabled, and to 0x50000000 with caching disabled. The PowerPC reset vector is at 0xFFF00100 so RedBoot is normally programmed from 0xFFF00000.

SDRAM
: This is located at address 0x00000000 of the physical memory space. The first 0x3000 bytes are used for the exception entry trampolines. The following 512 bytes contain the VSR table and the next 256 bytes are normally used for the eCos virtual vectors, allowing RAM-based applications to use services provided by the ROM monitor. For ROM and JTAG startup, all remaining SDRAM is available. For RAM startup, available RAM starts at location 0x00100000, with the bottom 1MiB reserved for use by RedBoot. The SDRAM is mapped 1-1 with cache enabled at virtual address 0x00000000 and uncached at 0x20000000.

SRAM
: The 128KiB of on-chip SRAM is mapped 1-1 at 0x30000000. This memory is not used by eCos and is therefore available for application use.

Peripherals
: All on-chip peripherals are accessed relative to the address in the IMMBAR register. Both the PEEDI configuration file and eCos itself set this to 0xE0000000. The CPLD is mapped to 0xE2000000 and is accessible just beyond the IMMBAR peripherals.

Real-time characterization

The tm_basic kernel test gives statistics gathered about the real-time characterization and performance of the kernel. The sample output is shown here for information.

Example 350.1. ads512101 Real-time characterization

```
            Startup, main stack : stack used  1032 size  5920
             Startup :  Interrupt stack used   571 size  4096
             Startup : Idlethread stack used   480 size  2048

eCos Kernel Timings
Notes: all times are in microseconds (.000001) unless otherwise stated

Reading the hardware clock takes 0 'ticks' overhead
... this value will be factored out of all other measurements
```

```
Clock interrupt took    1.49 microseconds (74 raw clock ticks)

Testing parameters:
   Clock samples:            32
   Threads:                  64
   Thread switches:         128
   Mutexes:                  32
   Mailboxes:                32
   Semaphores:               32
   Scheduler operations:    128
   Counters:                 32
   Flags:                    32
   Alarms:                   32


                                 Confidence
     Ave     Min     Max     Var  Ave  Min  Function
  ======  ======  ======  ====== ========== ========
    2.91    2.38    3.68    0.29   46%  32% Create thread
    0.28    0.26    1.00    0.02   98%  98% Yield thread [all suspended]
    0.29    0.26    0.98    0.04   90%  90% Suspend [suspended] thread
    0.27    0.26    0.62    0.01   96%  96% Resume thread
    0.40    0.38    1.10    0.03   96%  96% Set priority
    0.03    0.02    0.24    0.01   98%  73% Get priority
    0.72    0.68    1.90    0.06   84%  84% Kill [suspended] thread
    0.26    0.26    0.34    0.00   98%  98% Yield [no other] thread
    0.41    0.38    0.86    0.03   67%  85% Resume [suspended low prio] thread
    0.26    0.26    0.28    0.00   96%  96% Resume [runnable low prio] thread
    0.33    0.32    0.42    0.01   62%  35% Suspend [runnable] thread
    0.27    0.26    0.28    0.01   73%  73% Yield [only low prio] thread
    0.26    0.26    0.34    0.01   84%  84% Suspend [runnable->not runnable]
    0.69    0.68    0.84    0.02   90%  90% Kill [runnable] thread
    0.64    0.60    2.16    0.06   93%  93% Destroy [dead] thread
    1.14    1.10    1.44    0.05   79%  79% Destroy [runnable] thread
    3.93    2.88    5.30    0.38   54%   7% Resume [high priority] thread
    0.67    0.62    2.36    0.06   85%  75% Thread switch

    0.02    0.02    0.18    0.00   99%  99% Scheduler lock
    0.21    0.20    0.30    0.01   62%  62% Scheduler unlock [0 threads]
    0.21    0.20    0.22    0.01   61%  61% Scheduler unlock [1 suspended]
    0.21    0.20    0.30    0.01   68%  68% Scheduler unlock [many suspended]
    0.21    0.20    0.22    0.01   60%  60% Scheduler unlock [many low prio]

    0.18    0.04    0.80    0.11   43%  43% Init mutex
    0.37    0.34    1.28    0.06   96%  96% Lock [unlocked] mutex
    0.38    0.34    1.72    0.08   96%  96% Unlock [locked] mutex
    0.30    0.26    1.02    0.04   96%  96% Trylock [unlocked] mutex
    0.26    0.26    0.34    0.00   96%  96% Trylock [locked] mutex
    0.06    0.04    0.42    0.02   96%  96% Destroy mutex
    1.57    1.54    1.82    0.02   93%   3% Unlock/Lock mutex

    0.33    0.14    1.22    0.15   62%  50% Create mbox
    0.01    0.00    0.04    0.01   50%  46% Peek [empty] mbox
    0.36    0.32    1.44    0.07   96%  96% Put [first] mbox
    0.02    0.02    0.10    0.00   96%  96% Peek [1 msg] mbox
    0.35    0.32    0.46    0.04   71%  71% Put [second] mbox
    0.02    0.02    0.10    0.00   96%  96% Peek [2 msgs] mbox
    0.34    0.30    1.28    0.06   96%  96% Get [first] mbox
    0.31    0.30    0.40    0.01   50%  46% Get [second] mbox
    0.29    0.26    0.92    0.04   96%  96% Tryput [first] mbox
    0.31    0.28    0.86    0.03   96%  96% Peek item [non-empty] mbox
    0.34    0.32    0.80    0.03   96%  96% Tryget [non-empty] mbox
    0.27    0.26    0.34    0.01   96%  59% Peek item [empty] mbox
    0.27    0.26    0.36    0.01   96%  50% Tryget [empty] mbox
    0.03    0.02    0.12    0.01   71%  71% Waiting to get mbox
    0.03    0.02    0.12    0.01   71%  71% Waiting to put mbox
    0.08    0.06    0.50    0.03   96%  96% Delete mbox
    1.17    1.16    1.34    0.01   96%  96% Put/Get mbox

    0.07    0.04    0.26    0.04   75%  75% Init semaphore
    0.26    0.26    0.34    0.01   93%  93% Post [0] semaphore
    0.33    0.32    0.42    0.01   50%  46% Wait [1] semaphore
    0.27    0.24    0.74    0.03   96%  96% Trywait [0] semaphore
    0.25    0.24    0.36    0.01   96%  65% Trywait [1] semaphore
    0.05    0.04    0.22    0.02   96%  62% Peek semaphore
    0.06    0.04    0.32    0.02   50%  46% Destroy semaphore
    1.09    1.08    1.20    0.01   96%  62% Post/Wait semaphore

    0.21    0.06    0.66    0.08   50%  25% Create counter
    0.03    0.02    0.20    0.02   96%  75% Get counter value
    0.02    0.02    0.10    0.00   96%  96% Set counter value
    0.31    0.30    0.56    0.02   96%  93% Tick counter
    0.06    0.04    0.32    0.02   59%  37% Delete counter

    0.08    0.04    0.54    0.06   93%  71% Init flag
    0.30    0.26    1.00    0.04   96%  96% Destroy flag
    0.26    0.24    0.90    0.04   96%  96% Mask bits in flag
    0.30    0.28    0.88    0.04   96%  96% Set bits in flag [no waiters]
    0.41    0.36    1.42    0.06   96%  96% Wait for flag [AND]
    0.35    0.34    0.44    0.01   96%  50% Wait for flag [OR]
    0.38    0.36    0.38    0.00   96%   3% Wait for flag [AND/CLR]
    0.35    0.34    0.46    0.01   96%  50% Wait for flag [OR/CLR]
    0.02    0.02    0.12    0.01   96%  96% Peek on flag

    0.34    0.18    1.14    0.08   71%  21% Create alarm
    0.41    0.38    1.18    0.05   96%  96% Initialize alarm
    0.25    0.24    0.42    0.02   96%  62% Disable alarm
    0.39    0.36    0.88    0.03   96%  96% Enable alarm
    0.30    0.28    0.46    0.02   96%  50% Delete alarm
    0.35    0.34    0.42    0.01   96%  59% Tick counter [1 alarm]
    2.13    2.12    2.14    0.01   53%  53% Tick counter [many alarms]
    0.57    0.56    0.68    0.01   96%  59% Tick & fire counter [1 alarm]
    9.50    9.50    9.58    0.00   96%  96% Tick & fire counters [>1 together]
    2.36    2.36    2.36    0.00  100% 100% Tick & fire counters [>1 separately]
    1.18    1.18    1.32    0.00   99%  99% Alarm latency [0 threads]
    1.45    1.18    1.72    0.16   51%  27% Alarm latency [2 threads]
    4.51    3.24    6.08    0.56   50%  21% Alarm latency [many threads]
    2.11    2.10    3.44    0.02   99%  99% Alarm -> thread resume latency

    0.57    0.32    2.36    0.00            Clock/interrupt latency

    0.75    0.38    2.88    0.00            Clock DSR latency

   18      0    1177  (main stack:  1272)  Thread stack used (1960 total)
           All done, main stack : stack used  1272 size  5920
            All done :  Interrupt stack used   263 size  4096
            All done : Idlethread stack used  1117 size  2048

Timing complete - 29950 ms total

PASS:<Basic timing OK>
EXIT:<done>
```

Other Issues

The ADS512101 platform HAL does not affect the implementation of other parts of the eCos HAL specification. The MPC55XX variant HAL, and the PowerPC architectural HAL documentation should be consulted for further details.

Chapter 351. Freescale MPC5554DEMO Board Support

Name

eCos Support for the MPC5554DEMO Board — Overview

Description

This document covers the Freescale MPC5554DEMO and MPC5554EVB boards. These boards are essentially identical and will be referred to collectively as the MPC5554DEMO throughout this document. The board contains an MPC5554 microprocessor, 512KB of RAM and external connections for one serial channel. There is also 2MB of on-chip flash memory and 64KB of internal SRAM.

For typical eCos development, a RedBoot image is programmed into the on-chip flash memory, and the board will boot this image from reset. RedBoot provides gdb stub functionality so it is then possible to download and debug stand-alone and eCos applications via the gdb debugger using the serial line.

Supported Hardware

The on-chip flash memory consists of 20 blocks in a variety of sizes between 16KiB and 128KiB, occupying 2MB. In a typical setup, RedBoot is programmed into flash at 0x20000 and occupies the next 256KiB. The topmost block is used to manage the flash and hold RedBoot fconfig values. The first 128KiB and blocks between 0x00060000 and 0x001DFFFF may be used by application code.

There is a serial driver `CYGPKG_IO_SERIAL_POWERPC_ESCI` which supports both the eSCI based on-chip serial devices. Only eSCI device A is actually brought out to an external connector. This device can be used by RedBoot for communication with the host. The serial driver package is loaded automatically when configuring for the Mpc5554demo target.

eCos manages the on-chip interrupt controller. The architecture-defined decrementer is used to implement the eCos system clock and the microsecond delay function. Other on-chip devices (Caches, PIO, UARTs) are initialized only as far as is necessary for eCos to run. Other devices (CAN, eTPU, eMIOS etc.) are not touched.

Tools

The MPC5554DEMO port is intended to work with GNU tools configured for a powerpc-eabi target. The original port was undertaken using powerpc-eabi-gcc version 3.3.3, powerpc-eabi-gdb version 6.1, and binutils version 2.14.

Name

Setup — Preparing the MPC5554DEMO board for eCos Development

Overview

In a typical development environment, the MPC5554DEMO board boots from flash into the RedBoot ROM monitor. eCos applications are configured for RAM startup and then downloaded and run on the board via the debugger **powerpc-eabi-gdb**. Preparing the board therefore usually involves programming a suitable RedBoot image into flash memory.

The following RedBoot configurations are supported:

Configuration	Description	Use	File
ROM	RedBoot running from on-chip ROM	`MPC5554DEMO_redboot_ROM.ecm`	`MPC5554DEMO_redboot_ROM.bin`
RAM	RedBoot running in external RAM	`MPC5554DEMO_redboot_RAM.ecm`	`MPC5554DEMO_redboot_RAM.bin`
JTAG	RedBoot running in external RAM, loaded by JTAG	`MPC5554DEMO_redboot_JTAG.ecm`	`MPC5554DEMO_redboot_JTAG.bin`

For serial communications, all versions run with 8 bits, no parity, and 1 stop bit at 38400 baud. The baud rate can be changed using the flash configuration console baud rate option. RedBoot also supports flash management.

Initial Installation

The simplest approach to installing RedBoot is to make use of the **eSys Flasher** utility and the P&E Wiggler shipped with the board. The reader is referred to the documentation for this utility for details of its use; the following is a simple step-by-step guide to using it to install RedBoot on the board:

1. Install the **eSys Flasher** utility on a suitable PC running Windows XP.

2. Copy `MPC5554DEMO_redboot_ROM.srec` to a suitable location on the Windows PC.

3. Connect a straight-through (not null modem) serial cable between the COM-1 serial port of the board and a serial port on a convenient host (which need not be the PC running **eSys Flasher**). Run a terminal emulator (Hyperterm or minicom) at 38400 baud.

4. Connect the P&E Wiggler to the MPC5554DEMO board and via a USB cable to the Windows PC. Connect the power supply to the MPC5554DEMO board and power it on. At some point during this process XP may ask you to install a device driver. The necessary files will have been installed with the utility, so just follow the directions to install the driver.

5. Start **eSys Flasher** and select "P&E Wiggler (USB)" from the initial dialog. If the program connects to the board then the MCU and Part ID should be displayed at the top right of the next dialog.

6. Click on the "Program Flash" button. Select "S-Record" in the following dialog (and optionally "Verify after Program"), and press "Program". In the "Open" dialog navigate to where the `MPC5554DEMO_redboot_ROM.srec` file is located and select it. The utility will now erase, program and optionally verify the flash. When finished, press the "Close" button to exit the utility.

7. Pressing the reset button on the board should cause RedBoot to start up and display the following output:

   ```
   +**Warning** FLASH configuration checksum error or invalid key
   Use 'fconfig -i' to [re]initialize database
   ```

```
RedBoot(tm) bootstrap and debug environment [ROM]
Non-certified release, version UNKNOWN - built 18:06:16, Nov 26 2007

Platform: MPC5554DEMO (PowerPC MPC5554)
Copyright (C) 2000, 2001, 2002, 2003, 2004 Free Software Foundation, Inc.
Copyright (C) 2003, 2004, 2005, 2006, 2007 eCosCentric Limited

RAM: 0x20000000-0x20080000, [0x20006b88-0x20051000] available
FLASH: 0x00000000-0x001fffff, 1 x 0x4000 blocks, 2 x 0xc000 blocks, 1 x 0x4000 blocks, 2 x 0x10000 blocks, 14 x 0x20
RedBoot>
```

8. RedBoot's flash management and configuration should be initialized as follows:

```
RedBoot> fis init
About to initialize [format] FLASH image system - continue (y/n)? y
*** Initialize FLASH Image System
... Unlocking from 0x001e0000-0x001fffff: .
... Erase from 0x001e0000-0x001fffff: .
... Program from 0x20060000-0x20080000 to 0x001e0000: .
... Locking from 0x001e0000-0x001fffff: .
RedBoot> fconfig -i
Initialize non-volatile configuration - continue (y/n)? y
Run script at boot: false
Update RedBoot non-volatile configuration - continue (y/n)? y
... Unlocking from 0x001e0000-0x001fffff: .
... Erase from 0x001e0000-0x001fffff: .
... Program from 0x20060000-0x20080000 to 0x001e0000: .
... Locking from 0x001e0000-0x001fffff: .
RedBoot>
```

Rebuilding RedBoot

Should it prove necessary to rebuild a RedBoot binary, this is done most conveniently at the command line. The steps needed to rebuild the the ROM version of RedBoot for the MPC5554DEMO are:

```
$ mkdir redboot_mpc5554demo_rom
$ cd redboot_mpc5554demo_rom
$ ecosconfig new mpc5554demo redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/powerpc/mpc5554demo/current/misc/mpc5554demo_redboot_ROM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

At the end of the build the `install/bin` subdirectory should contain the file `redboot.srec`.

Name

Configuration — Platform-specific Configuration Options

Overview

The MPC5554DEMO platform HAL package is loaded automatically when eCos is configured for an `mpc5554demo` target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The MPC5554DEMO platform HAL package supports three separate startup types:

RAM
: This is the startup type which is normally used during application development. The board has RedBoot programmed into flash at location 0x00020000 and boots from that location. powerpc-eabi-gdb is then used to load a RAM startup application into memory and debug it. It is assumed that the hardware has already been initialized by RedBoot. By default the application will use the eCos virtual vectors mechanism to obtain certain services from RedBoot, including diagnostic output.

ROM
: This startup type can be used for finished applications which will be programmed into flash at location 0x00020000. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization.

JTAG
: This startup type can be used for finished applications which will be loaded into external RAM via a JTAG debugger. The application will be self-contained with no dependencies on services provided by other software. The JTAG debugger should initialize the hardware enough to load the code into RAM, eCos startup code will perform any further hardware initialization.

RedBoot and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building RedBoot.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then the serial port will be claimed for HAL diagnostics.

Flash Driver

The MPC5554 microcontroller contains 2MiB of flash memory. The `CYGPKG_DEVS_FLASH_MPC5500` package contains all the code necessary to support access to the flash. The MPC5554DEMO platform HAL package contains definitions that customize the driver to the MPC5554DEMO board.

System Clock

By default, the system clock interrupts once every 10ms, corresponding to a 100Hz clock. This can be changed by the configuration option `CYGNUM_HAL_RTC_DENOMINATOR` which corresponds to the clock frequency. Other clock-related settings are recalculated automatically if the denominator is changed.

Compiler Flags

The platform HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos. There are two flags specific to this port:

`-mcpu=8540`
The powerpc-eabi-gcc compiler supports many variants of the PowerPC architecture. A `-m` option should be used to select the specific variant in use. The MPC5554 is a Book E processor, and the current tools do not have an option to select this processor directly so instead we select a processor that is also Book E based and which is supported: the MPC8540.

`-msoft-float`
The PowerPC processor used in the MPC5554DEMO boards does not have a floating point unit. Therefore it is necessary to translate any floating point operations into software emulation. This option tells the compiler to do that.

Name

JTAG support — Usage

Use of JTAG for debugging

JTAG can be used to single-step and debug applications loaded in RAM, or even applications resident in ROM.

Debugging of ROM applications is only possible if using hardware breakpoints. The MPC5554 core only supports four such hardware breakpoints, and so they should be used sparingly. If using a GDB front-end such as Eclipse, check it has not set unnecessary extra breakpoints. Some JTAG devices give the option of whether to set hardware or software breakpoints by default. Be sure to configure your device appropriately.

Abatron BDI3000 notes

On the Abatron BDI3000, the `bdi3000.mpc5554demo.cfg` file should be used to setup and configure the hardware to an appropriate state to load programs. This includes setting up the TLB, PLL, external SRAM cache and flash memory controller.

The `bdi3000.mpc5554demo.cfg` file also contains an option to define whether hardware or software breakpoints are used by default, using the `BREAKMODE` directive in the `[TARGET]` section. Edit this file if you wish to use software break points, and remember to use the **boot** command on the BDI3000 command line interface to make the changes take effect.

On the BDI3000, debugging can be performed either via the telnet interface or using **powerpc-eabi-gdb** and the bdiGDB interface. In the case of the latter, **powerpc-eabi-gdb** needs to connect to TCP port 2001 on the BDI3000's IP address. For example:

```
(gdb) target remote 111.222.333.444:2001
```

By default when the BDI3000 is powered up, the target will always run the initialization section of the `bdi3000.mpc5554demo.cfg` file (which configures the CPU clock among other things), and halts the target. This behaviour is repeated with the **reset halt** command.

If the board is reset when in '**reset halt**' mode (either with the '**reset halt**' or '**reset**' commands, or by pressing the reset button) and the '**go**' command is then given, then the board will boot from ROM as normal.

It is also possible for the target to always run, without initialization, after the reset button has been pressed. This mode is selected with the **reset run** command. This conveniently allows the target to be connected to the JTAG debugger, and be able to reset it with the reset button, without being required to always type '**go**' every time. Thereafter, invoking the **reset** command will repeat the previous reset style. Also in this mode, exceptions will be handled by board software, rather than causing the JTAG debugger to halt the CPU.

Suitably configured RAM applications can be loaded either via GDB, or directly via the telnet CLI. For example:

```
MPC5554>load 0x20000500 test.bin bin
Loading test.bin , please wait ....
Loading program file passed
MPC5554>go 0x20000540
```

Consult the BDI3000 documentation for information on other formats.

Ronetix PEEDI notes

On the Ronetix PEEDI, the `peedi.mpc5554demo.cfg` file should be used to setup and configure the hardware to an appropriate state to load programs. This includes setting up the TLB, PLL, external SRAM cache and flash memory controller.

The `peedi.mpc5554demo.cfg` file also contains an option to define whether hardware or software breakpoints are used by default, using the `CORE0_BREAKMODE` directive in the `[PLATFORM_MPC5500]` section. Edit this file if you wish to use software break points, and remember to reset the PEEDI using the reset button or with the **reboot** command on the PEEDI command line interface to make the changes take effect.

On the PEEDI, debugging can be performed either via the telnet interface or using **powerpc-eabi-gdb** and the GDB interface. In the case of the latter, **powerpc-eabi-gdb** needs to connect to TCP port 2000 on the PEEDI's IP address. For example:

```
(gdb) target remote 111.222.333.444:2000
```

By default when the PEEDI is powered up, the target will always run the initialization section of the `peedi.mpc5554demo.cfg` file (which configures the CPU clock among other things), and halts the target. This behaviour is repeated with the **reset reset** command.

If the board is reset (either with the '**reset**', or by pressing the reset button) and the '**go**' command is then given, then the board will boot from ROM as normal.

It is also possible for the target to always run, without initialization, after the reset button has been pressed. This mode is selected with the **reset run** command. This conveniently allows the target to be connected to the JTAG debugger, and be able to reset it with the reset button, without being required to always type '**go**' every time. Thereafter, invoking the **reset** command will repeat the previous reset style. Also in this mode, exceptions will be handled by board software, rather than causing the JTAG debugger to halt the CPU.

Suitably configured RAM applications can be loaded either via GDB, or directly via the telnet CLI. For example:

```
mpc5554>mem load tftp://192.168.1.1/test.bin bin 0x20000500
++ info: Loading image file:  tftp://192.168.1.1/test.bin
++ info: At absolute address: 0x20000500
loading at 0x20000500
loading at 0x20008500
loading at 0x20010500
loading at 0x20018500

Successfully loaded 128KB (131072 bytes) in 0.3s
mpc5554>go 0x20000540
```

Consult the PEEDI documentation for information on other formats.

Configuration of JTAG applications

If the JTAG device has initialized the processor, such as by using the `peedi.mpc5554.cfg` configuration on the PEEDI, or the `bdi3000.mpc5554.cfg` configuration on the BDI3000, applications can be loaded directly into RAM without requiring a ROM monitor. This loading can be done directly through the JTAG device, or where supported by the JTAG device, through GDB.

In order to configure the application to support this mode, some configuration settings are required. Firstly `CYGSEM_HAL_USE_ROM_MONITOR` must be disabled. Secondly the `CYGDBG_HAL_DIAG_TO_DEBUG_CHAN` option should be disabled in order to prevent HAL diagnostic output being encoded into GDB ($O) packets. Selecting the JTAG startup type in the configuration tool sets these options automatically.

Running RAM applications

Once loaded and running via JTAG, HAL diagnostic output will appear by default on eSCI device A.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the MPC5554DEMO hardware, and should be read in conjunction with that specification. The MPC5554DEMO platform HAL package complements the PowerPC architectural HAL and the MPC55XX variant HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset the HAL will initialize or reinitialize most of the on-chip peripherals. There is an exception for RAM startup applications which depend on a ROM monitor for certain services.

For ROM startup, the HAL will perform additional initialization, setting up the external RAM and programming the various internal registers. This is all done in the `hal_hardware_init` function in the assembler source file `mpc5554demo.S`.

Linker Scripts and Memory Maps

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

Flash

This is located at address 0x00000000 of the physical memory space. It is mapped by the BAM 1-1 to virtual address 0x00000000 with caching enabled, and by eCos to 0x10000000 with caching disabled.

SRAM

This is located at address 0x20000000 of the physical memory space. The first 512 bytes are used for the VSR table and the next 256 bytes are normally used for the eCos virtual vectors, allowing RAM-based applications to use services provided by the ROM monitor. The next 512 bytes are used for a shared interrupt state table, recording mask and priority values for each interrupt source. For ROM and JTAG startup, all remaining SRAM is available. For RAM startup, available RAM starts at location 0x00008000, with the bottom 32kiB reserved for use by RedBoot. The SRAM is mapped 1-1 with cache enabled at virtual address 0x20000000 and uncached at 0x30000000.

on-chip peripherals

These are available via 1-1 uncached mappings at 0xFFF00000 and 0xC3F00000.

Real-time characterization

The tm_basic kernel test gives statistics gathered about the real-time characterization and performance of the kernel. The sample output is shown here for information.

Example 351.1. mpc5554demo Real-time characterization

```
            Startup, main stack : stack used   708 size  5664
             Startup :  Interrupt stack used   856 size  4096
             Startup : Idlethread stack used   220 size  2048

eCos Kernel Timings
Notes: all times are in microseconds (.000001) unless otherwise stated

Reading the hardware clock takes 13 'ticks' overhead
... this value will be factored out of all other measurements
Clock interrupt took    6.71 microseconds (858 raw clock ticks)

Testing parameters:
   Clock samples:            32
   Threads:                  17
   Thread switches:         128
   Mutexes:                  32
   Mailboxes:                32
   Semaphores:               32
   Scheduler operations:    128
   Counters:                 32
   Flags:                    32
   Alarms:                   32


                                 Confidence
     Ave     Min     Max     Var  Ave  Min  Function
  ======  ======  ======  ====== ========== ========
   17.12   15.38   34.45    2.10   94%  88% Create thread
    1.33    1.22    3.09    0.21   94%  94% Yield thread [all suspended]
    1.97    1.52    6.89    0.63   94%  88% Suspend [suspended] thread
    1.53    1.13    7.45    0.70   94%  94% Resume thread
    1.91    1.72    3.33    0.17   52%  41% Set priority
    0.09    0.07    0.48    0.05   94%  94% Get priority
    5.12    3.79   22.52    2.05   94%  94% Kill [suspended] thread
    1.37    1.22    3.71    0.28   94%  94% Yield [no other] thread
    2.25    2.00    6.18    0.46   94%  94% Resume [suspended low prio] thread
    1.18    1.14    1.88    0.08   94%  94% Resume [runnable low prio] thread
    1.65    1.52    3.00    0.21   82%  82% Suspend [runnable] thread
    1.58    1.22    7.39    0.68   94%  94% Yield [only low prio] thread
    1.14    1.10    1.69    0.06   94%  94% Suspend [runnable->not runnable]
    4.23    3.61   11.97    0.92   94%  94% Kill [runnable] thread
    3.71    2.78   15.52    1.39   94%  94% Destroy [dead] thread
    5.83    5.39   12.48    0.78   94%  94% Destroy [runnable] thread
   13.28   10.80   26.84    2.26   82%  88% Resume [high priority] thread
    2.69    2.59   12.10    0.19   98%  98% Thread switch

