eCos™ Reference Manual

September 2000

Copying terms

Trademarks

300-400-1010049-03

Contents

Part I: Preliminaries

1

eCos kernel overview

This is an overview of the internal workings of the **eCos**™ kernel.

The scheduler

At the core of the kernel is the scheduler. This defines the way in which threads are run, and provides the mechanisms by which they may synchronize. It also controls the means by which interrupts affect thread execution. No single scheduler can cover all possible system configurations. For different purposes we will need to cover several scheduling polices. In this release two schedulers are provided (described in more detail in “Sched subdirectory” on page 10):

- a *bitmap scheduler*
- a *multi-level queue scheduler*

At present the system will only support a single scheduler at any one time. Future systems may allow multiple schedulers to co-exist, but this will be hidden behind the scheduler API in the current release.

To make scheduling safe we need a mechanism to protect the scheduler data structures from concurrent access. The traditional approach to this is to disable interrupts during the critical regions. Unfortunately this increases the maximum interrupt dispatch latency, which is to be avoided in any real-time system.

The mechanisms chosen for **eCos** is to maintain a counter, **`Scheduler::sched_lock`**

that, if non-zero, prevents any rescheduling. The current thread can claim the lock by calling **`Scheduler::lock()`**. This increments the counter and prevents any further scheduling. The function **`Scheduler::unlock()`** decrements the counter and if it returns to zero, allows scheduling to continue.

For this to work in the presence of interrupts, it is necessary for the Interrupt Service Routines (ISR) to defer any scheduler-oriented operations until the lock is about to go zero. We do this by splitting the work of an ISR into two parts, with the second part, the Deferred Service Routine (DSR), being queued until the scheduler decides it is safe to run. This is covered in more detail in “Interrupts” on page 5 and “Interrupt and exception handlers” on page 18.

On a uni-processor, **`Scheduler::lock()`** is a simple increment of Scheduler::sched_lock. It does not need to be a read-modify-write cycle since the lock is strictly nested. The mere fact that the current thread is running implies that the lock has not been claimed by another thread, so it is always claimable.

`Scheduler::unlock()` is generic to all scheduler implementations.

Thread synchronization

To allow threads to cooperate and compete for resources, it is necessary to provide mechanisms for synchronization and communication. The classic synchronization mechanisms are mutexes/condition variables and semaphores. These are provided in the **eCos** kernel, together with other synchronization/communication mechanisms that are common in real-time systems, such as event flags and message queues.

One of the problems that must be dealt with in any real-time systems is priority inversion. This is where a high priority thread is (wrongly) prevented from continuing by one at lower priority. The normal example is of a high priority thread waiting at a mutex already held by a low priority thread. If the low priority thread is preempted by a medium priority thread then priority inversion has occurred since the high priority thread is prevented from continuing by an unrelated thread of lower priority.

This problem got much attention recently when the Mars Pathfinder mission had to reset the computers on the ground exploration robot repeatedly because a priority inversion problem would cause it to hang.

There are several solutions to this problem. The simplest is to employ a priority ceiling protocol where all threads that acquire the mutex have their priority boosted to some predetermined value. This has a number of disadvantages: it requires the maximum priority of the threads using the mutex to be known in advance; if the ceiling priority is too high it acts as a global lock disabling all scheduling and it is pessimistic, taking action to prevent the problem even when it does not arise.

A better solution is to use priority inheritance protocol. Here, the priority of the thread

that owns the mutex is boosted to equal that of the highest priority thread that is waiting for it. This technique does not require prior knowledge of the priorities of the threads that are going to use the mutex, and the priority of the owning thread is only boosted when a higher priority thread is waiting. This reduces the effect on the scheduling of other threads, and is more optimistic than the priority ceiling protocol. A disadvantage of this mechanism is that the cost of each synchronization call is increased since the inheritance protocol must be obeyed each time.

A third approach to priority inversion is to recognize that relative thread priorities have been poorly chosen and thus the system in which it occurs is faulty. In this case the kernel needs the ability to detect when priority inversion has taken place, and to raise an exception when it occurs to aid debugging. Then this code is removed from the shipping version.

The current **eCos** release provides a relatively simple implementation of mutex priority inheritance. This implementation will only work in the multi-level queue scheduler, and it does not handle the rare case of nested mutexes completely correctly. However it is both fast and deterministic. Mutex priority inheritance can be disabled if the application does not require it. This will reduce both code size and data space.

Future releases will provide alternative implementations of mutex priority inheritance, and application developers will be able to choose the implementation appropriate to their application.

Exceptions

An exception is a synchronous event caused by the execution of a thread. These include both the machine exceptions raised by hardware (such as divide-by-zero, memory fault and illegal instruction) and machine exceptions raised by software (such as deadline overrun). The standard C++ exception mechanism is too expensive to use for this, and in any case has the wrong semantics for the exception handling in an RTOS.

The simplest, and most flexible, mechanism for exception handling is to call a function. This function needs context in which to work, so access to some working data is required. The function may also need to be handed some data about the exception raised: at least the exception number and some optional parameters.

The exception handler receives a data argument which is a value that was registered with the handler and points to context information. It also receives an exception_number which identifies the exception taken, and an error code which contains any additional information (such as a memory fault address) needed to handle the exception. Returning from the function will allow the thread to continue.

Exception handlers may be either global or per-thread, or both, depending on

configuration options. If exceptions are per-thread, it is necessary to have an exception handler attached to each thread.

Interrupts

Interrupts are asynchronous events caused by external devices. They may occur at any time and are not associated in any way with the thread that is currently running.

The handling of interrupts is one of the more complex areas in RTOS design, largely because it is the least well defined. The ways in which interrupt vectors are named, how interrupts are delivered to the software and how interrupts are masked are all highly architecture- (and in some cases board-) specific. The approach taken in **eCos** is to provide a generalized mechanism with sufficient hooks for system-specific code to be inserted where needed.

Let us start by considering the issue of interrupt vectors. Hardware support differs greatly here: from the Intel Architecture and the 680X0 having support for vectoring individual interrupts to their own vectors, to most RISC architectures that only have a single vector. In the first case it is possible to attach an ISR directly to the vector and know that it need only concern itself with the device in question. In the second case it is necessary to determine which device is actually interrupting and then vector to the correct ISR. Where there is an external interrupt controller, it will be possible to query that and provide what is essentially a software implementation of hardware vectoring. Otherwise the actual hardware devices must be tested, by calling the ISRs in turn and letting them make the determination. Since it is possible for two devices to interrupt simultaneously, it is necessary to call all ISRs each time an interrupt occurs.

Interrupt masking has a similar variety of support. Most processors have a simple interrupt mask bit in a status register. The 680X0 has seven levels of masking. Any board with a interrupt controller can be programmed to provide similar multi-level masking. It is necessary to keep the interrupt masking mechanism simple and efficient, and use only architectural support. The cost of manipulating an on-board interrupt controller may be too high. However, individual device drivers may want access to their individual mask bits in the interrupt controller, so support for this must be provided.

Most of the infrastructure necessary for a (somewhat) portable treatment of interrupts is implemented in the **eCos** Hardware Abstraction Layer (HAL), which is documented in “The eCos Hardware Abstraction Layer (HAL)” on page 66.

Counters, clocks, alarms and timers

If the hardware provides a periodic clock or timer, it will be used to drive timing-related features of the system. Many CPU architectures now have built in timer registers that can provide a periodic interrupt. This should be used to drive these features where possible. Otherwise an external timer/clock chip must be used.

We draw a distinction between Counters, Clocks, Alarms and Timers. A Counter maintains a monotonically increasing counter that is driven by some source of ticks. A Clock is a counter driven by a regular source of ticks (i.e. it counts time). Clocks have a *resolution* associated with them. A default system Clock is driven by the periodic interrupt described above, and tracks real-time. Other interrupt sources may drive other Counters that may or may not track real-time at different resolutions. Some Counters may be driven by aperiodic events and thus have no relation to real-time at all.

An Alarm is attached to a Counter and provides a mechanism for generating single-shot or periodic events based on the counter's value. A Timer is simply an Alarm that is attached to a Clock.

The system (including the kernel) represents time in units of *ticks*. These are clock-specific time units and are usually the period of the timer interrupt, or a multiple thereof. Conversion of ticks into conventional time and date units should occur only when required via library functions. Equivalence between Clock time and real-time can be made with an RTC (real-time clock), NTP (network time protocol) or user input.

The representation of the current tick count needs to be 64 bit. This requires either compiler support for 64 bit integers, or assembly code. Even at the extreme of a 1 ns tick (ticks will typically be >1ms), this gives a 584 year rollover period.

The Clock API and configuration options that affect clock, counter and alarm behavior are described in detail in "Counters, clocks and alarms" on page 34.

2

A tour of the kernel sources

This description takes the form of a tour around the sources explaining their structure and describing the functionality of each component.

The kernel is divided into two basic parts, the largely machine independent parts in `Cygnus/eCos/packages/kernel/v1_3_x`, and the architecture- and platform-specific parts that comprise the Hardware Abstraction Layer (HAL) in `Cygnus/eCos/packages/hal`. These will be described separately. Also note that the HAL is described in great detail in its own chapter (see "The eCos Hardware Abstraction Layer (HAL)" on page 66).

Kernel headers

Kernel header files (in `Cygnus/eCos/packages/kernel/v1_3_x/include`) provide external interfaces and configuration control for the various kernel objects. In general there is an include file for each major kernel class. Those header files having to do with configuration live in the `pkgconf` subdirectory.

The base name of a header file and the source file that implements it are usually the same. So, for example, the member functions defined in `sched.hxx` are implemented in `sched.cxx`. For a number of classes there are also header files that define inline functions, for example `sched.inl`.

There are some kernel objects that are implemented using C++ templates to allow

code re-use in future; it is not intended that these template classes be used generally by applications. The appropriate concrete kernel classes should be used instead.

Now we examine the files one by one for reference:

`include/bitmap.hxx`

Bitmap scheduler definition. See source file `sched/bitmap.cxx`

`include/clock.hxx`,

`include/clock.inl`

Counter, clock and alarm functions. See source file `common/clock.cxx`

`include/diag.h`

Diagnostic routines. See source file `trace/diag.c`

`include/errors.h`

Kernel error codes. See source file `common/except.cxx`

`include/except.hxx`

Exception handling.

`include/flag.hxx`

Flag synchronization objects. See source file `sync/flag.cxx`

`include/instrmnt.h`

Instrumentation. See source file `instrmnt/meminst.cxx`

`include/intr.hxx`

Interrupts. See source file `intr/intr.cxx`

`include/kapi.h`,

`include/kapidata.h`

Native 'C' API to the kernel. See source file `common/kapi.cxx`

`include/ktypes.h`

Kernel types.

`include/llistt.hxx`

A simple doubly linked-list template class used elsewhere in the kernel.

`include/lottery.hxx`

Not used.Lottery scheduler implementation. See source file `sched/lottery.cxx`

`include/mbox.hxx`,

`include/mboxt.hxx`,

`include/mboxt2.hxx`,

`include/mboxt.inl`,

`include/mboxt2.inl`

Message boxes. See source file `sync/mbox.cxx`; `mboxt.hxx` and `mboxt2.hxx` and `mboxt.inl` and `mboxt2.inl` implement the underlying template function.

NOTE The files with a 2 suffix are used by default and provide precise µITRON semantics.

`include/memfixed.hxx`,

`include/mempoolt.hxx`,

`include/mempolt2.hxx`,

`include/mempoolt.inl`,

`include/mempolt2.inl`,

`include/mfiximpl.hxx`,

`include/mfiximpl.inl`

Fixed-block allocation memory pools. See source file `mem/memfixed.cxx`; mempoolt[2] and mfiximpl are a thread-safety template function and underlying memory manager respectively.

NOTE The files with a 2 suffix are used by default and provide precise µITRON semantics.

`include/memvar.hxx`,

`include/mempoolt.hxx`,

`include/mempolt2.hxx`,

`include/mempoolt.inl`,

`include/mempolt2.inl`,

`include/mvarimpl.hxx`,

`include/mvarimpl.inl`

Variable-block allocation memory pools. See source file `mem/memvar.cxx`; mempoolt[2] and mvar are a thread-safety template function and underlying memory manager respectively.

NOTE The files with a 2 suffix are used by default and provide precise µITRON semantics.

`include/mlqueue.hxx`

Multi-level queue scheduler. See source file `sched/mlqueue.cxx`

`include/mutex.hxx`

Mutexes. See source file `sync/mutex.cxx`

`include/sched.hxx`,

`include/sched.inl`

General scheduler functions. See source file `sched/sched.cxx`

`include/sema.hxx`,

`include/sema2.hxx`

Semaphores. See source files `sync/cnt_sem.cxx` and `sync/bin_sem.cxx` for counting or binary semaphores respectively.

NOTE The file with a 2 suffix is used by default and provides precise µITRON semantics.

`include/thread.hxx`,

`include/thread.inl`

Threads, regardless of scheduler. See `common/thread.cxx`

Kernel source files

The kernel source directory (`Cygnus/eCos/packages/kernel/v1_3_x/src`) is divided into a number of sub-directories each containing the source files for a particular kernel subsystem. These sources divide into two classes: those that are generic to all configurations, and those that are specific to a particular configuration.

Sched subdirectory

`sched/sched.cxx`

This contains the implementation of the base scheduler classes. The most important function here is **`Cyg_Scheduler::unlock_inner()`** which runs DSRs and performs any rescheduling and thread switching.

`sched/bitmap.cxx`

This contains the bitmap scheduler implementation. It represents each runnable thread with a bit in a bitmap. Each thread must have a unique priority and there is a strict upper limit on the number of threads allowed.

`sched/mlqueue.cxx`

This contains the multi-level queue scheduler implementation. It implements a number of thread priorities and is capable of timeslicing between threads at the same priority. This scheduler can also support priority inheritance.

`sched/lottery.cxx`

This contains a lottery scheduler implementation. This implements a CPU share scheduler based on threads holding a number of lottery tickets. At the start of each time quantum, a random number is generated and the thread holding the matching ticket is scheduled. Compensation tickets and ticket donation allow fair sharing

for I/O bound threads and an equivalent mechanism to priority inheritance.

NOTE This scheduler is experimental, and is meant to test the behavior of other parts of the kernel with a non-orthodox scheduler. It is not meant to be used for real applications. It is currently under development and is incomplete and unusable.

Common subdirectory

`common/thread.cxx`

This implements the basic thread classes. The functions in this file implement the basic thread controls to sleep and wake threads, change priorities and delay and time-out. Also defined here is the idle thread that runs when there is nothing else to do.

`common/clock.cxx`

This implements the counter, clock and alarm functions. Also defined here is the system real-time clock that is used to drive timeslicing, delays and time-outs.

`common/kapi.cxx`

This implements a C API to the basic kernel functions.

`common/memcpy.c`,

`common/memset.c`

Standard ANSI memcpy and memset operations; these are here because the compiler may invoke them for structure operations regardless of the presence of a C library.

Interrupt subdirectory

`intr/intr.cxx`

This implements the Interrupt class. Most of this code is concerned with posting and calling DSRs. The remainder of the interrupt handling code is machine specific and is in `hal_intr.cxx` in the HAL directory.

Synchronization subdirectory

`sync/mutex.cxx`

This contains the implementation of mutexes and condition variables. Mutexes can optionally be configured to use a priority inheritance mechanism supplied by the scheduler.

`sync/cnt_sem.cxx`

This contains the implementation of counting semaphores.

`sync/cnt_sem2.cxx`

This contains the alternate implementation of counting semaphores which implements precise µITRON semantics.

`sync/bin_sem.cxx`

This contains the implementation of binary semaphores.

`sync/mbox.cxx`

This contains wrapper functions for a message box of (void *) values. The implementation is the template defined in `include/mboxt.hxx` which `include/mboxt.inl` implements in turn. Message boxes exist in the kernel specifically to support µITRON compatibility.

`sync/flag.cxx`

This contains the implementation of flag objects. Flag objects exist in the kernel specifically to support µITRON compatibility.

Memory management subdirectory

`mem/memfixed.cxx`

This contains the wrapper functions for a fixed-block allocation memory manager. The actual implementation is in two parts: `include/mfiximpl.hxx` implements the fixed-block memory management algorithms, and template `include/mempoolt.hxx` implements thread safety and waiting for memory management classes. These are combined in `memfixed.cxx`. Memory pools exist in the kernel specifically to support µITRON compatibility.

`mem/memvar.cxx`

This contains the wrapper functions for a variable-block allocation memory manager. The actual implementation is in two parts: `include/mvarimpl.hxx` implements the variable-block memory management algorithms, and template `include/mempoolt.hxx` implements thread safety and waiting for memory management classes. These are combined in `memvar.cxx`. Memory pools exist in the kernel specifically to support µITRON compatibility.

Instrumentation subdirectory

`instrmnt/meminst.cxx`

This contains an implementation of the instrumentation mechanism that stores instrumentation records in a circular buffer in memory. The size of this buffer is configurable. The instrumentation flags mechanism allows the generation of instrumentation records to be controlled on a per-record basis. The header file `cyg/kernel/instrmnt.h` contains macros to generate instrumentation records in various places, and may be configured to only generate instrumentation records where required.

`instrmnt/nullinst.cxx`

This contains an implementation of the instrumentation mechanism that does nothing. By substituting its object file `nullinst.o` for `meminst.o` in a build, the instrumentation mechanism may be disabled without recompiling.

Trace subdirectory

`trace/simple.cxx`

This contains an implementation of the trace and assert mechanisms that output textual messages via a set of externally defined functions. These are currently supplied by the code in `trace/diag.c` but may be supplied by a device driver in the future.

`trace/fancy.cxx`

This contains a (fancier) implementation of the trace and assert mechanisms that output textual messages via a set of externally defined functions. These are currently supplied by the code in `trace/diag.c` but may be supplied by a device driver in the future.

This more elaborate view was introduced mainly to validate the trace and assertion macros during development.

`trace/null.cxx`

This contains an implementation of the trace and assert mechanisms that do nothing. By substituting its object file `null.o` for `simple.o` in a build, the trace mechanisms may be disabled without recompiling.

`trace/diag.c`

This contains a number of diagnostic routines that use the HAL supplied diagnostic output mechanism to format and print strings and numbers. There is currently no formatted output.

`trace/tcdiag.c`

This contains an implementation of the testing internal API which uses the kernel's diagnostic routines to perform output.

Sload subdirectory

This contains the sources of a simple S-Record loader that may be used in a ROM for various microprocessor development boards to download code via a serial port.

HAL source files

The HAL is divided into architecture- and platform-specific files. For each architecture supported, there is an `arch` directory, containing files generic to that

architecture, and a `platform` directory, containing files specific to each platform supported.

Amongst the architectures supported are: the PowerPC, the Tx39 and the MN10300. To find the code corresponding to each architecture, substitute “powerpc”, “mips” and “mn10300”, respectively, for “ARCH” in the following file descriptions. Similarly substitute the appropriate platform name representing your development board for “PLATFORM”.

Architecture files

`ARCH/arch/v1_3_x/include/basetype.h`

This file is used to define the base architecture configuration such as endianness and word size.

`ARCH/arch/v1_3_x/include/hal_arch.h`

This file contains macros that implement various architecture-specific functions. The most important macros here are the thread context initialization and switch macros that are used to implement multithreading.

`ARCH/arch/v1_3_x/include/hal_intr.h`

This file contains the HAL support for interrupt management and clock support.

`ARCH/arch/v1_3_x/include/hal_io.h`

This file contains the HAL support for accessing hardware registers. It provides a portable API that allows more generic device drivers to be written.

`ARCH/arch/v1_3_x/include/hal_cache.h`

This file contains macros to control any caches that may be present.

`ARCH/arch/v1_3_x/include/ARCH_stub.h`

This file contains architectural information for a GDB stub, such as the register layout in a GDB packet.

`ARCH/arch/v1_3_x/src/vectors.S`

This is an assembly code file that contains the code to handle interrupt and exception vectors. Since system reset can also be considered an exception, this is handled here also. Interrupts are currently handled by placing a stub routine in the hardware vector which calls a Vector Service Routine via an indirection table. There is a API to allow user-defined VSRs to be installed. The default VSR reads the interrupt controller registers and decodes the interrupt source into an offset into a further table of interrupt service routines. It also handles interrupt cleanup, which may result in the execution of deferred service routines (DSRs) and the preemption of the current thread.

`ARCH/arch/v1_3_x/src/context.S`

If present, this is an assembly code file that contains the code to support thread contexts. The routines to switch between various contexts, as well as initialize a thread context may be present in this file.

`ARCH/arch/v1_3_x/src/hal_misc.c`

This file contains the implementation of various miscellaneous HAL routines that are needed by the kernel or C++ runtime.

`ARCH/arch/v1_3_x/src/ARCH_stub.c`

This file contains the architectural part of a GDB stub. This deals with CPU-specific details of the stub, such as the setting of breakpoints and translating exception data into signals that GDB understands.

`ARCH/arch/v1_3_x/src/ARCH.ld`

This file is the linker script. During preprocessing it includes linker script fragments that define the memory layout.

Platform files

`ARCH/PLATFORM/v1_3_x/include/hal_diag.h`

This file contains the definitions of macros that support the HAL diagnostic output mechanism.

`ARCH/PLATFORM/v1_3_x/include/plf_stub.h`

This file contains a set of macros that allow the common GDB stub code to access the platform-specific minimal serial driver functions.

`ARCH/PLATFORM/v1_3_x/src/hal_diag.c`

This file contain the implementation of the HAL diagnostic output mechanism.

`ARCH/PLATFORM/v1_3_x/src/plf_stub.c`

This file contains a minimal serial driver for the target platform that is used by the GDB stub.

`ARCH/PLATFORM/v1_3_x/src/PLATFORM.S`

This is an assembler file that contains any platform-specific code. It often contains platform initialization code called from vectors.S.

Part II: Kernel APIs

3

Requirements for programs

eCos programs do not have to satisfy any unusual requirements, but there are always some differences between a program written for a real-time operating system as opposed to one written for a time sharing, virtual memory system like UNIX or Windows NT.

This chapter contains checklist of things to remember when writing **eCos** programs.

cyg_user_start()

The entry point for **eCos** user programs is usually **`cyg_user_start()`** instead of **`main()`**, although **`main()`** *can* be used if the ISO C library package is selected. Complete detail on the start-up sequence is given in “System start-up” on page 21.

Necessary headers

Any program which uses **eCos** system calls must have the following line at the top of the file:

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

and the programmer must make sure that `cyg/kernel/kapi.h` is available in the compiler include path. This can be done by setting the *C_INCLUDE_PATH* environment variable or by including the *-I* flag on the compiler command line.

Necessary link instructions

The **eCos** configuration and building process (described in *Getting Started with **eCos*** and ***eCos** User's Guide*) builds a single library, `libtarget.a`, which contains the selected **eCos** components. The `libtarget.a` library does *not* contain any user libraries: If you put some of your source in libraries, you will have to explicitly include those libraries in the linking instruction.

You also need to link to the **GNU C Compiler** runtime support library (`libgcc.a`).

You should *not* link to the standard C++ library. This can be achieved with the *-nostdlib* option.

You should only link to `libtarget.a` and `libgcc.a` using the linker script `target.ld` provided with **eCos**. The command line for linking should look like

```
gcc [options] [object files] -Ttarget.ld -nostdlib
```

Interrupt and exception handlers

In **eCos** a distinction is made between *exceptions* and *interrupts*.

`exceptions`

: are the result of some action by the currently executing code. Examples of exceptions are divide by zero, illegal instruction, bad memory access, etc.

`interrupts`

: are the result of a signal source which is conceptually asynchronous with the currently executing code. Examples of interrupts sources are the real-time clock, external and on chip peripherals and so forth.

This distinction is made in the **eCos** hardware abstraction layer (HAL) to provide a cleaner and more portable mechanism for installing interrupt handlers and exception handlers. Individual hardware platforms can have different ways of naming and handling interrupts, which is why this abstraction layer was chosen.

Interrupts and exceptions are both associated with *vectors*, which are labeled by *vector numbers* (see "Exception handling" on page 31 and "Interrupt handling" on page 32).

There are distinct spaces for exception and interrupt vectors. These are called "exception vector numbers" and "interrupt vector numbers". System calls which install exception handlers use the exception vector number, and the system calls which install interrupt handlers use the interrupt vector number to specify which interrupt or exception should be handled by the handler.

The details of the vector layout depend on the microprocessor and interrupt controller, and are documented in the relevant API sections.

Interrupt handlers are actually a *pair* of functions, one of which (the *interrupt service routine*, or ISR) is executed immediately and runs with that interrupt disabled. Since interrupts are disabled for the duration of the ISR, the ISR should be very brief and should not use any system services.

After the ISR exits, but before the kernel scheduler is invoked again, a *delayed service routine* (DSR) will be invoked. It executes with scheduling disabled, but with interrupts enabled, so that further invocations of the same DSR can be queued. The DSR can use some producer-side system calls, but it should be carefully crafted to avoid using any call that might put its thread to sleep. One of the few examples of safe calls is **`cyg_semaphore_post()`**; the non-blocking versions of some system calls are also safe. A call that is unsafe is **`cyg_mutex_lock()`**, since it will block if the mutex is already locked by another thread.

Finally, **eCos** has a formalism for installing *low level handlers* which bypass the kernel mechanisms described above. A program can install a *vector service routine* (VSR) which will be invoked instead of the kernel's usual exception or interrupt handling. The VSR will typically be written in assembly language.

VSRs are associated with vector numbers in the exception space, just like exception handlers (although there are some variations — architectures in which there are no exceptions in the **eCos** sense). The main difference between VSRs and exception handlers is that VSRs bypass the kernel's usual mechanisms.

Memory allocation

Most **eCos** system calls expect you to pass the address of pre-allocated memory for the objects created in that system call. This is frequently the preferred way of doing things for embedded applications, where programmers want to allocate all memory statically and have fine control over that resource.

In contrast, some **eCos** system calls also allow a ***`NULL`*** pointer to be passed. In such a case the kernel will allocate the memory or select default size. This feature is not supported in the current release, and a warning flag is placed in the documentation for those routines (like **`cyg_thread_create()`**).

eCos provides dynamic memory allocation, based on *memory pools*, a useful and flexible approach to memory management inspired by the µITRON compatibility layer. These are described in "Memory pools" on page 42.

If you configure your system to use the Standard C Library you can also use the standard **`malloc()`** library call.

Assertions and bad parameter handling

This section describes how the **eCos** kernel and basic packages behave when system calls are invoked with bad parameters.

In **eCos**, the basic kernel assertion behavior is configuration-dependent.

By default, assertions are turned off in the kernel. If the kernel is configured to turn them *on*, the kernel will make basic assertions, such as checking for invalid parameters when system calls are invoked. If an assertion fails, the kernel will print a message to the diagnostic output channel and stop executing.

If the kernel is configured with assertions disabled (usually when the application has been thoroughly debugged), it will not do any checking.

The configuration sections referenced above also describe the use of preconditions, postconditions and loop invariants. These are no different from ordinary assertions, but they are used in specialized circumstances, and the programmer would wish to select their presence individually.

4

System start-up

We describe here the steps performed by **eCos** upon start-up, mentioning how a programmer can introduce custom start-up routines.

System start-up — the HAL

The HAL (Hardware Abstraction Layer, see “The eCos Hardware Abstraction Layer (HAL)” on page 66) is the **eCos** package which contains all start-up code. Its start-up procedure is outlined in detail in “HAL startup” on page 79, but the main steps can be summarized here:

1. The HAL initializes the hardware, coordinates with the ROM monitor, and performs diagnostics.
2. The HAL invokes all static and global C++ constructors.
3. The HAL jumps to **`cyg_start()`**, which has the following prototype:

```
void cyg_start( void )
```

System start-up — cyg_start()

`cyg_start()` is the core of the start-up mechanism. The default definition is in `infra/current/src/startup.cxx`

It calls, in turn,

cyg_prestart()
cyg_package_start()
cyg_user_start()

and then starts the **eCos** scheduler if the system has been configured to have a kernel and scheduler.

You can override the default **cyg_start()** routine by providing your own function by the same name with the following prototype:

```
void cyg_start( void )
```

WARNING Overriding **cyg_start()** should rarely, if ever, be done. The functions **cyg_prestart()** and **cyg_user_start()** described just below allow enough flexibility for installing user initialization code safely for almost all applications.

NOTE If you are supplying your own definition of this function from a C++ file, make sure it has "C" linkage.

System startup — cyg_prestart()

The default **cyg_prestart()** function does not do anything; it is meant to be overwritten if the programmer needs to do any initialization *before* other system level initialization.

You can override the default **cyg_prestart()** routine by providing your own function by the same name with the following prototype:

```
void cyg_prestart( void )
```

NOTE If you are supplying your own definition of this function from a C++ file, make sure it has "C" linkage.

System startup — cyg_package_start()

The **cyg_package_start()** allows individual packages to do their initialization before the main user program is invoked.

Two of the packages shipped with this release of **eCos** have code in the default **cyg_package_start()**: the µITRON and the ISO standard C library compatibility packages (see "µITRON API" on page 51 and "C and math library overview" on page 146).

The infrastructure package has configuration options *CYGSEM_START_UITRON_COMPATIBILITY* and *CYGSEM_START_ISO_C_COMPATIBILITY* which control specialized

initialization.

You can override the default **cyg_package_start()** routine by providing your own function by the same name with the following prototype:

```
void cyg_package_start( void )
```

but you should be careful to initialize the default packages (if you are using them). An example user-supplied function might look like:

```
void cyg_package_start(void)
{
 #ifdef CYGSEM_START_UITRON_COMPATABILITY
cyg_uitron_start(); /* keep the µITRON initialization */
 #endif
my_package_start(); /* make sure I initialize my package */
}
```

NOTE If you are supplying your own definition of this function from a C++ file, make sure it has “C” linkage.

System startup — cyg_user_start()

This is the normal entry point for your code. Although a default empty version is provided by **eCos**, this is a good place to set up your threads (see “Thread operations” on page 27).

If you are not including the ISO standard C library package then there will not be a **main()** function, so it becomes mandatory to provide this function (see “C library startup” on page 153).

To set up your own **cyg_user_start()** function, create a function by that name with the following prototype:

```
void cyg_user_start( void )
```

When you return control from **cyg_user_start()**, **cyg_start()** will then invoke the scheduler, and any threads you created and resumed in **cyg_user_start()** will be executed. The preferred approach is to allow the scheduler to be started automatically, rather than to start it explicitly in **cyg_user_start()**.

CAUTION Remember that **cyg_user_start()** is invoked before the scheduler (and frequently the scheduler is invoked as the last step in **cyg_user_start()**), so it should not use any kernel services that require the scheduler.

NOTE If you are supplying your own definition of this function from a C++ file, make sure it has “C” linkage.

5

Native kernel C language API

The **eCos** kernel, like many other real-time kernels, is a library to which the programmer links an application. System calls resemble library API calls, and there is no trap mechanism to switch from user to system mode.

We present here the **eCos** kernel API and the APIs for other kernels provided as compatibility layers on top of **eCos**.

Since this API sits on top of a configurable system, the semantics are only weakly defined. The exact semantics and even the API itself depend on the configuration. For example if returned error codes were supported this would change the prototype of the functions. The semantics given in this chapter describe the default configuration.

As mentioned above, all source files which use the kernel C API should have the following `#include` statement:

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

at the head of the file.

Types used in programming eCos

We now describe the types defined for use with **eCos**. These are available to programs that include `kapi.h`.

Most of these types are meant to be *opaque* — in other words, programmers do not need to know (and probably should not know) how they are defined. But the types that

are numeric are marked, since it can be useful to use comparison operators.

The definitions for these types can be found in the installed tree, in the file `include/cyg/kernel/kapi.h`.

The **eCos** kernel uses the following naming convention for types:

- Types that can be treated as completely opaque usually have _t suffix.
- Types for which it is necessary to know the implementation do not have a _t suffix.

cyg_addrword_t

A type which is large enough to store the larger of an address and a machine word. This is used for convenience when a function is passed data which could be either a pointer to a block of data or a single word.

cyg_handle_t

A *handle* is a variable used to refer to **eCos** system objects (such as a thread or an alarm). Most **eCos** system calls that create system objects will return a handle that is used to access that object from then on.

cyg_priority_t

A numeric type used to represent the priority of a thread, or the priority of an interrupt level. A lower number means a higher (i.e. more important) priority thread.

cyg_code_t

A numeric type used for various error or status codes, such as exception numbers.

cyg_vector_t

A numeric type used to identify an interrupt vector. Its value is called the interrupt vector *id*. This type is used for both ISR vector ids and VSR vector ids.

cyg_tick_count_t

A numeric type used to count counter ticks. The resolution and other details regarding tick quantities depend on the configuration, but this is a 64 bit type, and no matter what configuration is chosen it should still last for centuries before it overflows.

cyg_bool_t

A boolean type whose values can be false (0) or true (1).

cyg_thread_entry_t

A function type for functions that are entry points for threads. It is used in the thread creation call **`cyg_thread_create()`**.