    0.05    0.04    1.36    0.02   99%  99% Scheduler lock
    0.85    0.84    1.98    0.02   99%  99% Scheduler unlock [0 threads]
    0.84    0.84    0.98    0.00   99%  99% Scheduler unlock [1 suspended]
    0.84    0.84    0.98    0.00   99%  99% Scheduler unlock [many suspended]
    0.85    0.84    1.07    0.00   99%  99% Scheduler unlock [many low prio]

    0.86    0.16    3.54    0.17   87%   6% Init mutex
    1.76    1.39    7.95    0.39   96%  96% Lock [unlocked] mutex
    2.00    1.58   15.00    0.81   96%  96% Unlock [locked] mutex
    1.41    1.21    7.68    0.39   96%  96% Trylock [unlocked] mutex
    1.16    1.10    2.88    0.11   96%  96% Trylock [locked] mutex
    0.16    0.15    0.55    0.02   96%  96% Destroy mutex
    8.00    7.90   10.75    0.19   93%  93% Unlock/Lock mutex

    1.29    0.33   13.13    0.74   90%  96% Create mbox
    0.87    0.20    2.05    0.74   59%  59% Peek [empty] mbox
    1.97    1.52   13.41    0.72   96%  96% Put [first] mbox
    0.03    0.02    0.60    0.04   96%  96% Peek [1 msg] mbox
    1.58    1.52    3.07    0.10   93%  93% Put [second] mbox
    0.03    0.02    0.44    0.03   96%  96% Peek [2 msgs] mbox
    2.26    1.65   19.64    1.09   96%  96% Get [first] mbox
    1.78    1.65    3.17    0.18   90%  68% Get [second] mbox
    1.42    1.28    5.59    0.26   96%  96% Tryput [first] mbox
    1.42    1.30    5.07    0.23   96%  96% Peek item [non-empty] mbox
    1.53    1.42    5.02    0.22   96%  96% Tryget [non-empty] mbox
    1.27    1.24    1.85    0.05   93%  93% Peek item [empty] mbox
    1.33    1.27    3.45    0.13   96%  96% Tryget [empty] mbox
    0.10    0.05    1.82    0.11   96%  96% Waiting to get mbox
    0.07    0.05    0.46    0.04   93%  93% Waiting to put mbox
    0.42    0.34    3.06    0.17   96%  96% Delete mbox
    5.90    5.57   13.93    0.62   93%  93% Put/Get mbox

    0.46    0.15    1.38    0.08   84%  12% Init semaphore
    1.13    1.04    1.82    0.13   90%  71% Post [0] semaphore
    1.30    1.25    1.64    0.08   87%  87% Wait [1] semaphore
    1.17    1.09    3.89    0.17   96%  96% Trywait [0] semaphore
    1.04    1.04    1.23    0.01   96%  96% Trywait [1] semaphore
    0.18    0.15    1.05    0.05   96%  96% Peek semaphore
    0.17    0.15    0.73    0.04   96%  96% Destroy semaphore
    5.01    4.98    5.89    0.06   96%  96% Post/Wait semaphore

    1.02    0.24    4.36    0.21   90%   6% Create counter
    0.69    0.06    3.77    0.66   78%  78% Get counter value
    0.07    0.04    1.02    0.06   96%  96% Set counter value
    1.31    1.26    1.85    0.09   84%  84% Tick counter
    0.18    0.15    1.30    0.07   96%  96% Delete counter

    0.50    0.15    1.72    0.09   87%   6% Init flag
    1.46    1.23    6.98    0.34   96%  96% Destroy flag
    1.08    1.02    3.10    0.13   96%  96% Mask bits in flag
    1.37    1.22    6.13    0.30   96%  96% Set bits in flag [no waiters]
    1.96    1.70   10.25    0.52   96%  96% Wait for flag [AND]
    1.65    1.63    2.24    0.04   96%  96% Wait for flag [OR]
    1.71    1.70    2.16    0.03   96%  96% Wait for flag [AND/CLR]
    1.64    1.63    1.96    0.02   96%  96% Wait for flag [OR/CLR]
    0.00    0.00    0.00    0.00  100% 100% Peek on flag

    2.24    0.68   11.62    0.59   93%   3% Create alarm
    3.04    2.24   14.94    0.97   87%  84% Initialize alarm
    1.14    1.03    4.33    0.20   96%  96% Disable alarm
    2.09    1.83    9.42    0.47   96%  96% Enable alarm
    1.33    1.22    4.84    0.22   96%  96% Delete alarm
    1.53    1.41    5.14    0.23   96%  96% Tick counter [1 alarm]
    7.59    7.47   11.20    0.23   96%  96% Tick counter [many alarms]
    2.49    2.44    4.22    0.11   96%  96% Tick & fire counter [1 alarm]
   44.24   44.22   44.78    0.03   96%  96% Tick & fire counters [>1 together]
    8.64    8.62    9.40    0.05   96%  96% Tick & fire counters [>1 separately]
    5.57    5.53    8.68    0.05   99%  99% Alarm latency [0 threads]
    6.86    5.54    7.84    0.58   67%  32% Alarm latency [2 threads]
    6.85    5.53   11.23    0.67   76%  19% Alarm latency [many threads]
   10.28   10.05   31.93    0.44   97%  97% Alarm -> thread resume latency

    1.12    1.08    6.98    0.00            Clock/interrupt latency

    2.77    2.26   12.51    0.00            Clock DSR latency

  635    604     665  (main stack:  1244)  Thread stack used (1704 total)
           All done, main stack : stack used  1244 size  5664
            All done :  Interrupt stack used   288 size  4096
            All done : Idlethread stack used   617 size  2048

Timing complete - 30880 ms total

PASS:<Basic timing OK>
EXIT:<done>
```

Other Issues

The MPC5554DEMO platform HAL does not affect the implementation of other parts of the eCos HAL specification. The MPC55XX variant HAL, and the PowerPC architectural HAL documentation should be consulted for further details.

Chapter 352. MPC8309KIT Board Support

Name

eCos Support for the MPC8309KIT Board — Overview

Description

This document covers the MPC8309KIT board. The board consists of a MPC8309SOM card plugged in to a MPC830X carrier board. The SOM contains an MPC8309 microprocessor, 256MiB of RAM and 8MiB of Flash. There are external connections for a single RS232 UART and the Fast Ethernet Controller.

For typical eCos development, a RedBoot image is programmed into the flash memory, and the board will boot this image from reset. RedBoot provides gdb stub functionality so it is then possible to download and debug stand-alone and eCos applications via the gdb debugger using the serial line or Ethernet.

Supported Hardware

The Flash memory consists of 127 blocks each of 64KiB, and 8 blocks of 8KiB, occupying 8MiB. In a typical setup, RedBoot is programmed into the base of flash at 0xFE000000 and occupies the next 768KiB. The topmost 64KiB block is used to manage the flash and holds RedBoot fconfig values. The remainder may be used by application code.

There is a serial driver `CYGPKG_IO_SERIAL_GENERIC_16X5X` which supports the 16X5X compatible DUARTs. The package `CYGPKG_IO_SERIAL_POWERPC_MPC8309KIT` provides definitions to configure the generic driver to the board. Only UART0 is actually brought out to a usable DB9 external connector via an RS232 transceiver; UART1 is delivered to the second DB9 via an RS485 transceiver. This device can be used by RedBoot for communication with the host. The serial driver package is loaded automatically when configuring for the MPC8309KIT target.

The UEC Ethernet driver, `CYGPKG_DEVS_ETH_POWERPC_UEC` is used to control the QUICC Engine UCC based Ethernet device. This driver only supports a single Ethernet interface at present: the RJ-45 socket on the SOM board.

eCos manages the on-chip interrupt controller. The architecture-defined decrementer is used to implement the eCos system clock and the microsecond delay function. A GTM is used to implement a profiling timer. Other on-chip devices (Caches, GPIO, UARTs) are initialized only as far as is necessary for eCos to run. The remaining devices (PCI, CAN, USB etc.) are not touched.

Interrupt Nesting

eCos normally operates with a sequential interrupt model, where each ISR is run with interrupts disabled and coincident ISRs are run in turn. However, the PowerPC HAL and kernel are designed to support nested interrupts where required. The MPC83XX variant HAL and MPC8309KIT platform HAL have been validated for nested interrupt support.

No special configuration is required to use nested interrupts, support for this is always present. Nested interrupts can be enabled simply by re-enabling interrupts in an ISR. However, there are a number of issues that need to be considered:

1. All ISRs are entered with interrupts disabled and should be exited with interrupts disabled. So the ISR must bracket any code that can be preempted with enable and disable calls.

2. An ISR must cancel the cause of its own interrupt before re-enabling interrupts otherwise it could put the CPU into an interrupt loop. This should include a call to `cyg_interrupt_acknowledge()`, maybe writing to device registers, or even a call to `cyg_interrupt_mask()` to block the interrupt source.

3. Stack usage within ISRs and the level of nesting may require the value of `CYGNUM_HAL_COMMON_INTERRUPTS_STACK_SIZE` to be increased. Note that DSRs are also run on the interrupt stack, so excessive stack usage in any DSR must also be accounted for.

4. ISRs for standard devices do not enable interrupts, and will thus run to completion with interrupts off. These ISRs will impose latency on the start of any nested ISR and will preempt it while interrupts are enabled. However, the only ISRs currently supported are for the Ethernet, serial, I²C and system timer, all of which are either minimal, or very simple.

5. There is no prioritisation of ISRs, hardware prioritisation only determines which of any simultaneously pending ISRs is delivered to the CPU next. Any ISR can interrupt any other, low priority ISRs are not blocked by high priority ones. If some sort of prioritisation is required, it must be implemented in software by selectively masking and unmasking vectors as appropriate.

The following shows the suggested layout of an ISR that supports nesting:

```
cyg_uint32 nested_isr( cyg_uint32 vector, CYG_ADDRWORD data )
{
    CYG_INTERRUPT_STATE ints;

    cyg_interrupt_acknowledge( vector );

    // Cancel or mask interrupt here

    // Enable CPU interrupts
    HAL_ENABLE_INTERRUPTS();

    // Code here is preemptable

    // Disable interrupts before return
    HAL_DISABLE_INTERRUPTS(ints);

    return 1;
}
```

See the Timers Test for an example of a program that uses nested interrupts.

Tools

The MPC8309KIT port is intended to work with GNU tools configured for a powerpc-eabi target. The original port was undertaken using powerpc-eabi-gcc version 4.4.5, powerpc-eabi-gdb version 7.2, and binutils version 2.20.1.

Name

Setup — Preparing the MPC8309KIT board for eCos Development

Overview

In a typical development environment, the MPC8309KIT board boots from flash into the RedBoot ROM monitor. eCos applications are configured for RAM startup and then downloaded and run on the board via the debugger **powerpc-eabi-gdb**. Preparing the board therefore usually involves programming a suitable RedBoot image into flash memory.

The following RedBoot configurations are supported:

Configuration	Description	Use	File
ROM	RedBoot running directly from flash	`redboot_ROM.ecm`	`redboot_ROM.bin`
RAM	RedBoot running in external RAM	`redboot_RAM.ecm`	`redboot_RAM.bin`
JTAG	RedBoot running in external RAM, loaded by JTAG	`redboot_JTAG.ecm`	`redboot_JTAG.bin`

Under normal circumstances the ROM RedBoot is used. The JTAG RedBoot is used to install the ROM RedBoot, and the RAM RedBoot may be used to update it.

For serial communications, all versions run with 8 bits, no parity, and 1 stop bit at 115200 baud. The baud rate can be changed using the flash configuration console baud rate option. RedBoot also supports flash management.

Initial Installation

The simplest approach to installing RedBoot is to make use of a JTAG device to run a JTAG version of RedBoot on the board and use that to download and install ROM RedBoot. The following is a simple step-by-step guide to installing RedBoot on the board using a Ronetix PEEDI JTAG emulator:

1. Set up the PEEDI as described in the Ronetix documentation. The `peedi.mpc8309kit.cfg` file should be used to setup and configure the hardware.

2. Connect a null modem serial cable between the MPC8309KIT board and a suitable host. Run a serial terminal emulator (Hyperterm, Teraterm or minicom) on the host, connecting to the serial device at 115200 baud with no flow control.

3. Connect an Ethernet cable between the board and your local network.

4. From the `loaders/mpc8309kit` sub-directory of your eCosPro installation, copy `redboot_JTAG.srec` and `redboot_ROM.bin` to the data area of a TFTP server the PEEDI can access.

5. Connect a telnet session to the PEEDI and issue a reset command to the PEEDI to put the device into a known state:

```
mpc8309> reset stop
++ info: user reset
mpc8309>
++ info: HRESET, SRESET and TRST asserted
++ info: TRST released
++ info: BYPASS check passed
++ info: 1 TAP controller(s) detected
++ info: TAP : IDCODE = 0x16AC101D, MPC8308
++ info: overriding RCW (0xA0600000 0x44050008)
```

```
      ++ info: HRESET and SRESET released
      ++ info: CPU PVR is 0x80850020 (e300c3)
      ++ info: CPU SVR is 0x81100011
      ++ info: setting breakpoint at 0x00000100
      ++ info: core 0: initialized
```

6. Now issue the following command, substituting your own TFTP server address:

```
      mpc8309> mem load tftp://192.168.7.22/redboot_JTAG.srec srec
      * warning: default file for this core not specified
      ** warning: use CORE_FILE parameter to specify default file
      ++ info: Loading image file:  tftp://192.168.7.22/redboot_JTAG.srec
      ++ info: At absolute address: 0x00000000
      loading at 0x0
      loading at 0x3500
      loading at 0xB500
      loading at 0x13500
      loading at 0x1B500
      loading at 0x22220
      loading at 0x2A220
      loading at 0x32220
      loading at 0x327E8
      loading at 0x3A7E8

      Successfully loaded 248KB (253956 bytes) in 1.7s
      mpc8309>
```

7. Now issue the go command:

```
      mpc8309> go 0x100
```

You should see something similar to the following output on the board serial line.

```
      +**Warning** FLASH configuration checksum error or invalid key
      Use 'fconfig -i' to [re]initialize database
      Ethernet eth0: MAC address 00:04:9f:ef:03:01
      IP: 192.168.7.171/255.255.255.0, Gateway: 192.168.7.1
      Default server: 0.0.0.0
      DNS server IP: 192.168.7.3, DNS domain name: <null>

      RedBoot(tm) bootstrap and debug environment [JTAG]
      Non-certified release, version UNKNOWN - built 13:34:41, Mar  2 2012

      Copyright (C) 2000-2009 Free Software Foundation, Inc.
      Copyright (C) 2003-2012 eCosCentric Limited
      RedBoot is free software, covered by the eCos license, derived from the
      GNU General Public License. You are welcome to change it and/or distribute
      copies of it under certain conditions. Under the license terms, RedBoot's
      source code and full license terms must have been made available to you.
      Redboot comes with ABSOLUTELY NO WARRANTY.

      Platform: MPC8309KIT (PowerPC MPC8309)
      RAM: 0x00000000-0x10000000 [0x00049bc0-0x0ffe1000 available]
      FLASH: 0xfe000000-0xfe7fffff, 127 x 0x10000 blocks, 8 x 0x2000 blocks
      RedBoot>
```

8. RedBoot's flash management and configuration should be initialized as follows:

```
      RedBoot> fis init
      About to initialize [format] FLASH image system - continue (y/n)? y
      *** Initialize FLASH Image System
      ... Unlocking from 0xfe7f0000-0xfe7fffff: ........
      ... Erase from 0xfe7f0000-0xfe7fffff: ........
      ... Program from 0x0fff0000-0x10000000 to 0xfe7f0000: ........
      ... Locking from 0xfe7f0000-0xfe7fffff: ........
      RedBoot> fconfig -i
      Initialize non-volatile configuration - continue (y/n)? y
```

```
Run script at boot: false
Use BOOTP for network configuration: true
Default server IP address: 192.168.7.22
Console baud rate: 115200
DNS domain name: example.com
DNS server IP address: 192.168.7.3
Network hardware address [MAC] for eth0: 0x00:0x04:0x9F:0xEF:0x03:0x073
GDB connection port: 9000
Force console for special debug messages: false
Network debug at boot time: false
Update RedBoot non-volatile configuration - continue (y/n)? y
... Unlocking from 0xfe7f0000-0xfe7fffff: ........
... Erase from 0xfe7f0000-0xfe7fffff: ........
... Program from 0x0fff0000-0x10000000 to 0xfe7f0000: ........
... Locking from 0xfe7f0000-0xfe7fffff: ........
RedBoot>
```

For the "Default server IP address", enter the IP address of the TFTP server on which the `redboot_ROM.bin` is to be found.

9. Now we need to download and program a ROM version of RedBoot. From RedBoot, issue the following command, substituting the IP address of your TFTP server:

```
RedBoot> load -r -b %{freememlo} -h 192.168.7.22 redboot_ROM.bin
Using default protocol (TFTP)
Raw file loaded 0x00049c00-0x00089cd7, assumed entry at 0x00049c00
RedBoot>
```

10. Program the RedBoot into the board:

```
RedBoot> fis create RedBoot
An image named 'RedBoot' exists - continue (y/n)? y
... Unlocking from 0xfe000000-0xfe0bffff: ...........
... Erase from 0xfe000000-0xfe0bffff: ...........
... Program from 0x00049c00-0x00089cd8 to 0xfe000000: .....
... Locking from 0xfe000000-0xfe0bffff: ...........
... Unlocking from 0xfe7f0000-0xfe7fffff: ........
... Erase from 0xfe7f0000-0xfe7fffff: ........
... Program from 0x0fff0000-0x10000000 to 0xfe7f0000: ........
... Locking from 0xfe7f0000-0xfe7fffff: ........
RedBoot>
```

11. RedBoot is now programmed into the board. Detach the PEEDI and reset the board and you should see the following output:

```
+Ethernet eth0: MAC address 00:04:9f:ef:03:73
IP: 192.168.7.182/255.255.255.0, Gateway: 192.168.7.1
Default server: 192.168.7.22
DNS server IP: 192.168.7.3, DNS domain name: <null>

RedBoot(tm) bootstrap and debug environment [ROM]
Non-certified release, version UNKNOWN - built 10:42:26, Mar  2 2012

Copyright (C) 2000-2009 Free Software Foundation, Inc.
Copyright (C) 2003-2012 eCosCentric Limited
RedBoot is free software, covered by the eCos license, derived from the
GNU General Public License. You are welcome to change it and/or distribute
copies of it under certain conditions. Under the license terms, RedBoot's
source code and full license terms must have been made available to you.
Redboot comes with ABSOLUTELY NO WARRANTY.

Platform: MPC8309KIT (PowerPC MPC8309)
RAM: 0x00000000-0x10000000 [0x0001a9b8-0x0ffe1000 available]
FLASH: 0xfe000000-0xfe7fffff, 127 x 0x10000 blocks, 8 x 0x2000 blocks
RedBoot>
```

To reinstall RedBoot, the above process can be repeated, or a RAM RedBoot can be loaded by the ROM RedBoot and used like the JTAG RedBoot to load and program a new ROM RedBoot.

Rebuilding RedBoot

Should it prove necessary to rebuild a RedBoot binary, this is done most conveniently at the command line. The steps needed to rebuild the the ROM version of RedBoot for the MPC8309KIT are:

```
    $ mkdir redboot_mpc8309kit_rom
    $ cd redboot_mpc8309kit_rom
    $ ecosconfig new mpc8309kit redboot
    $ ecosconfig import $ECOS_REPOSITORY/hal/powerpc/mpc8309kit/current/misc/mpc8309kit_redboot_ROM.ecm
    $ ecosconfig resolve
    $ ecosconfig tree
    $ make
```

At the end of the build the `install/bin` subdirectory should contain the file `redboot.bin`.

Name

Configuration — Platform-specific Configuration Options

Overview

The MPC8309KIT platform HAL package is loaded automatically when eCos is configured for an `mpc8309kit` target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The MPC8309KIT platform HAL package supports three separate startup types:

RAM
: This is the startup type which is normally used during application development. The board has ROM RedBoot using RAM from 0x00000000 and applications will be loaded from 0x00100000. powerpc-eabi-gdb is used to load a RAM startup application into memory and debug it. It is assumed that the hardware has already been initialized by RedBoot. By default the application will use the eCos virtual vectors mechanism to obtain certain services from RedBoot, including diagnostic output.

ROM
: This startup type can be used for finished applications which will be programmed into flash at location 0xFE000000. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization.

JTAG
: This startup type can be used for finished applications which will be loaded into external RAM via a JTAG debugger. The application will be self-contained with no dependencies on services provided by other software. The JTAG debugger should initialize the hardware enough to load the code into RAM, eCos startup code will perform any further hardware initialization.

RedBoot and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building RedBoot.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then the serial port will be claimed for HAL diagnostics.

Flash Driver

The MPC8309KIT board contains 8MiB of flash memory. The `CYGPKG_DEVS_FLASH_AMD_AM29XXXXX_V2` package contains all the code necessary to support access to the flash. The MPC8309KIT platform HAL package contains definitions that customize the driver to the MPC8309KIT board.

System Clock

By default, the system clock interrupts once every 10ms, corresponding to a 100Hz clock. This can be changed by the configuration option `CYGNUM_HAL_RTC_DENOMINATOR` which corresponds to the clock frequency. Other clock-related settings are recalculated automatically if the denominator is changed.

Ethernet Driver

The MPC8309KIT board uses the QUICC Engine UCC UEC1 Ethernet device attached to an external PHY. The `CYGPKG_DEVS_ETH_POWERPC_UEC` package contains all the code necessary to support this device. This driver only supports a single Ethernet interface at present: the RJ-45 socket on the SOM board. This driver is not active until the generic Ethernet support package, `CYGPKG_IO_ETH_DRIVERS`, is included in the configuration.

RTC Driver

The `CYGPKG_DEVICES_WALLCLOCK_MPC83XX` package supports the MPC8309 RTC device. The driver supports only clocking from the 32kHz RTC input clock. This driver is not active until the generic wallclock device support package, `CYGPKG_IO_WALLCLOCK`, is included in the configuration.

Watchdog Driver

The MPC8309KIT board uses the MPC8309 internal watchdog support. This device is compatible with the watchdog on the MPC512X family and so shares that device driver. The `CYGPKG_DEVICES_WATCHDOG_MPC512X` package contains the code necessary to support this device. Within that package the `CYGNUM_DEVICES_WATCHDOG_POWERPC_MPC512X_DESIRED_TIMEOUT_MS` configuration option controls the watchdog timeout, and by default will force a reset of the board upon timeout. This driver is not active until the generic watchdog device support package, `CYGPKG_IO_WATCHDOG`, is included in the configuration.

UART Serial Driver

The MPC8309KIT board uses the 16X5X compatible serial DUARTs. Two serial UART adaptors are available on the board. Only UART0 is actually brought out to a usable DB9 external connector via an RS232 transceiver; UART1 is delivered to the second DB9 via an RS485 transceiver. The generic 16X5X driver `CYGPKG_IO_SERIAL_GENERIC_16X5X` supports the 16X5X compatible DUARTs. The package `CYGPKG_IO_SERIAL_POWERPC_MPC8309KIT` provides definitions to configure the generic driver to the board.

I2C Driver

The MPC512X HAL contains a driver for the I^{2}C busses on the board. There are several devices attached to the busses, see the board documentation for a description. The file `mpc512x_i2c.c` and the header `plf_io.h` contain definitions for some of these devices. The test programs `pca9534_1.c` and `pca9534_2.c` contain tests that exercise the I^{2}C bus by flashing the LEDs on the carrier board.

Compiler Flags

The platform HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos. There is one flag specific to this port:

`-mcpu=e300c3`

The powerpc-eabi-gcc compiler supports many variants of the PowerPC architecture. A `-m` option should be used to select the specific variant in use. The MPC8309 contains an e300c3 processor, and this option allows the compiler to optimize code for this processor variant.

Name

JTAG support — Usage

Use of JTAG for debugging

JTAG can be used to single-step and debug applications loaded in RAM, or even applications resident in ROM.

The MPC8309 core only supports a limited number of hardware breakpoints, so they should be used sparingly. If using a GDB front-end such as Eclipse, check it has not set unnecessary extra breakpoints. Some JTAG devices give the option of whether to set hardware or software breakpoints by default. Be sure to configure your device appropriately.

Ronetix PEEDI Notes

On the Ronetix PEEDI, the `peedi.mpc8309kit.cfg` file should be used to setup and configure the hardware to an appropriate state to load programs. This includes setting up the clocks, chip selects and SDRAM controller.

The `peedi.mpc8309kit.cfg` file also contains an option to define whether hardware or software breakpoints are used by default, using the `CORE0_BREAKMODE` directive in the `[PLATFORM_MPC8300]` section. Edit this file if you wish to use hardware break points, and remember to restart the PEEDI to make the changes take effect.

On the PEEDI, debugging can be performed either via the telnet interface or using **powerpc-eabi-gdb** and the GDB interface. In the case of the latter, **powerpc-eabi-gdb** needs to connect to TCP port 2000 on the PEEDI's IP address. For example:

```
    (gdb) target remote 111.222.333.444:2000
```

By default when the PEEDI is powered up, the target will always run the initialization section of the `peedi.mpc8309kit.cfg` file, and halts the target. This behaviour is repeated whenever the board is reset.

If the '**reset run**' command is given, then the board will boot as normal. If a RedBoot is resident in flash, it will be run.

Consult the PEEDI documentation for information on other features.

Configuration of JTAG applications

If the JTAG device has initialized the processor, such as by using the `peedi.mpc8309kit.cfg` configuration on the PEEDI, applications can be loaded directly into RAM without requiring a ROM monitor. This loading can be done directly through the JTAG device, or where supported by the JTAG device, through GDB.

In order to configure the application to support this mode, some configuration settings are required. Firstly `CYGSEM_HAL_USE_ROM_MONITOR` must be disabled. Secondly the `CYGDBG_HAL_DIAG_TO_DEBUG_CHAN` option should be disabled in order to prevent HAL diagnostic output being encoded into GDB ($O) packets. Selecting the JTAG startup type in the configuration tool sets these options automatically.

Running JTAG applications

Once loaded and running via JTAG, HAL diagnostic output will appear by default on the serial line.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the MPC8309KIT hardware, and should be read in conjunction with that specification. The MPC8309KIT platform HAL package complements the PowerPC architectural HAL and the MPC83XX variant HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset the HAL will initialize or reinitialize the on-chip peripherals that eCos uses. There is an exception for RAM startup applications which depend on a ROM monitor for certain services.

For ROM startup, the HAL will perform additional initialization, setting up the external RAM and programming the various internal registers. This is all done in the `hal_hardware_init` function in the assembler source file `mpc8309kit.S`.

Linker Scripts and Memory Maps

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

Flash
: This is located at address 0xFE000000 of the physical memory space. It is mapped by the BAR registers 1-1 to virtual address 0xFE000000 with caching enabled, and to 0x5E000000 with caching disabled. While the PowerPC reset vector is at 0xFFF00100 the chip bootstrap mechanism means that ROM applications actually boot from 0x00000100, where the flash is remapped during startup. Initialization code remaps the flash to 0xFE000000 and moves execution there.

SDRAM
: This is located at address 0x00000000 of the physical memory space. The first 0x3000 bytes are used for the exception entry trampolines. The following 512 bytes contain the VSR table and the next 256 bytes are normally used for the eCos virtual vectors, allowing RAM-based applications to use services provided by the ROM monitor. For ROM and JTAG startup, all remaining SDRAM is available. For RAM startup, available RAM starts at location 0x00100000, with the bottom 1MiB reserved for use by RedBoot. The SDRAM is mapped 1-1 with cache enabled at virtual address 0x00000000 and uncached at 0x20000000.

Peripherals
: All on-chip peripherals are accessed relative to the address in the IMMBAR register. Both the PEEDI configuration file and eCos itself set this to 0xE0000000.

Real-time characterization

The tm_basic kernel test gives statistics gathered about the real-time characterization and performance of the kernel. The sample output is shown here for information.

Example 352.1. mpc8309kit Real-time characterization