To help write thread entry point functions, here is how cyg_thread_entry_t is defined:

```
typedef void cyg_thread_entry_t(void *);
```

Examples of thread functions can be found in the programming tutorial in *Getting Started with **eCos***.

cyg_exception_handler_t

A function type used for installing exception handlers. It is defined as:

```
typedef void cyg_exception_handler_t(
 cyg_addrword_t data,
 cyg_code_t exception_number,
 cyg_addrword_t info
);
```

cyg_thread, cyg_interrupt, cyg_counter, cyg_clock, cyg_alarm, cyg_mbox, cyg_mempool_var, and cyg_mempool_fix

These types are of the appropriate size to contain the memory used by the respective kernel objects. These types are only used in the corresponding create call where the programmer allocates the memory for the object and passes the address to the kernel. After creation the provided handle is used to reference the object.

cyg_mempool_info

Contains information about a memory pool.

```
typedef struct {
 cyg_int32 totalmem;
 cyg_int32 freemem;
 void *base;
 cyg_int32 size;
 cyg_int32 blocksize;
 cyg_int32 maxfree; // The largest free block
} cyg_mempool_info;
```

cyg_sem_t, cyg_mutex_t, and cyg_cond_t

These types are of the appropriate size to contain the memory used by their respective kernel objects. These objects are always referred to by a pointer to an object of this type.

cyg_VSR_t, cyg_ISR_t, and cyg_DSR_t

These are function types used when vector, interrupt and delayed service routines are installed.

```
typedef void cyg_VSR_t();
typedef cyg_uint32 cyg_ISR_t(cyg_vector_t vector,
 cyg_addrword_t data);
typedef void cyg_DSR_t(cyg_uint32 vector,
 cyg_ucount32 count,
 cyg_addrword_t data);
```

cyg_resolution_t

Stores the resolution of a clock. The resolution is defined to be (dividend/divisor) nanoseconds per tick.

```
typedef struct { cyg_uint32 dividend;
 cyg_uint32 divisor; }
cyg_resolution_t;
```

cyg_alarm_t

The function type used for alarm handlers.

```
typedef void cyg_alarm_t(cyg_handle_t alarm,
 cyg_addrword_t data);
```

Thread operations

`void` **`cyg_scheduler_start`**`( void )`

Starts the scheduler with the threads that have been created. It never returns. The scheduler has been chosen at configuration time. **eCos** currently ships with three schedulers: a bitmap scheduler, a multi-level scheduler (selected by default), and an experimental “lottery” scheduler which is currently incomplete and unusable.

The configuration tool can be used to select between schedulers. The configuration options are *`CYGSEM_SCHED_BITMAP`*, *`CYGSEM_SCHED_MLQUEUE`* and *`CYGSEM_SCHED_LOTTERY`*.

NOTE Interrupts are not enabled until the scheduler has been started with **`cyg_scheduler_start()`**.

`void` **`cyg_scheduler_lock`**`( void )`

Locks the scheduler so that a context switch cannot occur. This can be used to protect data shared between a thread and a DSR, or between multiple threads, by surrounding the critical region with **`cyg_scheduler_lock()`** and **`cyg_scheduler_unlock()`**.

`void` **`cyg_scheduler_unlock`**`( void )`

Unlocks the scheduler so that context switching can occur again.

`void` **`cyg_thread_create`**`(`
`cyg_addrword_t` *sched_info*,
`cyg_thread_entry_t *`*entry*,
`cyg_addrword_t` *entry_data*,
`char *`*name*,
`void *`*stack_base*,
`cyg_ucount32` *stack_size*,
`cyg_handle_t *`*handle*,
`cyg_thread *`*thread)*

Creates a thread in a suspended state. The thread will not run until it has been resumed with **`cyg_thread_resume()`** and the scheduler has been started with **`cyg_scheduler_start()`**.

Here is a description of the parameters of **`cyg_thread_create()`**:

`sched_info`

Information to be passed to the scheduler. For almost all schedulers this is a simple priority value, and you can simply pass a non-negative integer when you create the thread. Even when this holds, some schedulers may have restrictions on how priorities can be used. For example, the bitmap scheduler can onlyhave one thread at each priority, so if an already-occupied priority slot is quoted, the next free slot of lower priority is chosen.

`entry`

A user-supplied function: it is a routine that begins execution of the new thread. This function takes a single argument of type cyg_addrword_t, which is usually a pointer to a block of data, which allows **`cyg_scheduler_start()`** to pass data to this particular thread.

Here is a typedef for the *`entry`* function:

```
typedef void cyg_thread_entry_t(cyg_addrword_t);
```

`entry_data`

A data value passed to the *`entry`* function. This may be either a machine word datum or the address of a block of data.

`name`

A C string with the name of this thread.

`stack_base`

The address of the stack base. If this value is *`NULL`* then **`cyg_thread_create()`** will choose a stack base.

NOTE Passing a stack base of *NULL* is not supported in this release. You must pass a real address for the stack base.

stack_size

The size of the stack for this thread. If this is 0, the default stack size will be used for this thread.

NOTE Passing a stack size of 0 is not supported in this release. You must pass a real stack size.

handle

cyg_thread_create() returns the thread handle in this location.

thread

The thread housekeeping information is placed in the memory pointed to by this parameter. If this pointer is *NULL* then the memory will be allocated.

NOTE Passing a NULL value for the thread data structure address is not supported in this release. You must pass a valid address.

`void` **`cyg_thread_exit`**`( void )`

Exits the current thread. At present this simply puts the thread into suspended state.

`void` **`cyg_thread_suspend`**`(`
`cyg_handle_t` *thread)*

Suspends the *thread*. A thread may be suspended multiple times, in which case it will need to be resumed the same number of times before it will run.

`void` **`cyg_thread_resume`**`(`
`cyg_handle_t` *thread)*

Resumes *thread*. If a thread has been suspended multiple times it will need to be resumed the same number of times before it will run. Threads are created in a suspended state and must be resumed before they will run.

`void` **`cyg_thread_yield`**`( void )`

Yields control to the next runnable thread of equal priority. If no such thread exists, then this function has no effect.

`void` **`cyg_thread_kill`**`(`
`cyg_handle_t` *thread)*

Kills *thread*.

`cyg_bool_t` **`cyg_thread_delete`**`(`
`cyg_handle_t` *thread*`)`

Kills *thread* and deletes it from the scheduler. If necessary, it will kill *thread* first using **`cyg_thread_kill`**(*thread*). If *thread* does not terminate in response to the kill message, this function returns false, indicating failure.

This function differs from **cyg_thread_kill()** (or calling **cyg_thread_exit()** for the current thread) by deregistering the thread from the scheduler. As a result, the thread handle, thread stack and space passed for the thread housekeeping information can then be reused. This is not the case if just **cyg_thread_kill()** or **cyg_thread_exit()** is invoked for the thread.

NOTE **cyg_thread_delete()** only deregisters the thread from the scheduler, it does not free up any resources that had been allocated by the thread such as dynamic memory, nor does it unlock any synchronization objects owned by the thread. This is the responsibility of the programmer. Additionally, unlike **cyg_thread_kill()**, the **cyg_thread_delete()** function cannot be self-referencing.

EXAMPLE
```
// Delete another thread. This must be done in a loop, waiting
// for the call to return true. If it returns false, go to sleep
// for a while, so that the killed thread gets a chance to run
// and complete its business.
while (!cyg_thread_delete(<thread_handle>) {
    cyg_thread_delay(1);
}
```

```
cyg_handle_t cyg_thread_self( void )
```

Returns the handle of the current thread.

```
void cyg_thread_release(
    cyg_handle_t thread )
```

Break the thread out of any wait it is currently in. Exactly how the thread returns from the wait operation, and how, if at all, the break is indicated, depends on the synchronization object it was waiting on.

```
cyg_ucount32 cyg_thread_new_data_index( void )
```

Allocates a new per-thread data index from those still available. If no more indexes are available, and assertions are enabled, an assertion will be raised.

```
void cyg_thread_free_data_index(
     cyg_ucount32 index )
```

Return the per-thread data index to the pool.

```
CYG_ADDRWORD cyg_thread_get_data(
     cyg_ucount32 index )
```

Retrieve the per-thread data at the given index for the current thread.

```
CYG_ADDRWORD *cyg_thread_get_data_ptr(
     cyg_ucount32 index )
```

Return a pointer to the per-thread data at the given index for the current thread. This should be used with some care since in some future implementation the per-thread data may be managed by a dynamic mechanism that might invalidate this pointer at

any time. This pointer should only be considered valid until the next call to the per-thread data functions.

```
void cyg_thread_set_data(
     cyg_ucount32 index, CYG_ADDRWORD data )
```

Store the data in the per-thread data for the current thread at the given index.

Priority manipulation

```
void cyg_thread_set_priority(
     cyg_handle_t thread,
     cyg_priority_t priority )
```

Sets the priority of the given thread to the given value. The smaller the value, the higher the priority of the thread.

Allowed priorities range between 1 and 64. The values of these parameters are configuration-dependent because they depend on which scheduler has been selected, and what value has been configured for the *CYGNUM_KERNEL_SCHED_PRIORITIES* configuration parameter (see “Thread operations” on page 27 and “Option: Number of priority levels” in Section V”).

There is always an idle thread, owned by the kernel, running at *CYG_THREAD_MIN_PRIORITY*. Because of this, ordinary threads should never be run at the lowest priority.

```
cyg_priority_t cyg_thread_get_priority(
      cyg_handle_t thread )
```

Returns the priority of the given thread.

```
void cyg_thread_delay(
     cyg_tick_count_t delay )
```

Puts the current thread to sleep for *delay* ticks. In a default configuration there are approximately 100 ticks a second. The actual length of the ticks is given by the resolution of the real-time clock. See “Counters, clocks and alarms” on page 34 for more information on counter resolution.

Exception handling

Exception handlers can be installed to deal with various system-level exceptions, such as alignment errors, resets, timers and so forth. Exception handling is a configurable feature of **eCos** and is enabled by default.

The range of values for the *exception_number* parameter in the functions below is hardware-dependent, as are the individual exceptions. See

`hal/ARCH/arch/v1_3_x/include/hal_intr` for the exception vector definitions specific to a given architecture.

The exception handler is a function of the following type:

```
typedef void cyg_exception_handler_t(
 cyg_addrword_t data,
 cyg_code_t exception_number,
 cyg_addrword_t info
);
```

cyg_exception_handler_t is the type used for functions which are called as a result of an exception. It is used in the function **`cyg_exception_set_handler()`**.

`void` **`cyg_exception_set_handler`**`(`
`cyg_code_t` *exception_number,*
`cyg_exception_handler_t *`*new_handler,*
`cyg_addrword_t` *new_data,*
`cyg_exception_handler_t **`*old_handler,*
`void **`*old_data)*

Replace current exception handler. This may apply to either the thread, or to a global exception handler, according to how exception handling was configured (global or per-thread). The exception may be ignored, or used to specify a particular handler.

`void` **`cyg_exception_call_handler`**`(`

`cyg_handle_t` *thread,*

`cyg_code_t` *exception_number,*

`cyg_addrword_t` *exception_info)*

Invoke exception handler for the given exception number. The exception handler will be invoked with *`exception_info`* as its third argument.

Interrupt handling

Interrupt handling is by nature machine-specific. The **eCos** kernel aims to provide efficiency and flexibility in this area, while maintaining a very low interrupt latency. To allow the programmer direct access to hardware, the semantics and the interface can vary from one architecture to another.

The interrupt vectors for a given architecture are defined in `hal/ARCH/arch/v1_3_x/include/hal_intr.h` where also special semantics and caveats of the interrupt capabilities would be described.

```
typedef void cyg_VSR_t();
typedef cyg_uint32 cyg_ISR_t(cyg_vector_t vector,
```

```
            cyg_addrword_t data);
           typedef void cyg_DSR_t(cyg_vector_t vector,
            cyg_ucount32 count,
            cyg_addrword_t data);

           enum cyg_ISR_results
           {
            CYG_ISR_HANDLED = 1, /* Interrupt was handled */
            CYG_ISR_CALL_DSR = 2 /* Schedule DSR */
           };
```

`void` **`cyg_interrupt_create`**(

`cyg_vector_t` *vector,*

`cyg_priority_t` *priority,*

`cyg_addrword_t` *data,*

`cyg_ISR_t *`*isr,*

`cyg_DSR_t *`*dsr,*

`cyg_handle_t *`*handle,*

`cyg_interrupt *`*intr)*

Creates an interrupt object and returns a handle to it. The object contains information about which interrupt *`vector`* to use and the ISR and DSR that will be called after the interrupt object is attached. The interrupt object will be allocated in the memory passed in the *`intr`* parameter. The interrupt object is not immediately attached; it must be attached with the **`cyg_interrupt_attach()`** call.

`void` **`cyg_interrupt_delete`**(

`cyg_handle_t` *interrupt)*

Detaches the *`interrupt`* from the vector and frees the corresponding memory.

`void` **`cyg_interrupt_attach`**(

`cyg_handle_t` *interrupt)*

Attaches *`interrupt`*.

`void` **`cyg_interrupt_detach`**(

`cyg_handle_t` *interrupt)*

Detaches *`interrupt`*.

`void` **`cyg_interrupt_get_vsr`**(

`cyg_vector_t` *vector,*

`cyg_VSR_t **`*vsr)*

Returns a pointer to the VSR currently installed on *`vector`*.

`void` **`cyg_interrupt_set_vsr`**(

`cyg_vector_t` *vector,*

cyg_VSR_t *vsr)*

Sets the current VSR on *vector*. A VSR directly attaches to the hardware interrupt vector and needs to be written in assembler.

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

Disables all interrupts.

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

Enables all interrupts.

`void` **`cyg_interrupt_mask`**`(`

cyg_vector_t *vector)*

Programs the interrupt controller to stop delivery of interrupts on *vector*. On some architectures this will also disable all lower priority interrupts while on others they remain enabled.

`void` **`cyg_interrupt_unmask`**`(`

cyg_vector_t *vector)*

Programs the interrupt controller to allow delivery of interrupts on the given interrupt *vector*.

`void` **`cyg_interrupt_acknowledge`**`(`

cyg_vector_t *vector)*

Should be used from inside an ISR to acknowledge receipt of the interrupt. The interrupt must be acknowledged. If an interrupt is not acknowledged, the interrupt may trigger immediately after the ISR returns, causing the ISR to be called again in a loop.

`void` **`cyg_interrupt_configure`**`(`

cyg_vector_t *vector,*

cyg_bool_t *level,*

cyg_bool_t *up)*

On some interrupt controllers the way an interrupt is detected may be configured. The *level* parameter chooses between level- or edge-triggered interrupts. The *up* parameter chooses between high and low level for level triggered interrupts or rising and falling edges for edge triggered interrupts.

Counters, clocks and alarms

Counters

The counter objects provided by the kernel provide an abstraction of the clock facility that is generally provided. Application code can associate alarms with counters, where

an alarm is identified by the number of ticks until it triggers, the action to be taken on triggering, and whether or not the alarm should be repeated.

There are two different implementations of the counter objects. The first stores all alarms in a single linked list. The alternative implementation uses a table of linked lists, with the size of the table being a separate configurable option. A single list is more efficient in terms of memory usage and is generally adequate when the application only makes use of a small number of alarms. For more complicated operations it is better to have a table of lists since this reduces the amount of computation whenever the timer goes off. Assuming a table size of 8 (the default value) on average the timer code will only need to check 1/8 of the pending alarms instead of all of them.

The configuration options which select the counter implementation are *CYGIMP_KERNEL_COUNTERS_MULTI_LIST* ("Option: Implement counters using a table of lists" in Section V) and *CYGIMP_KERNEL_COUNTERS_SINGLE_LIST* ("Option: Implement counters using a single list", in Section V).

The following functions can be used to create and manipulate counters:

```
void cyg_counter_create(
        cyg_handle_t *counter,
        cyg_counter *the_counter )
```

Creates a new counter and places it in the space pointed to by *counter*. A counter stores a value that is incremented by **cyg_counter_tick()**. Alarms may be attached to counters, and the alarms will trigger when the counter reaches a specified value.

```
void cyg_counter_delete(
        cyg_handle_t counter )
```

Deletes the given counter and frees the corresponding memory.

```
cyg_tick_count_t cyg_counter_current_value(
        cyg_handle_t counter )
```

Returns the current value of the given counter.

```
void cyg_counter_set_value(
        cyg_handle_t counter,
        cyg_tick_count_t new_value )
```

Sets the counter's value to *new_value*.

```
void cyg_counter_tick(
        cyg_handle_t counter )
```

Advances the counter by one tick.

Clocks

Clocks are counters which are associated with a stream of ticks that represent time periods. Clocks have a resolution associated with them, whereas counters do not.

The most frequently used clock is the *real-time clock* which serves two special purposes. First, it is necessary to support clock and alarm related functions such as **cyg_thread_delay()**. Second, it is needed to implement timeslicing in the mlqueue and lottery schedulers. If the application does not require either of these facilities, then it is possible to disable the real-time clock support completely. It is also possible to disable just timeslicing with the configuration option *CYGSEM_KERNEL_SCHED_TIMESLICE*, or just the clock and alarm functions, using the option *CYGFUN_KERNEL_THREADS_TIMER*..