```
            Startup, main stack : stack used  1080 size  6048
             Startup :  Interrupt stack used  4064 size  4096
             Startup : Idlethread stack used   508 size  2048

eCos Kernel Timings
Notes: all times are in microseconds (.000001) unless otherwise stated

Reading the hardware clock takes 1 'ticks' overhead
... this value will be factored out of all other measurements
Clock interrupt took    2.38 microseconds (79 raw clock ticks)
```

```
Testing parameters:
   Clock samples:            32
   Threads:                  64
   Thread switches:         128
   Mutexes:                  32
   Mailboxes:                32
   Semaphores:               32
   Scheduler operations:    128
   Counters:                 32
   Flags:                    32
   Alarms:                   32


                                 Confidence
    Ave     Min     Max     Var  Ave  Min  Function
  ======  ======  ======  ====== ========== ========
   17.15   16.32   20.55    0.34   65%  14% Create thread
    0.98    0.90    3.06    0.10   95%  75% Yield thread [all suspended]
    0.92    0.84    2.31    0.10   87%  75% Suspend [suspended] thread
    0.92    0.84    1.98    0.10   87%  75% Resume thread
    1.16    1.08    3.39    0.12   85%  84% Set priority
    0.24    0.18    0.66    0.05   75%  78% Get priority
    1.91    1.74    7.26    0.21   96%  82% Kill [suspended] thread
    0.96    0.90    2.61    0.09   87%  75% Yield [no other] thread
    1.28    0.96    3.06    0.14   79%   9% Resume [suspended low prio] thread
    0.91    0.84    2.07    0.10   87%  75% Resume [runnable low prio] thread
    1.23    0.99    3.60    0.13   85%   1% Suspend [runnable] thread
    0.98    0.90    3.06    0.10   95%  75% Yield [only low prio] thread
    0.93    0.84    2.16    0.10   87%  70% Suspend [runnable->not runnable]
    1.77    1.62    6.69    0.17   96%  90% Kill [runnable] thread
    2.22    2.04    5.13    0.12   70%  43% Destroy [dead] thread
    3.31    2.88    8.94    0.21   87%  10% Destroy [runnable] thread
    8.08    6.87   14.22    0.34   76%   1% Resume [high priority] thread
    2.00    1.92    3.99    0.08   86%  75% Thread switch

    0.06    0.03    0.57    0.01   74%  25% Scheduler lock
    0.68    0.66    1.65    0.02   99%  61% Scheduler unlock [0 threads]
    0.66    0.66    1.29    0.01   99%  99% Scheduler unlock [1 suspended]
    0.68    0.66    1.23    0.02   99%  50% Scheduler unlock [many suspended]
    0.68    0.66    1.41    0.02   50%  49% Scheduler unlock [many low prio]

    0.24    0.09    1.47    0.10   75%  31% Init mutex
    1.16    0.96    3.69    0.17   75%  78% Lock [unlocked] mutex
    1.23    0.99    4.41    0.20   75%  81% Unlock [locked] mutex
    0.98    0.81    3.06    0.15   75%  84% Trylock [unlocked] mutex
    0.96    0.78    2.97    0.15   71%  71% Trylock [locked] mutex
    0.12    0.06    0.87    0.08   93%  71% Destroy mutex
    5.43    5.01   10.05    0.66   90%  90% Unlock/Lock mutex

    0.37    0.24    1.71    0.09   71%  46% Create mbox
    0.13    0.06    0.33    0.06   59%  71% Peek [empty] mbox
    1.29    1.11    3.72    0.18   68%  87% Put [first] mbox
    0.13    0.06    0.33    0.07   53%  68% Peek [1 msg] mbox
    1.33    1.20    3.90    0.18   96%  90% Put [second] mbox
    0.13    0.06    0.33    0.07   50%  71% Peek [2 msgs] mbox
    1.30    1.08    3.84    0.18   71%  81% Get [first] mbox
    1.32    1.20    3.78    0.17   96%  87% Get [second] mbox
    1.28    1.17    3.24    0.13   96%  84% Tryput [first] mbox
    1.16    1.05    2.97    0.13   93%  87% Peek item [non-empty] mbox
    1.20    1.08    3.39    0.15   96%  87% Tryget [non-empty] mbox
    1.11    0.96    3.03    0.15   68%  71% Peek item [empty] mbox
    1.09    0.93    2.91    0.14   93%  59% Tryget [empty] mbox
    0.15    0.09    0.48    0.07   87%  75% Waiting to get mbox
    0.15    0.09    0.48    0.08   81%  75% Waiting to put mbox
    0.36    0.24    1.47    0.10   62%  53% Delete mbox
    3.55    3.39    7.59    0.25   96%  96% Put/Get mbox
    0.18    0.06    1.56    0.10   78%  46% Init semaphore
    0.82    0.72    1.95    0.09   50%  46% Post [0] semaphore
    0.91    0.78    2.43    0.11   81%  46% Wait [1] semaphore
    0.83    0.72    2.01    0.10   53%  46% Trywait [0] semaphore
    0.77    0.72    1.35    0.07   90%  71% Trywait [1] semaphore
    0.16    0.06    1.20    0.09   59%  46% Peek semaphore
    0.15    0.06    1.02    0.08   46%  46% Destroy semaphore
    2.96    2.85    5.31    0.16   96%  96% Post/Wait semaphore

    0.28    0.21    1.71    0.09   96%  96% Create counter
    0.13    0.00    0.45    0.06   59%  21% Get counter value
    0.02    0.00    0.15    0.04   81%  81% Set counter value
    0.96    0.87    1.86    0.08   81%  84% Tick counter
    0.18    0.03    0.69    0.10   65%  43% Delete counter

    0.18    0.06    1.35    0.10   78%  46% Init flag
    0.87    0.72    2.40    0.12   71%  46% Destroy flag
    0.79    0.72    1.65    0.09   96%  71% Mask bits in flag
    0.88    0.75    2.01    0.09   50%  46% Set bits in flag [no waiters]
    0.99    0.87    3.12    0.15   96%  71% Wait for flag [AND]
    0.97    0.87    2.76    0.14   96%  81% Wait for flag [OR]
    0.99    0.87    3.21    0.15   96%  84% Wait for flag [AND/CLR]
    0.96    0.87    2.67    0.13   96%  84% Wait for flag [OR/CLR]
    0.03    0.00    0.24    0.05   71%  71% Peek on flag

    0.41    0.27    1.47    0.12   68%  56% Create alarm
    1.21    1.02    3.39    0.20   93%  71% Initialize alarm
    0.93    0.84    1.89    0.11   84%  71% Disable alarm
    1.19    0.99    3.48    0.21   96%  71% Enable alarm
    1.05    0.90    2.61    0.16   84%  71% Delete alarm
    0.90    0.81    1.71    0.09   93%  71% Tick counter [1 alarm]
    3.04    2.97    3.96    0.10   96%  71% Tick counter [many alarms]
    1.20    1.11    2.58    0.12   93%  84% Tick & fire counter [1 alarm]
   12.84   12.75   14.22    0.12   96%  71% Tick & fire counters [>1 together]
    3.36    3.27    4.86    0.11   96%  71% Tick & fire counters [>1 separately]
    1.51    1.50    2.79    0.02   99%  99% Alarm latency [0 threads]
    2.22    1.50    3.72    0.36   67%  14% Alarm latency [2 threads]
    7.42    5.28    9.93    1.13   47%  30% Alarm latency [many threads]
    3.83    3.78    7.95    0.10   95%  94% Alarm -> thread resume latency

    0.88    0.39    3.75    0.00            Clock/interrupt latency

    1.39    0.54    6.45    0.00            Clock DSR latency

   17      0    2024  (main stack:  6047)  Thread stack used (2024 total)
           All done, main stack : stack used  1592 size  6048
            All done :  Interrupt stack used   756 size  4096
            All done : Idlethread stack used  1172 size  2048

Timing complete - 31290 ms total

PASS:<Basic timing OK>
EXIT:<done>
```

Other Issues

The MPC8309KIT platform HAL does not affect the implementation of other parts of the eCos HAL specification. The MPC55XX variant HAL, and the PowerPC architectural HAL documentation should be consulted for further details.

Name

GPIO Support — Details

Synopsis

#include <cyg/hal/hal_io.h>

`cyg_uint32 pin = **CYGHWR_HAL_MPC83XX_GPIO**(*ctlr*, *bit*, *mode*);`

CYGHWR_HAL_MPC83XX_GPIO_SET (*pin*);

CYGHWR_HAL_MPC83XX_GPIO_OUT (*pin*, *val*);

CYGHWR_HAL_MPC83XX_GPIO_IN (*pin*, *val*);

Description

This section describes how to use macros provided by eCos to manage GPIO functionality on the MPC83XX processors.

The MPC83XX HAL provides a number of macros to support the encoding of GPIO pin identity and configuration into a single 32 bit descriptor. This is useful to drivers and other packages that need to configure and use different lines for different devices.

A descriptor is created with `CYGHWR_HAL_MPC83XX_GPIO(ctlr, bit, mode)` which takes the following arguments:

ctlr
This identifies the GPIO controller to which the pin is attached. It may take the value 1 or 2.

bit
This gives the bit or pin number within the port. These are numbered from 0 to 32. Note that the bit numbers conform to the Freescale big-endian numbering scheme.

mode
This defines the mode in which the pin is to be used. There are 4 main options: `INPUT` sets the pin as an input, `OUTPUT` sets it as an output. `INPUT_OPENDRAIN` and `OUTPUT_OPENDRAIN` set it as input or output with an open drain.

The following examples show how this macro may be used:

```
// Define controller 1 pin 10 as an input
#define CYGHWR_HAL_MPC8309_PIN1              CYGHWR_HAL_MPC83XX_GPIO( 1, 10, INPUT )

// Define controller 2 pin 23 as an open drain output
#define CYGHWR_HAL_MPC8309_PIN23             CYGHWR_HAL_MPC83XX_GPIO( 2, 23, OUTPUT_OPENDRAIN )
```

Additionally, the macro `CYGHWR_HAL_MPC83XX_GPIO_NONE` may be used in place of a pin descriptor and has a value that no valid descriptor can take. It may therefore be used as a placeholder where no GPIO pin is present or to be used.

The remaining macros all take a GPIO pin descriptor as an argument. `CYGHWR_HAL_MPC83XX_GPIO_SET` configures the pin according to the descriptor and must be called before any other macros. `CYGHWR_HAL_MPC83XX_GPIO_OUT` sets the output to the value of the least significant bit of the *val* argument. The *val* argument of `CYGHWR_HAL_MPC83XX_GPIO_IN` should be a pointer to an int, which will be set to 0 if the pin input is zero, and 1 otherwise.

Name

Test Programs — Details

Test Programs

The MPC8309KIT HAL contains some test programs which allow various aspects of the microcontroller or the architecture to be tested.

Timers Test

The `timers` test checks the functionality of the microcontroller timers and in particular the interrupt nesting mechanism. This test also acts as and example of how to handle nested interrupts. The test programs the four available GTM timers to interrupt at a variety of different rates and records various parameters. The timers are programmed to interrupt at higher rates for higher interrupt priority timers. This information is summarized at the start of the run:

```
Options:
    LOOPS        24
    LOITER       1
    DSRS         1
    MHZ          10
    SYSTICK      1
    DELAY        0
    LATENCY      1
    LATENCY_HIST 20
    LATENCY_BASE 0

    CSB clock    133333332Hz

T  Interval Frequency    Tick Prescaler Vector
4:    127us  10256410    97ns        13     73
2:    355us  10256410    97ns        13     79
3:    731us  10256410    97ns        13     85
1:    999us  10256410    97ns        13     91
```

The options indicate what compile-time options have been applied. See the source of the test for a brief description of each. The table gives for each timer the requested interval between interrupts, the programmed frequency, the resulting length of a tick, the prescaler used to achieve this and the interrupt vector. Each timer is programmed to run at approximately 10MHz and the tick value is the resulting tick duration at that frequency. Each timer will actually interrupt every `Interval*Tick*MHZ` nanoseconds. The vector numbers also define the static priority of the interrupts, so timer 4 is highest priority, and timer 1 lowest. After initialization the test outputs a sequence of tables of the following format every 5 seconds:

```
ISRs max_nesting 5 max_nesting_seen 6
Spurious interrupts: 0

ISR Preempt:
T  Ticks    0    1    2    3    4
4:  944k 688k  81k  89k  84k   66
2:  338k 245k  28k    0  29k  34k
3:  164k 118k  13k  15k    0  16k
1:  120k  86k    0  10k  10k  12k
ISR Nesting:
T      1    2    3    4    5    6
4:  688k 204k  43k 6810  966    0
2:  245k  73k  16k 1991   63    0
3:  118k  36k 8144 1002   24    2
1:   86k  26k 5975  792    4    1
DSRs
T:               0    1    2    3    4
4:  preempt:  671k 5522 5524 5510  369
      count:     0  93k  317    0    0
2:  preempt:  239k 1933    2 1973 1969
      count:     0  33k    0    0    0
3:  preempt:  115k  955 1063    0  955
      count:     0  16k    0    0    0
1:  preempt:   84k    0  711  689  691
      count:     0  12k    0    0    0
```

```
ISR Latency
 T:  Max  Ave  Histogram (ns)...
      ns   ns    0   97  194  291  388  485  582  679  776  873  970 1067 1164 1261 1358 1455 1552 1649 1746 1843+
 4: 1455  485    0    0  261  466  496 700k 233k 2644 1150 1037  941  889  875  714  355  194    1    2    0   61
 2: 1649  485    0    0   79   84   75 248k  81k 1200  784  751  672  577  626  609  550  388  261 1083    9   73
 3: 1649  485    0    0   16   15   20 120k  38k  620  397  396  368  352  343  329  319  243  182  721    8   60
 1: 2716  485    0    0    0    0    0  87k  28k  515  293  293  263  252  275  253  252  200  145  576   14   59
```

The first line shows the depth of ISR nesting seen since the last report, plus the maximum seen throughout the run. The second line counts the number of spurious interrupts seen, and should always be zero. The above example is taken from the end of a run, although there are only 4 timers, this run has seen a nesting level of 5 and a whole run total of 6.

The ISR Preempt table contains a row for each timer. The *Ticks* column shows the total number of ISRs called for this timer. The *0* column shows how many ISR calls interrupted thread state. The remaining columns show how many ISR calls preempted the ISR for the given timer. For example, the ISR for timer 3 preempted the ISR for timer 2 about 15000 times. Mostly the ISRs do not interrupt themselves, but timer 4's ISR has interrupted itself 66 times. This is because the ISRs delay for a while to improve the possibility of preemption, and the accumulation of latencies occasionally results in an ISR still running when the next interrupt occurs. Note that this only records the ISR immediately below the current one on the stack, not every nested ISR.

The ISR Nesting table indicates for each ISR how deeply nested the ISRs are when each is run. In each line, the 1 column indicates how many times the ISR was first on the stack, the 2 column how many times there was one preempted ISR, the 3 column how many times there were two preempted ISRs and so on. For example, the ISR for timer 1 has been at the base of the stack about 86000 times, preempted one other ISR about 26000 times and preempted a stack of five ISRs just once.

The DSRs table contains two rows for each timer. The *preempt:* row shows how many times the ISR preempted the DSR for the given timer. The zero column correspond to thread state as before. For example the ISR for timer 2 preempted the DSR for timer 4 1969 times. The *count:* row shows the range of `count` values passed to the DSR and indicates the number of DSR calls not matched exactly to ISR calls. The ISR calls the DSR every 10 ticks, so the total counts should be one tenth of the ISR Ticks value. In this run only timer 4 has accumulated any instances where a new ISR occurred before a previously posted DSR could run.

The ISR latency table shows, for each ISR the range of ISR latencies. This is done by reading the timer counter on entry to the ISR and calculating the delay from the point at which the timer triggered the interrupt. For each timer the maximum latency seen is recorded, together with the average for the last 5 seconds. In the histogram, each column represents an additional tick of the 10MHz frequency of each timer, multiplied up to its duration in nanoseconds. Entries count the number of ISRs that were seen with that latency and the 1843+ column accumulates all larger latencies.

I2C Tests

Two programs are supplied to test the functioning of I²C. The MPC8309KIT board has a number of I²C devices, but no external access to the busses. These tests access the only device that provides visible confirmation of its functions: by manipulating the LEDs attached to a PCA9534 GPIO expander on the carrier board on I²C bus 2.

There are two programs; `pca9534_1` operates the I²C device in polled mode and `pca9534_2` operates in in interrupt driven mode. Otherwise they are identical.

Chapter 353. MPC512X Variant Support

Name

`CYGPKG_HAL_POWERPC_MPC512X` — eCos Support for the MPC512X Microprocessor Family

Description

The Freescale MPC512X series of PowerPC microcontrollers is supported by eCos with an eCos processor variant HAL and a number of device drivers supporting some of the on-chip peripherals. These include device drivers for PSC serial and SPI, I²C, FEC Ethernet, watchdog devices and the PATA interface. In addition it provides common functionality and definitions that MPC512X based platform ports may require, as well as definitions useful to application developers.

This documentation covers the MPC512X functionality provided but should be read in conjunction with the specific HAL documentation for the platform port. That documentation will cover issues that are platform-specific and are not covered here, and may also describe differences that override or supersede what the STM32 variant HAL provides. The areas that are specific to platform HALs and not the STM32 variant HAL include:

- memory map and related configuration and setup
- Clock parameters
- GPIO and pin configuration
- Any special handling for external interrupts, or additional interrupts
- Diagnostic I/O baud rates
- Additional diagnostic I/O devices, if any

Name

On-chip Subsystems and Peripherals — Hardware Support

Hardware support

This section describes the devices controlled by this HAL.

Cache Handling

The MPC512X contains both data and instruction caches. The `cyg/hal/hal_cache.h` header defines the cache sizes for the processor variants and defines the eCos standard macros for operating on the caches.

PSC support

The MPC512X contains a number of general purpose Programmable Serial Controllers (PSCs) which can be used as UARTs or SPI master or slave controllers (along with other modes that eCos does not support).

For each PSC, *N*, there are a number of configuration options:

CYGHWR_HAL_POWERPC_MPC512X_PSC*N*

> This defines the mode in which this PSC is to be used. If set to `UNUSED` then the PSC is not used and its external pins are available for alternate device functions. If it is set to `UART` it is used as a UART. If set to `SPI` then it is an SPI master, and if set to `SPISLAVE` an SPI slave device.

CYGHWR_HAL_POWERPC_MPC512X_PSC*N*_TXFIFO_SIZE

> This defines the size of the transmit FIFO for this PSC, and must be a multiple of 4. The default value is chosen depending on the mode.

CYGHWR_HAL_POWERPC_MPC512X_PSC*N*_RXFIFO_SIZE

> This defines the size of the receive FIFO for this PSC, and must be a multiple of 4. The default value is chosen depending on the mode.

CYGHWR_HAL_POWERPC_MPC512X_PSC*N*_TXFIFO_ADDRESS

> This defines the address in the shared FIFO RAM of the transmit FIFO for this PSC. This value is usually calculated from the defined sizes of the PSCs and should not be changed without good reason.

CYGHWR_HAL_POWERPC_MPC512X_PSC*N*_RXFIFO_ADDRESS

> This defines the address in the shared FIFO RAM of the transmit FIFO for this PSC. This value is usually calculated from the defined sizes of the PSCs and should not be changed without good reason.

It is normally the responsibility of the platform HAL to define the mode in which each PSC is to be used.

The MPC512X variant HAL supports basic polled HAL diagnostic I/O over any of the PSC UART devices. There is also a fully interrupt-driven serial device driver suitable for eCos applications for all on-chip serial devices. The serial driver consists of an eCos package: `CYGPKG_IO_SERIAL_POWERPC_PSC` which provides all support for the MPC512X PSC serial devices. Using the HAL diagnostic I/O support, any of these devices can be used by the ROM monitor or RedBoot for communication with GDB. If a device is needed by the application, either directly or via the serial driver, then it cannot also be used for GDB communication using the HAL I/O support. An alternative serial port should be used instead.

Any PSC which is put into UART mode will be included in both the HAL's list of available polled serial devices and be available for use by the serial driver. The MPC512X UARTs provide TX and RX data lines plus hardware flow control using RTS/CTS for those UARTs that have them connected.

A separate SPI master driver is available as the package `CYGPKG_DEVS_SPI_POWERPC_PSC`, and SPI slave support is present in this variant HAL and is described later.

Interrupts

The MPC512X HAL provides standard support for interrupt decoding and delivery. The available interrupt vectors are defined in the `cyg/hal/hal_intr.h` header. Interrupts from the FIFO controller are additionally decoded into their own separate set of vectors.

The MPC512X has a somewhat unusual interrupt priority mechanism. Vectors are collected together into groups and the priority of an interrupt can only be varied within that group. The unusual part is that instead of assigning a priority to a vector, a vector is assigned to a priority. It is not permitted to assign a vector to more than one priority. eCos does not enforce unique priority assignments, this is left to the user. Priorities range from 0 to 7; any value outside that range will leave the priority unchanged at the hardware default. Hence in many placed, a priority value of 8 is used to effectively not change the setting.

GPIO and Pin Configuration

The variant HAL provides support for packaging the configuration of a GPIO line into a single 16-bit descriptor that can then be used with macros to configure the pin and set and read its value. Similar descriptor based support is also available for controlling the configuration of external IO pins.

I²C Support

This variant HAL contains an I²C driver that may be used with the standard I²C infrastructure. For each I²C bus there are a number of configuration options:

CYGINT_HAL_POWERPC_MPC512X_I2C_BUS*N*

> This interface must be `implement`ed by the platform HAL to indicate that the given I²C bus is connected to devices.

CYGHWR_HAL_POWERPC_MPC512X_I2C_BUS*N*_CLOCK

> This is the I²C bus clock speed in Hz. Usually frequencies of either 100kHz or 400kHz are chosen, the latter sometimes known as fast mode.

CYGHWR_HAL_POWERPC_MPC512X_I2C_BUS*N*_INTR_PRI

> This is the I²C bus interrupt priority. It may range from 0 to 7; the default of 8 selects the default hardware setting.

Profiling Support

The MPC512X HAL contains support for **gprof**-base profiling using a sampling timer. The default timer used is GPT7. The timer used is selected by a set of `#define`s in `src/var_misc.c` which can be changed to refer to a different timer if required. This timer is only enabled when the gprof profiling package (`CYGPKG_PROFILE_GPROF`) is included and enabled in the eCos configuration, otherwise it remains available for application use.

Clock Control

The platform HAL must provide the input clock frequency (`CYGHWR_HAL_POWERPC_MPC512X_CLOCK_REF_CLK`) in its CDL file. This is then combined with the following options defined in this package to calculate the system clocks:

CYGHWR_HAL_POWERPC_MPC512X_CLOCK_SPMF

This defines the system PLL multiplier and should match the value supplied by the reset configuration word.

CYGHWR_HAL_POWERPC_MPC512X_CLOCK_SYS_DIV

This option defines the system clock divider multiplied by 10. It must match the value supplied by the reset configuration word. CDL does not currently handle real values, so this value must be represented by a scaled integer. Not all values that can be represented by this option are valid.

CYGHWR_HAL_POWERPC_MPC512X_CLOCK_CPMF

This option defines the Core PLL multiplier multiplied by 10. It must match the value supplied by the reset configuration word. CDL does not currently handle real values, so this value must be represented by a scaled integer.

CYGHWR_HAL_POWERPC_MPC512X_CLOCK_SDHC_DIV

This option defines the SDHC divider multiplied by 100. It must match the value programmed into the SCFR2 register by the platform HAL initialization code. The default value equates to the hardware default setting for the register. CDL does not currently handle real values, so this value must be represented by a scaled integer. Not all values that can be represented by this option are valid.

CYGHWR_HAL_POWERPC_MPC512X_CLOCK_DIU_DIV

This option defines the DIU divider multiplied by 100. It must match the value programmed into the SCFR1 register by the platform HAL initialization code. The default value equates to the hardware default setting for the register. CDL does not currently handle real values, so this value must be represented by a scaled integer. Not all values that can be represented by this option are valid.

CYGHWR_HAL_POWERPC_MPC512X_CLOCK_IPS_DIV

This option defines the IPS divider. It must match the value programmed into the SCFR1 register by the platform HAL initialization code. The default value equates to the hardware default setting for the register.

CYGHWR_HAL_POWERPC_MPC512X_CLOCK_PCI_DIV

This option defines the PCI divider. It must match the value programmed into the SCFR1 register by the platform HAL initialization code. The default value equates to the hardware default setting for the register.

CYGHWR_HAL_POWERPC_MPC512X_CLOCK_LPC_DIV

This option defines the LPC divider. It must match the value programmed into the SCFR1 register by the platform HAL initialization code. The default value equates to the hardware default setting for the register.

CYGHWR_HAL_POWERPC_MPC512X_CLOCK_NFC_DIV

This option defines the NFC divider. It must match the value programmed into the SCFR1 register by the platform HAL initialization code. The default value equates to the hardware default setting for the register.

These setting are used to calculate a variety of clock values which are then used in the HALs and drivers to set baud rates, timers and other clock-related features.

Note that when changing or configuring any of these clock settings, you should consult the relevant processor datasheet as there may be both upper and lower constraints on the frequencies of some clock signals, including intermediate clocks. There are also some clocks where, while there is no strict constraint, clock stability is improved if values are chosen wisely. Finally, be aware that increasing clock speeds using this package may have an effect on platform specific properties, such as memory timings which may have to be adjusted accordingly.

Name

SPI Slave — Hardware Support for SPI Slave Device

Synopsis

#include <cyg/hal/mpc512x_spislave.h>

typedef void **hal_mpc512x_spi_slave_rx**(*hal_mpc512x_spi_slave *slave*, *cyg_uint8 *buf*);

hal_mpc512x_spi_slave ***hal_mpc512x_spi_slave_init**(*int psc*, *cyg_uint32 tfr_size*, *cyg_uint32 flags*, *hal_mpc512x_spi_slave_rx *rx_callback*, *void *user_data*);

int **hal_mpc512x_spi_slave_tx**(*hal_mpc512x_spi_slave *slave*, *cyg_uint8 *buf*);

Introduction

SPI slave support is provided by a module in the MPC512X variant HAL. It comprises a data structure, two functions and the prototype of a function that must be supplied by the user. All of these may be defined by including the `cyg/hal/mpc512x_spislave.h` header file.

Configuration

Any PSC that is to be used as an SPI slave must be configured into `SPISLAVE` mode. If this is done then the following configuration options become available:

CYGHWR_HAL_POWERPC_MPC512X_PSCX_SPI_SLAVE_MAX

> This option defines the maximum transfer size that any SPI slave device can handle. This is used to define the size of the buffers allocated to any SPI slave device. Individual SPI slaves may define FIFO sizes less than or equal to this value.

CYGHWR_HAL_POWERPC_MPC512X_PSC*N*_SPI_SLAVE_MAX

> This option defines the maximum transfer size that the SPI slave device on PSC*N* can handle. This is used to control the size of the FIFOs allocated to this device. At initialization an application can choose an actual transfer size equal to or less than this value.

CYGHWR_HAL_POWERPC_MPC512X_PSC*N*_SPI_SLAVE_INTR_PRI

> This option defines interrupt priority for the SPI slave on PSC*N*. The priority may range from 0 to 7. The default value of 8 selects the hardware default level.

Usage

The SPI protocol is highly asymmetric. The timing of when a transfer starts, the frequency at which it is clocked and any gap between individual bytes is under the control of the master. The slave device has no mechanism for influencing any of this. For example, with an 8MHz clock the slave would have to supply one byte every microsecond, and the gap between bytes is only 125ns. In a processor that has many other demands on its time, this kind of latency is hard to guarantee. Therefore, the SPI slave support provided makes use of the hardware characteristics to avoid any software being involved in the main part of an SPI transfer.

The SPI slave support makes use of the hardware FIFOs associated with each PSC. By pre-loading the transmit FIFO with data and keeping the transfer size to less than the FIFO size, the entire transfer can occur without software involvement. Software only needs to get involved at the end of the transfer, to empty the data sent by the master from the receive FIFO, and to load data for the next transfer into the transmit FIFO. To make this work, all transfers must be less than the size configured for the FIFOs, and all transfers must be of the same pre-defined size.

An SPI slave PSC is initialized by calling `hal_mpc512x_spi_slave_init()`. The *psc* parameter identifies the PSC to be initialized, which must have been configured in `SPISLAVE` mode. The *tfr_size* parameter defines the transfer size to be used, and must be less than or equal to this PSC's maximum transfer size. *rx_callback* is a pointer to a function that will be called when data is available and *user_data* is a user-supplied value. The *flags* parameter contains configuration flags, at present these can be used to set the SPI CPHA and CPOL parameters used to control data sampling and clocking.

On return the init function will return a pointer to a hal_mpc512x_spi_slave structure which is used in the other API calls. If the initialization fails for some reason, NULL is returned. After a successful initialization, the SPI slave is ready for the master to initiate a transfer.

When the master performs a transfer, bytes will be clocked in to the receive FIFO and bytes will be clocked out of the transmit FIFO. Initialization will have pre-primed the transmit FIFO with *tfr_size* zeroes, so on the first transfer the master can only send data to the slave. Once the transfer is complete the PSC will raise an interrupt and call the *rx_callback()* function. This will be provided with a pointer to a buffer containing the received data. The *user_data* may be accessed via the *slave* pointer. This function is called from DSR mode, so should not call any functions that potentially cause a context switch; generally it should use a semaphore or other synchronization object to wake up a thread to perform any further processing. The receive buffer will be overwritten by the next transfer, so should be copied out to private memory if it needs to be preserved.

To supply data to be sent during the next transfer, the user should call `hal_mpc512x_spi_slave_tx()`. The *buf* argument points to *tfr_size* bytes to be sent. There is sufficient buffering for a single pending transfer, in addition to the contents of the FIFO, so this function may be called before the previous transfer completes. On completion of the transfer, the transmit FIFO will be filled from the pending buffer. If the pending buffer is already full when this function is called the thread will be made to wait until the buffer is empty.

The following code (with some irrelevant details omitted) gives the basic outline of how a dedicated SPI slave might be structured:

```
// Omitted: standard headers

#include <cyg/hal/mpc512x_spislave.h>

#define PSC             3
#define TFR_SIZE        32

cyg_sem_t sem;

cyg_uint8 tx_buf[CYGHWR_HAL_POWERPC_MPC512X_PSCX_SPI_SLAVE_MAX];
cyg_uint8 rx_buf[CYGHWR_HAL_POWERPC_MPC512X_PSCX_SPI_SLAVE_MAX];

// SPI slave callback
void rx_callback( hal_mpc512x_spi_slave *slave, cyg_uint8 *buf )
{
    // Copy received data to private buffer
    memcpy( rx_buf, buf, TFR_SIZE );

    // Wake up thread
    cyg_semaphore_post( &sem );
}

// Entry function for SPI slave handling thread
// Omitted: thread creation
void spi_slave(cyg_addrword_t arg)
{
    hal_mpc512x_spi_slave *slave;

    // Initialize semaphore
    cyg_semaphore_init( &sem, 0 );

    // Initialize SPI slave on the PSC
    slave = hal_mpc512x_spi_slave_init( PSC,
                                        TFR_SIZE,
                                        HAL_SPI_SLAVE_CPHA0|HAL_SPI_SLAVE_CPOL0,
                                        &rx_callback,
                                        NULL );

    // Omitted: raise error on slave == NULL

    // Loop forever handling transfers
    while( 1 )
    {
        // Wait for a transfer to complete
        cyg_semaphore_wait( &sem );

        // Omitted: deal with received data.

        // Omitted: create transmit data for next transfer.

        // Queue up for next transfer
        hal_mpc512x_spi_slave_tx( slave, tx_buf );
    }
}
```

A test program in the ADS512101 board HAL (the only board on which this device could be tested) demonstrates the use of the SPI slave support in a real program.

Part LXXXV. SH Architecture

Table of Contents

Chapter 354. Renesas SDK7780 Development Board Support

Name

eCos Support for the Renesas SDK7780 Development Board — Overview

Description

The Renesas SDK7780 Development Board (henceforth just "SDK7780") has an SH7780 processor, 128MB of external SDRAM, 128MB of external flash memory, an SMSC LAN91C111 ethernet controller with integrated PHY, two 9-pin SCIF serial interfaces plus required support chips for all the on-chip peripherals.

For typical eCos development, a RedBoot image is programmed into the flash memory, and the board will boot this image from reset. RedBoot provides gdb stub functionality so it is then possible to download and debug stand-alone and eCos applications via the gdb debugger. This can happen over either a serial line or over ethernet.

Supported Hardware

The SDK7780 contains two banks of two Spansion S29GL256M flash parts. The banks may be swapped between CS0 and CS1 by SW4-1. Bank A contains ETS and the TFTP bootloader and is left alone. Bank B contains RedBoot which is usually installed by the bootloader. Flipping SW4-1 causes the system to boot from Bank B. Only Bank B, accessed via CS0, is managed by eCos. Bank A is left untouched to preserve ETS and allow RedBoot to be reprogrammed. In a typical setup, the first two flash blocks (256K bytes) are reserved for use for the ROM RedBoot image. The topmost block is used to manage the flash and hold RedBoot fconfig values. The remaining blocks between 0x80040000 and 0x83FDFFFF can be used by application code.

The board is fitted with a JTAG socket allowing use of the E10A JTAG interface to perform hardware debugging. At present there is no GDB support for this device, howver it may be used from HEW to debug application in ROM or loaded via serial or ethernet.

There is a serial driver `CYGPKG_DEVS_SERIAL_SH_SCIF` which supports the two on-chip serial devices. Either of these devices can be used by RedBoot for communication with the host. If a device is needed by the application, either directly or via the serial driver, then it cannot also be used for RedBoot communication. Either the alternative serial port, or another communication channel such as ethernet should be used instead. The serial driver package is loaded automatically when configuring for the sdk7780 target.

There is an ethernet driver `CYGPKG_DEVS_ETH_SH_SDK7780` which provided configuration parameters for the onboard ethernet device. The device itself is supported by the `CYGPKG_DEVS_ETH_SMSC_LAN91CXX` package. These drivers are also loaded automatically when configuring for the sdk7780 target, although not activated until generic ethernet package support is also added.

There is a watchdog driver `CYGPKG_DEVICES_WATCHDOG_SH_SH4A`. This driver is also loaded automatically when configuring for the SDK7780 target.

There is a driver for the on-chip real-time clock (RTC) at `CYGPKG_DEVICES_WALLCLOCK_SH3`. This driver is also loaded automatically when configuring for the SDK7780 target.

eCos manages the on-chip interrupt controller. Timer 0 is used to implement the eCos system clock, and timer 1 is used to implement a microsecond delay function. Timer 2 is unused and left for the application. Other on-chip devices (FEMI, EMI, LMI, INTC, TMU, CAC, UBC, CPG) are initialized only as far as is necessary for eCos to run. Other devices (eg DMAC, MMCIF, HAC, SSI, FLCTL etc) are not touched.

Tools

The SDK7780 port is intended to work with GNU tools configured for an sh-elf target. The original port was done using sh-elf-gcc version 3.4.4, sh-elf-gdb version 6.3, and binutils version 2.16.

Name

Setup — Preparing the SDK7780 board for eCos Development

Overview

In a typical development environment, the SDK7780 board boots from flash into the RedBoot ROM monitor. eCos applications are configured for RAM startup and then downloaded and run on the board via the debugger sh-elf-gdb. Preparing the board therefore usually involves programming a suitable RedBoot image into flash memory.

The following RedBoot configurations are supported:

Configuration	Description	Use	File
ROM	RedBoot running from the board's flash	`redboot_ROM.ecm`	`redboot_ROM.bin`
RAM	Used for upgrading ROM version	`redboot_RAM.ecm`	`redboot_RAM.bin`

For serial communications, all versions run with 8 bits, no parity, and 1 stop bit at 115200 baud. This baud rate can be changed via the configuration option `CYGNUM_HAL_SH_SH4_SCIF_BAUD_RATE` and rebuilding RedBoot. RedBoot also supports ethernet communication and flash management.

Initial Installation

Flash Installation

This process makes use of the ETS software programmed into flash bank A in order to write RedBoot into flash bank B. To do this you will need to set up a TFTP server on a machine that is accessible from the development board.

Before downloading and programming the RedBoot image, ETS and the TFTP Bootloader must be configured. Full details for doing this are available in the documentation that accompanies the board. The following steps should be read in conjunction with that.

The first step is to connect an RS232 cable between the upper of the SDK7780 serial ports and the host PC. Next start a terminal emulation application such as HyperTerminal or minicom on the host PC and set the serial communication parameters to 115200 baud, 8 data bits, no parity, 1 stop bit (8N1) and no flow control (handshaking). Also connect an ethernet cable that is on the same network as the TFTP server.

Apply power to the SDK7780 and press the power button. The board should boot automatically into ETS and after an initial report of the board configuration should display the `ETS MAIN MENU`. Select option 2 `Boot Configuration Menu` and then option 1 `Change Configuration`. Now select option 1 `TFTP` and accept the default 5s delay. Check that SW4-4&5 are off and answer Y to the remaining questions. Once the process is finished, reset the board to restart.

On restart the board will boot into the TFTP Boot Loader after a 5s delay. It will immediately drop into a configuration dialog. Decide whether to use DHCP or static IP. Static IP is recommended for which you will need to enter this board's IP address and that of the TFTP server. For the kernel filename enter `redboot.bin`; answer Y to download to flash bank B; enter B for the file format; accept the default load address; answer N for the disk image and command line; enter F to select the flash boot option and Y for SW4-4 state. The output should look something like this:

```
Press 'SPACE Bar' key or wait for 5 seconds to enter TFTP Boot Loader.

Press any other key to start ETS.


Starting TFTP Boot Loader...
```

```
Renesas SDK7780 (Little Endian Mode)
SH TFTP Bootloader Version 2.30
   Copyright (C) 2000 Free Software Foundation, Inc.
   Copyright (C) 2004 Renesas Technology Europe Limited

This software comes with ABSOLUTELY NO WARRANTY.
This is free software, and you are welcome to redistribute it under
under the terms of the GNU Lesser General Public License version 2.1 or later.

Platform MAC Address - 0000:87D6:3EB2

No configuration found in flash, entering setup...

Setup TFTP Bootloader Configuration:

At any prompt press 'Enter' to leave unchanged or 'Esc' key to quit.

Use DHCP to setup network?(Default=Static IP)[Y/N]: n

Enter board IP address [NNN.NNN.NNN.NNN]         : 10.0.3.2
Enter server IP address [NNN.NNN.NNN.NNN]        : 10.0.1.2
Enter kernel (user program) filename             : redboot.bin
Download to flash Bank B?(Default=Bank A) [Y/N]  : y
Download file format Binary/S-record? [B/S]      : b
Binary load address [default=H'88210000]         :
Load a disk image? [Y/N]                         : n
Setup kernel command line? [Y/N]                 : n
Boot option: TFTP, Flash, Don't Boot(TFTP Bootloader Main Menu)? [T/F/N]: f

Is flash Bank A write protect switch SW4-4 = 'OFF'? [Y/N] : y

Updating Boot Flash - Do not switch-off, reset or disconnect until complete.
Please wait, saving the new Boot configuration...

Successfully saved new TFTP Bootloader configuration.



TFTP Bootloader Main Menu
-------------------------
  1. Boot from network
  2. Boot from flash
  3. Boot from FAT32 format MMC card
  4. Load program into flash
  5. Display configuration
  6. Change configuration
Command:>
```

The bootloader will now drop into its main menu. We are now ready to download and program the RedBoot image. Locate the file `redboot_ROM.bin` in the release `loaders` directory and copy it to the TFTP server's directory naming it `redboot.bin`. At the TFTP bootloader select option 4 which should it to fetch the `redboot.bin` file from the server. Answer Y for the SW4-5 state and wait for the flash to be written. The output should look something like this:

```
TFTP Bootloader Main Menu
-------------------------
  1. Boot from network
  2. Boot from flash
  3. Boot from FAT32 format MMC card
  4. Load program into flash
  5. Display configuration
  6. Change configuration
Command:>4
Board IP address : 10.0.3.2
Server IP address: 10.0.1.2
Downloading redboot.bin to flash bank B
Downloaded 175144 bytes, crc 471083609
```

```
Is flash Bank B write protect switch SW4-5 = 'OFF'? [Y/N] : y
Flash: S29GL256M (67108864 bytes)
Sectors: 1024 (131072 bytes each)
Writing to flash...
b 00000001 add A4000000

Toggle flash bank select switch to boot from bank B

TFTP Bootloader Main Menu
-------------------------
  1. Boot from network
  2. Boot from flash
  3. Boot from FAT32 format MMC card
  4. Load program into flash
  5. Display configuration
  6. Change configuration
Command:>
```

Toggle SW4-1 to select bank B for booting and reset the board. The following output should appear:

```
+**Warning** FLASH configuration checksum error or invalid key
Use 'fconfig -i' to [re]initialize database
Ethernet eth0: MAC address 00:00:d6:87:b2:3e
IP: 10.0.2.4/255.0.0.0, Gateway: 10.0.0.3
Default server: 0.0.0.0

RedBoot(tm) bootstrap and debug environment [ROM]
Non-certified release, version UNKNOWN - built 13:52:55, Jan 22 2007

Platform: Renesas SDK7780 (SH7780)
Copyright (C) 2000, 2001, 2002, 2003, 2004 Free Software Foundation, Inc.
Copyright (C) 2003, 2004, 2005, 2006, 2007 eCosCentric Limited

RAM: 0x88000000-0x8a000000, [0x8800b758-0x89fc1000] available
FLASH: 0x80000000-0x83ffffff, 512 x 0x20000 blocks
RedBoot>
```

At this stage the RedBoot flash management initialization has not yet happened so the warning about the configuration checksum error is expected. To perform this initialization use the **fis init -f** command:

```
RedBoot> fis init -f
About to initialize [format] FLASH image system - continue (y/n)? y
*** Initialize FLASH Image System
... Erase from 0x83fe0000-0x83ffffff: .
... Program from 0x89fd0000-0x89ff0000 to 0x83fe0000: .
RedBoot>
```

At the end, the block of flash at location 0x83FE0000 holds information about the various flash blocks, allowing other flash management operations to be performed.

Flash Configuration

The next step is to set up RedBoot's non-volatile configuration values:

```
RedBoot> fconfig -i
Initialize non-volatile configuration - continue (y/n)? y
Run script at boot: false
Use BOOTP for network configuration: true
Default server IP address:
Console baud rate: 115200
DNS server IP address:
Set eth0 network hardware address [MAC]: false
GDB connection port: 9000
Force console for special debug messages: false
Network debug at boot time: false
```

```
Update RedBoot non-volatile configuration - continue (y/n)? y
... Erase from 0x83fe0000-0x83ffffff: .
... Program from 0x89fd0000-0x89ff0000 to 0x83fe0000: .
RedBoot>
```

For most of these configuration variables, the default value is correct, although you may wish to provide a default server used for TFTP retrieval, and default DNS server. If there is no suitable BOOTP or DHCP service running on the local network then BOOTP should be disabled and, instead, RedBoot will prompt for a fixed IP address, netmask, and addresses for the local gateway and DNS server.

Once you have set appropriate RedBoot flash configuration values you may reset the board. When RedBoot issues its prompt, it is ready to accept connections from sh-elf-gdb, allowing applications to be downloaded and debugged. Connections can be made via either serial port, or by TCP to port 9000 (or an alternative port if manually set by the **fconfig** command).

Occasionally it may prove necessary to update the installed RedBoot image. This can be done simply by repeating the above process by toggling SW4-1 back to boot ETS and the TFTP bootloader. Alternatively, the existing RedBoot install can be used to load the RAM-resident version in which case the standard RAM RedBoot build can be used. See the RedBoot documentation for instruction on how to do this.

Rebuilding RedBoot

Should it prove necessary to rebuild a RedBoot binary, this is done most conveniently at the command line. Assuming your `PATH` and `ECOS_REPOSITORY` environment variables have been set correctly, the steps needed to rebuild the RAM version of RedBoot are:

```
$ mkdir redboot_ram
$ cd redboot_ram
$ ecosconfig new sdk7780 redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/sh/sdk7780/VERSION/misc/redboot_RAM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

At the end of the build the `install/bin` subdirectory should contain the file `redboot.bin`.

Rebuilding the ROM version involve basically the same process. The ROM version uses the file `redboot_ROM.ecm` and generates a file named `redboot.bin`. Make sure you don't mix up the different redboot.bin files; rename them to something more memorable such as `redboot_RAM.bin` and `redboot_ROM.bin`.

Name

Configuration — Platform-specific Configuration Options

Overview

The SDK7780 platform HAL package is loaded automatically when eCos is configured for an `sdk7780` target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The SDK7780 platform HAL package supports two separate startup types:

RAM This is the startup type which is normally used during application development. The board has RedBoot programmed into flash at location 0x80000000/0xA0000000 and boots from that location. sh-elf-gdb is then used to load a RAM startup application into memory and debug it. It is assumed that the hardware has already been initialized by RedBoot. By default the application will use the eCos virtual vectors mechanism to obtain certain services from RedBoot, including diagnostic output.

ROM This startup type can be used for finished applications which will be programmed into flash at location 0x80000000/0x-A0000000. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization.

RedBoot and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building RedBoot.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then serial port COM 0 will be claimed for HAL diagnostics.

Flash Driver

The SDK7780 board contains 128M bytes of Flash, in two banks of two Spansion S29GL256M parts in parallel. Each part consists of 512 blocks of 64KB each, but since two parts are used in parallel they are viewed as 512 blocks of 128KB each. Flash bank A is not touched by eCos, so effectively the board is viewed as having a single 64MB bank of flash.

These are AMD 29xxxxx compatible parts, and as such the `CYGPKG_DEVS_FLASH_AMD_AM29XXXXX_V2` package contains all the code necessary to support these parts. The SDK7780 platform HAL contains definitions that customize the driver to the SDK7780 board.

Ethernet Driver

The SDK7780 board contains an SMSC LAN91C111 ethernet device. The `CYGPKG_DEVS_ETH_SMSC_LAN91CXX` package contains all the code necessary to support this part and the `CYGPKG_DEVS_ETH_SH_SDK7780` package contains definitions that customize the driver to the SDK7780 board.

The ethernet will automatically auto-negotiate 10Mbps or 100Mbps operation with its link peer, as well as full duplex or half duplex mode.

The driver usually reads the MAC address (ESA) from the EEPROM connected to the MAC. Alternatively an address can be set in the CDL configuration in the component CYGSEM_DEVS_ETH_SH_SDK7780_ETH0_SET_ESA within the SDK7780 ethernet driver; or an address can be set in the Flash configuration of RedBoot using the **fconfig** command. If both are set, the Flash configuration is used in preference.

PCI Driver

The SDK7780 board is fitted with four 3V PCI slots, which are accessed via the PCIC funtional unit on the SH7780. eCos supports PCI devices inserted in these slots and if the PCI library is selected in the eCos configuration, a driver will usually call `cyg_pci_init()` which will automatically configure memory and I/O base address registers, as well as any interrupts the device requires.

The CPU is able to access the PCI memory space through the memory window at 0xb0000000, and the PCI I/O space through the memory window at 0xFE200000. PCI bus master devices may access system memory (usually for DMA), at windows starting at address 0x0 in both memory and I/O PCI spaces.

Note that the PCI INTD line is on a pin shared with the CTS line for SCIF0. If this interrupt is to be used, SW4-8 must be set to OFF.

SCIF Serial Driver

The SDK7780 board uses the SH7780 internal SCIF serial support Two serial ports are available: SCIF0 which is mapped to virtual vector channel 0 in the HAL diagnostic driver or "`/dev/ser0`" in the interrupt-driven driver; and SCIF1 which is mapped to virtual vector channel 1 and "`/dev/ser1`". Only SCIF0 supports modem control signals such as those used for hardware flow control, and SW4-8 must be set to ON for CTS to be routed to the DB9 connector.

System Clock

By default, the system clock interrupts once every 10ms, corresponding to a 100Hz clock. This can be changed by the configuration option `CYGNUM_HAL_RTC_DENOMINATOR` which corresponds to the clock frequency. Other clock-related settings are recalculated automatically if the denominator is changed.

RTC Driver

The SDK7780 board uses the SH7780 internal RTC support. The `CYGPKG_DEVICES_WALLCLOCK_SH3` package contains all the code necessary to support this device. This driver is not active until the generic wallclock device support package, `CYGPKG_IO_WALLCLOCK`, is included in the configuration.

Watchdog Driver

The SDK7780 board uses the SH7780 internal watchdog support. The `CYGPKG_DEVICES_WATCHDOG_SH_SH4A` package contains all the code necessary to support this device. Within that package the `CYGNUM_DEVS_WATCHDOG_ARM_AT91RM9200_DESIRED_TIMEOUT_MS` configuration option controls the watchdog timeout, and by default will force a reset of the board upon timeout. This driver is not active until the generic watchdog device support package, `CYGPKG_IO_WATCHDOG`, is included in the configuration.

Compiler Flags

The platform HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos. There are two flags specific to this port:

`-m4`
: The sh-elf-gcc compiler supports many variants of the SH architecture, from the SH2 onwards. A `-m` option should be used to select the specific variant in use, and with current tools `-m4` is the correct option for the SH7780.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the SDK7780 hardware, and should be read in conjunction with that specification. The SDK7780 platform HAL package complements the SH architectural HAL and the SH4 variant HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset the HAL will initialize or reinitialize most of the on-chip peripherals. There is an exception for RAM startup applications which depend on a ROM monitor for certain services.

For ROM startup, the HAL will perform additional initialization, setting up the external SDRAM and programming the various internal registers including clocks, EMI and LMI. The values used for most of these registers are assigned fixed values from a table in the header `cyg/hal/platform.inc`.

Linker Scripts and Memory Maps

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

off-chip Flash

This is located at address 0x00000000 of the physical memory space and is therefore accessible in the P1 region at location 0x80000000. An uncached shadow of this memory is available in the P2 region at 0xA0000000. The contents of the flash are organized as described earlier.

external SDRAM

This is located at address 0x08000000 of the physical memory space and is therefore accessable in the P1 region at location 0x88000000. An uncached shadow of this memory is available in the P2 region at 0xA8000000. The first 512 bytes are used for hardware exception vectors. The next 256 bytes are normally used for the eCos virtual vectors, allowing RAM-based applications to use services provided by the ROM monitor. For ROM startup, all remaining SDRAM is available. For RAM startup, available SDRAM starts at location 0x88020000, with the bottom 128Kbytes reserved for use by RedBoot.

on-chip peripherals

These are accessible via the P4 region at location 0xE0000000 onwards. The on-chip PCI controller has apertures into the PCI memory bus at 0xB0000000 and 0xFD000000 in the P2 region, and access to to the PCI IO space at 0xFE200000. Other base addresses of on-chip peripherals can be found in the SH7780 Hardware Manual.

off-chip peripherals

All off-chip peripherals used by eCos are accessed via on-chip bus controllers such as LMI or PCI. All others are left untouched.

Clock Support

The platform HAL provides configuration options for the eCos system clock. This always uses the hardware timer 0, which should not be used directly by application code. Timer 1 is used to implement a microsecond resolution busy delay service. Timers 2 to 5 are not used by eCos so application code is free to manipulate these as required. The actual HAL macros for managing the clock are provided by the SH architecture processor HAL.

There is a software model of the structure of the SH family clock supply subsystem which performs the correct calculations to yield not only the inputs for the CPU clock but also the peripheral clocks fed to the serial device, memory controllers and other devices. The values for the master crystal, the PLL multipliers and various dividers are supplied by the platform HAL. Some care must be taken in defining these since wrong values will cause the timers and the SCIF baud rate to be miscalculated (resulting visibly in garbage on the serial output).

The SH7780 extends the SH family clock model by providing a CLOCKGEN subsystem allowing the hardware clock frequency to be controlled. The CLOCKGENA.PLL1CR register is the primary means to do this, and is initialised by switches 1, 2, 3 and 7 on DIP switch block MD_SW. The delivery default settings for these switches is to select clock mode 0 giving a 400MHz CPU clock, 100MHz local bus and 50MHz peripheral clock.

If the DIP switches are changed from the default then the values of `CYGHWR_HAL_SH_OOC_DIVIDER_IFC`, `CYGHWR_HAL_SH_OOC_DIVIDER_BFC` and `CYGHWR_HAL_SH_OOC_DIVIDER_PFC` must be changed to match.

Other Issues

The SDK7780 platform HAL does not affect the implementation of other parts of the eCos HAL specification. The SH4 variant HAL, and the SH architectural HAL documentation should be consulted for further details.

It should be noted that the floating point support in the SH HAL has a caveat that, if the FPSCR register is changed, it may get reverted at a later stage by certain operations performed by the GCC compiler. This behaviour is intentional as the alternative would be to update the GCC compiler's internal state about the FPSCR at every context switch which would be expensive for a feature that is unlikely to be used frequently. If the FPSCR is to be changed by the application, the developer should call the function `__set_fpscr(int)`, passing it the new FPSCR value.

Chapter 355. SuperH SH4-202 MicroDev Board Support

Name

eCos Support for the SuperH SH4-202 MicroDev Board — Overview

Description

The SuperH SH4-202 MicroDev board (henceforth just "MicroDev") has an SH4-202 processor, 64MB of external SDRAM, 32MB of external flash memory, an SMSC LAN91C111 ethernet controller and connectors plus required support chips for all the on-chip peripherals.