The real-time clock is available if the configuration option *CYGVAR_KERNEL_COUNTERS_CLOCK* is defined.

Clock resolution is stored in variables of type cyg_resolution_t (see “cyg_resolution_t” on page 27).

```
void cyg_clock_create(
        cyg_resolution_t resolution,
        cyg_handle_t *handle,
        cyg_clock *clock )
```

Creates a clock object with the given *resolution* and places it in the space pointed to by *clock*. A clock is a counter driven by a regular source of ticks. For example the system real-time clock is driven by a clock interrupt.

```
void cyg_clock_delete(
        cyg_handle_t clock )
```

Deletes a clock object and frees the associated memory.

```
void cyg_clock_to_counter(
        cyg_handle_t clock,
        cyg_handle_t *counter )
```

Converts a clock handle to a counter handle. The counter functions can then be used with the counter handle.

```
void cyg_clock_set_resolution(
        cyg_handle_t clock,
        cyg_resolution_t resolution )
```

Changes the resolution of a given clock object.

```
cyg_resolution_t cyg_clock_get_resolution(
        cyg_handle_t clock )
```

Returns the resolution of *clock*.

```
cyg_handle cyg_real_time_clock( void )
```

Returns a handle to the system-supplied real-time clock.

```
cyg_tick_count_t cyg_current_time( void )
```

Returns the real-time clock's counter. This is equivalent to executing the code:

```
cyg_clock_to_counter(cyg_real_time_clock(), &h),
cyg_counter_current_value(h);
```

Alarms

```
typedef void cyg_alarm_t(cyg_handle_t alarm,
 cyg_addrword_t data);
```

cyg_alarm_t is the type used for functions which are used to handle alarm events. It is used in the function **cyg_alarm_create()**.

```
void cyg_alarm_create(
    cyg_handle_t counter,
    cyg_alarm_t *alarm_fn,
    cyg_addrword_t data,
    cyg_handle_t *handle,
    cyg_alarm *alarm )
```

Creates an alarm object. The alarm is attached to the *counter* and is created in the memory pointed to by *alarm*. When the alarm triggers, the handler function *alarmfn* is called and is passed *data* as a parameter. The alarm handler executes in the context of the function that incremented the counter and thus triggered the alarm.

NOTE If the alarm is associated with the real-time clock, the alarm handler *alarmfn* will be invoked by the delayed service routine (DSR) that services the real-time clock. This means that real-time clock alarm handlers (which are possibly the most frequently used) must follow the rules of behavior for DSRs. These rules are outlined in "Interrupt and exception handlers" on page 18.

```
void cyg_alarm_delete(
    cyg_handle_t alarm )
```

Disables the alarm, detaches from the counter, invalidates handles, and frees memory if it was dynamically allocated by **cyg_alarm_create()**.

```
void cyg_alarm_initialize(
    cyg_handle_t alarm,
    cyg_tick_count_t trigger,
```

cyg_tick_count_t *interval)*

Initialize an alarm. This sets it to trigger at the tick with value *trigger*. When an alarm triggers, this event is dealt with by calling the *alarmfn* parameter which was passed when the alarm was created using **cyg_alarm_create()**. If *interval* is non-zero, then after the alarm has triggered it will set itself to trigger again after *interval* ticks. Otherwise, if *interval* is zero, the alarm is will be disabled automatically once it has triggered.

void **cyg_alarm_enable**(

cyg_handle_t *alarm)*

Enables an alarm that has been disabled by calling **cyg_alarm_disable()**.

void **cyg_alarm_disable**(

cyg_handle_t *alarm)*

Disables an alarm. After an alarm is disabled it will not be triggered unless it is subsequently re-enabled by calling **cyg_alarm_enable()** or is reinitialized by calling **cyg_alarm_initialize()**.

Note, though, that if a periodic alarm that has been disabled is re-enabled without reinitializing it will be in phase with the *original* sequence of alarms. If it is *reinitialized*, the new sequence of alarms will be in phase with the moment in which **cyg_alarm_initialize()** was called.

Synchronization

Semaphores

The semaphores defined by the type cyg_sem_t are counting semaphores. These objects are not referred to by handles, but rather by the pointer to the variable in which the semaphore is created.

void **cyg_semaphore_init**(

cyg_sem_t **sem*,

cyg_ucount32 *val)*

Initializes a semaphore. The initial semaphore count is set to *val*.

void **cyg_semaphore_destroy**(

cyg_sem_t **sem)*

Destroys a semaphore. This must not be done while there are any threads waiting on it.

void **cyg_semaphore_wait**(

cyg_sem_t **sem)*

If the semaphore count is zero, the current thread will wait on the semaphore. If the count is non-zero, it will be decremented and the thread will continue running.

`cyg_bool_t` **`cyg_semaphore_trywait`**(

`cyg_sem_t *`*sem)*

A non-blocking version of **`cyg_semaphore_wait()`**. This attempts to decrement the semaphore count. If the count is positive, then the semaphore is decremented and *`true`* is returned. If the count is zero then the semaphore remains unchanged, and *`false`* is returned, but the current thread continues to run.

`cyg_bool_t` **`cyg_semaphore_timed_wait`**(

`cyg_sem_t *`*sem*,

`cyg_tick_count_t` *abstime)*

A time-out version of **`cyg_semaphore_wait()`**. This attempts to decrement the semaphore count. If the count is positive, then the semaphore is decremented and *`true`* is returned. If the count is zero, it will wait for the semaphore to increment. If however the *`abstime`* time-out is reached first, it will return *`false`* without changing state, and the current thread will continue to run.

The *cyg_tick_count_t* parameter is an absolute time. If a relative time is required, you should use **`cyg_current_time`** with an offset. For example, to time out 200 ticks from the present you would use:

```
cyg_semaphore_timed_wait(&sem, cyg_current_time() + 200);
```

`cyg_semaphore_timed_wait()` is only available if the configuration option *`CYGFUN_KERNEL_THREADS_TIMER`* is set.

`void` **`cyg_semaphore_post`**(

`cyg_sem_t *`*sem)*

If there are threads waiting on this semaphore this will wake exactly one of them. Otherwise it simply increments the semaphore count.

`void` **`cyg_semaphore_peek`**(

`cyg_sem_t *`*sem*,

`cyg_count32 *`*val)*

Returns the current semaphore count in the variable pointed to by *`val`*.

Mutexes

Mutexes (mutual exclusion locks) are used in a similar way to semaphores. A mutex only has two states, locked and unlocked. Mutexes are used to protect accesses to shared data or resources. When a thread locks a mutex it becomes the owner. Only the mutex's owner may unlock it. While a mutex remains locked, the owner should not lock it again, as the behavior is undefined and probably dangerous.

If non-owners try to lock the mutex, they will be suspended until the mutex is available again, at which point they will own the mutex.

`void` **`cyg_mutex_init`**`(`

`cyg_mutex_t *`*`mutex )`

Initializes a mutex. It is initialized in the unlocked state.

`void` **`cyg_mutex_destroy`**`(`

`cyg_mutex_t *`*`mutex )`

Destroys a mutex. A mutex should not be destroyed in the locked state, as the behavior is undefined.

`cyg_bool_t` **`cyg_mutex_lock`**`(`

`cyg_mutex_t *`*`mutex )`

Changes the nutex from the unlocked state to the locked state. When this happens the mutex becomes owned by the current thread. If the mutex is locked, the current thread will wait until the mutex becomes unlocked before performing this operation. The result of this function will be TRUE if the mutex has been locked, or FALSE if it has not. A FALSE result can result if the thread has been released from its wait by a call to **`cyg_thread_release()`** or **`cyg_mutex_release()`**.

`void` **`cyg_mutex_unlock`**`(`

`cyg_mutex_t *`*`mutex )`

Changes the mutex from the locked state to the unlocked state. This function may only be called by the thread which locked the mutex, and should not be called on an unlocked mutex.

`void` **`cyg_mutex_release`**`(`

`cyg_mutex_t *`*`mutex )`

Release all threads waiting on the mutex pointed to by the mutex argument. These threads will return from cyg_mutex_lock() with a FALSE result and will not have claimed the mutex. This function has no effect on any thread that may haev the mutex claimed.

Condition Variables

Condition variables are a synchronization mechanism which (used with a mutex) grants several threads mutually exclusive access to shared data and to broadcast availability of that data to all the other threads.

A typical example of the use of condition variables is when one thread (the producer) is producing data and several other (consumer) threads are waiting for that data to be ready. The consumers will wait by invoking **`cyg_cond_wait()`**. The producer will lock access to the data with a mutex, and when it has generated enough data for the other processes to consume, it will invoke **`cyg_cond_broadcast()`** to wake up the

consumers. The *Getting Started with **eCos*** book has example programs which use condition variables to implement a simple message passing system between threads.

```
void cyg_cond_init(
    cyg_cond_t *cond,
    cyg_mutex_t *mutex )
```

Initializes the condition variable. A condition variable is attached to a specific mutex.

```
void cyg_cond_destroy(
    cyg_cond_t *cond )
```

Destroys the condition variable *cond*. This must not be done on a condition variable which is in use. After it has been destroyed, it may be subsequently reinitialized.

```
void cyg_cond_wait(
    cyg_cond_t *cond )
```

Causes the current thread to wait on the condition variable, while simultaneously unlocking the corresponding mutex. **cyg_cond_wait()** may be called by a thread which has the corresponding mutex locked.

The thread can only be awakened by a call to **cyg_cond_signal()** or **cyg_cond_broadcast()** on the same condition variable. When the thread is awakened, the mutex will be reclaimed before this function proceeds. Since it may have to wait for this, **cyg_cond_wait()** should only be used in a loop since the condition may become false in the meantime. This is shown in the following example:

```
extern cyg_mutex_t mutex;
extern cyg_cond_t cond;

cyg_mutex_lock( &mutex );
...

while( condition_not_true )
{
 cyg_cond_wait( &cond );
}

...

cyg_mutex_unlock( &mutex );
```

```
cyg_bool_t cyg_cond_timed_wait(
    cyg_cond_t *cond,
    cyg_tick_count_t abstime )
```

A time-out version of **cyg_cond_wait()** which waits for a signal or broadcast. If a signal or broadcast is received it returns *true*, but if one is not received by

abstime, it returns *false*.

The *cyg_tick_count_t* parameter is an absolute time. If a relative time is required, you should use **cyg_current_time** with an offset. For example, to time out 200 ticks from the present you would use:

```
cyg_cond_timed_wait(&sem, cyg_current_time() + 200);
```

cyg_cond_timed_wait() is only available if the configuration option *CYGMFN_KERNEL_SYNCH_CONDVAR_TIMED_WAIT* is set.

`void` **cyg_cond_signal**(

`cyg_cond_t *`*cond)*

Wakes up at least one thread which is waiting on the condition variable. When a thread is awakened it will become the owner of the mutex. **cyg_cond_signal()** may be called by the thread which currently owns the mutex to which the condition variable is attached.

`void` **cyg_cond_broadcast**(

`cyg_cond_t *`*cond)*

Wakes *all* the threads waiting on the condition variable. Each time a thread is awakened it will become the current owner of the mutex.

Memory pools

There are two sorts of memory pools. A variable size memory pool is for allocating blocks of any size. A fixed size memory pool, has the block size specified when the pool is created and only provides blocks of that size.

Blocking, non-blocking and “blocking with time-out” versions of these calls are provided.

`void` **cyg_mempool_var_create**(

`void *`*base*,

`cyg_int32` *size*,

`cyg_handle_t *`*handle*,

`cyg_mempool_var *`*var)*

Creates a variable size memory pool. The parameters are:

base

base of memory to use for pool

size

size of memory pool in bytes

handle

returned handle of memory pool

var

space to put pool structure in

`void` **`cyg_mempool_var_delete`**`(`

`cyg_handle_t` *varpool)*

Deletes the variable size memory pool *varpool*.

`void *`**`cyg_mempool_var_alloc`**`(`

`cyg_handle_t` *varpool*,

`cyg_int32` *size)*

Allocates a block of length *size*. This will block until the memory becomes available.

`void *`**`cyg_mempool_var_timed_alloc`**`(`

`cyg_handle_t` *varpool*,

`cyg_int32` *size*,

`cyg_tick_count_t` *abstime)*

Allocates a block of length size. If the requested amount of memory is not available, it will wait until *abstime* before giving up and returning *NULL*.

`void *`**`cyg_mempool_var_try_alloc`**`(`

`cyg_handle_t` *varpool*,

`cyg_int32` *size)*

Allocates a block of length size. *NULL* is returned if not enough is available.

`void` **`cyg_mempool_var_free`**`(`

`cyg_handle_t` *varpool*,

`void *`*p)*

Frees memory back into variable size pool.

`cyg_bool_t` **`cyg_mempool_var_waiting`**`(`

`cyg_handle_t` *varpool)*

Returns true if any threads are waiting for memory in *pool*.

```
typedef struct {
 cyg_int32 totalmem;
 cyg_int32 freemem;
 void *base;
 cyg_int32 size;
 cyg_int32 blocksize;
 cyg_int32 maxfree; // The largest free block
} cyg_mempool_info;
```

`void` **`cyg_mempool_var_get_info`**`(`
`cyg_handle_t` *varpool*,
`cyg_mempool_info *`*info)*
Puts information about a variable memory pool into the structure provided.

`void` **`cyg_mempool_fix_create`**`(`
`void *`*base*,
`cyg_int32` *size*,
`cyg_int32` *blocksize*,
`cyg_handle_t *`*handle*,
`cyg_mempool_fix *`*fix)*
Create a fixed size memory pool. This function takes the following parameters:

`base`
base of memory to use for pool

`size`
size of total space requested

`blocksize`
size of individual elements

`handle`
returned handle of memory pool

`fix`
space to put pool structure in

`void` **`cyg_mempool_fix_delete`**`(`
`cyg_handle_t` *fixpool)*
Deletes the given fixed size memory pool.

`void *`**`cyg_mempool_fix_alloc`**`(`
`cyg_handle_t` *fixpool)*
Allocates a block. If the memory is not available immediately, this blocks until the memory becomes available.

`void *`**`cyg_mempool_fix_timed_alloc`**`(`
`cyg_handle_t` *fixpool,*
`cyg_tick_count_t` *abstime)*
Allocates a block. If the memory is not already available, it will try until *`abstime`* before giving up and returning a *`NULL`*.

`void *`**`cyg_mempool_fix_try_alloc`**`(`

cyg_handle_t *fixpool)*

Allocates a block. NULL is returned if no memory is available.

void **cyg_mempool_fix_free**(
cyg_handle_t *fixpool,*
void **p)*

Frees memory back into fixed size pool.

cyg_bool_t **cyg_mempool_fix_waiting**(
cyg_handle_t *fixpool)*

Returns true if there are any threads waiting for memory in the given memory pool.

void **cyg_mempool_fix_get_info**(
cyg_handle_t *fixpool,*
cyg_mempool_info **info)*

Puts information about a variable memory pool into the structure provided.

The fixed size memory pool simply returns blocks of memory of exactly the blocksize requested. If the pool is being used to allocate memory for a type that has alignment constraints (such as 4-byte alignment), then it is up to the user to align the memory appropriately for the type in question. Alternatively, choose a blocksize that is an exact multiple of the required alignment.

The memory available from the memory pools will not be the same size as the memory supplied to it. Some of the memory is used for internal data structures of the allocator. `cyg_mempool_fix_get_info()` and `cyg_mempool_var_get_info()` may be used to determine the available memory.

Message boxes

Message boxes are a primitive mechanism for exchanging messages between threads, inspired by the µITRON specification. A message box can be created with **`cyg_mbox_create()`** before the scheduler is started, and two threads in a typical producer/consumer relationship can access it. One thread, the producer, will use **`cyg_mbox_put()`** to make data available to the consumer thread which uses **`cyg_mbox_get()`** to access the data.

The size of the internal message queue is configured by the *`CYGNUM_KERNEL_SYNCH_MBOX_QUEUE_SIZE`* parameter (see “Message box queue size”, in Section V). The default value is 10.