For typical eCos development, a RedBoot image is programmed into the flash memory, and the board will boot this image from reset. RedBoot provides gdb stub functionality so it is then possible to download and debug stand-alone and eCos applications via the gdb debugger. This can happen over either a serial line or over ethernet.

Supported Hardware

The flash memory consists of 128 blocks of 256k bytes each. In a typical setup, the first flash block is used for the ROM RedBoot image and the second is used to store a version of RedBoot that can run out of RAM. The topmost two blocks are used to manage the flash and hold RedBoot fconfig values. The remaining 124 blocks between 0xA0080000 and 0xA1F7FFFF can be used by application code.

The board is fitted with a PLCC socket suitable for an EEPROM (or PROM) such as the 1Mbit ST M29WO10B. This is enabled by toggling two DIP switches, after which the EEPROM is mapped into the same address as the flash memory. Therefore, the flash is not accessible if booting from the EEPROM.

There is a serial driver `CYGPKG_DEVS_SERIAL_SH_SCIF` which supports the on-chip serial device. This device can be used by RedBoot for communication with the host. If this device is needed by the application, either directly or via the serial driver, then it cannot also be used for RedBoot communication. Another communication channel such as ethernet should be used instead. The serial driver package is loaded automatically when configuring for the MicroDev target.

There is an ethernet driver `CYGPKG_DEVS_ETH_SH_MICRODEV` for the on-chip ethernet device. This driver is also loaded automatically when configuring for the MicroDev target.

eCos manages the on-chip interrupt controller. Timer 0 is used to implement the eCos system clock, and timer 1 is used to implement a microsecond delay function. Timer 2 is unused and left for the application. Other on-chip devices (FEMI, EMI, INTC, TMU, CAC, UBC) are initialized only as far as is necessary for eCos to run. Other devices (eg RTC, DMAC, etc) are not touched.

Tools

The MicroDev port is intended to work with GNU tools configured for an sh-elf target. The original port was done using sh-elf-gcc version 3.2.1, sh-elf-gdb version 5.3, and binutils version 2.13.1.

Name

Setup — Preparing the MicroDev board for eCos Development

Overview

In a typical development environment, the MicroDev board boots from flash into the RedBoot ROM monitor. eCos applications are configured for RAM startup and then downloaded and run on the board via the debugger sh-elf-gdb. Preparing the board therefore usually involves programming a suitable RedBoot image into flash memory. Alternatively RedBoot may be programmed into a PLCC EEPROM and inserted into socket U21, although in that case, the flash memory is not accessible.

The following RedBoot configurations are supported:

Configuration	Description	Use	File
ROM	RedBoot running from the board's flash	`redboot_ROM.ecm`	`redboot_ROM.bin`
EEPROM	RedBoot running from the board's socketed EEPROM	`redboot_EEPROM.ecm`	`redboot_EEPROM.bin`
RAM	Used for upgrading ROM version	`redboot_RAM.ecm`	`redboot_RAM.bin`

For serial communications, all versions run with 8 bits, no parity, and 1 stop bit at 38400 baud. This baud rate can be changed via the configuration option `CYGNUM_HAL_SH_SH4_SCIF_BAUD_RATE` and rebuilding RedBoot. RedBoot also supports ethernet communication and flash management.

Initial Installation

Flash Installation

This process assumes that the board is connected to a SuperH Micro Probe. The Micro Probe should be set up as described in Appendix A of the "SH4 Development Tools User Guide". You should also have access to the SuperH development tools since it is necessary to use the version of GDB that comes with those tools to access the Micro Probe, **sh-elf-gdb** will not work.

Programming the RedBoot ROM monitor into flash memory requires an application that can manage flash blocks. RedBoot itself has this capability. Rather than have a separate application that is used only for flash management during the initial installation, a special RAM-resident version of RedBoot is loaded into memory and run. This version can then be used to load the normal flash-resident version of RedBoot and program it into the flash.

The first step is to connect an RS232 null modem cable between the MicroDev serial port and the host PC. Next start a terminal emulation application such as HyperTerminal or minicom on the host PC and set the serial communication parameters to 38400 baud, 8 data bits, no parity, 1 stop bit (8N1) and no flow control (handshaking).

Now run the **sh4gdb** command, giving it the name of the RAM redboot ELF file, connect to the Micro Probe, load the executable and run it. The entire session should look like this:

```
$ sh4gdb redboot_RAM.elf
GNU gdb 5.2.1
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "--host=i686-pc-linux-gnu --target=sh-superh-elf"...
(gdb) sh4si superh
```

```
The target is assumed to be little endian
The target architecture is assumed to be sh4
0xa0000000 in ?? ()
(gdb) load
Loading section .vectors, size 0x9e0 lma 0x88010000
Loading section .text, size 0x1ab20 lma 0x880109e0
Loading section .rodata, size 0x3e6c lma 0x8802b500
Loading section .data, size 0xf30 lma 0x8802f370
Start address 0x88010000, load size 131740
Transfer rate: 351306 bits/sec, 433 bytes/write.
(gdb) cont
Continuing.
```

The required `redboot_RAM.elf` file is normally supplied with the eCos release in the `loaders` directory. If it needs to be rebuilt then instructions for this are supplied below.

If this sequence fails in any way then check the setup and connections of the Micro Probe. It if is successful then you should see the following printed out on the serial line:

```
+FLASH configuration checksum error or invalid key
... waiting for BOOTP information
Ethernet eth0: MAC address 00:08:ee:00:0b:37
Can't get BOOTP info for device!

RedBoot(tm) bootstrap and debug environment [RAM]
Non-certified release, version UNKNOWN - built 14:28:55, Sep  8 2003

Copyright (C) 2000, 2001, 2002, Free Software Foundation, Inc.

RAM: 0x88000000-0x8c000000, 0x8812cca0-0x8bfb1000 available
FLASH: 0xa0000000 - 0xa2000000, 128 blocks of 0x00040000 bytes each.
RedBoot>
```

If the ethernet cable is not plugged in there may be a fairly long wait after the "... waiting for BOOTP information" message. At this stage the RedBoot flash management initialization has not yet happened so the warning about the configuration checksum error is expected. To perform this initialization use the **fis init -f** command:

```
RedBoot> fis init -f
About to initialize [format] FLASH image system - continue (y/n)? y
*** Initialize FLASH Image System
... Unlock from 0xa1fc0000-0xa2000000: .
... Erase from 0xa1fc0000-0xa2000000: .
... Program from 0x8bfbf000-0x8bfff000 at 0xa1fc0000: .
... Lock from 0xa1fc0000-0xa2000000: .
RedBoot>
```

At the end, the block of flash at location 0xA1FC0000 holds information about the various flash blocks, allowing other flash management operations to be performed. The next step is to set up RedBoot's non-volatile configuration values:

```
RedBoot> fconfig -i
Initialize non-volatile configuration - continue (y/n)? y
Run script at boot: false
Use BOOTP for network configuration: true
Console baud rate: 38400
DNS server IP address:
Set eth0 network hardware address [MAC]: false
GDB connection port: 9000
Force console for special debug messages: false
Network debug at boot time: false
Update RedBoot non-volatile configuration - continue (y/n)? y
... Unlock from 0xa1f80000-0xa1f81000: .
... Erase from 0xa1f80000-0xa1f81000: .
... Program from 0x8bfb2000-0x8bfb3000 at 0xa1f80000: .
... Lock from 0xa1f80000-0xa1f81000: .
RedBoot>
```

For most of these configuration variables, the default value is correct. If there is no suitable BOOTP service running on the local network then BOOTP should be disabled and, instead, RedBoot will prompt for a fixed IP address, netmask, and addresses for the local gateway and DNS server.

It is now possible to load the flash-resident version of RedBoot. Because of the way that flash chips work, it is better to first load it into RAM and then program it into flash.

```
RedBoot> load -r -m xmodem -b %{freememlo}
```

The file `redboot_ROM.bin` should now be uploaded using the terminal emulator. The file is a raw binary and should be transferred using the X-modem protocol.

```
Raw file loaded 0x8812d000-0x8814e32f, assumed entry at 0x8812d000
xyzModem - CRC mode, 1064(SOH)/0(STX)/0(CAN) packets, 2 retries
RedBoot>
```

Once RedBoot has been loaded into RAM it can be programmed into flash:

```
RedBoot> fis create RedBoot -b %{freememlo}
An image named 'RedBoot' exists - continue (y/n)? y
... Erase from 0xa0000000-0xa0040000: .
... Program from 0x8812d000-0x8816d000 at 0xa0000000: .
... Unlock from 0xa1fc0000-0xa2000000: .
... Erase from 0xa1fc0000-0xa2000000: .
... Program from 0x8bfbf000-0x8bfff000 at 0xa1fc0000: .
... Lock from 0xa1fc0000-0xa2000000: .
RedBoot>
```

The flash-resident version of RedBoot has now been programmed at location 0xA0000000, and the flash info block at 0xA1FC0000 has been updated. The initial setup is now complete. Power off the Micro Probe and reset the MicroDev board using S6. You should see the following:

```
+... waiting for BOOTP information
Ethernet eth0: MAC address 00:08:ee:00:0b:37
Can't get BOOTP info for device!

RedBoot(tm) bootstrap and debug environment [ROM]
Non-certified release, version UNKNOWN - built 14:22:57, Sep  8 2003

Copyright (C) 2000, 2001, 2002, Free Software Foundation, Inc.

RAM: 0x88000000-0x8c000000, 0x8800db98-0x8bfb1000 available
FLASH: 0xa0000000 - 0xa2000000, 128 blocks of 0x00040000 bytes each.
RedBoot>
```

When RedBoot issues its prompt, it is also ready to accept connections from sh-elf-gdb, allowing applications to be downloaded and debugged.

Occasionally it may prove necessary to update the installed RedBoot image. This can be done simply by repeating the above process, using the Micro Probe. Alternatively, the existing RedBoot install can be used to load the RAM-resident version. You can even install the RAM resident RedBoot in the "RedBoot[backup]" flash region. See the RedBoot documentation for instruction on how to do this.

EEPROM Installation

The board has a 32-pin PLCC socket suitable for an EEPROM, silk screened U21. To use RedBoot running from EEPROM, you must first program the file `redboot_EEPROM.bin` (normally supplied with the eCos release in the `loaders` directory) into the EEPROM using an appropriate programmer. No byte swapping is required. If RedBoot needs to be rebuilt, then instructions for this are supplied below, and the import file `redboot_EEPROM.ecm` should be used.

To configure the board to boot from the EEPROM instead of flash, you must power off the board and change the following DIP switch settings, which may both be found on DIP switch 2 (silk screened S2): switch 2 (silk screened FEMI SIZ1) should be set to

ON, which will change the access width for FEMI area 0 from 32-bit to 8-bit; switch 6 (silk screened FPGA SW3) should be set to OFF to configure the FPGA to map memory accesses for FEMI area 0 to point at the EEPROM instead of flash. In this mode, it is no longer possible to access flash memory as the EEPROM is mapped into the same area in the address space.

Note that it is usually preferable to boot from flash instead of EEPROM as flash is accessed 32-bits at a time, whereas the EEPROM is accessed 8-bits at a time, which therefore affects performance as this requires 4 times as many read cycles.

Rebuilding RedBoot

Should it prove necessary to rebuild a RedBoot binary, this is done most conveniently at the command line. The steps needed to rebuild the RAM version of RedBoot are:

```
$ mkdir redboot_ram
$ cd redboot_ram
$ ecosconfig new sh4_202_md redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/sh/sh4_202_md/v2_0_2/misc/redboot_RAM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

At the end of the build the `install/bin` subdirectory should contain the file `redboot.bin`.

Rebuilding the ROM versions involves basically the same process. The ROM version uses the file `redboot_ROM.ecm` and generates a file `redboot.bin`. Make sure you don't mix up the different redboot.bin files; rename them to something more memorable such as `redboot_RAM.bin` and `redboot_ROM.bin`.

Name

Configuration — Platform-specific Configuration Options

Overview

The MicroDev platform HAL package is loaded automatically when eCos is configured for an sh4_202_md target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The MicroDev platform HAL package supports two separate startup types:

RAM This is the startup type which is normally used during application development. The board has RedBoot programmed into flash at location 0xA0000000 and boots from that location. sh-elf-gdb is then used to load a RAM startup application into memory and debug it. It is assumed that the hardware has already been initialized by RedBoot. By default the application will use the eCos virtual vectors mechanism to obtain certain services from RedBoot, including diagnostic output.

ROM This startup type can be used for finished applications which will be programmed into flash at location 0xA0000000. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization.

RedBoot and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option CYGSEM_HAL_ROM_MONITOR should be set. Typically this option is set only when building RedBoot.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option CYGSEM_HAL_USE_ROM_MONITOR should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then the serial port will be claimed for HAL diagnostics.

Flash Driver

The MicroDev board contains 32Mb of Intel StrataFlash, specifically, two E28F128 parts in parallel. The CYGPKG_DEVS_FLASH_STRATA package contains all the code necessary to support these parts and the CYGPKG_DEVS_FLASH_SH_MICRODEV package contains definitions that customize the driver to the MicroDev board.

Note that if booting from EEPROM instead of flash, the flash driver will not be able to detect or use the flash parts.

Ethernet Driver

The MicroDev board contains an SMSC LAN91C111 ethernet device. The CYGPKG_DEVS_ETH_SMSC_LAN91CXX package contains all the code necessary to support this part and the CYGPKG_DEVS_ETH_SH_MICRODEV package contains definitions that customize the driver to the MicroDev board.

System Clock

By default, the system clock interrupts once every 10ms, corresponding to a 100Hz clock. This can be changed by the configuration option CYGNUM_HAL_RTC_DENOMINATOR which corresponds to the clock frequency. Other clock-related settings are recalculated automatically if the denominator is changed.

Compiler Flags

The platform HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos. There are two flags specific to this port:

`-m4`
: The sh-elf-gcc compiler supports many variants of the SH architecture, from the SH2 onwards. A `-m` option should be used to select the specific variant in use, and with current tools `-m4` is the correct option for the SH4-202.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the MicroDev hardware, and should be read in conjunction with that specification. The MicroDev platform HAL package complements the SH architectural HAL and the SH4 variant HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset the HAL will initialize or reinitialize most of the on-chip peripherals. There is an exception for RAM startup applications which depend on a ROM monitor for certain services.

For ROM startup, the HAL will perform additional initialization, setting up the external DRAM and programming the various internal registers. The values used for most of these registers are assigned fixed values from a table in the header `cyg/hal/platform.inc`.

Linker Scripts and Memory Maps

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

off-chip Flash

This is located at address 0x00000000 of the physical memory space and is therefore accessible in the P1 region at location 0x80000000. An uncached shadow of this memory is available in the P2 region at 0xA0000000. The contents of the flash are organized as described earlier.

off-chip EEPROM

If selected by the DIP switches, this occupies the same addresses as the off-chip flash, and the flash is no longer visible.

external SDRAM

This is located at address 0x08000000 of the physical memory space and is therefore accessable in the P1 region at location 0x88000000. An uncached shadow of this memory is available in the P2 region at 0xA8000000. The first 256 bytes are used for hardware exception vectors. The next 256 bytes are normally used for the eCos virtual vectors, allowing RAM-based applications to use services provided by the ROM monitor. For ROM startup, all remaining SDRAM is available. For RAM startup, available SDRAM starts at location 0x80100000, with the bottom 1MB reserved for use by RedBoot.

on-chip peripherals

These are accessible via the P4 region at location 0xE0000000 onwards.

off-chip peripherals

The ethernet device is located at 0xA7500000. The FPGA interrupt controller is located at 0x06110000. These are the only off-chip peripherals accessed by eCos. All others are left untouched.

Clock Support

The platform HAL provides configuration options for the eCos system clock. This always uses the hardware timer 0, which should not be used directly by application code. Timer 1 is used to implement a microsecond resolution busy delay service. Timer 2 is not used by eCos so application code is free to manipulate this as required. The actual HAL macros for managing the clock are provided by the SH architecture processor HAL.

There is a software model of the structure of the SH family clock supply subsystem which performs the correct calculations to yield not only the inputs for the CPU clock but also the peripheral clocks fed to the serial device, memory controllers and other devices.

The values for the master crystal, the PLL multipliers and various dividers are supplied by the platform HAL. Some care must be taken in defining these since wrong values will cause the timers and the SCIF baud rate to be miscalculated. If the OSCAR chip switches are changed from the default then the value of `CYGHWR_HAL_SH_OOC_XTAL` must be changed to match.

Other Issues

The MicroDev platform HAL does not affect the implementation of other parts of the eCos HAL specification. The SH4 variant HAL, and the SH architectural HAL documentation should be consulted for further details.

It should be noted that the floating point support in the SH HAL has a caveat that, if the FPSCR register is changed, it may get reverted at a later stage by certain operations performed by the GCC compiler. This behaviour is intentional as the alternative would be to update the GCC compiler's internal state about the FPSCR at every context switch which would be expensive for a feature that is unlikely to be used frequently. If the FPSCR is to be changed by the application, the developer should call the function `__set_fpscr(int)`, passing it the new FPSCR value.

Chapter 356. STMicroelectronics ST40 Evaluation Board Support

Name

eCos Support for the STMicroelectronics ST40 Evaluation Board — Overview

Description

The STMicroelectronics ST40 Evaluation Board (henceforth just "ST40EB") has an ST40RA166XH processor, 32MB of external SDRAM, 4MB of external flash memory, an STE10/100A ethernet controller with integrated PHY (Intel i21143/DEC Tulip compatible), two 9-pin SCIF serial interfaces, LCD display panel and connectors plus required support chips for all the on-chip peripherals.

For typical eCos development, a RedBoot image is programmed into the flash memory, and the board will boot this image from reset. RedBoot provides gdb stub functionality so it is then possible to download and debug stand-alone and eCos applications via the gdb debugger. This can happen over either a serial line or over ethernet.

Supported Hardware

The flash memory consists of two ST M29W160DB parts in parallel, giving a total of 4M bytes of external Flash. In a typical setup, the first four flash blocks (256K bytes) are reserved for use for the ROM RedBoot image and the subsequent two are used to store a version of RedBoot that can run out of RAM. The topmost block is used to manage the flash and hold RedBoot fconfig values. The remaining blocks between 0xA0080000 and 0xA03DFFFF can be used by application code.

The board is fitted with a HUDI socket allowing use of the ST40-Connect/SH JTAG interface to perform hardware debugging. To use this, a set of GDB macro files are provided with a GNU GDB toolset provided by ST, and to connect to the board in this case the command to be used from the GDB prompt is:

```
(gdb) mb360 XX.XX.XX.XX
```

XX.XX.XX.XX is the IP address (or DNS name) allocated to the ST40-Connect.

Images loaded in this way must be for RAM startup, and should ensure that the eCos CDL configuration option "Work with a ROM monitor" (`CYGSEM_HAL_USE_ROM_MONITOR`) is disabled. This happens automatically for RedBoot.

There is a serial driver `CYGPKG_DEVS_SERIAL_SH_SCIF` which supports the two on-chip serial devices. Either of these devices can be used by RedBoot for communication with the host. If a device is needed by the application, either directly or via the serial driver, then it cannot also be used for RedBoot communication. Either the alternative serial port, or another communication channel such as ethernet should be used instead. The serial driver package is loaded automatically when configuring for the st40raeb target.

There is an ethernet driver `CYGPKG_DEVS_ETH_SH_ST40EB` for the ST40EB's onboard ethernet device. The device is accessed via the PCI bus. This driver is also loaded automatically when configuring for the st40raeb target, although not activated until generic ethernet package support is also added.

eCos manages the on-chip interrupt controller. Timer 0 is used to implement the eCos system clock, and timer 1 is used to implement a microsecond delay function. Timer 2 is unused and left for the application. Other on-chip devices (FEMI, EMI, LMI, INTC, TMU, CAC, UBC, CPG) are initialized only as far as is necessary for eCos to run. Other devices (eg RTC, DMAC, ST expansion module interface (STEMI) etc) are not touched.

Tools

The ST40EB port is intended to work with GNU tools configured for an sh-elf target. The original port was done using sh-elf-gcc version 3.2.1, sh-elf-gdb version 5.3, and binutils version 2.13.1.

Name

Setup — Preparing the ST40EB board for eCos Development

Overview

In a typical development environment, the ST40EB board boots from flash into the RedBoot ROM monitor. eCos applications are configured for RAM startup and then downloaded and run on the board via the debugger sh-elf-gdb. Preparing the board therefore usually involves programming a suitable RedBoot image into flash memory.

The following RedBoot configurations are supported:

Configuration	Description	Use	File
ROM	RedBoot running from the board's flash	`redboot_ROM.ecm`	`redboot_ROM.bin`
RAM	Used for upgrading ROM version	`redboot_RAM.ecm`	`redboot_RAM.bin`
RAM_NOETH	Used for programming RedBoot the first time	`redboot_RAM_NOETH.ecm`	`redboot_RAM_NOETH.bin`

For serial communications, all versions run with 8 bits, no parity, and 1 stop bit at 115200 baud. This baud rate can be changed via the configuration option `CYGNUM_HAL_SH_SH4_SCIF_BAUD_RATE` and rebuilding RedBoot. RedBoot also supports ethernet communication and flash management.

Initial Installation

Flash Installation

This process assumes that the board is connected to an ST40-Connect/SH. The ST40-Connect should be set up as described in the ST Micro Connect Manual. You should also have access to the SuperH development tools since it is necessary to use the version of GDB that comes with those tools to access the ST40-Connect, **sh-elf-gdb** will not work.

Programming the RedBoot ROM monitor into flash memory requires an application that can manage flash blocks. RedBoot itself has this capability. Rather than have a separate application that is used only for flash management during the initial installation, a special RAM-resident version of RedBoot is loaded into memory and run. This version can then be used to load the normal flash-resident version of RedBoot and program it into the flash.

The first step is to connect an RS232 null modem cable between either of the ST40EB serial ports and the host PC. Next start a terminal emulation application such as HyperTerminal or minicom on the host PC and set the serial communication parameters to 115200 baud, 8 data bits, no parity, 1 stop bit (8N1) and no flow control (handshaking).

Now run the **sh4gdb** command, giving it the name of the RAM_NOETH redboot ELF file, connect to the ST40-Connect, load the executable and run it. The entire session should look like this:

```
$ sh4gdb -nw redboot_RAM_NOETH.elf
GNU gdb 5.2.1
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "--host=i686-pc-linux-gnu --target=sh-superh-elf"...
(gdb) mb360 172.31.1.90
The target is assumed to be little endian
The target architecture is assumed to be sh4
0xa0000000 in ?? ()
(gdb) load
Loading section .vectors, size 0x9d0 lma 0x88020000
Loading section .text, size 0x1d650 lma 0x880209d0
Loading section .rodata, size 0x4364 lma 0x8803e020
Loading section .data, size 0x12b8 lma 0x880423a0
Start address 0x88020000, load size 144956
Transfer rate: 1159648 bits/sec, 3814 bytes/write.
(gdb) cont
Continuing.
```

The required `redboot_RAM_NOETH.elf` file is normally supplied with the eCos release in the `loaders` directory. If it needs to be rebuilt then instructions for this are supplied below. The RAM_NOETH build is used this time instead of plain RAM as it does not contain any PCI or ethernet support. PCI support has been observed to cause complications with using the ST40-Connect.

If this sequence fails in any way then check the setup and connections of the ST40-Connect. It if is successful then you should see the following printed out on the serial line:

```
+FLASH configuration checksum error or invalid key

RedBoot(tm) bootstrap and debug environment [RAM]
Non-certified release, version UNKNOWN - built 09:27:11, Sep 20 2004

Platform: ST40 Eval Board (ST40)
Copyright (C) 2000, 2001, 2002, Free Software Foundation, Inc.
Copyright (C) 2003, 2004, eCosCentric Limited

RAM: 0x88000000-0x8a000000, [0x880491e0-0x89fd1000] available
FLASH: 0x80000000 - 0x80400000, 32 blocks of 0x00020000 bytes each.
RedBoot>
```

At this stage the RedBoot flash management initialization has not yet happened so the warning about the configuration checksum error is expected. To perform this initialization use the **fis init -f** command:

```
RedBoot> fis init -f
About to initialize [format] FLASH image system - continue (y/n)? y
*** Initialize FLASH Image System
*** Initialize FLASH Image System
... Erase from 0x80040000-0x803e0000: .............................
... Erase from 0x80400000-0x80400000:
... Erase from 0x803e0000-0x80400000: .
... Program from 0x89fe0000-0x8a000000 at 0x803e0000: .
RedBoot>
```

At the end, the block of flash at location 0x89FE0000 holds information about the various flash blocks, allowing other flash management operations to be performed.

It is now possible to load the flash-resident version of RedBoot. Because of the way that flash chips work, it is better to first load it into RAM and then program it into flash.

```
RedBoot> load -r -m ymodem -b %{freememlo}
```

The file `redboot_ROM.bin` should now be uploaded using the terminal emulator on your host. The file is raw binary and should be transferred using the Y-modem protocol.

```
Raw file loaded 0x8804fc00-0x88075137, assumed entry at 0x8804fc00
xyzModem - CRC mode, 1197(SOH)/0(STX)/0(CAN) packets, 4 retries
RedBoot>
```

Once the ROM version of RedBoot has been loaded into RAM it can be programmed into flash:

```
RedBoot> fis create RedBoot -b %{freememlo}
An image named 'RedBoot' exists - continue (y/n)? y
```

```
... Erase from 0x80000000-0x80040000: ..
... Program from 0x8804fc00-0x8808fc00 at 0x80000000: ..
... Erase from 0x803e0000-0x80400000: .
... Program from 0x89fe0000-0x8a000000 at 0x803e0000: .
RedBoot>
```

The flash-resident version of RedBoot has now been programmed at location 0x80000000/0xA0000000, and the flash info block at 0x89FE0000/0xA9FE0000 has been updated (on the SuperH, addresses beginning 0xA are the non-cacheable shadow versions in the P2 region of addresses beginning 0x8 from the P1 region). The initial setup is now complete. Power off the ST40-CONNECT and reset the ST40EB board using the reset button SW2. You should see something similar to the following:

```
+FLASH configuration checksum error or invalid key
... waiting for BOOTP information
Ethernet eth0: MAC address 00:80:e1:12:00:3b
IP: 172.31.1.99/255.255.255.0, Gateway: 172.31.1.1
Default server: 172.31.1.2, DNS server IP: 172.31.1.1

RedBoot(tm) bootstrap and debug environment [ROM]
Non-certified release, version UNKNOWN - built 01:21:36, Sep 20 2004

Platform: ST40 Eval Board (ST40)
Copyright (C) 2000, 2001, 2002, Free Software Foundation, Inc.
Copyright (C) 2003, 2004, eCosCentric Limited

RAM: 0x88000000-0x8a000000, [0x8800e380-0x89fd1000] available
FLASH: 0x80000000 - 0x80400000, 32 blocks of 0x00020000 bytes each.
RedBoot>
```

If the ethernet cable is not plugged in there may be a fairly long wait after the "... waiting for BOOTP information" message.

Flash Configuration

The next step is to set up RedBoot's non-volatile configuration values:

```
RedBoot> fconfig -i
Initialize non-volatile configuration - continue (y/n)? y
Run script at boot: false
Use BOOTP for network configuration: true
Default server IP address:
Console baud rate: 115200
DNS server IP address:
Set eth0 network hardware address [MAC]: false
GDB connection port: 9000
Force console for special debug messages: false
Network debug at boot time: false
Update RedBoot non-volatile configuration - continue (y/n)? y
... Erase from 0x803e0000-0x80400000: .
... Program from 0x89fe0000-0x8a000000 at 0x803e0000: .
RedBoot>
```

For most of these configuration variables, the default value is correct, although you may wish to provide a default server used for TFTP retrieval, and default DNS server. If there is no suitable BOOTP service running on the local network then BOOTP should be disabled and, instead, RedBoot will prompt for a fixed IP address, netmask, and addresses for the local gateway and DNS server.

Once you have set appropriate RedBoot flash configuration values you may reset the board. When RedBoot issues its prompt, it is ready to accept connections from sh-elf-gdb, allowing applications to be downloaded and debugged. Connections can be made via either serial port, or by TCP to port 9000 (or an alternative port if manually set by the **fconfig** command).

Occasionally it may prove necessary to update the installed RedBoot image. This can be done simply by repeating the above process, using the ST40-Connect. Alternatively, the existing RedBoot install can be used to load the RAM-resident version in which case the standard RAM RedBoot build can be used instead of the RAM_NOETH build. You can even install the RAM resident RedBoot in the "RedBoot[backup]" flash region. See the RedBoot documentation for instruction on how to do this.

Rebuilding RedBoot

Should it prove necessary to rebuild a RedBoot binary, this is done most conveniently at the command line. Assuming your `PATH` and `ECOS_REPOSITORY` environment variables have been set correctly, the steps needed to rebuild the RAM version of RedBoot are:

```
$ mkdir redboot_ram
$ cd redboot_ram
$ ecosconfig new st40raeb redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/sh/st40eb/VERSION/misc/redboot_RAM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

At the end of the build the `install/bin` subdirectory should contain the file `redboot.bin`.