Blocking, non-blocking and “blocking with time-out” versions of these calls are provided.

void **cyg_mbox_create**(

cyg_handle_t *handle,

cyg_mbox *mbox)

Creates a message box using the space provided in the *mbox* parameter, and returns a handle for future access to that message box.

void **cyg_mbox_delete**(

cyg_handle_t *mbox)

Deletes the given message box.

void ***cyg_mbox_get**(

cyg_handle_t *mbox)

Waits for a message to be available, then retrieves it and returns the address of the data.

void ***cyg_mbox_timed_get**(

cyg_handle_t *mbox,

cyg_tick_count_t *timeout)

Waits for a message to be available, but times out if *timeout* time passes. This version of the function is only available if the configuration option *CYGFUN_KERNEL_THREADS_TIMER* is turned on.

void ***cyg_mbox_tryget**(

cyg_handle_t *mbox)

Checks to see if a message is ready. If no message is available it returns immediately with a return value of *NULL*. If a message is available it retrieves it and returns the address of the data.

void ***cyg_mbox_peek_item**(

cyg_handle_t *mbox)

Checks to see if a message is ready, and if one is available returns the address of the data *without* removing the message from the queue. If no message is available it returns *NULL*.

cyg_bool_t **cyg_mbox_put**(

cyg_handle_t *mbox,

void *item)

Places a message in the given message box. If the queue is full it will block until the message can be sent. It returns true if the message was successfully sent, and false if the message was not sent and its sleep was awakened by the kernel before the message could be sent.

The **cyg_mbox_put()** function is only available if the *CYGMTH_MBOXT_PUT_CAN_WAIT* configuration has been selected.

cyg_bool_t **cyg_mbox_timed_put**(

cyg_handle_t *mbox*,

void **item*,

cyg_tick_count_t *abstime)*

A time-out version of **cyg_mbox_put()**. This will try to place the message in the given message box. If the queue is full, it will wait until *abstime* before giving up and returning *false*.

The **cyg_mbox_timed_put()** function is only available if the both the *CYGMFN_KERNEL_SYNCH_MBOXT_PUT_CAN_WAIT* and *CYGFUN_KERNEL_THREADS_TIMER* configuration have been selected.

cyg_bool_t **cyg_mbox_tryput**(

cyg_handle_t *mbox*,

void **item)*

Tries to place a message in the given message box. It returns *true* if the message was successfully sent, and *false* if the message could not be sent immediately, usually because the queue was full.

cyg_count32 **cyg_mbox_peek**(

cyg_handle_t *mbox)*

Takes a peek at the queue and returns the number of messages waiting in it.

cyg_bool_t **cyg_mbox_waiting_to_get**(

cyg_handle_t *mbox)*

Queries the kernel to see if other processes are waiting to receive a message in the given message box. Returns *true* if other processes are waiting, *false* otherwise.

cyg_bool_t **cyg_mbox_waiting_to_put**(

cyg_handle_t *mbox)*

Queries the kernel to see if *other* processes are waiting to send a message in the given message box. Returns *true* if other processes are waiting, *false* otherwise.

Flags

Flags are a synchronization mechanism which allow a thread to wait for a single condition or a combination of conditions. The conditions are represented by bits in a 32 bit word. Flags are inspired by the µITRON specification.

Flags are of type cyg_flag_t, which are 32 bit words, and routines are provided to set or mask some bits in the flag value.

A “consumer side” thread can wait for a “producer side” thread to set the entire collection of bits, or any subset of them.

When a thread sets some bits in a flag, all threads whose requirements are now satisfied are woken up; thus flags have broadcast semantics. A variation on the wait call can specify that the flag value be cleared when the wait call is satisfied, in which case the setting of bits would not be a broadcast.

Blocking, non-blocking, and "blocking with time-out" versions of the wait calls are provided.

`void` **`cyg_flag_init`**(

`cyg_flag_t` **flag)*

Initializes a flag variable.

`void` **`cyg_flag_destroy`**(

`cyg_flag_t` **flag)*

Destroys a flag variable.

`void` **`cyg_flag_setbits`**(

`cyg_flag_t` **flag*,

`cyg_flag_value_t` *value)*

Sets the bits in *flag* which are set in *value*.

A side effect of **`cyg_flag_setbits()`** is that the kernel wakes up any waiting threads whose requirements are now satisfied.

flag

A pointer to the flag whose bits are being set. The new setting of *flag* will be `*flag -> (*flag | value)`.

value

A word whose 1 bits will be also set in **flag*.

`void` **`cyg_flag_maskbits`**(

`cyg_flag_t` **flag*,

`cyg_flag_value_t` *value)*

Clear the bits in the given flag which are zero in the *value*. This cannot result in new threads being eligible for awakening.

flag

A pointer to the flag whose bits are being cleared. The new setting of *flag* will be `*flag -> (*flag & value)`.

value

A word whose 0 bits will be also cleared in **flag*.

We now describe the **`cyg_flag_wait()`**, which frequently uses the following macros:

```
#define CYG_FLAG_WAITMODE_AND ((cyg_flag_mode_t)0)
```

```
#define CYG_FLAG_WAITMODE_OR ((cyg_flag_mode_t)2)
#define CYG_FLAG_WAITMODE_CLR ((cyg_flag_mode_t)1)
cyg_flag_value_t cyg_flag_wait(
    cyg_flag_t *flag,
    cyg_flag_value_t pattern,
    cyg_flag_mode_t mode )
```

Wait for all the bits which are one in *pattern* to be set in the *flag* value (if *mode* is *CYG_FLAG_WAITMODE_AND*) or for any of the bits which are one in pattern to be set in the flag value (if *mode* is *CYG_FLAG_WAITMODE_OR*).

When **cyg_flag_wait()** returns, meaning that the condition is met, the flag value which succeeded is returned from the call; in other circumstances (such as a bad value for *mode* or *pattern*), zero is returned to indicate the error.

If the mode is one of those above plus *CYG_FLAG_WAITMODE_CLR*, the whole of the flag value is cleared to zero when the condition is met.

cyg_flag_wait() takes the following parameters:

flag

The value of the flag (set by the thread that called **cyg_flag_setbits()** or **cyg_flag_maskbits()**) is placed in here.

pattern

The set of bits which, if set, will cause the calling thread to be woken up.

mode

A parameter which modifies the conditions for wake-up. It can take the following values:

CYG_FLAG_WAITMODE_AND

Only wake up if *all* the bits in *mask* is set in the flag.

CYG_FLAG_WAITMODE_OR

Wake up if *any* of the bits in *mask* is set in the flag.

CYG_FLAG_WAITMODE_AND + *CYG_FLAG_WAITMODE_CLR*,

CYG_FLAG_WAITMODE_OR + *CYG_FLAG_WAITMODE_CLR*

Like *CYG_FLAG_WAITMODE_AND* and *CYG_FLAG_WAITMODE_OR*, but the entire flag is cleared to zero when the condition is met, whereas normally only the bits that are set in *pattern* would be cleared.

Waiting threads are queued depending on the semantics of the underlying scheduler. In release 1.3.x, this means that, if the multi-level queue scheduler is selected, queueing is in FIFO ordering, while the bitmap scheduler supports thread priority ordered queueing. When some flag value bits become signalled by a call to

cyg_flag_setbits(), the queue is scanned in order, and each waiting thread in turn is awoken or re-queued depending on its request. When a thread is awoken, if it made the wait call with *CYG_FLAG_WAITMODE_CLR*, the flag value is cleared to zero, and the scan of queued threads is terminated.

```
cyg_flag_value_t cyg_flag_timed_wait(
    cyg_flag_t *flag,
    cyg_flag_value_t pattern,
    cyg_flag_mode_t mode,
    cyg_tick_count_t abstime )
```

A time-out version of **cyg_flag_wait()**. This waits for the condition required by pattern and mode to be met, or until the abstime time-out is reached, whichever is first. If the time-out is reached first, zero is returned. This call is only available if the configuration option *CYGFUN_KERNEL_THREADS_TIMER* is enabled.

```
cyg_flag_value_t cyg_flag_poll(
    cyg_flag_t *flag,
    cyg_flag_value_t pattern,
    cyg_flag_mode_t mode )
```

A non-blocking version of **cyg_flag_wait()**. If the condition required by pattern and mode is met, the flag value is returned, otherwise zero is returned. The flag value may be cleared in the event of success by specifying *CYG_FLAG_WAITMODE_CLR* in the mode, as usual.

```
cyg_flag_value_t cyg_flag_peek(
    cyg_flag_t *flag )
```

Returns the current flag value.

```
cyg_bool_t cyg_flag_waiting(
    cyg_flag_t *flag )
```

Returns true if there are threads waiting on this flag.

6

µITRON API

The µITRON specification defines a highly flexible operating system architecture designed specifically for application in embedded systems. The specification addresses features which are common to the majority of processor architectures and deliberately avoids virtualization which would adversely impact real-time performance. The µITRON specification may be implemented on many hardware platforms and provides significant advantages by reducing the effort involved in understanding and porting application software to new processor architectures.

Several revisions of the µITRON specification exist. In this release, **eCos** supports the µITRON version 3.02 specification, with complete "Standard functionality" (level S), plus many "Extended" (level E) functions. The definitive reference on µITRON is Dr. Sakamura's book µ*ITRON 3.0, An Open and Portable Real-Time Operating System for Embedded Systems*. If you have purchased the **eCos Developer's Kit**, you will have received a copy of this book. Otherwise, the book can be purchased from the IEEE Press, and an ASCII version of the standard can be found online at

`http://www.itron.gr.jp/`

(The old address

`http://tron.um.u-tokyo.ac.jp/TRON/ITRON/`

still exists as a mirror site.)

The **eCos** kernel implements the functionality used by the µITRON compatibility subsystem. The configuration of the kernel influences the behavior of µITRON programs.

In particular, the default configuration has time slicing (also known as round-robin scheduling) switched on; this means that a task can be moved from RUN state to READY state at any time, in order that one of its peers may run. This is not strictly conformant to the µITRON specification, which states that timeslicing may be implemented by periodically issuing a `rot_rdq(0)` call from within a periodic task or cyclic handler; otherwise it is expected that a task runs until it is pre-empted in consequence of synchronization or communications calls it makes, or the effects of an interrupt or other external event on a higher priority task cause that task to become READY. To disable timeslicing functionality in the kernel and µITRON compatibility environment, please disable the *CYGSEM_KERNEL_SCHED_TIMESLICE* configuration option in the kernel package. A description of kernel scheduling is in “Thread operations” on page 27.

For another example, the semantics of task queueing when waiting on a synchronization object depend solely on the way the underlying kernel is configured. As discussed above, the multi-level queue scheduler is the only one which is µITRON compliant, and it queues waiting tasks in FIFO order. Future releases of that scheduler might be configurable to support priority ordering of task queues. Other schedulers might be different again: for example the bitmap scheduler can be used with the µITRON compatibility layer, even though it only allows one task at each priority and as such is not µITRON compliant, but it supports only priority ordering of task queues. So which queueing scheme is supported is not really a property of the µITRON compatibility layer; it depends on the kernel.

In this version of the µITRON compatibility layer, the calls to disable and enable scheduling and interrupts (**`dis_dsp()`**, **`ena_dsp()`**, **`loc_cpu()`** and **`unl_cpu()`**) call underlying kernel functions; in particular, the **`xxx_dsp()`** functions lock the scheduler entirely, which prevents dispatching of DSRs; functions implemented by DSRs include clock counters and alarm timers. Thus time “stops” while dispatching is disabled with **`dis_dsp()`**.

Like all parts of the **eCos** system, the detailed semantics of the µITRON layer are dependent on its configuration and the configuration of other components that it uses. The µITRON configuration options are all defined in the file `pkgconf/uitron.h`, and can be set using the configuration tool or editing this file by hand.

An important configuration option for the µITRON compatibility layer is *CYGSEM_UITRON_BAD_PARAMS_RETURN_ERRORS* (see “Option: Return Error Codes for Bad Params”, in Section V), which allows a lot of the error checking code in the µITRON compatibility layer to be removed; of course this leaves a program open to undetected errors, so it should only be used once an application is fully debugged and tested. Its benefits include reduced code size and faster execution. However, it affects the API significantly, in that with this option enabled, bad calls do not return errors, but either cause an assert failure (if that is itself enabled) or

malfunction internally. There is discussion in more detail about this in each section below.

We now give a brief description of the µITRON functions which are implemented in this release. Note that all C and C++ source files should have the following `#include` statement:

```
#include <cyg/compat/uitron/uit_func.h>
```

Task Management Functions

The following functions are fully supported in this release:

```
ER sta_tsk(
           ID tskid,
           INT stacd )
void ext_tsk( void )
void exd_tsk( void )
ER dis_dsp( void )
ER ena_dsp( void )
ER chg_pri(
           ID tskid,
            PRI tskpri )
ER rot_rdq(
           PRI tskpri )
ER get_tid(
           ID *p_tskid )
ER ref_tsk(
           T_RTSK *pk_rtsk,
           ID tskid )
ER ter_tsk(
           ID tskid )
ER rel_wai(
           ID tskid )
```

The following two functions are supported in this release, when enabled with the configuration option CYGPKG_UITRON_TASKS_CREATE_DELETE with some restrictions:

```
ER cre_tsk(
```

```
        ID tskid,
        T_CTSK *pk_ctsk )
ER del_tsk(
        ID tskid )
```

These functions are restricted as follows:

Because of the static initialization facilities provided for system objects, a task is allocated stack space statically in the configuration. So while tasks can be created and deleted, the same stack space is used for that task (task ID number) each time. Thus the stack size (pk_ctsk->stksz) requested in **cre_tsk()** is checked for being less than that which was statically allocated, and otherwise ignored. This ensures that the new task will have enough stack to run. For this reason **del_tsk()** does not in any sense free the memory that was in use for the task's stack.

The task attributes (pk_ctsk->tskatr) are ignored; current versions of **eCos** do not need to know whether a task is written in assembler or C/C++ so long as the procedure call standard appropriate to the CPU is followed.

Extended information (pk_ctsk->exinf) is ignored.

Error checking

For all these calls, an invalid task id (tskid) (less than 1 or greater than the number of configured tasks) only returns E_ID when bad params return errors (*CYGSEM_UITRON_BAD_PARAMS_RETURN_ERRORS* is enabled, see above).

Similarly, the following conditions are only checked for, and only return errors if *CYGSEM_UITRON_BAD_PARAMS_RETURN_ERRORS* is enabled:

- pk_crtk in **cre_tsk()** is a valid pointer, otherwise return E_PAR
- **ter_tsk()** or **rel_wai()** on the calling task returns E_OBJ
- the CPU is not locked already in **dis_dsp()** and **ena_dsp()**; returns E_CTX
- priority level in **chg_pri()** and **rot_rdq()** is checked for validity, E_PAR
- return value pointer in **get_tid()** and **ref_tsk()** is a valid pointer, or E_PAR

The following conditions are checked for, and return error codes if appropriate, regardless of the setting of *CYGSEM_UITRON_BAD_PARAMS_RETURN_ERRORS*:

- When create and delete functions **cre_tsk()** and **del_tsk()** are supported, all calls which use a valid task ID number check that the task exists; if not, E_NOEXS is returned
- When supported, **cre_tsk()**: the task must not already exist; otherwise E_OBJ
- When supported, **cre_tsk()**: the requested stack size must not be larger than that statically configured for the task; see "Option: Static initializerst", in Section V and "Option: Default stack size", in Section V. Else E_NOMEM
- When supported, **del_tsk()**: the underlying **eCos** thread must not be running -

this would imply either a bug or some program bypassing the µITRON compatibility layer and manipulating the thread directly. E_OBJ

- **sta_tsk()**: the task must be dormant, else E_OBJ
- **ter_tsk()**: the task must not be dormant, else E_OBJ
- **chg_pri()**: the task must not be dormant, else E_OBJ
- **rel_wai()**: the task must be in WAIT or WAIT-SUSPEND state, else E_OBJ

Task-Dependent Synchronization Functions

These functions are fully supported in this release:

```
ER sus_tsk(
    ID tskid )
ER rsm_tsk(
    ID tskid )
ER frsm_tsk(
    ID tskid )
ER slp_tsk( void )
ER tslp_tsk(
    TMO tmout )
ER wup_tsk(
    ID tskid )
ER can_wup(
     INT *p_wupcnt,
    ID tskid )
```

Error checking

The following conditions are only checked for, and only return errors if *CYGSEM_UITRON_BAD_PARAMS_RETURN_ERRORS* is enabled (see "Option: Return Error Codes for Bad Params", in Section V):

- invalid tskid; less than 1 or greater than CYGNUM_UITRON_TASKS returns E_ID
- **wup_tsk()**, **sus_tsk()**, **rsm_tsk()**, **frsm_tsk()** on the calling task returns E_OBJ
- dispatching is enabled in **tslp_tsk()** and **slp_tsk()**, or E_CTX
- tmout must be positive, otherwise E_PAR
- return value pointer in **can_wup()** is a valid pointer, or E_PAR

The following conditions are checked for, and can return error codes, regardless of the

setting of *CYGSEM_UITRON_BAD_PARAMS_RETURN_ERRORS*:

- When create and delete functions **cre_tsk()** and **del_tsk()** are supported, all calls which use a valid task ID number check that the task exists; if not, E_NOEXS is returned
- **sus_tsk()**: the task must not be dormant, else E_OBJ
- **frsm/rsm_tsk()**: the task must be suspended, else E_OBJ
- **tslp/slp_tsk()**: return codes E_TMOUT, E_RLWAI and E_DLT are returned depending on the reason for terminating the sleep
- **wup_tsk()** and **can_wup()**: the task must not be dormant, or E_OBJ is returned

Synchronization and Communication Functions

These functions are fully supported in this release:

```
ER sig_sem(
    ID semid )
ER wai_sem(
    ID semid )
ER preq_sem(
    ID semid )
ER twai_sem(
    ID semid,
    TMO tmout )
ER ref_sem(
    T_RSEM *pk_rsem ,
    ID semid )
ER set_flg(
    ID flgid,
    UINT setptn )
ER clr_flg(
    ID flgid,
    UINT clrptn )
ER wai_flg(
    UINT *p_flgptn,
    ID flgid ,
    UINT waiptn ,
    UINT wfmode )
```

```
ER pol_flg(
    UINT *p_flgptn,
    ID flgid ,
    UINT waiptn ,
    UINT wfmode )
ER twai_flg(
    UINT *p_flgptn
    ID flgid ,
    UINT waiptn ,
    UINT wfmode,
    TMO tmout )
ER ref_flg(
    T_RFLG *pk_rflg,
    ID flgid )
ER snd_msg(
    ID mbxid,
    T_MSG *pk_msg )
ER rcv_msg(
    T_MSG **ppk_msg,
    ID mbxid )
ER prcv_msg(
    T_MSG **ppk_msg,
    ID mbxid )
ER trcv_msg(
    T_MSG **ppk_msg,
    ID mbxid ,
    TMO tmout )
ER ref_mbx(
    T_RMBX *pk_rmbx,
    ID mbxid )
```

The following functions are supported in this release (with some restrictions) if enabled with the appropriate configuration option for the object type (for example *CYGPKG_UITRON_SEMAS_CREATE_DELETE*) :

```
ER cre_sem(
    ID semid,
    T_CSEM *pk_csem )
ER del_sem(
    ID semid )
ER cre_flg(
    ID flgid,
```

```
    T_CFLG *pk_cflg )
ER del_flg(
    ID flgid )
ER cre_mbx(
    ID mbxid,
    T_CMBX *pk_cmbx )
ER del_mbx(
    ID mbxid )
```

In general the queueing order when waiting on a synchronization object depends on the underlying kernel configuration. The multi-level queue scheduler is required for strict µITRON conformance and it queues tasks in FIFO order, so requests to create an object with priority queueing of tasks (`pk_cxxx->xxxatr = TA_TPRI`) are rejected with E_RSATR. Additional undefined bits in the attributes fields must be zero.

In general, extended information (pk_cxxx->exinf) is ignored.

For semaphores, the initial semaphore count (pk_csem->isemcnt) is supported; the new semaphore's count is set. The maximum count is not supported, and is not in fact defined in type pk_csem.

For flags, multiple tasks are allowed to wait. Because single task waiting is a subset of this, the W bit (TA_WMUL) of the flag attributes is ignored; all other bits must be zero. The initial flag value is supported.

For mailboxes, the buffer count is defined statically by kernel configuration option *CYGNUM_KERNEL_SYNCH_MBOX_QUEUE_SIZE*; therefore the buffer count field is not supported and is not in fact defined in type pk_cmbx. Queueing of messages is FIFO ordered only, so TA_MPRI (in pk_cmbx->mbxatr) is not supported.

Error checking

The following conditions are only checked for, and only return errors if *CYGSEM_UITRON_BAD_PARAMS_RETURN_ERRORS* is enabled:

- invalid object id; less than 1 or greater than CYGNUM_UITRON_TASKS/SEMAS/MBOXES as appropriate returns E_ID
- dispatching is enabled in any call which can sleep, or E_CTX
- tmout must be positive, otherwise E_PAR
- pk_cxxx pointers in **cre_xxx()** must be valid pointer, or E_PAR
- return value pointers in **ref_xxx()** is a valid pointer, or E_PAR
- flag wait pattern must be non-zero, and mode must be valid, or E_PAR
- return value pointer in flag wait calls is a valid pointer, or E_PAR

The following conditions are checked for, and can return error codes, regardless of the setting of *CYGSEM_UITRON_BAD_PARAMS_RETURN_ERRORS*:

- When create and delete functions **cre_xxx()** and **del_xxx()** are supported, all

calls which use a valid object ID number check that the object exists. If not, E_NOEXS is returned.

- In create functions **`cre_xxx()`**, when supported, if the object already exists, then E_OBJ
- In any call which can sleep, such as **`twai_sem()`**: return codes E_TMOUT, E_RLWAI, E_DLT or of course E_OK are returned depending on the reason for terminating the sleep
- In polling functions such as **`preq_sem()`**return codes E_TMOUT or E_OK are returned depending on the state of the synchronization object
- In creation functions, the attributes must be compatible with the selected underlying kernel configuration: in **`cre_sem()`** `pk_csem->sematr` must be equal to `TA_TFIFO` else E_RSATR.
- In **`cre_flg()`** `pk_cflg->flgatr` must be either `TA_WMUL` or `TA_WSGL` else E_RSATR.
- In **`cre_mbx()`** `pk_cmbx->mbxatr` must be `TA_TFIFO + TA_MFIFO` else E_RSATR.

Extended Synchronization and Communication Functions

None of these functions are supported in this release.

Interrupt management functions

These functions are fully supported in this release:

```
void ret_int( void )
ER loc_cpu( void )
ER unl_cpu( void )
ER dis_int(
    UINT eintno )
ER ena_int(
    UINT eintno )
voidret_wup(
    ID tskid )
ER iwup_tsk(
    ID tskid )
ER isig_sem(
```

```
    ID semid )
ER iset_flg(
    ID flgid ,
    UID setptn)
ER isend_msg(
    ID mbxid ,
    T_MSG *pk_msg)
```

Note that **ret_int()** and the **ret_wup()** are implemented as macros, containing a "return" statement.

Also note that**ret_wup()** and the **ixxx_yyy()** style functions will only work when called from an ISR whose associated DSR is **cyg_uitron_dsr()**, as specified in include file <cyg/compat/uitron/uit_ifnc.h>, which defines the **ixxx_yyy()** style functions also. Do not use them from a DSR: use plain **ixxx_yyy()** style functions instead.

The following functions are not supported in this release:

```
ER def_int(
    UINT dintno,
        T_DINT *pk_dint )
ER chg_iXX(
    UINT iXXXX )
ER ref_iXX(
    UINT * p_iXXXX )
```

These unsupported functions are all Level C (CPU dependent). Equivalent functionality is available via other **eCos**-specific APIs.

Error checking

The following conditions are only checked for, and only return errors if *CYGSEM_UITRON_BAD_PARAMS_RETURN_ERRORS* is enabled:

- **loc/unl_cpu()**: these must only be called in a µITRON task context, else E_CTX.
- **dis/ena_int()**: the interrupt number must be in range as specified by the platform HAL in qustion, else E_PAR.

Memory pool Management Functions

These functions are fully supported in this release:

```
ER get_blf(
    VP *p_blf,
```

```
    ID mpfid )
ER pget_blf(
    VP *p_blf,
    ID mpfid )
ER tget_blf(
    VP *p_blf,
    ID mpfid,
    TMO tmout )
ER rel_blf(
    ID mpfid,
    VP blf )
ER ref_mpf(
    T_RMPF *pk_rmpf,
    ID mpfid )
ER get_blk(
    VP *p_blk,
    ID mplid,
    INT blksz )
ER pget_blk(
    VP *p_blk,
    ID mplid,
    INT blksz )
ER tget_blk(
    VP *p_blk,
    ID mplid,
    INT blksz,
    TMO tmout )
ER rel_blk(
    ID mplid,
    VP blk )
ER ref_mpl(
    T_RMPL *pk_rmpl,
    ID mplid )
```

Note that of the memory provided for a particular pool to manage in the static initialization of the memory pool objects, some memory will be used to manage the pool itself. Therefore the number of blocks * the blocksize will be less than the total memory size.

The following functions are supported in this release, when enabled with CYGPKG_UITRON_MEMPOOLVAR_CREATE_DELETE or CYGPKG_UITRON_MEMPOOLFIXED_CREATE_DELETE as appropriate, with

some restrictions:

```
ER cre_mpl(
    ID mplid,
    T_CMPL *pk_cmpl )
ER del_mpl(
    ID mplid )
ER cre_mpf(
    ID mpfid,
    T_CMPF *pk_cmpf )
ER del_mpf(
    ID mpfid )
```

Because of the static initialization facilities provided for system objects, a memory pool is allocated a region of memory to manage statically in the configuration. So while memory pools can be created and deleted, the same area of memory is used for that memory pool (memory pool ID number) each time. The requested variable pool size (pk_cmpl->mplsz) or the number of fixed-size blocks (pk_cmpf->mpfcnt) times the block size (pk_cmpf->blfsz) are checked for fitting within the statically allocated memory area, so if a create call succeeds, the resulting pool will be at least as large as that requested. For this reason **del_mpl()** and **del_mpf()** do not in any sense free the memory that was managed by the deleted pool for use by other pools; it may only be managed by a pool of the same object id.

For both fixed and variable memory pools, the queueing order when waiting on a synchronization object depends on the underlying kernel configuration. The multi-level queue scheduler is required for strict µITRON conformance and it queues tasks in FIFO order, so requests to create an object with priority queueing of tasks (pk_cxxx->xxxatr = TA_TPRI) are rejected with E_RSATR. Additional undefined bits in the attributes fields must be zero.

In general, extended information (pk_cxxx->exinf) is ignored.

Error checking

The following conditions are only checked for, and only return errors if *CYGSEM_UITRON_BAD_PARAMS_RETURN_ERRORS* is enabled:

- invalid object id; less than 1 or greater than *CYGNUM_UITRON_MEMPOOLVAR/MEMPOOLFIXED* as appropriate returns E_ID
- dispatching is enabled in any call which can sleep, or E_CTX
- tmout must be positive, otherwise E_PAR
- pk_cxxx pointers in **cre_xxx()** must be valid pointer, or E_PAR
- return value pointers in **ref_xxx()** is a valid pointer, or E_PAR

- return value pointers in get block routines is a valid pointer, or E_PAR
- blocksize request in get variable block routines is greater than zero, or E_PAR

The following conditions are checked for, and can return error codes, regardless of the setting of *CYGSEM_UITRON_BAD_PARAMS_RETURN_ERRORS*:

- When create and delete functions **cre_xxx()** and **del_xxx()** are supported, all calls which use a valid object ID number check that the object exists. If not, E_NOEXS is returned.
- When create functions **cre_xxx()** are supported, if the object already exists, then E_OBJ
- In any call which can sleep, such as **get_blk()**: return codes E_TMOUT, E_RLWAI, E_DLT or of course E_OK are returned depending on the reason for terminating the sleep
- In polling functions such as **pget_blk()**return codes E_TMOUT or E_OK are returned depending on the state of the synchronization object
- In creation functions, the attributes must be compatible with the selected underlying kernel configuration: in **cre_mpl()** `pk_cmpl->mplatr` must be equal to `TA_TFIFO` else E_RSATR.
- In **cre_mpf()** `pk_cmpf->mpfatr` must be equal to `TA_TFIFO` else E_RSATR.
- In creation functions, the requested size of the memory pool must not be larger than that statically configured for the pool else E_RSATR; see “Option: Static initializers”, in Section V. In **cre_mpl()** `pk_cmpl->mplsz` is the field of interest
- In **cre_mpf()** the product of `pk_cmpf->blfsz` and `pk_cmpf->mpfcnt` must be smaller than the memory statically configured for the pool else E_RSATR
- In functions which return memory to the pool **rel_blk()** and **rel_blf()**, if the free fails, for example because the memory did not come from that pool originally, then E_PAR is returned

Time Management Functions

These functions are fully supported in this release:

```
ER set_tim(
    SYSTIME *pk_tim )
```

CAUTION Setting the time may cause erroneous operation of the kernel, for example a task performing a wait with a time-out may never awaken.

```
ER get_tim(
    SYSTIME *pk_tim )
ER dly_tsk(
```

```
    DLYTIME dlytim )
ER def_cyc(
    HNO cycno,
    T_DCYC *pk_dcyc )
ER act_cyc(
    HNO cycno,
    UINT cycact )
ER ref_cyc(
    T_RCYC *pk_rcyc,
    HNO cycno )
ER def_alm(
    HNO almno,
    T_DALM *pk_dalm )
ER ref_alm(
    T_RALM *pk_ralm,
    HNO almno )
void ret_tmr( void )
    Error checking
```

The following conditions are only checked for, and only return errors if *CYGSEM_UITRON_BAD_PARAMS_RETURN_ERRORS* is enabled:

- invalid handler number; less than 1 or greater than *CYGNUM_UITRON_CYCLICS/ALARMS* as appropriate, or E_PAR
- dispatching is enabled in **dly_tsk()**, or E_CTX
- dlytim must be positive or zero, otherwise E_PAR
- return value pointers in **ref_xxx()** is a valid pointer, or E_PAR
- params within pk_dalm and pk_dcyc must be valid, or E_PAR
- cycact in **act_cyc()** must be valid, or E_PAR
- handler must be defined in **ref_xxx()** and **act_cyc()**, or E_NOEXS
- parameter pointer must be a good pointer in **get_tim()** and **set_tim()**, otherwise E_PAR is returned

The following conditions are checked for, and can return error codes, regardless of the setting of *CYGSEM_UITRON_BAD_PARAMS_RETURN_ERRORS*:

- **dly_tsk()**: return code E_RLWAI is returned depending on the reason for terminating the sleep

System Management Functions

These functions are fully supported in this release:

```
ER get_ver(
    T_VER *pk_ver )
ER ref_sys(
    T_RSYS *pk_rsys )
ER ref_cfg(
    T_RCFG *pk_rcfg )
```

Note that the information returned by each of these calls may be configured to match the user's own versioning system, and the values supplied by the default configuration may be inappropriate.

These functions are not supported in this release:

```
ER def_svc(
    FN s_fncd,
    T_DSVC *pk_dsvc )
ER def_exc(
    UINT exckind,
    T_DEXC *pk_dexc )
```

Error checking

The following conditions are only checked for, and only return errors if *CYGSEM_UITRON_BAD_PARAMS_RETURN_ERRORS* is enabled:

- return value pointer in all calls is a valid pointer, or E_PAR

Network Support Functions

None of these functions are supported in this release.

7

The eCos Hardware Abstraction Layer (HAL)

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

Caveat This document is an informal description of the HAL capabilities and is not intended to be full documentation, although it may be used as a source for such. It also describes the HAL as it is currently implemented for the architectures targeted in this release. Further work (described in “Future developments” on page 83), is needed to complete it.

Architecture, implementation and platform

We have identified three levels at which the HAL must operate. The *architecture* HAL abstracts the basic CPU architecture and includes things like interrupt delivery, context switching, CPU startup etc. The *platform* HAL abstracts the properties of the current platform and includes things like platform startup, timer devices, I/O register access and interrupt controllers. The *implementation* HAL abstracts properties that lie

between these two, such as architecture variants and on-chip devices. The boundaries between these three HAL levels are necessarily blurred.

In the current HAL structure, there are separate directory trees for the architectural and platform HALs. The implementation HAL is currently supported in one or other of these by means of conditional compilation depending on how generic a particular feature is expected to be. Thus processor variants are handled in the architectural HAL since they are likely to be generic to several implementations. On-chip devices are handled in the platform HAL, if they impact the kernel, or as proper device drivers (and are thus outside the HAL).

The one area where there is significant interaction between these HAL layers is in the interrupt delivery VSR. Here the VSR, which is in the architectural HAL, may need to interrogate an interrupt controller to dispatch the correct ISR. The interrupt controller may be defined by the platform or implementation HAL. This is normally only a few instructions so is currently handled by conditional compilation. If this proves to become unwieldy, a mechanism for including platform code in the architectural HAL may be needed.

General principles

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

1. The HAL is implemented in C and assembler, although the **eCos** kernel is largely implemented in C++. This is to permit the HAL the widest possible applicability.
2. All interfaces to the HAL are implemented by CPP macros. This allows them to be implemented as inline C code, inline assembler or function calls to external C or assembler code. This allows the most efficient implementation to be selected without affecting the interface. It also allows them to be redefined if the platform HAL needs to replace or enhance a definition from the architecture HAL.
3. The HAL provides simple, portable mechanisms for dealing with the hardware of a wide range of architectures and platforms. It is always possible to bypass the HAL and program the hardware directly, but this may lead to a loss of portability.

Architectural HAL files

`hal/ARCH/arch/`**`v1_3_x`**`/include/basetype.h`

This file defines the properties of the base architecture that are used to compile the portable parts of the kernel. It is included automatically by `cyg/infra/cyg_type.h`. The following definitions may be included.