Rebuilding the ROM or RAM_NOETH versions involve basically the same process. The ROM version uses the file `redboot_ROM.ecm` and the RAM_NOETH version uses the file `redboot_RAM_NOETH.ecm` and both versions generate a file named `redboot.bin`. Make sure you don't mix up the different redboot.bin files; rename them to something more memorable such as `redboot_RAM.bin` and `redboot_ROM.bin`.

Name

Configuration — Platform-specific Configuration Options

Overview

The ST40EB platform HAL package is loaded automatically when eCos is configured for an `st40raeb` target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The ST40EB platform HAL package supports two separate startup types:

RAM
: This is the startup type which is normally used during application development. The board has RedBoot programmed into flash at location 0x80000000/0xA0000000 and boots from that location. sh-elf-gdb is then used to load a RAM startup application into memory and debug it. It is assumed that the hardware has already been initialized by RedBoot. By default the application will use the eCos virtual vectors mechanism to obtain certain services from RedBoot, including diagnostic output.

  This startup type can also be used with the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` disabled to allow eCos applications loaded into RAM to be run using the ST40-Connect and debugged using the SuperH **sh4gdb** version of GDB in the same way as the RAM version of RedBoot was loaded earlier.

ROM
: This startup type can be used for finished applications which will be programmed into flash at location 0x80000000/0xA0000000. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization.

RedBoot and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building RedBoot.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, as a testing step before switching to ROM startup, or for standalone RAM applications loaded and debugged with **sh4gdb**.

If the application does not rely on a ROM monitor for diagnostic services then serial port COM 0 will be claimed for HAL diagnostics.

Flash Driver

The ST40EB board contains 4M bytes of Flash, specifically, two ST M29W160DB parts in parallel. Each part starts with bootblocks of 16K bytes, 8K bytes, 8K bytes and 32K bytes respectively, followed by 31 blocks of 64K bytes each. These are AMD 29xxxxx compatible parts, and as such the `CYGPKG_DEVS_FLASH_AMD_AM29XXXXX` package contains all the code necessary to support these parts and the `CYGPKG_DEVS_FLASH_SH_ST40EB` package contains definitions that customize the driver to the ST40EB board.

Ethernet Driver

The ST40EB board contains an STE10/100A ethernet device. This is largely compatible with Intel i21143/DEC Tulip parts. The device is accessed via the PCI bus. The `CYGPKG_DEVS_ETH_INTEL_I21143` package contains all the code necessary to support this part and the `CYGPKG_DEVS_ETH_SH_ST40EB` package contains definitions that customize the driver to the ST40EB board.

The ethernet will automatically auto-negotiate 10Mbps or 100Mbps operation with its link peer, as well as full duplex or half duplex mode.

The driver usually reads the MAC address (ESA) from the EEPROM connected to the STE. Alternatively an address can be set in the CDL configuration in the component CYGSEM_DEVS_ETH_SH_ST40EB_ETH0_SET_ESA within the ST40EB ethernet driver; or an address can be set in the Flash configuration of RedBoot using the **fconfig** command. If both are set, the Flash configuration is used in preference.

PCI Driver

The ST40EB board is fitted with three 5V PCI slots, which are accessed via a PCI-PCI bridge, as is the onboard STE10/100A ethernet device. The 3.3V PCI slot is accessed directly. eCos supports PCI devices inserted in these slots and if the PCI library is selected in the eCos configuration, a driver will usually call `cyg_pci_init()` which will automatically configure memory and I/O base address registers, as well as any interrupts the device requires.

The CPU is able to access the PCI memory space through the memory window from 0xb0000000 to 0xb5ffffff, and the PCI I/O space through the memory window from 0xb6000000 to 0xb6ffffff. PCI devices may access system memory (usually for DMA), at windows starting at address 0x0 in both memory and I/O PCI spaces.

System Clock

By default, the system clock interrupts once every 10ms, corresponding to a 100Hz clock. This can be changed by the configuration option `CYGNUM_HAL_RTC_DENOMINATOR` which corresponds to the clock frequency. Other clock-related settings are recalculated automatically if the denominator is changed.

Compiler Flags

The platform HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos. There are two flags specific to this port:

`-m4`

The sh-elf-gcc compiler supports many variants of the SH architecture, from the SH2 onwards. A `-m` option should be used to select the specific variant in use, and with current tools `-m4` is the correct option for the ST40.

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the ST40EB hardware, and should be read in conjunction with that specification. The ST40EB platform HAL package complements the SH architectural HAL and the SH4 variant HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset the HAL will initialize or reinitialize most of the on-chip peripherals. There is an exception for RAM startup applications which depend on a ROM monitor for certain services.

For ROM startup, the HAL will perform additional initialization, setting up the external SDRAM and programming the various internal registers including clocks, EMI and LMI. The values used for most of these registers are assigned fixed values from a table in the header `cyg/hal/platform.inc`.

Linker Scripts and Memory Maps

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

off-chip Flash
: This is located at address 0x00000000 of the physical memory space and is therefore accessible in the P1 region at location 0x80000000. An uncached shadow of this memory is available in the P2 region at 0xA0000000. The contents of the flash are organized as described earlier.

external SDRAM
: This is located at address 0x08000000 of the physical memory space and is therefore accessable in the P1 region at location 0x88000000. An uncached shadow of this memory is available in the P2 region at 0xA8000000. The first 256 bytes are used for hardware exception vectors. The next 256 bytes are normally used for the eCos virtual vectors, allowing RAM-based applications to use services provided by the ROM monitor. For ROM startup, all remaining SDRAM is available. For RAM startup, available SDRAM starts at location 0x80020000, with the bottom 128Kbytes reserved for use by RedBoot.

on-chip peripherals
: These are accessible via the P4 region at location 0xE0000000 onwards. The on-chip PCI controller is located at base address in the P2 region at 0xB0000000, and a memory-mapped view of the PCI space configuration registers at 0xB7000000. Other base addresses of on-chip peripherals can be found in the ST40RA datasheet (P/N: ADCS 7260755H).

off-chip peripherals
: All off-chip peripherals used by eCos are accessed via on-chip bus controllers such as LMI, EMI or PCI. All others are left untouched.

Clock Support

The platform HAL provides configuration options for the eCos system clock. This always uses the hardware timer 0, which should not be used directly by application code. Timer 1 is used to implement a microsecond resolution busy delay service. Timer 2 is not used by eCos so application code is free to manipulate this as required. The actual HAL macros for managing the clock are provided by the SH architecture processor HAL.

There is a software model of the structure of the SH family clock supply subsystem which performs the correct calculations to yield not only the inputs for the CPU clock but also the peripheral clocks fed to the serial device, memory controllers and other devices. The values for the master crystal, the PLL multipliers and various dividers are supplied by the platform HAL. Some care must be taken in defining these since wrong values will cause the timers and the SCIF baud rate to be miscalculated (resulting visibly in garbage on the serial output).

The ST40 extends the SH family clock model by providing a CLOCKGEN subsystem allowing the hardware clock frequency to be controlled. The CLOCKGENA.PLL1CR register is the primary means to do this, and is initialised by switches 1, 2 and 3 on DIP switch block SW3. As the ST40EB is fitted with an ST40RA166 processor, it is assumed that a speed of 166MHz has been selected. This corresponds to SW3-1 set to OFF, SW3-2, set to OFF and SW3-3 set to ON.

If the DIP switches are changed from the default then the value of `CYGHWR_HAL_SH_OOC_XTAL` must be changed to match. Consult the ST40RA documentation on appropriate values for the clock and associated divider options for the subclocks if you wish these to be altered from the default.

Other Issues

The ST40EB platform HAL does not affect the implementation of other parts of the eCos HAL specification. The SH4 variant HAL, and the SH architectural HAL documentation should be consulted for further details.

It should be noted that the floating point support in the SH HAL has a caveat that, if the FPSCR register is changed, it may get reverted at a later stage by certain operations performed by the GCC compiler. This behaviour is intentional as the alternative would be to update the GCC compiler's internal state about the FPSCR at every context switch which would be expensive for a feature that is unlikely to be used frequently. If the FPSCR is to be changed by the application, the developer should call the function `__set_fpscr(int)`, passing it the new FPSCR value.

Part LXXXVI. TILE-Gx Architecture

Table of Contents

Chapter 357. TILE-Gx Architectural Support

Name

Overview — eCos Support for the TILE-Gx Family of Processors

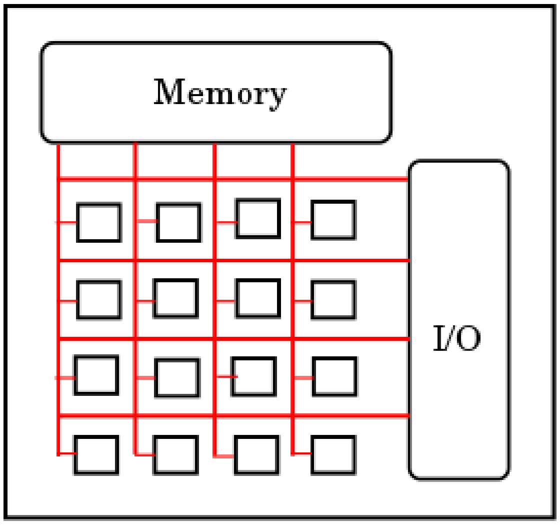
Description

This package `CYGPKG_HAL_TILEGX` provides architectural HAL support for running eCos aplications on a Tilera TILE-Gx processor. In many ways this is a very unusual architecture for running eCos, and therefore the TILE-Gx eCos port is also somewhat different from ports to other architectures.

This section of the package documentation gives a brief description of the TILE-Gx hardware, how it is typically used, and the main limitations of the eCos port. Later sections describe the process of preparing hardware for running an eCos application, the various configuration options provided by the port, and some of the implementation details.

The Hardware

A TILE-Gx chip contains of an array of CPU cores, known as tiles. It also contains a set of memory interfaces and I/O peripherals. The cores, memory and peripherals are connected via a mesh interconnect. One of the peripherals, USB, can be used to connect the TILE-Gx chip to a host PC for software development.



Each tile is a fully-fledged and fairly conventional CPU, aimed at running an advanced operating system like Linux. It has a register bank of 64 64-bit registers, some of which have a dedicated purpose such as communication over the mesh network. There are also a considerable number of special purpose registers or SPRs. Many of these are intended for system software, for example to control the memory management unit and the interrupt subsystem.

Each tile has a split instruction/data primary cache and a secondary cache. The hardware also implements a distributed tertiary cache over the mesh interconnect. Assuming the MMU is set up correctly on each tile the hardware will maintain cache coherency. Shared memory can be used for exchanging large amounts of data between tiles. The mesh interconnect also provides two buses, the I/O Dynamic Network or IDN and the User Dynamic Network or UDN. These allow the transfer of fairly small packets, up to a couple of hundred 64-bit words of data, between tiles.

On power up the hardware runs a hypervisor boot image, typically loaded from a serial ROM although alternative images can be loaded from a host PC over USB. Usually the hypervisor proceeds to run an SMP Linux kernel on all the tiles, and applications run as processes on top of Linux in a protected environment. The kernel runs on top of the hypervisor, and control over the various peripherals and memory resources is shared between the two in complicated ways. It is possible to partition the chip's tiles such that Linux only runs on some of them, while others run a bare metal executable or BME application. A BME application replaces the hypervisor and runs with full access to the hardware. The eCos port uses this functionality to run eCos applications on some of the tiles.

Limitations

The eCos port to the TILE-Gx architecture is subject to some important restrictions, and application developers should be aware of these.

1. At the time of writing eCos does not provide full SMP functionality. It is possible to run eCos applications on multiple tiles at the same time, but these tiles all run their own private instance of eCos. Each tile is allocated its own private memory. Once the eCos applications are running on the various tiles they can communicate over the UDN bus or over shared memory, and they can also communicate with Linux processes running inside the Linux partition. The TMC support package `CYGPKG_HAL_TILEGX_TMC` in `packages/hal/tilegx/tmc` provides some support code for this, and more importantly some detailed examples.

2. Also at the time of writing eCos does not support operating in 64-bit mode. In practice this is not a major problem. Although TILE-Gx is a 64-bit architecture the instruction set provides full support for running in 32-bit mode, and **tile-gcc** has a *`-m32`* flag to support this. Running in 32-bit mode should have no detrimental effect on performance. In fact it may improve performance very slightly because pointers in data structures will consume less memory, allowing more data to fit into the caches and reducing the memory bandwidth requirements. The main restriction is that the address space of an eCos application is limited to two gigabytes (it should be four gigabytes but **tile-gdb** appears to get confused at times when dealing with 32-bit pointers with the top bit set).

3. The eCos ports depends on various bits of Tilera software. The port has been performed with the TileraMDE-4.1.0.148119 release of the Multicore Development Environment. If there are incompatible changes to the MDE, for example if the hypervisor code for loading and starting a BME application is rewritten, then such changes could stop the eCos port working in strange ways. The TMC package's examples for setting up shared memory between a Linux process and an eCos application are also particularly vulnerable to breakage given the rapid rate of development of the Linux kernel.

4. The Tilera support for BME applications is limited. That means that sharing resources between the Linux partition and BME tiles running eCos is generally difficult, and at times may be impossible. For example there is no easy way for eCos to allocate additional physical memory: after bootstrap all unallocated memory belongs to the Linux partition and the hypervisor. If more memory is needed then the allocation has to be performed by a Linux process and the details of the allocation can then be passed on to the eCos application, allowing the latter to map the memory into its address space.

   Particular difficulties are likely to arise when it comes to peripherals. Sharing a peripheral between an eCos application and a Linux device driver will typically be impossible: the latter will not have been written to allow sharing, and any spinlocks or other locking mechanisms that may exist will not be accessible to an eCos application.

Name

Setup — preparing the hardware for eCos development

Overview

Just as the TILE-Gx hardware is rather different from more conventional hardware used to run eCos, the process of setting up the hardware is also rather different. Most importantly eCos only runs on some of the tiles in a chip, with Linux and the hypervisor running on the other tiles. It is the hypervisor that is responsible for booting all the tiles. Setting up the hardware involves constructing and running a suitable hypervisor image. Such an image contains the following:

1. The hypervisor executable, `hv`. This is a 64-bit ELF executable. A default build is provided with the Tilera Multicore Development Environment (MDE) in the `$TILERA_ROOT/tile/boot` subdirectory, but a custom build may be necessary to cope with specific hardware such as ethernet PHY chips.

2. A hypervisor configuration file, typically with `.hvc` suffix. An example can be found in the MDE file `$TILERA_ROOT/tile/etc/hvc/vmlinux.hvc`. Amongst other functionality this configuration file allows some of the tiles to be designated as BME or bare metal executable tiles, which will run a dedicated application directly on the hardware instead of layered on top of Linux and the hypervisor. eCos applications run in a BME tile.

3. A Linux kernel, for example `$TILERA_ROOT/tile/boot/vmlinux`. This is another 64-bit ELF executable. Again a custom build of the Linux kernel may be needed on some hardware. Note that the default `vmlinux` file has not been stripped of debug information and has not been compressed, so it will be some tens of megabytes in size.

4. An initial RAM file system, for example `$TILERA_ROOT/tile/boot/initramfs.cpio.gz`. This contains executables, shared libraries, and other files needed by the Linux system at run-time. Applications are very likely to involve a custom version of this RAM file system containing additional files, for example the application executables. Usually ELF files will be stripped off their debug information as they are incorporated into the initramfs file, and of course the latter is compressed.

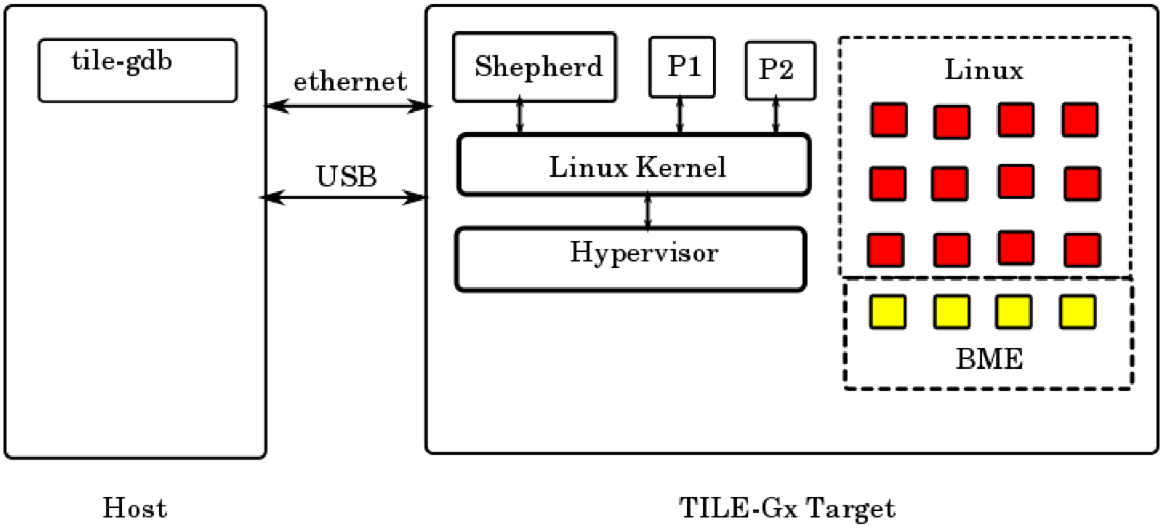
5. If the hypervisor configuration file specifies that one or more tiles should form a BME partition, the executable that should run on those tiles must also be incorporated into the hypervisor image since it is the hypervisor that will launch that executable.

A hypervisor image is created directly by the **tile-mkboot** command, or indirectly by **tile-monitor**. An initramfs file is created by **tile-gen-initramfs**. Full information on these commands is provided in the Tilera documentation, especially UG509 The Multicore Development Environment System Programmer's Guide. That information is not repeated here, instead this document focusses only on eCos-specific aspects. It is assumed throughout that the application developer has a working MDE installation and that the hardware is already set up for developing Linux applications.

The setup process is different for debug and production systems. A debug system allows eCos applications to be loaded and debugged via **tile-gdb**, and will be used for most of the development process. In a production system the eCos application is loaded and starts running automatically during bootstrap, but cannot be debugged via **tile-gdb** (except under certain circumstances when running on the simulator instead of real hardware). There are also three different scenarios: running on real hardware with the hypervisor image loaded over USB; running on real hardware with the hypervisor image booting automatically from a serial ROM; and running on the simulator.

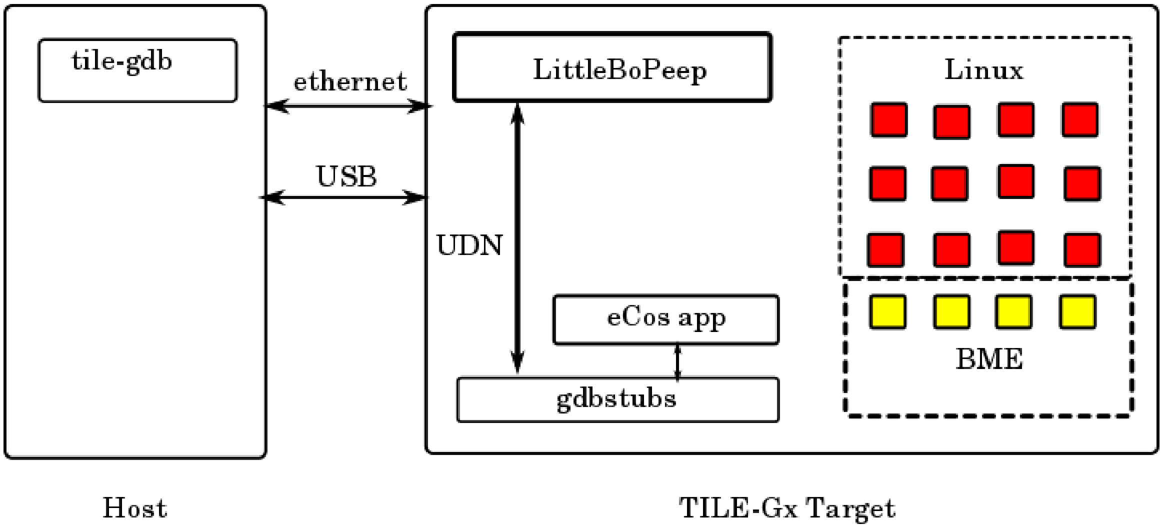
Debugging Overview

Debugging an application running on a TILE-Gx chip generally involves software rather than any hardware debug technology. First consider a process running in the Linux partition of a TILE-Gx chip:



This shows a 4x4 TILE-Gx chip with 12 of the tiles configured as a Linux partition and 4 tiles configured as a BME partition. There are two application processes running in the Linux partition, P1 and P2, both sitting on top of the Linux kernel which in turn sits on top of the hypervisor. There is also an auxiliary process, the shepherd. A host PC is connected to the TILE-Gx target via USB and ethernet. Debugging an application on the TILE-Gx target involves running **tile-gdb** on the host PC. This debugger instance connects to the shepherd process on the target, and communication between the two uses the gdb remote protocol over TCP/IP. When for example **tile-gdb** needs to read a memory location inside process P1 it constructs a memory read remote protocol request and sends this to the shepherd on the target. The shepherd decodes the request and then makes a `ptrace` system call into the Linux kernel. The kernel performs the desired operation and passes the result back to the shepherd. This constructs a remote protocol reply message which gets sent back to **tile-gdb** on the host, and the debugger now has the required information.

Debugging an eCos application running inside a BME tile involves a similar but not identical process. The key difference is that a Linux kernel running in the TILE-Gx chip has no control over any of the BME tiles, so the `ptrace` system call is useless. Instead the main debug functionality is provided by a special eCos application, gdbstubs, held in the file `gdb_module.64`. The hypervisor boots this eCos application into each BME tile. The user's eCos application is loaded and run on top of gdbstubs. Now, unfortunately gdbstubs does not have the same ready access to the outside world as the Linux shepherd: the target's network and USB interfaces are managed by the Linux side so gdbstubs cannot easily provide TCP/IP communication. Instead there is another process, LittleBoPeep running inside the Linux partitition alongside the shepherd. LittleBoPeep accepts TCP/IP connections from **tile-gdb** instances on the host. Remote protocol requests are passed on to gdbstubs running on the appropriate BME tiles over the TILE-Gx internal UDN communication network. gdbstubs decodes the request, performs the appropriate operation such as reading a memory location, and sends the reply back to LittleBoPeep over UDN. LittleBoPeep then forwards the reply to **tile-gdb** on the host, and the debugger now has the required information.



UDN messages are addressed to a specific tile, so for LittleBoPeep to receive messages from gdbstubs it must bind itself to a specific tile within the Linux partition. It will always select the last tile in that partition. This may cause problems if other Linux applications attempt to use UDN communications. By default a user-level process like LittleBoPeep does not have permission to access the UDN network, so it has to make a call into the Linux kernel to obtain access. The Linux kernel will only allow one process per tile to perform UDN communication, which is a somewhat strange restriction since the UDN hardware supports four separate communication channels and LittleBoPeep only needs one of them. Therefore if some other process runs on the same tile as LittleBoPeep and claims UDN access first, LittleBoPeep's UDN initialization will fail. Alternatively if LittleBoPeep initializes first then the other process' attempt at claiming UDN access will fail. This problem cannot be worked around without changing the Linux kernel.

Hardware, USB Bootstrap, Debug system

The first scenario involves a debug system running on real hardware with the hypervisor image booted via USB and **tile-monitor**. This is likely to be the most common scenario during software development. The first step is to take an existing `.hvc` hypervisor configuration file, for example `$TILERA_ROOT/tile/etc/hvc/vmlinux.hvc`, and append lines like the following:

```
# Define the BME tiles
bme bmeapp private 3,0 3,1 3,2 3,3
  memory 0 default
  pertile va=0x6c000000
```

These lines define a BME partition consisting of four tiles: 3,0 3,1 3,2 and 3,3. The BME partition can be given more, fewer, or different tiles by changing this list of tile addresses. The application that will be run on each tile is given an alias of `bmeapp`, and that alias will be mapped on to a real filename via a **tile-monitor** command line argument. The keyword `private` is essential: it informs the hypervisor that each BME tile is independent from the other and needs its own memory. The alternative would be a BME partition running a single SMP application with the memory shared between the BME tiles, and SMP is not supported with the current TILE-Gx eCos port.

The line `memory 0 default` determines which memory controller is used for the BME tiles' memory. This particular line is cloned from the MDE's `tilegx/examples/bme/client_server/sim.hvc`. Application developers may wish to use different settings for this as per the Tilera manual UG509, section 8.4.3.

The bulk of the memory allocated to a BME application is determined automatically by the hypervisor from information in the ELF executable. However the hypervisor will allocate an additional block of pertile memory to hold the initial stack, some hypervisor data structures, and so on. The line `pertile va=0x6c000000` places that additional block within the 32-bit address space supported by an eCos application. This line must not be changed since eCos expects to find the memory at that address.

Note that the Tilera tools support only a single BME partition definition in the hypervisor configuration file. Therefore it is not possible to run different applications on different BME tiles, or to have different memory settings for different tiles. Usually this will not be an issue for a debug system because the executable is always a gdbstubs binary.