Byte order

`CYG_BYTEORDER`

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

Label translation

`CYG_LABEL_NAME(name)`

This is a wrapper used in some C and C++ files which specify labels defined in assembly code or the linker script. It need only be defined if the default implementation in `cyg/kernel/ktypes.h`, which passes the name argument unaltered, is inadequate. The most usual alternative definition of this macro prepends an underscore to the label name. This depends on the labeling convention of the tool set.

Base types

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

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

Atomic types

```
cyg_halatomic
CYG_ATOMIC
```

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

`hal/ARCH/arch/`**`v1_3_x`**`/include/hal_arch.h`

This file contains definitions that are related to the basic architecture of the CPU.

Register save format

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

```
} HAL_SavedRegisters;
```

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

Thread context initialization

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

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

sp

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

arg

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

entry

The address of an entry point function. This will be called according the C calling conventions, and the value of *`arg`* will be passed as the first argument.

id

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

Thread context switching

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

This macro implements the thread switch code. The arguments are:

from

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

to

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

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

Note that interrupts are not disabled during this process, any interrupts that occur will be delivered onto the stack to which the current value of the CPU stack pointer points. Hence the stack pointer should never be invalid, or loaded with a value that might cause the saved state to become corrupted by an interrupt.

Bit indexing

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

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

Idle thread activity

```
HAL_IDLE_THREAD_ACTION( count )
```

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

Reorder barrier

```
HAL_REORDER_BARRIER()
```

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

Breakpoint support

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

These macros provide support for breakpoints.

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

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

GDB support

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

These macros provide support for interfacing GDB to the HAL.

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

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

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

Setjmp and longjmp support

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

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

`hal/ARCH/arch/v1_3_x/include/hal_intr.h`

This file contains definitions related to interrupt handling.

Vector numbers

```
CYGNUM_HAL_VECTOR_XXX
CYGNUM_HAL_VSR_MIN
CYGNUM_HAL_VSR_MAX
CYGNUM_HAL_ISR_MIN
CYGNUM_HAL_ISR_MAX
CYGNUM_HAL_EXCEPTION_MIN
CYGNUM_HAL_EXCEPTION_MAX
CYGNUM_HAL_ISR_COUNT
CYGNUM_HAL_VSR_COUNT
CYGNUM_HAL_EXCEPTION_COUNT
```

All possible interrupt and exception vectors should be specified here, together with

maximum and minimum values for range checking.

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

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

Where a CPU supports synchronous exceptions, the range of such exceptions allowed are defined by *CYGNUM_HAL_EXCEPTION_MIN* and *CYGNUM_HAL_EXCEPTION_MAX*. The actual exception numbers will normally correspond to the VSR exception range. In future other exceptions generated by the system software (such as stack overflow) may be added.

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

Interrupt state control

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

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

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

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

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

HAL_QUERY_INTERRUPTS() stores the state of the interrupt mask in the variable passed in the *state* argument.

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

the other interrupt types.

ISR and VSR management

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

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

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

HAL_INTERRUPT_DETACH() detaches the ISR from the vector.

HAL_VSR_SET() replaces the VSR attached to the *vector* with the replacement supplied in *vsr*. The old VSR is returned in the location pointed to by *pvsr*.

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

Interrupt controller management

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

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

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

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

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

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

vector

The interrupt to be configured.

level

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

up

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

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

vector

The interrupt whose level is to be set.

level

The priority level to which the interrupt is to set. In some architectures the set interrupt level is also used as an interrupt enable/disable. Hence this function and **`HAL_INTERRUPT_MASK()`** and **`HAL_INTERRUPT_UNMASK()`** may interfere with each other.

Clock control

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

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

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

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

`HAL_CLOCK_READ()` reads the current value of the clock counter and puts the value in the location pointed to by *`pvalue`*. The value stored will always be the number of clock ‘‘ticks” since the last interrupt, and hence ranges between zero and the initial period value.

`hal/ARCH/arch/v1_3_x/include/hal_io.h`

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

Register address

`HAL_IO_REGISTER`

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

Values of variables and constants of this type will usually be supplied by configuration mechanisms.

Register read

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

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

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

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

Register write

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

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

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

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

`hal/ARCH/arch/v1_3_x/include/hal_cache.h`

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

There are versions of the macros defined here for both the Data and Instruction caches. these are distinguished by the use of either DCACHE or ICACHE in the macro names. In the following descriptions, XCACHE is also used to stand for either

of these. Where there are issues specific to a particular cache, this will be explained in the text.

There might be restrictions on the use of some of the macros which it is the user's responsibility to comply with. Such restrictions are documented in the `hal_cache.h` file.

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

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

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

Cache dimensions

```
HAL_XCACHE_SIZE
HAL_XCACHE_LINE_SIZE
HAL_XCACHE_WAYS
HAL_XCACHE_SETS
```

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

HAL_XCACHE_SIZE

gives the total size of the cache in bytes.

HAL_XCACHE_LINE_SIZE

gives the cache line size in bytes.

HAL_XCACHE_WAYS

gives the number of ways in each set and defines its level of associativity. This would be 1 for a direct mapped cache.

HAL_XCACHE_SETS

gives the number of sets in the cache, and is derived from the previous values.

Global cache control

```
HAL_XCACHE_ENABLE()
```

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

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

HAL_XCACHE_ENABLE() and HAL_XCACHE_DISABLE()

enable and disable the cache.

HAL_XCACHE_INVALIDATE_ALL()

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

HAL_XCACHE_SYNC()

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

HAL_XCACHE_BURST_SIZE()

allows the size of cache to/from memory bursts to be controlled. This macro will only be defined if this functionality is available.

HAL_DCACHE_WRITE_MODE()

controls the way in which data cache lines are written back to memory. There will be definitions for the possible modes. Typical definitions are HAL_DCACHE_WRITEBACK_MODE and HAL_DCACHE_WRITETHRU_MODE. This macro will only be defined if this functionality is available.

HAL_XCACHE_LOCK()

causes data to be locked into the cache. The base and size arguments define the memory region that will be locked into the cache. It is architecture dependent whether more than one locked region is allowed at any one time, and whether this operation causes the cache to cease acting as a cache for addresses outside the region during the duration of the lock. This macro will only be defined if this functionality is available.

HAL_XCACHE_UNLOCK()

cancels the locking of the memory region given. This should normally correspond

to a region supplied in a matching lock call. This macro will only be defined if this functionality is available.

HAL_XCACHE_UNLOCK_ALL()

cancels all existing locked memory regions. This may be required as part of the cache initialization on some architectures. This macro will only be defined if this functionality is available.

Cache line control

```
HAL_DCACHE_ALLOCATE( base , size )
HAL_DCACHE_FLUSH( base , size )
HAL_XCACHE_INVALIDATE( base , size )
HAL_DCACHE_STORE( base , size )
HAL_DCACHE_READ_HINT( base , size )
HAL_DCACHE_WRITE_HINT( base , size )
HAL_DCACHE_ZERO( base , size )
```

All of these macros apply a cache operation to all cache lines that match the memory address region defined by the base and size arguments. These macros will only be defined if the described functionality is available. Also, it is not guaranteed that the cache function will only be applied to just the described regions, in some architectures it may be applied to the whole cache.

HAL_DCACHE_ALLOCATE()

allocates lines in the cache for the given region without reading their contents from memory, hence the contents of the lines is undefined. This is useful for preallocating lines which are to be completely overwritten, for example in a block copy operation.

HAL_DCACHE_FLUSH()

invalidates all cache lines in the region after writing any dirty lines to memory.

HAL_XCACHE_INVALIDATE()

invalidates all cache lines in the region. Any dirty lines are invalidated without being written to memory.

HAL_DCACHE_STORE()

writes all dirty lines in the region to memory, but does not invalidate any lines.

HAL_DCACHE_READ_HINT()

hints to the cache that the region is going to be read from in the near future. This may cause the region to be speculatively read into the cache.

HAL_DCACHE_WRITE_HINT()

hints to the cache that the region is going to be written to in the near future. This may have the identical behavior to HAL_DCACHE_READ_HINT().

HAL_DCACHE_ZERO()

allocates and zeroes lines in the cache for the given region without reading memory. This is useful if a large area of memory is to be cleared.

`hal/ARCH/arch/v1_3_x/src/ARCH.ld`

This is the architecture specific linker script file. It defines the section types required for the architecture. During preprocessing, the memory layout specified for the chosen platform and startup type is included, defining region, alignment and location parameters for the sections.

`hal/ARCH/arch/v1_3_x/src/vectors.S`

This file contains code to deal with exception and interrupt vectors. Since the reset entry point is usually implemented as one of these it also deals with system startup.

The exact implementation of this code is under the control of the HAL implementer. So long as it interacts correctly with the macros defined in `hal_intr.h` it may take any form. However, all current implementation follow the same pattern, and there should be a very good reason to break with this. The rest of this section describes how the standard HAL implementation operates.

This file usually contains the following sections of code:

- Startup and initialization code.
- Exception delivery.
- Default handling of synchronous exception.
- Default handling of interrupts.

HAL startup

Execution normally begins at the reset vector with the machine in a minimal startup state.

The following is a list of the jobs that need to be done in approximately the order in which they should be accomplished. Many of these will not be needed in some configurations.

- Initialize various CPU status registers. Most importantly, the CPU interrupt mask should be set to disable interrupts.
- Set up any CPU memory controller to access RAM, ROM and I/O devices correctly. Until this is done it may not be possible to access RAM.
- Enable the cache, if it is to be used. This may require enabling the CPU's memory management system since that is often the only way of controlling the cacheability of memory. If this is necessary, a direct one-to-one mapping between physical and virtual memory is most desirable.
- Set up the stack pointer, this allows subsequent initialization code to make procedure calls.

- Initialize any global pointer register needed for access to globally defined variables. This allows subsequent initialization code to access global variables.
- Perform any platform specific initialization. This is best accomplished by calling an initialization routine in `PLATFORM.S` (see "`hal/ARCH/PLATFORM/v1_3_x/src/PLATFORM.S`" on page 83).
- If the system is starting from ROM, copy the ROM template of the `.data` section out to its correct position in RAM. (See "`hal/ARCH/arch/v1_3_x/src/ARCH.ld`" on page 79).
- Zero the `BSS` section.
- Create a suitable C call stack frame.
- Call **`cyg_hal_invoke_constructors()`** to run any static constructors.
- Call **`cyg_start()`**. If **`cyg_start()`** returns, drop into an infinite loop.

Vectors and VSRs

The CPU delivers all exceptions whether synchronous or interrupts to a set of vectors. Depending on the architecture, these may be implemented in a number of different ways. Examples of existing mechanisms are:

PowerPC

Exceptions are vectored to locations 256 bytes apart starting at either zero or `0xFFF00000`. There are 16 such vectors defined by the architecture and extra vectors may be defined by specific implementations.

MIPS

All exceptions are vectored to a single address and software is responsible for reading the exception code from a CPU register to discover its true source.

MN10300

External interrupts are vectored to an address stored in one of seven interrupt vector registers. These only supply the lower 16 bits of the address, the upper 16 bits are fixed to `0x4000XXXX`. Hence the service routine is constrained to the 64k range starting at `0x40000000`.

Pentium

Exceptions are delivered via an Interrupt Descriptor Table (IDT) which is essentially an indirection table indexed by exception type. The IDT may be placed anywhere in memory. In PC hardware the interrupt controller can be programmed to deliver the external interrupts to a block of 16 vectors at any offset in the IDT.

680X0

Exceptions are delivered via an indirection table described by a CPU base register (for X > 0). External interrupts are either delivered via a set of level-specific

vectors defined by the architecture, or a vector number may be supplied by the device in which case another entry in the table may be used.

The model adopted by the HAL is that VSRs should be easily replaceable with a pointer to an alternative routine. Of the above architectures, only the Pentium and 680X0 allow this directly in the hardware. In the other three, extra software is required. The code attached directly to the vector is a short trampoline that indirects by way of a HAL supplied VSR table to the true VSR. In the PowerPC and MN10300 the table offset is implicit in the vector routine called, for the MIPS the code reads the *cause* register and indirects through the appropriate table entry.

Default exception handling

Most synchronous exception vectors will point to a default exception VSR which is responsible for handling all exceptions in a generic manner.

Since most exceptions handled by this VSR are errors (or breakpoints when a program is being debugged), its default behavior should be to save the entire machine state, disable interrupts, and invoke the debugger's entry point, passing it a pointer to the saved state.

If the debugger returns then the saved state is restored and the interrupted code resumed. Since the debugger may adjust the saved state while it runs a little care must be taken to restore the state correctly.

Default interrupt handling

Most external interrupt vectors will point to a default interrupt VSR which decode the actual interrupt being delivered and invokes the appropriate ISR.

The default interrupt VSR has a number of responsibilities if it is going to interact with the Kernel cleanly and allow interrupts to cause thread preemption.

To support this VSR an ISR vector table is needed. For each valid vector three pointers need to be stored: the ISR, its data pointer and an interrupt object pointer needed by the kernel. It is implementation defined whether these are stored in a single table of triples, or in three separate tables.

The VSR should follow the following approximate plan:

- Save the CPU state. In non-debug configurations, it may be possible to get away with saving less than the entire machine state.
- Increment the kernel scheduler lock. This is a static member of the Cyg_Scheduler class. It may be necessary to look at a objdump or assembler listing of `sched.cxx` to discover its mangled label.
- (Optional) Switch to an interrupt stack if not already running on it. This allows nested interrupts to be delivered without needing every thread to have a stack large enough to take the maximum possible nesting. It is implementation defined

how to detect whether this is a nested interrupt.

- (Optional) Re-enable interrupts to permit nesting.
- Decode the actual external interrupt being delivered from the interrupt controller. This will yield the ISR vector number.
- Using the ISR vector number as an index, retrieve the ISR pointer and its data pointer from the ISR vector table.
- Construct a C call stack frame.
- Call the ISR, passing the vector number and data pointer. The vector number and a pointer to the saved state should be preserved across this call, preferably by storing them in registers that are defined to be callee-saved by the calling conventions.
- If this is an un-nested interrupt and a separate interrupt stack is being used, switch back to the interrupted thread's own stack.
- (Optional) If interrupts were not enabled above, enable them here since the **`interrupt_end()`** function must be called with interrupts enabled.
- Use the saved ISR vector number to get the interrupt object pointer from the ISR vector table.
- Call **`interrupt_end()`** passing it the return value from the ISR, the interrupt object pointer and a pointer to the saved CPU state. This function is implemented by the Kernel and is responsible for finishing off the interrupt handling. Specifically, it may post a DSR depending on the ISR return value, and will decrement the scheduler lock. If the lock is zeroed by this then it may result in a thread context switch.
- When **`interrupt_end()`** returns, restore the machine state and resume execution of the interrupted thread. Depending on the architecture, it may be necessary to disable interrupts again for part of this.

The detailed order of these steps may vary slightly depending on the architecture, in particular where interrupts are enabled and disabled.

`hal/ARCH/arch/v1_3_x/src/hal_misc.c`

This file contains any miscellaneous functions that are reference by the HAL. Typical functions that might go here are C implementations of the least- and most- significant bit index routines, constructor calling functions such as **`cyg_hal_invoke_constructors()`** and support routines for the exception and interrupt vector handling.

`hal/ARCH/PLATFORM/v1_3_x/include/pkgconf/STARTUP.mlt`

For each startup type (STARTUP) the memory layout of the sections is defined. This information may be edited using the **Configuration Tool** only.

`hal/ARCH/PLATFORM/v1_3_x/include/pkgconf/STARTUP.ldi`

For each startup type (STARTUP) the memory layout of the sections is exported by the **Configuration Tool** as a linker script fragment suitable for inclusion within the architecture-specific linker script file during preprocessing. The linker script fragment to be included is specified by the CYGHWR_MEMORY_LAYOUT_LDI macro in the `system.h` header file. The linker script fragments will be overwritten by the **Configuration Tool** and should only edited manually where the **Configuration Tool** is not in use.

`hal/ARCH/PLATFORM/v1_3_x/include/hal_diag.h`

During early development it is useful to have the ability to output messages to some default destination. This may be a memory buffer, a simulator supported output channel, a ROM emulator virtual UART or a serial line. This file defines set of macros that provide simple, polled output for this purpose.

`HAL_DIAG_INIT()` performs any initialization required on the device being used to generate diagnostic output. This may include setting baud rate, and stop, parity and character bits.

`HAL_DIAG_WRITE_CHAR(c)` writes the character supplied to the diagnostic output device.

These macros may either implement the required functionality directly, or may call functions elsewhere in the HAL to do it. In the latter case these should be in the file hal/ARCH/PLATFORM/v1_3_x/src/hal_diag.c.