Once the hypervisor configuration file is ready it is possible to boot up the system with **tile-monitor**:

```
tile-monitor --verbose --dev /usb0 --hvc ecos.hvc \
             --mkboot-args -+- --no-strip -+- \
             --bme bmeapp=<path0>/gdb_module.64 \
             --upload <path1>/LittleBoPeep /LittleBoPeep \
             --launch - /LittleBoPeep 3,0 3,1 3,2 3,3 -
```

The various options are as follows:

- `--verbose` is optional. It enables additional diagnostics within **tile-monitor** which may help to track down problems.
- `--dev /usb0` tells **tile-monitor** how to interact with the target hardware, in this case through the `tileusb0` device provided by the Tilera USB device driver. If the hardware is accessed via some other means then argument will need to be adjusted accordingly.
- `--hvc ecos.hvc` identifies the hypervisor configuration file that should be used.
- `--mkboot-args -+- --no-strip -+-` causes **tile-monitor** to pass the argument `--no-strip` when it invokes a sub-process **tile-mkboot** to create the hypervisor image. By default **tile-mkboot** will strip all debug information out of ELF executables that go into the hypervisor image, including the BME executable. Unfortunately **tile-strip** is not compatible with eCos executables and will corrupt them, preventing the hypervisor from correctly loading these executables into the BME tiles. Suppressing the automatic stripping bypasses this problem.
- `--bme bmeapp=<eCos executable>`. The BME lines in the hypervisor configuration file specified bmeapp as an alias for the executable that should be run on all BME tiles. Here we specify exactly which file corresponds to that alias. For a debug system the executable should always be a gdbstubs binary `gdb_module.64`. A prebuilt version of that executable should be included in the release, or alternatively a binary can be rebuilt as described below.
- `--upload <path1>/LittleBoPeep /LittleBoPeep`. This uploads the LittleBoPeep executable to the target, storing in in the root of the RAM file system.
- `--launch - /LittleBoPeep 3,0 3,1 3,2 3,3 -` Once LittleBoPeep has been uploaded to the Linux system it is started with the appropriate arguments. These arguments specify the BME tiles running gdbstubs and should match the list in the hypervisor configuration file.

The above invocation of **tile-monitor** uses the default hypervisor executable, the default Linux kernel executable and the default initramfs file from `$TILERA_ROOT/tile/boot`. Alternative versions of these can be specified if desired using `--hv-bin-dir`, `--vmlinux` or `--initramfs` options. See the **tile-monitor** documentation for more information.

Once the hypervisor image has been created, downloaded over USB, and started the hypervisor will set up the BME tiles and start the bmeapp application on each one. When gdbstubs is that application it will pause early on during initialization, waiting to be contacted by LittleBoPeep over the UDN network. Some time later, when uploading and launching LittleBoPeep **tile-monitor** should report the following:

```
[monitor] Uploading...
[monitor] Uploading complete.
[monitor] Process 551 created using '/LittleBoPeep'.
```

The exact process number may vary depending on what else is running. LittleBoPeep will now start running, connect to gdbstubs running on the specified tiles, and report the status of each one on the system console:

```
LittleBoPeep: starting.
LittleBoPeep: tile 3,0, gdbstubs active, listening on port 10300
LittleBoPeep: tile 3,1, gdbstubs active, listening on port 10301
```

```
LittleBoPeep: tile 3,2, gdbstubs active, listening on port 10302
LittleBoPeep: tile 3,3, gdbstubs active, listening on port 10303
```

At this point LittleBoPeep, running in the Linux partition, is ready to accept TCP connections from **tile-gdb** on any network interface to any of the specified ports. Assume that the target-side Linux system has been set up with TCP/IP networking enabled and that the gbe0 network interface has been assigned the network address 10.1.1.42. Also assume that the user has configured and built a RAM-startup eCos application, for example the Hello World example, as per the eCos User Guide. It is now possible to load and run the eCos executable on one of the BME tiles:

```
% tile-gdb --quiet hello
Reading symbols from .../hello...done.
(gdb) target remote 10.1.1.42:10300
Remote debugging using 10.1.1.42:10300
0x00016550 in ?? ()
(gdb) load
Loading section .text, size 0x15d80 lma 0x10010000
Loading section .data, size 0x180 lma 0x10025d80
Start address 0x10010000, load size 89856
Transfer rate: 37 KB/sec, 987 bytes/write.
(gdb) break exit
Breakpoint 1 at 0x10021d40: file .../src/exit.cxx, line 75.
(gdb) continue
Continuing.
Hello, eCos world!
[Switching to Thread 2]

Breakpoint 1, exit (status=0x0) at .../src/exit.cxx:75
75  {
(gdb) maintenance packet r
sending: "r"
received: ""
(gdb) quit
A debugging session is active.

    Inferior 1 [Remote target] will be killed.

Quit anyway? (y or n) y
```

Obviously the sizes, transfer rate, and so on will vary somewhat. The command **target remote 10.1.1.42:10300** tells tile-gdb to establish a TCP/IP connection to IP address 10.1.1.42, port 10300, and communicate using the gdb remote protocol. The IP address corresponds to the target's gbe0 network interface - note that any of the target's network interfaces may be used, gbe0 is used here simply as an example. The port number is one of the ones reported by LittleBoPeep and corresponds to tile 3,0. The eCos application is loaded and runs to completion, hitting a breakpoint at the `exit()` function. Note that the application's console output goes over the debug channel and is reported by **tile-gdb**. LittleBoPeep supports concurrent debug sessions to every BME tile running gdbstubs, but only one **tile-gdb** instance can interact with a given tile at a time.

Normally when gdb exits the target-side application is left running. Most of the time this is not what is wanted when developing an eCos application. Instead the desired behaviour is that the tile gets restarted, ready for loading and running another build of the application being debugged. That is achieved by the **maintenance packet r** command. Quitting **tile-gdb** after that command has been issued will cause the tile to restart, and LittleBoPeep will report:

```
LittleBoPeep: tile 3,0 is resetting.
LittleBoPeep: tile 3,0, gdbstubs active, listening on port 10300
```

The port assignment used by LittleBoPeep is straightforward: given a gdbstubs instance running on tile x,y, LittleBoPeep will accept TCP connections on port (10000 + 100x + y). This numbering scheme should support chips with up to 10000 tiles, although of course trying to debug applications on even a small fraction of that number of tiles will prove problematical.

eCos applications run with limited memory protection, and have unrestricted access to system resources such as the special purpose registers used for controlling the MMU and interrupts. Therefore an eCos application can crash the tile it is running on, for example by accidentally overwriting some critical area of memory. At that point **tile-gdb** and LittleBoPeep will no longer be able to communicate with the gdbstubs executable running on the target tile, and LittleBoPeep will report this:

```
LittleBoPeep has lost one of her sheep.
  Tile 3,0 has stopped responding to UDN packets.
  This tile is now marked as crashed and cannot be used again until the hardware is reset.
```

Most commonly this condition will be detected when the application stops producing output and the user attempts to interrupt it with a ctrl-C. Unfortunately the gdb remote protocol provides no easy way for LittleBoPeep to reliably send a suitable diagnostic message to **tile-gdb**, so the debugger will simply report that the connection to the target has been closed.

It should be noted that eCos does run with the MMU enabled and that the eCos application only has access to the memory allocated to its tile by the hypervisor during bootstrap, plus any additional memory explicitly mapped into the address space typically by calling `hal_tilegx_mmap()`. That additional memory may correspond to memory-mapped I/O devices. It may also be memory allocated by a Linux application using the TMC library whose details are then passed on to the eCos application. eCos does not have access to other memory in the system so for example it cannot overwrite kernel data structures on some other tile running Linux.

The above **tile-gdb** example assumes that the system is set up with Linux TCP/IP networking enabled, so that **tile-gdb** can connect to LittleboPeep. If this is not the case, for example because all network interfaces are needed for non-TCP/IP communications, then a slightly different approach is needed. **tile-monitor** and the target-side shepherd process provide tunnelling support: **tile-monitor** will accept connections on the host PC and forward any data to the shepherd over the USB connection; the shepherd process will establish a matching TCP/IP connection within the Linux partition and forward any data to its destination within the target. Tunnelling requires some additional arguments when invoking **tile-monitor**:

```
tile-monitor --verbose --dev /usb0 --hvc ecos.hvc \
             --mkboot-args -+- --no-strip -+- \
             --bme bmeapp=<path0>gdb_module.64 \
             --tunnel 10300 10300 --tunnel 10301 10301 \
             --tunnel 10302 10302 --tunnel 10303 10303 \
             --upload <path1>/LittleBoPeep /LittleBoPeep \
             --launch - /LittleBoPeep 3,0 3,1 3,2 3,3 -
```

tile-monitor on the host PC will accept connections on port 10300, and the shepherd will establish a matching target-side connection to port 10300. Each tunnel argument establishes one such pairing. Inside the **tile-gdb** session it is necessary to use a remote address of `localhost:10300` instead of `10.1.1.42:10300`, connecting via **tile-monitor** instead of directly to the remote port.

Hardware, ROM Bootstrap, Debug System

This scenario also allows eCos applications to be debugged via **tile-gdb** and LittleBoPeep. However the hypervisor image is not booted into the target hardware via **tile-monitor** and USB. Instead the hardware boots automatically from a serial ROM. Setting up a system like this requires two steps: constructing a suitable hypervisor boot image; and programming that image into the serial ROM.

Creating the boot image involves either the **tile-mkboot** command or **tile-monitor --create-bootrom** which implicitly invokes the former. The latter approach is taken here. As before a boot image incorporates a hypervisor executable, a hypervisor configuration file, a Linux kernel, an initramfs file, the executable to run on any BME tiles, and a small number of support files. The hypervisor executable and configuration file can be the same as before, and `gdb_module.64` should again be used as the BME executable. However the Linux kernel and the initramfs file need special attention.

A typical serial ROM is comparatively small, usually 16MB. Worse, that ROM normally holds a primary boot loader and two separate boot images, the current image plus a backup image to allow recovery if and when things go wrong. That means a new boot image file has to be a bit less than 8MB. Given that an uncompressed unstripped Linux kernel is some tens of megabytes, clearly it is necessary to strip and compress it when generating the boot image. **tile-mkboot** will do this by default to all executables, but this behaviour can be suppressed with the *`--no-strip`* option.

Unfortunately **tile-mkboot** will also attempt to strip and compress any BME executables, and does not provide any finer-grained control over this behaviour. Applying **tile-strip** to an eCos executable will corrupt it, and the hypervisor's BME loader code does not support loading compressed executables. Therefore if **tile-mkboot** stripping and compression is enabled then the eCos application cannot be loaded into a BME tile, but if it is disabled then the resulting boot image file will be far too large for the serial ROM.

The solution is to pre-strip and pre-compress the Linux kernel file before constructing the boot image.

```
$ tile-strip -o vmlinux_stripped $TILERA_ROOT/tile/boot/vmlinux
$ bzip2 -9 vmlinux_stripped
```

That takes care of the Linux kernel. It is also necessary to customize the initramfs file so that LittleBoPeep is started automatically during the Linux bootstrap process. The process of constructing a custom initramfs with **tile-gen-initramfs** is documented in the Tilera manual UG509, section 3.4.2. It involves editing a `contents.txt` file, adding the following lines:

```
file /usr/bin/LittleBoPeep DIR/LittleBoPeep 755 0 0
file /etc/rc.local DIR/rc.local 644 0 0
```

This assumes that the LittleBoPeep executable has been placed in the same directory as the edited `contents.txt` and that a file `rc.local` has been created alongside it. That file should contain:

```
/usr/bin/LittleBoPeep 3,0 3,1 3,2 3,3 &
```

(This assumes that the application being developed does not already involve an `rc.local` file to start various processes within the Linux partition. If that file already exists then the LittleBoPeep line can just be appended.)

The Linux bootstrap will automatically run `/etc/rc.local` if that exists, so LittleBoPeep will be started in the background and will connect to gdbstubs on the four tiles specified. Obviously if the hypervisor configuration file lists a different set of tiles then `rc.local` should be updated to match.

Once a suitable Linux kernel and initramfs file are ready the boot image can be created:

```
tile-monitor --no-dev --create-bootrom image.bootrom \
             --hvc ecos.hvc --mkboot-args -+- --no-strip -+- \
             --vmlinux vmlinux_stripped.bz2 \
             --initramfs ecos_initramfs.cpio.gz \
             --bme bmeapp=<path>/gdb_module.64
```

This should produce a suitable file `image.bootrom`. That file can now be transferred to a running TILE-Gx system and programmed into the serial ROM using **sbim -i image.bootrom**. When the hardware is rebooted the hypervisor will automatically set up the BME tiles as per the `ecos.hvc` configuration file, load and start `gdb_module.64` on each BME tile, and start the Linux kernel on the remaining tiles. The Linux kernel will go through its boot process and end up running `/etc/rc.local`, which in turn will start LittleBoPeep running. LittleBoPeep will connect to gdbstubs on the specified tiles, then it will accept debug connections from **tile-gdb** over TCP/IP as before.

Hardware, USB Bootstrap, Production System

Once the eCos application has been debugged to the developer's satisfaction it is time to switch from a debug system to a production system. This no longer involves gdbstubs or LittleBoPeep. Instead the eCos application is incorporated directly into the hypervisor boot image and started automatically. Note that this will happen fairly early on in the bootstrap process, before the Linux kernel is started let alone any Linux processes running on top of the kernel. Console output from the eCos application will go the target's system console.

The first step is to change the eCos configuration option **CYG_HAL_STARTUP** from RAM to ROM. For the TILE-Gx architecture RAM startup is for applications which will run on top of gdbstubs, while ROM startup is for applications which are incorporated into the hypervisor boot image. The eCos configuration should then be rebuilt and the application relinked. Assume this application is called `ecosapp`.

Now, an eCos application is a 32-bit executable. The hypervisor BME loading code only supports 64-bit executables. To work around this the TILE-Gx architectural HAL comes with a utility script **tile-ecos-32to64** which reads in a full 32-bit executable, discards anything which will not be needed on the target-side such as debug information, and outputs a 64-bit pseudo-executable.

```
tile-ecos-32to64 ecosapp ecosapp.64
```

The output file `ecosapp.64` will not be a fully-fledged ELF executable and other tools such as **tile-objdump** may be confused by it. It is intended only for use by the hypervisor's BME loader. If say a disassembly is required then **tile-objdump** should be applied to the original 32-bit file, not the generated 64-bit pseudo-executable. The system can now be started via **tile-monitor**:

```
tile-monitor --verbose --dev /usb0 --hvc ecos.hvc \
             --mkboot-args -+- --no-strip -+- \
             --bme bmeapp=ecosapp.64
```

Most of these arguments are the same as when booting a debug system over USB. The same hypervisor executable, hypervisor configuration file, Linux kernel and initramfs file can be used. A different file is associated with the `bmeapp` alias, and there is no need to upload and launch LittleBoPeep since there are no gdbstubs instances running in any of the BME tiles for it to talk to.

The Tilera hypervisor only supports a single application which will be run on all BME tiles. If the system needs different functionality on different tiles then the eCos application must be a union of all functionality, and a run-time decision must be made as to which tile runs what code.

Hardware, ROM Bootstrap, Production System

Setting up a production system which can boot from ROM is very similar to setting up a debug system. It will again be necessary to strip and compress the Linux kernel before generating the boot image. As far as eCos is concerned there is no need for anything extra in the initramfs file. In particular the LittleBoPeep executable would serve no purpose so does not need to be included or started from `rc.local`.

As far as the eCos executable is concerned, this must be processed in the same way as when booting a production system over USB. The eCos configuration must be changed to ROM startup and rebuilt, the application must be relinked, and the resulting 32-bit executable must be processed with **tile-ecos-32to64**. Once this is done the boot image can be created using **tile-monitor**:

```
tile-monitor --no-dev --create-bootrom image.bootrom \
             --hvc ecos.hvc --mkboot-args -+- --no-strip -+- \
             --vmlinux vmlinux_stripped.bz2 \
             --bme bmeapp=ecosapp.64
```

The resulting `image.bootrom` file can now be transferred to a running TILE-Gx system and programmed into the serial ROM using **sbim -i image.bootrom**.

Simulator, Debug System

Running an application in the simulator is mostly similar to running on the hardware, albeit very much slower. Also the simulator does not implement all the hardware functionality, for example ethernet emulation is very limited. Typically a slightly different hypervisor configuration file is used, see `$TILERA_ROOT/tile/etc/hvc/vmlinux-sim.hvc` as opposed to `$TILERA_ROOT/tile/etc/hvc/vmlinux.hvc`. The file will need to be edited to incorporate a BME partition.

The simulator is usually started by another **tile-monitor** invocation:

```
tile-monitor --verbose --simulator --config gx8016 \
             --console --functional --gdb-port 9000 \
             --bm-debug-on-panic --debug-on-crash \
             --hvc ecos-sim.hvc \
             --mkboot-args -+- --no-strip -+- \
             --bme bmeapp=<path0>/gdb_module.64 \
             --tunnel 10300 10300 --tunnel 10301 10301 \
             --tunnel 10302 10302 --tunnel 10303 10303 \
             --upload <path1>/LittleBoPeep /LittleBoPeep \
             --launch - /LittleBoPeep --spin 3,0 3,1 3,2 3,3 -
```

Here **tile-monitor** is instructed to start a functional simulation of a gx8016 chip and provide the system console. A hypervisor image is created containing `gdb_module.64` and that executable will be started automatically on all BME tiles defined in the hypervisor configuration file. Once Linux is up and running in the simulator LittleBoPeep is loaded and started. Given the simulator's very limited ethernet support TCP/IP networking will not be available, so **tile-gdb** will have to connect to LittleBoPeep over a tunnel set up by **tile-monitor** on the host PC and the shepherd process on the target.

LittleBoPeep is started with an additional argument, *`--spin`*. This works around another limitation within the TILE-Gx Linux world. When gdbstubs sends a gdb remote protocol message or reply to LittleBoPeep over the UDN network, there is no easy way for that UDN traffic to wake up a sleeping Linux process. Normally LittleBoPeep polls the UDN network for incoming data at 10 millisecond intervals, waking up every clock tick. That gives acceptable latencies and bandwidth when debugging on real hardware. However simulating a 10 millisecond interval takes many real seconds, impacting gdb communication performance sufficiently badly that **tile-gdb** becomes almost unusable. Running LittleBoPeep with *--spin* forces it to poll continuously for incoming UDN traffic instead of at intervals. This will greatly improve debug performance, but of course debugging will still be slow compared with real hardware. It should be noted that *--spin* disables the code in LittleBoPeep which detects crashed tiles, so these will no longer be reported.

Simulator, Production System

Although rarely useful, it is possible to run a production system on the simulator. As with production systems running on real hardware, the eCos configuration needs to be switched to ROM startup and rebuilt, the eCos application needs to be relinked, and **tile-ecos-32to64** has to be used to convert the 32-bit executable to a 64-bit pseudo-executable which can be read by the hypervisor's BME loader.

```
tile-monitor --verbose --simulator --config gx8016 \
             --console --functional --gdb-port 9000 \
             --bm-debug-on-panic --debug-on-crash \
             --hvc ecos-sim.hvc \
             --mkboot-args -+- --no-strip -+- \
             --bme bmeapp=ecosapp.64
```

The **tile-monitor** options are largely the same as when running a debug system. `ecosapp.64` is used instead of the gdbstubs `gdb_module.64` file. There is no need to upload or launch LittleBoPeep since there are no gdbstubs instances for it to interact with, and if LittleBoPeep is not used then there is no need to set up TCP/IP tunnels between the host and target.

The TILE-Gx architectural HAL does provide a configuration option which may prove useful in this environment: `CYGHWR_HAL_TILEGX_SIMULATOR`. Enabling this option causes the eCos application to trigger a backdoor provided by the simulator. The simulation will be halted until **tile-gdb** is attached to the simulator, and information will be output on exactly how that should be done. However note that the executable file used should be the original 32-bit one, not the 64-bit pseudo-executable since that no longer contains any debug information. The resulting debug session will have limited debug functionality compared with the gdbstubs/LittleBoPeep solution, for example it will not support thread-aware debugging, and of course the simulator does not provide a full simulation of all the hardware. Never the less this does provide a limited way of debugging eCos code in a production system, which is not possible on real hardware.

Rebuilding gdbstubs and LittleBoPeep

A full release of eCos for the TILE-Gx architecture should include prebuilt binaries of gdbstubs and LittleBoPeep. If for any reason it is necessary to rebuild these, the process is straightforward. First, LittleBoPeep. The source code for this is found in the `host` subdirectory of the TILE-Gx architectural HAL package, `packages/hal/tilegx/arch`, together with a makefile. Simply running **make** inside that directory produce a new LittleBoPeep executable. To rebuild a gdbstubs executable, create a new directory and inside that directory run the following commands:

```
$ ecosconfig new tilegx stubs
U CYGDBG_HAL_COMMON_INTERRUPTS_SAVE_MINIMUM_CONTEXT, new inferred value 0
U CYGDBG_HAL_COMMON_CONTEXT_SAVE_MINIMUM, new inferred value 0
$ ecosconfig tree
$ make
...
```

At the end of the build the `install/bin` subdirectory will contain a 32-bit `gdb_module.img` executable and a 64-bit pseudo-executable `gdb_module.64`. The latter file is suitable for including in a hypervisor image in a debug system.

Name

Options — Configuring the TILE-Gx Architectural HAL Package

Loading and Unloading the Package

The TILE-Gx architectural HAL package CYGPKG_HAL_TILEGX will be loaded automatically when eCos is configured for a tilegx target, as will other hardware-specific packages such as the TMC support library. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The configuration option CYG_HAL_STARTUP can take one of two values: RAM and ROM. RAM startup is the default except when rebuilding a gdbstubs image. RAM startup should be used in a debug system when the application runs on top of gdbstubs, allowing the use of **tile-gdb**. Applications linked against a RAM-startup build of eCos cannot be incorporated into a hypervisor boot image. ROM startup should be used in a production system, and when rebuilding a gdbstubs image. Applications linked against a ROM-startup build of eCos should be incorporated into a hypervisor boot image and run on the target, not on top of gdbstubs.

There are two related options: CYGSEM_HAL_ROM_MONITOR and CYGSEM_HAL_USE_ROM_MONITOR. These options need to exist only to satisfy expectations elsewhere in the eCos source base. There should be no need to change the default values.

RAM Sizing

On a typical eCos target all available memory will be used by eCos. Often the amount of memory will be fixed at compile-time, but some targets support run-time memory sizing. This approach will not work on a TILE-Gx system: typically most of the memory will be needed by the Linux partition, and each BME tile running eCos will need its own allocation.

Instead the memory size of each BME tile is determined at the point that the hypervisor loads the ROM-startup executable held in the boot image. That executable will be gdbstubs in a debug system, or the eCos application in a production system. The memory consists of a ROM region holding the code and a RAM region for data. In a debug system gdbstubs will use only a small amount of the RAM region and the rest will be used for the code and data of the RAM-startup application that will be loaded and run via **tile-gdb**. The hypervisor will also allocate a small amount of additional memory at location 0x6c000000 for the startup stack and some data structures, and some memory at the top of the address space for interrupt vectors.

It is possible but non-trivial to increase the amount of memory available at run-time, using the shared memory techniques described in the TMC support package. Essentially a process running in the Linux partition would allocate a large block of memory and pass the details on to the eCos application, which can then map the block into its own address space. However this additional memory has to be managed entirely by the application. There is no easy way to add it to the eCos system heap.

For a ROM startup application or for gdbstubs the RAM size is controlled by the configuration option CYGNUM_HAL_TILEGX_RAM_SIZE, and has a default value of 4096K or 4 megabytes. The size can be increased or decreased to match the application's actual needs. However the hypervisor will perform rounding of the requested size. For RAM sizes up to 16MB the hypervisor will round the requested size up to 64K, 256K, 1MB, 4MB or 16MB. For RAM sizes larger than 16MB the hypervisor will allocate multiple blocks of 16MB each. eCos will use the amount of memory actually allocated by the hypervisor so there is no point in specifying an intermediate value size of say 8MB: the hypervisor will round this up to 16MB and eCos will use all 16MB.

For a RAM startup application changing CYGNUM_HAL_TILEGX_RAM_SIZE usually has little or no effect. eCos will use the actual amount of memory allocated by the hypervisor to gdbstubs, irrespective of the value of this option. However the option is used in the linker script to specify the size of the RAM region. If the total application code and static data requirements exceed the option's 4MB default value then the application will fail to link because the linker believes that only 4MB are available. In these circumstances it is possible to increase the option's value and thus allow the link to succeed. Obviously it will still be necessary to run a suitably-sized gdbstubs executable so that the hypervisor really will allocate the desired amount of memory.

For a production system, changing this configuration option in the ROM startup eCos application is straightforward. For a debug system things are a little more complicated: the option would have to affect the gdbstubs executable, not the application being debugged, since it is the former that is loaded by the hypervisor. Rebuilding a custom gdbstubs executable with a different RAM size is not hard, but it is annoying. As an alternative the RAM size can also be set when using **tile-ecos-32to64** to convert a 32-bit executable to a 64-bit pseudo-executable:

```
$ tile-ecos-32to64 -m 16384 gdb_module.img gdb_module_16MB.64
```

This creates a variant of the usual `gdb_module.64` with an 16384K or 16MB RAM size instead of the default 4MB, using the prebuilt `gdb_module.img` file shipped with the release.

Diagnostics

The option `CYGHWR_HAL_TILEGX_DIAGNOSTICS_DESTINATION` controls what happens to eCos console output. For a ROM startup application eCos output will be sent to the hypervisor over the IDN bus and will be displayed on the system console. For a RAM startup application eCos output will by default be sent to LittleBoPeep and then on to **tile-gdb**. However it is possible to redirect the console output to the hypervisor and hence the system console if desired.

System Clock

The configuration options **CYGNUM_HAL_RTC_NUMERATOR** and `CYGNUM_HAL_RTC_DENOMINATOR` control the frequency of the eCos system clock, or more precisely they are used to calculate the value programmed into the TILE_TIMER_CONTROL special purpose register. The number of nanoseconds between clock ticks is given by NUMERATOR/DENOMINATOR. The default value of the NUMERATOR is 1000000000, the number of nanoseconds in a second. Therefore simple clock frequencies can be achieved simply setting the DENOMINATOR to that frequency. For example the default 100Hz system clock is achieved by a DENOMINATOR value of 100. For more complicated clock frequencies the calculation could involve unacceptable rounding errors, and it may be necessary to change both the NUMERATOR and DENOMINATOR to avoid these.

When running on the simulator the default 100Hz clock requires a simulation of 10 milliseconds between clock ticks, and that may take a few tens of second of real time. If the application spends much of its time waiting for the next clock tick then performance may be greatly improved by running with a faster clock.

Simulator Support

Any ROM startup application including gdbstubs can be run on the Tilera simulator if desired, without changing any configuration options. However it is possible to build the system specifically for running on the simulator by enabling the configuration option `CYGHWR_HAL_TILEGX_SIMULATOR`. This enables some extra code in the eCos startup code which causes the system to pause until **tile-gdb** is connected and resumes execution. The simulator will output details of exactly how **tile-gdb** should connect, but note that the file being debugged should be the original 32-bit executable and not the 64-bit pseudo-executable generated by **tile-ecos-32to64**. Enabling the option also causes the system to halt if and when a non-recoverable double fault exception occurs, which may make debugging such faults slightly less difficult. Finally it changes the behaviour of some eCos testcases, causing them to run far fewer iterations than a testcase running on real hardware.

Compiler Flags

The package has two sets of configuration options related to compiler and linker flags. The first set consists of `CYGBLD_GLOBAL_CFLAGS` and `CYGBLD_GLOBAL_LDFLAGS`, plus supporting options `CYGBLD_GLOBAL_COMMAND_PREFIX`, `CYGBLD_LINKER_SCRIPT`, and `CYGHWR_MEMORY_LAYOUT` needed by other parts of the eCos source base. `CFLAGS` defines the default compiler flags that will be used for all packages, `LDFLAGS` the default linker flags. The most important flag is `-m32` to force a 32-bit build.

The second consists of `CYGPKG_HAL_TILEGX_CFLAGS_ADD` and `CYGPKG_HAL_TILEGX_CFLAGS_REMOVE`, plus supporting options `CYGPKG_HAL_TILEGX_TESTS` and `CYGBLD_HAL_TILEGX_BUILD_STUBS64`. `CFLAGS_ADD` is used to specify additional compiler flags that should be used when compiling the TILE-Gx architectural HAL package, and `CFLAGS_REMOVE` can be used to remove some of the global flags.

Name

HAL Port — Implementation Details

Description

This section of the documentation gives an outline description of the most important parts of the architectural HAL package, and especially how the eCos HAL specification has been mapped on to the TILE-Gx hardware.

The TILE-Gx HAL is organized somewhat differently from HALs for other targets. Typically an eCos port involves three or four separate HAL packages: the architectural HAL handles features that are common to every chip within an architecture; a variant HAL copes with different families within an architecture, for example one family may support only on-chip memory and no MMU while another family is designed for use with external memory; within a family there may be different processors supporting different sets of peripherals; finally a platform HAL handles anything specific to the circuit board rather than to the chip, for example the amount of external memory. The TILE-Gx port does not require these complications. Most of the hardware variations will be handled by the hypervisor or within the Linux partition and do not affect the eCos side of things.

Data Types

The eCos port to the TILE-Gx architecture only supports 32-bit mode. `int`, `long` and pointers are all 32 bits, and a `long long` is 64 bit. The chip always runs in little-endian mode.

Header Files

The architectural HAL package provides the standard HAL header files `cyg/hal/hal_arch.h`, `cyg/hal/hal_intr.h`, `cyg/hal/hal_io.h`, and so on. However there is one important difference between the TILE-Gx versions of these headers and their equivalents for other architectures. Typically `hal_intr.h` provides definitions of all the interrupt and exception vectors, directly or indirectly. Similarly `hal_io.h` provides definitions for some or all of the on-chip peripherals. These definitions are required by some parts of eCos, for example the kernel needs to know which interrupt corresponds to the system clock, but they are specific to individual processors or variants. The gcc toolchain supports the architecture as a whole so cannot supply these definitions.

For TILE-Gx the situation is different. The multicore development environment comes with a full set of definitions in various headers below the `arch` subdirectory, for example `arch/interrupts.h` defines all the interrupt vectors, and `arch/spr_def.h` defines the special purpose registers. **tile-gcc** will find these header files automatically. The headers are intended for use by the Linux kernel but are mostly usable by eCos and eCos applications. There may be some instances where these header files reference Linux-specific functionality, or where they assume a 64-bit build. Duplicating all this information in the eCos header files would serve no purpose.

Memory Layout and the Linker Script

The memory layout for a ROM startup application or for gdbstubs is as follows:

Location	Purpose
0x00010000	Code (.text section, read-only data)
0x10000000	Data (.data and .bss sections, eCos heap)
0x6C000000	Startup stack and Hypervisor Data
0xFFFFFFFFFE000000	Interrupt vectors

The amount of memory allocated for the ROM region is just large enough to hold the application's code and read-only data, plus enough for a shadow copy of the .data initialized static data section. That shadow copy is needed to allow for platform restarts. The hypervisor will round up this amount to a suitable page boundary.

eCos memory accesses go through the MMU so the above memory locations are translated to physical addresses. Therefore one eCos tile's location 0x10000000 will correspond to a different physical memory address from another tile's location 0x10000000, and these memory regions are not shared between tiles.

The amount of memory allocated for the RAM region is determined by the **CYGNUM_HAL_TILEGX_RAM_SIZE** configuration option, but can be overridden by passing a suitable *-m <size>* option to **tile-ecos-32to64** when the executable is converted into a format suitable for including into a hypervisor boot image. The first 4K are reserved for a system data structure `hal_tilegx_global_state` which contains information such as the virtual vector table, interrupt-related data, and MMU settings. In a debug system this data structure must be shared between the ROM-startup gdbstubs and the RAM-startup application, which is most conveniently done by placing it at a well-known location.

In a debug system the next 60K of the RAM region, up to location 0x10010000, is reserved for use by gdbstubs. The remaining memory holds the application code, data, and heap.

The hypervisor allocates some additional memory at location 0x6c000000 (strictly, at the pertile va location specified in the hypervisor configuration file, but that location should be 0x6c000000). The hypervisor assumes that the application is an ordinary BME application which will need memory for its stack and heap, and which will also need information from the hypervisor such as the tile's CPU speed. The eCos requirements are different but there is no easy way to return this memory to the hypervisor. Instead eCos can still make good use of it for the startup and interrupt stack, and it does need some of the same information from the hypervisor. If an application wishes to access this hypervisor information it can do so via `hal_tilegx_global_state.hv_global_info`, as defined in `cyg/hal/hal_tilegx.h`.

When an interrupt occurs the cpu branches to a location determined by the current cpu protection level and the interrupt vector number. eCos always runs at protection level 2, which means that the relevant locations occupy approximately 16K starting at location 0xFFFF_FFFF_FE00_0000. eCos needs to provide these interrupt vectors so an executable contains a section for this. Strictly the location can be changed by manipulating the INTERRUPT_VECTOR_BASE_2 special purpose register but there is no good reason for doing so.

The linker script `src/tilegx.ld` defines all the above, in conjunction with `pkgconf/hal_tilegx.h` and `pkgconf/mlt_tilegx.h`.

For a RAM startup application running on top of gdbstubs, the application's code will be placed at location 0x10010000 onwards, immediately after the memory reserved for `hal_tilegx_global_state` and gdbstubs. The application's static data will follow immediately after the code. The rest of the memory will be allocated to the system heap for dynamic memory allocation. A RAM startup application will run in the memory map set up by the hypervisor, so the RAM size is determined by the `CYGNUM_HAL_TILEGX_RAM_SIZE` option used when gdbstubs was configured, or alternatively by the memory size passed to **tile-ecos-32to64**. The RAM startup initialization code will determine the actual amount of RAM and size the system heap accordingly.

Startup

During bootstrap the hypervisor will initialize all BME tiles as per the configuration file, allocating memory as per the executable's memory map, then jumping to the executable's entry point. The hypervisor runs at protection level 2, and starts the executable with the same protection level. There is never any need for eCos to change this protection level, and doing so would introduce various complications especially in the interrupt handling code.

For ROM startup the application entry point is `hal_tilegx_start` in `src/vectors.S`. If the application has been started by the hypervisor then that will have already taken care of much of the low-level initialization, for example zeroing the .bss uninitialized static data region. However it is also possible for an application to perform a restart. This happens most commonly when a **maintenance packet r** command is issued from inside `tile-gdb` and the debug session is then terminated, but a restart can also be caused by a double fault exception or by using the `HAL_PLATFORM_RESET()` macro defined in `cyg/hal/hal_intr.h`. After a restart the assembler initialization code needs to do rather more work, including restoring all initialized static data to their original values, zeroing all uninitialized static data, and switching to an appropriate stack.

If eCos has been built with configuration option `CYGHWR_HAL_TILEGX_SIMULATOR` enabled and if it is actually running inside the simulator then the application will halt at this point, allowing the user to attach **tile-gdb**.

Once the assembler initialization code has finished it jumps to the C function hal_tilegx_c_startup(), defined in src/tilegx.c. This performs initialization or reinitialization of various other subsystems including the memory management unit's translation lookaside buffers (TLBs), interrupt handling, virtual vectors, and gdbstubs as appropriate. Finally it runs through any C++ static constructors, including those for other eCos packages like the eCos kernel, and calls the generic cyg_start() routine.

For RAM startup the application entry point is again hal_tilegx_start in src/vectors.S, but the code executed is somewhat different from that for ROM startup. Again there is a jump to hal_tilegx_c_startup() in src/tilegx.c, and from there to cyg_start().

Thread Contexts

The HAL_SavedRegisters structure defined in cyg/hal/hal_arch.h defines the storage needed for saving and restoring a thread context during context switches and interrupt handling. Mostly it consists of the registers r0-r53, but there is some additional state which overlaps the stack frames defined by the TILE-Gx ABI. The details are generally of no interest to application developers.

There is one piece of system state which is not held in the saved context structure and which arguably should be: the special purpose register SPR_CMPEXCH_VALUE. This register is not used in ordinary code. It serves only to help implement shared memory spinlocks:

```
      int result;
      __insn_mtspr(SPR_CMPEXCH_VALUE, oldval);
      result = __insn_cmpexch4(&spinlock, newval);
```

It is possible for an interrupt to occur between setting SPR_CMPEXCH_VALUE and applying the cmpexch4 or cmpexch instructions, and the register may get overwritten before the code resumes. The Linux kernel saves and restores this special purpose register during interrupt handling, adding several cycles to the interrupt latency. The eCos HAL does not save this register on the assumption, and instead the spinlock code has to disable interrupts around the above pair of instructions:

```
      CYG_INTERRUPT_STATE ints_state;
      int result;
      HAL_DISABLE_INTERRUPTS(ints_state);
      __insn_mtspr(SPR_CMPEXCH_VALUE, oldval);
      result = __insn_cmpexch4(&spinlock, newval);
      HAL_RESTORE_INTERRUPTS(ints_state);
```

This makes interrupt handling more efficient but spinlocks more expensive. Since eCos does not support SMP operations spinlocks are unlikely to be used often, and it is expected that this approach will be a net performance gain.

Interrupts

Interrupt management requires several pieces of functionality. First it must be possible to disable and reenable interrupts, so that critical code sections can run atomically. Second it must be possible to mask and unmask individual interrupt sources. Third, if support for nested interrupts is enabled via the configuration option CYGSEM_HAL_COMMON_INTERRUPTS_ALLOW_NESTING then it should be possible to assign priorities to the various interrupts, such that inside an interrupt handler lower priority interrupts are masked and higher priority interrupts are unmasked. Finally other eCos code including the kernel and any device drivers must be able to register their own interrupt handling functions. There are two versions of such handlers: a low-level VSR must be written in assembler, but is called very early after an interrupt triggers; a higher-level ISR can be written in C, but the system needs to do more work before the ISR can be called.

Each TILE-Gx tile has two special purpose registers or SPRs which control how interrupts are handled. INTERRUPT_MASK_2 can be used to mask or unmask the various interrupt sources (there are other registers for protection levels 0, 1, and 3 but those are irrelevant to the eCos port). INTERRUPT_CRITICAL_SECTION can be used to block all maskable interrupts. At first glance this second register could be used to implement the disable/reenable functionality. Unfortunately that does not quite work. If a CPU exception occurs while INTERRUPT_CRITICAL_SECTION is set then that is treated as a non-recoverable double fault. Since gdbstubs depends on CPU exceptions for some of the debug functionality, the implementation takes a different approach.

During normal execution INTERRUPT_MASK_2 holds the set of all interrupts that are currently masked, as expected. A shadow copy of this set is held in the global hal_tilegx_global_state.global_interrupt_mask. Disabling interrupts involves setting INTERRUPT_MASK_2 to 0xFFFF_FFFF_FFFF_FFFF, and reenabling interrupts involves restoring INTERRUPT_MASK_2 as per the shadow copy.

The above explanation is actually oversimplified. Implementing prioritized nested interrupts requires some additional complications. Associated with each interrupt source is an interrupt mask holding the set of all interrupts with equal or lower priorities. There are also two pseudo-interrupt sources, none and disabled, with associated masks 0 and 0xFFFF_FFFF_FFFF_FFFF. At any time the value of the INTERRUPT_MASK_2 SPR is the union of the global interrupt mask and the current interrupt's mask. During normal execution the current interrupt is none so INTERRUPT_MASK_2 holds the same value as the global interrupt mask. When interrupts are disabled the current interrupt is disabled so INTERRUPT_MASK_2 holds 0xFFFF_FFFF_FFFF_FFFF. While processing an interrupt INTERRUPT_MASK_2 holds all globally masked interrupts and all interrupts masked for the current interrupt. Keeping everything up to date in the right order requires considerable care, but achieves the desired functionality.

Assuming an unmasked interrupt triggers, the hardware jumps to location 0xFFFF_FFFF_FE00_0000 + (0x100 * interrupt_number). The ROM startup executable or gdbstubs provides the code that resides at that location, as per the macro intvec in src/vectors.S. This initial code allocates space for a HAL_SavedRegisters structure on the stack , saves a small number of registers, loads a per-interrupt VSR function pointer from hal_tilegx_global_state, and jumps to that VSR. Usually that VSR will be hal_default_interrupt_vsr, again in src/vectors.S, but applications can install their own VSR functions if interrupt latency is particularly critical for an interrupt source. Any such VSR is likely to be based at least in part on the default one.

The default VSR saves additional registers, updates the current interrupt field in hal_tilegx_global_state and the INTERRUPT_MASK_2 SPR, synchronizes with the kernel, and enables nested interrupts. It then calls the ISR associated with the current interrupt. ISRs can be written in C but there are constraints on what they are allowed to do. More information on this is provided in the kernel documentation. When the ISR returns the VSR performs additional processing, possibly including a context switch to a higher-priority thread that is now runnable, before eventually returning to the interrupted code.

CPU Exceptions

On TILE-Gx exceptions like SIGILL, an illegal instruction exception, are implemented in much the same way as interrupts. However exceptions cannot be masked. The CPU jumps to a location near 0xFFFF_FFFF_FE00_0000, where the ROM startup executable or gdbstubs will have placed suitable code. That code jumps to a VSR, which this time will usually be hal_default_exception_vsr instead of hal_default_interrupt_vsr. This in turn calls hal_tilegx_exception_handler() in src/tilegx.c which will usually deliver the exception to the kernel. There are various special cases, for example a SIGILL exception may be the result of hitting a **tile-gdb** breakpoint.

One of the exceptions is special: double fault. This occurs when a CPU exception occurs while the INTERRUPT_CRITICAL_SECTION SPR is set. Double faults are not recoverable: critical information held in other SPRs will have been overwritten. If running in the simulator and CYGHWR_HAL_TILEGX_SIMULATOR is set then the simulation will be halted. Otherwise, in the absence of a better solution, an attempt will be made to restart the system. The structure field hal_tilegx_global_state.started_by will be set to hal_tilegx_started_by_double_fault, allowing application code to detect this after the restart and take any action that might be appropriate. Double faults should be rare, but application developers should be aware of the possibility.

The Idle Thread

The kernel's idle thread will execute the nap instruction, causing the tile to sleep until the next interrupt occurs.

The System Clock

The kernel clock has been implemented using the TILE_TIMER_CONTROL special purpose register, so that hardware is not available for use by application code. The auxiliary tile timer is available for use by the application, but note that the simulator does not implement that timer.

The counter value programmed into the TILE_TIMER_CONTROL register is determined from the system clock frequency, which is information provided by the hypervisor, and from the configuration options CYGNUM_HAL_RTC_NUMERATOR and

`CYGNUM_HAL_RTC_DENOMINATOR`. The hardware does not support automatic reloading of the counter when a clock interrupt occurs, so the interrupt handler has to reload it explicitly. That code attempts to compensate for the time taken from the interrupt triggering to the counter being reloaded. The accuracy cannot be completely guaranteed, especially in a debug system, so a small amount of clock drift may occur.

The Cache

The TILE-Gx architecture has a complicated caching system including per-tile primary and secondary caches and a distributed tertiary cache. The hardware maintains data cache coherency so there is very rarely any need for code to exercise fine-grained control over the cache such as flushing cachelines. Therefore the various cache-related eCos macros like `HAL_DCACHE_SYNC()` are defined as no-ops. The hardware does not maintain coherency between the instruction and data cache so eCos does define a number of instruction cache macros like `HAL_ICACHE_INVALIDATE()`. These are needed by the gdbstubs code to implement breakpoints, and are unlikely to be of any interest to application developers.

Diagnostics

The port supports two destinations for diagnostic output. Applications built for ROM startup will send their diagnostic output to the hypervisor over the IDN bus, and the hypervisor will output the text on the system console. For a RAM startup application diagnostic output will normally be sent to **tile-gdb** via LittleBoPeep, but the output can be redirected to the hypervisor if desired. This behaviour is controlled by the `CYGHWR_HAL_TILEGX_DIAGNOSTICS_DESTINATION` configuration option.

Other Functionality

The TILE-Gx architectural HAL provides two non-standard functions which can be used to manage the MMU settings:

```
#include <cyg/hal/hal_tilegx.h>

int hal_tilegx_mmap(unsigned long long virtual_address,
                    unsigned long long physical_address,
                    unsigned long long dtlb_attributes);

void hal_tilegx_munmap(unsigned long long virtual_address);
```

`hal_tilegx_mmap()` can be used to map a physical address into the tile's virtual address space with the specified attributes. The physical address can correspond to real memory. Typically this will be allocated by a process running in the Linux partition, and the details can then be forwarded to an eCos application which will map it into its address space. Alternatively the physical address can correspond to a memory-mapped device. The virtual address can be anywhere in the address space that is not already used, but preferably in the range 0x0000_0000 to 0x7FFF_FFFF to avoid problems with 32-bit pointers. Low memory is normally used for the system's ROM and RAM regions and 0x6C00_0000 is used for the hypervisor data, but anywhere between 0x4000_0000 to 0x6800_0000 or 0x7000_0000 to 0x7FFF_FFFF is normally fine. The address should be aligned to a boundary suitable for the block size. The final argument will be written to the `DTLB_CURRENT_ATTR` SPR and consists of numerous fields. The Tilera documentation should be consulted for more information. `hal_tilegx_mmap()` returns 0 if the operation fails, typically because all of the data TLBs are already in use, or 1 on success.

`hal_tilegx_munmap()` can be used to undo a previous `hal_tilegx_mmap()` call.

Chapter 358. TILE-Gx TMC Library

Name

Overview — eCos Port of a Subset of the TMC Library

Description

The Tilera Multicore Components or TMC Library provides a variety of primitives for building parallel programs. It is documented in UG527, “The Applications Libraries Reference Manual”. The eCos package `CYGPKG_HAL_TILEGX_TMC` implements a subset of this library. The subset supports communication between Linux applications running on some of the tiles on a TILE-Gx chip and eCos applications running on other tiles. The package also contains a number of example applications demonstrating the communication functionality.

The package `CYGPKG_HAL_TILEGX_TMC` is automatically included in any configuration for a TILE-Gx target. It does not have to be added to the configuration. The package does not add any overhead to eCos applications which do not use any of its functionality, so there is no reason for ever removing the package from the configuration.

The Tilera TMC library consists of the following components:

1. UDN helper routines for communication over the UDN bus. Most of these have been ported to eCos.
2. Performance tuning. The routine `tmc_perf_get_cpu_speed()` is implemented.
3. Spinning shared memory synchronization primitives. These have all been ported to eCos.
4. Scheduler shared memory synchronization primitives. These have not been ported. The primitives interact with the Linux kernel, and there is no way for an eCos application running on a BME tile to do that.
5. Specified-attribute memory page allocation. These have not been ported. Allocating memory pages involves calling into the hypervisor and the hypervisor is no longer present on BME tiles running eCos. However it is possible for a Linux application to allocate one or more pages, pass the details on to an eCos application, and have the latter map the pages into its address space.
6. Common memory, allowing pages to be mapped at the same virtual address in different processes. These have not been ported. Linux applications run in 64-bit mode with a 64-bit address space, whereas eCos applications run in 32-bit mode with 32-bit addresses. This makes it difficult to use the same virtual addresses.
7. CPU sets and affinitization. These have not been ported. The primitives are intended to allow threads to be bound to specific tiles. Since each eCos instance runs on only one tile there is no point in attempting to support such affinitization.
8. User space interrupt installation routines. These have not been ported. eCos has its own model of how interrupts should be handled and its own routines for managing interrupts. Trying to support the TMC's model of interrupt handling as well would complicate things for little or no gain.
9. Interprocessor interrupt event-handling. These have not been ported. It is not clear that they are actually useful since there is no primitive for generating an IPI interrupt. Instead UDN communications provides a way of sending data asynchronously to another tile, and if desired that UDN communication can be processed by an interrupt handler.
10. Using mspaces for standard malloc/free. These have not been ported. Under Linux they provide an alternative implementation of the C library's `malloc()` and `free()` routines, offering some control over home caching and memory page sizes. For eCos applications it makes more sense to use the standard eCos heap using memory provided by the hypervisor during startup.
11. Cache control. These functions have not been ported, and the Tilera documentation recommends against using these low-level shared memory primitives. The main functionality provided, memory fences to guarantee visibility of stores to cache coherent memory, is instead provided by the eCos `HAL_MEMORY_BARRIER()` macro.
12. Multiple heap allocation. These routines allow for the allocation of separate mspaces with control over cache homing and other functionality. They involve interaction with the hypervisor's memory page support which is not possible for an eCos application.

13. Task management and cleanup. These primitives relate to support for multiple processes and interaction with the Tilera shepherd process. Since eCos does not support multiple processes, only multiple threads, the primitives are not applicable.

The package only provides implements of the header files `<tmc/udn.h>`, `<tmc/spin.h>`, and `<tmc/perf.h>`. The other header files do exist but will generate a compile-time warning if they are included. Providing these dummy headers prevents the compiler from accidentally including the Linux TMC headers.

Restrictions

The eCos TMC support is subject to a number of important restrictions which application developers must be aware of.

When a Linux process uses `tmc_alloc_map()` or a similar routine to allocate a block of memory, that memory is owned by the Linux process. Details of the block including the physical address can be passed on to eCos applications which can then map it into their address space. If the Linux process exits or gets killed off the Linux kernel and hypervisor will reclaim the allocated block, which may then get reused for some other Linux process or for the kernel or hypervisor itself. Meanwhile the eCos application may still have a mapping to the underlying physical memory and may still write to it, corrupting memory that now belongs to some random other part of the system. Neither the hypervisor nor the Linux kernel have any way of keeping track of what memory has been mapped into an eCos application's address space, so they cannot do anything to avoid this problem.

The only solution is to make sure that the eCos application is always informed when the Linux process exits, so that it can unmap any shared memory pages. That is not always easy to achieve, especially in a debug environment, but it is the application developer's responsibility.

Separately, UDN communication is subject to a major restriction. By default a Linux application does not have access to the UDN bus, and must explicitly obtain such access from the Linux kernel by a call to `tmc_udn_init()`. The kernel only grants UDN access to one task per tile. In a debug environment LittleBoPeep runs on one of the tiles providing gdb debug functionality for eCos tiles, and LittleBoPeep needs to use the UDN bus for this. Therefore Linux applications requiring UDN access cannot run on the same tile as LittleBoPeep, usually the highest-numbered tile not used for BME.

UDN Support

The following UDN routines are supported:

`tmc_udn_header_from_cpu()`
`tmc_udn_send_buffer()`
`tmc_udn0_receive_buffer()`, `tmc_udn1_receive_buffer()` and `tmc_udn2_receive_buffer()`
`tmc_udn_send()`
`tmc_udn_send_1()` to `tmc_udn_send_20()`
`tmc_udn0_receive()` to `tmc_udn2_receive()`
`tmc_udn0_available_count()` to `tmc_udn2_available_count()`
`tmc_udn_available_mask()`

The functions related to UDN channel 3, `tmc_udn3_receive_buffer()`, `tmc_udn3_receive()` and `tmc_udn3_available_count()` are not supported. UDN channel 3 is used for communication between LittleBoPeep and gdbstubs, and if application code tried to use this channel as well then things would get very confusing with UDN traffic going to the wrong program.

There are four other UDN functions in the Tilera TMC library which are not supported under eCos: `tmc_udn_init()`, `tmc_udn_close()`, `tmc_udn_activate()`, and `tmc_udn_persist_after_exec()`. Under Linux `tmc_udn_init()` is needed to request access to the UDN bus from the Linux kernel. That is not necessary under eCos since eCos applications run at protection level 2, which is sufficient for UDN access. The three other functions are also related to access rights and are equally unnecessary.

Spinning Shared Memory Synchronization

The following are supported:

tmc_spin_mutex_t

```
TMC_SPIN_MUTEX_INIT                       tmc_spin_mutex_init()
tmc_spin_mutex_lock()                     tmc_spin_mutex_trylock()
tmc_spin_mutex_unlock()
```

tmc_spin_queued_mutex_t

```
TMC_SPIN_QUEUED_MUTEX_INIT                tmc_spin_queued_mutex_init()
tmc_spin_queued_mutex_lock()              tmc_spin_queued_mutex_trylock()
tmc_spin_queued_mutex_unlock()
```

tmc_spin_rwlock_t

```
TMC_SPIN_RWLOCK_INIT                      tmc_spin_rwlock_init()
tmc_spin_rwlock_rdlock()                  tmc_spin_rwlock_wrlock()
tmc_spin_rwlock_tryrdlock()               tmc_spin_rwlock_trywrlock()
tmc_spin_rwlock_rdunlock()                tmc_spin_rwlock_wrunlock()
tmc_spin_rwlock_unlock()
```

tmc_spin_barrier_t

```
TMC_SPIN_BARRIER_INIT()                   tmc_spin_barrier_init()
tmc_spin_barrier_wait()
```

These data types and functions have the same semantics as the Tilera TMC versions, so the Tilera documentation can be consulted for more details.

An important point about these routines is that the various unlock functions and the barrier wait function involve a memory barrier, guaranteeing that all memory writes are visible to other tiles. Therefore code that only manipulates shared data while owning a lock automatically avoids many memory consistency problems.

Example Applications

The package comes with a number of example applications in the `examples` subdirectory:

`udn0`

This example illustrates UDN communication between a TILE-Gx Linux application and one or more instances of an eCos application.

`udn1`

This example is derived from `udn0`. It adds direct communication between the instances of the eCos application.

`shm0`

This example sets up a block of shared memory between a Linux application and one or more instances of an eCos application. The shared memory is used to hold large amounts of data. The Linux application sends UDN messages to control what each eCos instance does with that shared data.

`shm1`

This example also sets up a block of shared memory. However UDN communication is used only during initialization, to set up the shared memory. All subsequent communication between Linux and eCos goes via the shared memory.

`spintest`

This is a testcase for the various spinning shared memory synchronization primitives.

`not61850`

This is an example involving a Linux host application, a TILE-GX Linux application, and one or more instances of an eCos worker application. It combines ethernet traffic using the gxio/mpipe library routines running in the TILE-Gx Linux application, and communication between that and the eCos workers over shared memory and the UDN bus. It was written to a specific customer's requirements and may be of limited interest to other users.

Real-time characterization of selected targets