`hal/ARCH/PLATFORM/v1_3_x/src/PLATFORM.S`

This is a platform specific assembly code file. Its main purpose is to contain any platform specific startup code called from `vectors.S`.

`hal/ARCH/PLATFORM/v1_3_x/src/context.S`

If present, this is an assembly code file that contains the code to support thread contexts. The routines to switch between various contexts, as well as initialize a thread context may be present in this file.

`hal/ARCH/PLATFORM/v1_3_x/src/hal_diag.c`

If present, this file contains the implementation of the HAL diagnostic support routines.

Future developments

The HAL is not complete, and will evolve and increase over time. Among the intended developments are:

- Common macros for interpreting the contents of a saved machine context. These would allow portable code, such as debug stubs, to extract such values as the

program counter and stack pointer from a state without having to interpret a `HAL_SavedRegisters` structure directly.

- Debugging support. Macros to set and clear hardware and software breakpoints. Access to other areas of machine state may also be supported.
- Floating point support. The saving and restoring of floating point state may need to added to the HAL for those architectures that support it. The exact mechanisms provided need to be defined.
- Static initialization support. The current HAL provides a dynamic interface to things like thread context initialization and ISR attachment. We also need to be able to define the system entirely statically so that it is ready to go on restart, without needing to run code. This will require extra macros to define these initializations. Such support may have a consequential effect on the current HAL specification.
- CPU state control. Many CPUs have both kernel and user states. Although it is not intended to run any code in user state for the foreseeable future, it is possible that this may happen eventually. If this is the case, then some minor changes may be needed to the current HAL API to accommodate this. These should mostly be extensions, but minor changes in semantics may also be required.
- Physical memory management. Many embedded systems have multiple memory areas with varying properties such as base address, size, speed, bus width, cacheability and persistence. An API is needed to support the discovery of this information about the machine's physical memory map.
- Memory management control. Some embedded processors have a memory management unit. In some cases this must be enabled to allow the cache to be controlled, particularly if different regions of memory must have different caching properties. For some purposes, in some systems, it will be useful to manipulate the MMU settings dynamically.
- Power management. Macros to access and control any power management mechanisms available on the CPU implementation. These would provide a substrate for a more general power management system that also involved device drivers and other hardware components.
- Generic serial line macros. Most serial line devices operate in the same way, the only real differences being exactly which bits in which registers perform the standard functions. It should be possible to develop a set of HAL macros that provide basic serial line services such as baud rate setting, enabling interrupts, polling for transmit or receive ready, transmitting and receiving data etc. Given these it should be possible to create a generic serial line device driver that will allow rapid bootstrapping on any new platform. It may be possible to extend this mechanism to other device types.

- Porting Guide. As the HAL develops it will become important to perform a port to a new architecture in the correct way.

Kernel porting notes

This section briefly describes the issues involved in porting **eCos** to a new target platform and/or architecture.

Porting overview

The effort required to port **eCos** to a new target varies. Adding support for a new platform/board may require almost no effort, while adding support for a new architecture is more demanding. Additionally, new device drivers may have to be written if there is no existing support for the target's devices.

Given that there are usually more target platforms using the same microprocessor or microcontroller, adding **eCos** support for a new target would often be a question of adding support for the new target platform. The architectures supported by **eCos** include the following: ARM7, MIPS (TX39), MN10300, PowerPC (MPC8xx), and SPARClite.

Adding a new architecture support is a bigger job and also requires tool support (**GCC**, **GDB** and binutils) which is a big undertaking in itself.

Platform support

Adding support for a new platform requires (a subset of):

1. Adding **eCos** configuration information.
2. Memory layout description.
3. Memory controller initialization.
4. Interrupt controller handling.
5. Minimal serial device driver for **GDB** interaction and simple diagnostics output.
6. System timer initialization and control.
7. Wallclock driver.

   A wallclock emulation based on the system timer is provided with the standard **eCos** distribution. For those hardware platforms where a battery backed-up clock device or other means of determining actual wallclock time exists, a wallclock driver may be implemented more fully.

If the architecture in question is a microcontroller (as opposed to a microprocessor), the job of porting may be as simple as adding configuration information and defining a new memory layout (items one and two). Currently **eCos** supports the following

microcontrollers: MN10300, MPC8xx, and TX39.

Architectural support

Adding support for a new architecture requires:

1. Adding **eCos** configuration information.
2. Writing a HAL for the CPU core's register model, interrupt and exception model, cache model, and possibly simple handling for the MMU model.
3. For microcontrollers the HAL should also support the memory controller, interrupt controller and a possible on-MCP serial controller for **GDB** interaction and simple diagnostics output, system timer initialization and control, and a wallclock driver.

If there is already support for a member of the same architecture family, the porting job may just consist of adding extra feature support to the existing HAL. Or if the new target architecture only defines a subset of the architecture family, the HAL may need additional configuration control, allowing parts of the existing HAL code to be disabled.

Adding configuration information

Architecture and platform configuration information resides in two top-level files `targets` and `packages` as well as in architecture and platform specific configuration files (`hal/<arch>/arch/current/include/pkgconf/hal_<arch>.h` and `hal/<arch>/<platform>/current/include/pkgconf/hal_<arch>_<platform>.h`. Furthermore, each platform must define memory layouts for each startup type.

targets

Architecture and platform information must be added to the `targets` file.

```
target powerpc {
    alias               { PowerPC powerpc-eabi }
    command_prefix      powerpc-eabi
    packages            { CYGPKG_HAL_POWERPC }
    hal                 hal/powerpc/arch

    cflags {
        ARCHFLAGS       "-mcpu=860 -D_SOFT_FLOAT"
        ERRFLAGS        "-Wall -Wpointer-arith -Wstrict-prototypes -Winline
-Wundef"
        CXXERRFLAGS     "-Woverloaded-virtual"
        LANGFLAGS       "-ffunction-sections -fdata-sections"
        DBGFLAGS        "-g -O2"
        CXXLANGFLAGS    "-fno-rtti -fno-exceptions -fvtable-gc -finit-priority"
        LDLANGFLAGS     "-Wl,--gc-sections -Wl,-static"
    }
```

```
    platform cogent {
        alias           { "Cogent board" }
        startup         { ram rom stubs }
        packages        {
            CYGPKG_HAL_POWERPC_COGENT
            CYGPKG_DEVICES_WALLCLOCK
            CYGPKG_DEVICES_WATCHDOG
        }
    }
}
```

pkgconf uses the entries in `targets` to create a build tree. The *--target* option matches target name (`powerpc`) or its aliases (`PowerPC powerpc-eabi`), just as the *--platform* option matches platform name (`cogent`) or its aliases (`Cogent board`). The same is true for the *--startup* option which matches on the list of valid startup types (`ram`, `rom` and `stubs`).

The `command_prefix` is the prefix on the cross compiler tools, usually the same target triplet used when configuring the tools (`powerpc-eabi`).

`packages` lists the hardware-related packages that should be enabled if this target is selected. Typically this will just be the appropriate architectural HAL package provided for this architecture (`CYGPKG_HAL_POWERPC`), while `hal` specifies the relative path of the source files.

`cflags` specifies the compiler and linker flags. The *-finit-priority* flag is required for proper initialization of **eCos**, while *-ffunction-sections*, *-fdata-sections*, and *-Wl,--gc-sections* are required to provide linker garbage collection which removes functions and initialized data that are not going to be used. The other `FLAGS` definitions can be set according to preference, taking care to ensure that `ARCHFLAGS` contains all necessary flags for the particular architecture.

The `platform` option is used to define a new target platform. There can be several of these for each architecture. The name and startup types are defined using `platform`, `alias`, and `startup` as described above. `packages` defines the set of packages supported by this particular platform. This set must include the platform HAL package (`CYGPKG_HAL_POWERPC_COGENT`), but can name other packages (`CYGPKG_DEVICES_WALLCLOCK` and `CYGPKG_DEVICES_WATCHDOG`) which will be enabled per default when selecting this architecture/platform configuration.

packages

The individual packages must be defined in the `packages` file.

```
package CYGPKG_HAL_POWERPC {
        alias           { "PowerPC common HAL" hal_powerpc powerpc_hal
powerpc_arch_hal }
        directory       hal/powerpc/arch
```

```
        include_dir     cyg/hal
        hardware
}

package CYGPKG_HAL_POWERPC_COGENT {
        alias           { "PowerPC Cogent board support" hal_powerpc_cogent
powerpc_cogent_hal }
        directory       hal/powerpc/cogent
        include_dir     cyg/hal
        hardware
}
```

These are the definitions of the two packages named in the `targets` file. The aliases can be used with the *--disable-* and *--enable-* options of **pkgconf**.

`directory` specifies the relative path of the source files, `include_dir` where header files provided by the package should be copied to in the install directory, and `hardware` specifies that these packages is normally associated with specific hardware and should only be enabled for the appropriate hardware.

Package-specific configuration

The package-specific configuration files provide presentation information used by the Configuration Tool, dependencies on other packages and of course additional fine-grained options that are architecture and/or target specific. See the two files `hal/powerpc/arch/current/include/pkgconf/hal_powerpc.h` and `hal/powerpc/cogent/current/include/pkgconf/hal_powerpc_cogent.h` for an example.

Memory layout information

For each target platform must be defined the memory layout used for any given startup type. This information resides in two files `mlt_<arch>_<platform>_<startup>.ldi` and `mlt_<arch>_<platform>_<startup>.mlt` in the directory `hal/<arch>/<platform>/current/include/pkgconf/`. The former is a linker script fragment, the latter a file describing the layout for the **eCos Configuration Tool**.

Redefining the memory layout can be done in the **Configuration Tool**, which will create the linker script (the `.ldi` file). It is also possible to do by hand, in which case only the linker script should be created; when no `.mlt` file exists, the **Configuration Tool** will not overwrite the default linker script.

Platform porting

Platform porting basically consists of making a copy of an existing platform directory and changing the code to match the new platform. The header and source files in the platform directory and their contents are described in “Architectural HAL files”

on page 67.

In particular the configuration information and memory layout need changing, as may the board initialization code and the minimal serial drivers used by `hal_diag.c` and `plf_stub.c`.

Another useful reference for porting to a new platform is the GNUPro documentation on `gdb` stubs, which can be found at

`http://www.cygnus.com/pubs/gnupro/3_GNUPro_Debugging_Tools/b_Debugging_with_GDB/gdbThe_GDB_remote_serial_protocol.html`

Architectural porting

The easiest way to make a new architectural port of **eCos** is to make a copy of an existing HAL and change the code to suit the new CPU. This guide will use the PowerPC Cogent board as an example. Wherever *powerpc*, *ppc*, or *cogent* is mentioned in this guide or in the source files, you should replace the strings with appropriate architecture and platform names. There are also a few files that need renaming.

If there is simulator support for the new CPU it is possible to test big parts of the HAL and the rest of the **eCos** kernel before a port to a specific platform is attempted. This is an advantage as doing a platform port can cause problems of it own, making it difficult to determine whether the architectural or platform parts of the port in progress are to blame when something is not working properly.

When no simulator support exists, the starting point of a port is to produce a minimal GDB stub for the target platform, which will allow code to be downloaded, executed and/or debugged on the board. This guide is based on a situation where no simulator exists as it would be the most likely scenario.

Writing an eCos GDB stub

A **GDB** stub has both a architectural part (description of the CPUs registers, exception decoding, breakpoint and stepping model, etc.) and a platform part (board initialization and simple serial driver).

Writing a stub is a subset of the work required to a full architectural and platform port of the HAL (and thus **eCos**). The below sections will be a rough list of minimal requirements for a stub; remaining elements of the files can be fleshed out when extending the port to include full **eCos** functionality. The files and their contents are described in “Architectural HAL files” on page 67.

TIP If the target board has an existing download stub (not necessarily **GDB** compliant), the **GDB** stub can be tested by changing it to run from RAM rather than ROM (using `ram` startup instead of `stubs` startup).

After downloading the stub and starting it, it should be possible to connect

GDB to the target. Note that trying to download another application may cause the memory of the stub to be overwritten, so some consideration is required when defining the memory layout.

If the target board does not have an existing download stub and requires a new EPROM to be burned for each testing cycle, you may want to start with writing a *minimal* stub which can only be used for downloading data to the target board.

For this purpose you can skip the exception support code in `vectors.S` and hack `hal/common/current/src/stubrom/stubrom.c` to jump directly to the stub code without using a breakpoint.

TIP While working on improving the stub code or other parts of the HAL you can use the simple diagnostics output functions (by way of diag_printf) as a crude way of providing debugging feedback until you get full **GDB** stub functionality in place.

TIP A good way of debugging the stub itself is to enable remote debugging in **GDB** (`set remotedebug 1`). This makes **GDB** display any communication between itself and the stub on the target. Consult the **GDB** file `remote.c` for details on the protocol.

Architecture files

`include/basetype.h`

Implement in full. Little effort.

`include/hal_arch.h`

The following macros are required for the stub: `HAL_SavedRegisters`, `HAL_BREAKPOINT`, `HAL_BREAKINST`, `HAL_BREAKINST_SIZE`, **`HAL_GET_GDB_REGISTERS`**, and **`HAL_SET_GDB_REGISTERS`**.

`include/hal_cache.h`

The macros in this file can be left as empty if caches are kept disabled. This is definitely the best way to start porting, avoiding cache problems entirely. The cache is not of much use until **eCos** can be used with applications anyway.

`include/hal_intr.h`

It is necessary to implement enough exception handling code to properly handle breakpoints.

As the porting job progresses, asynchronous break points (CYGDBG_HAL_DEBUG_GDB_BREAK_SUPPORT) may come in handy. These require a minimal interrupt system to be in place.

`include/hal_io.h`

Should be fully implemented. Usually zero effort.

`include/<arch>_regs.h`

Can be filled in piecemeal as the porting job progresses.

`include/<arch>_stub.h`

Redefine `NUMREGS`, `REGSIZE`, and `regnames` using the same register layout as **GDB**. The register definitions can be found in the `config/<arch>/tm-<arch>.h` file in the **GDB** sources. The definitions for the PowerPC were found in `config/rs6000/tm-rs6000.h`.

Discrepancies between what **GDB** expects and what is defined in the stub will show up when you use the `info reg` command in **GDB** (and know what the register contents on the target should be). Be careful to get the `REGSIZE` macro defined correctly.

`src/context.c`

Nothing here required by the stub.

`src/hal_misc.c`

The two functions **`cyg_hal_invoke_constructors`** and **`cyg_hal_exception_handler`** must be implemented. The former is the same on most architectures and the latter just needs to call **`__handle_exception`**.

`src/<arch>.ld`

The linker script must be properly defined.

`src/<arch>_stub.c`

This file must be fully implemented.

`__computeSignal` can be defined to just return SIGTRAP as a minimal implementation. Proper signal decoding may help debugging though.

Single-stepping can be implemented in one of two ways. Some architectures (such as the PowerPC) have hardware support to control single-stepping making it simple to implement. Other architectures require use of breakpoints to implement the functionality, which requires instruction decoding. Examples of the latter approach can be found in the ARM, MIPS, and MN10300 stubs. Implementing instruction decoding obviously requires more effort.

`src/vectors.S`

This is the core file of the architecture HAL. It is hard to define what the minimal implementation requirements are for stubs to work. It may be worth and/or necessary to do a full implementation of this file to start with, but here are some pointers anyway.

`_start` as defined for the PowerPC is about the minimum requirement, but you can ignore MMU and cache setup while working on the stub.

`__default_exception_vsr` and `restore_state` must preserve enough state to allow breakpoints without trashing CPU state for the application code. If you need asynchronous **GDB** breakpoints `__default_interrupt_vsr` must also be defined well enough to allow interrupts without trashing the CPU state of the interrupted application code.

Assorted tables also need to be defined, depending on how much of the exception and interrupt handlers is implemented.

Platform files

`include/hal_diag.h`

Shouldn't require any changes.

`include/plf_stub.h`

This file provides the interface to the platform stub functions for the `generic-stub.c` code.

The minimal stub (no asynchronous **GDB** breakpoints) only requires `HAL_STUB_PLATFORM_INIT_SERIAL`, `HAL_STUB_PLATFORM_GET_CHAR`, and `HAL_STUB_PLATFORM_PUT_CHAR` and the matching functions in `plf_stub.c` to be defined.

`src/<platform>.c`

This file defines `hal_hardware_init` which takes care of initializing the board. For the Cogent board this includes watchdog initialization and memory controller setup. Other boards may have different requirements.

`src/hal_diag.c`

This file defines three functions that provide simple diagnostics output; `hal_diag_init`, `hal_diag_write_char`, and `hal_diag_read_char`. Normally these would implement a very simple serial driver. They could also address an LCD or just some LEDs.

The simple serial driver for the Cogent board is implemented in a separate file, `cma_ser.c`, which is shared with the `plf_stub.c` file.

`src/plf_stub.c`

This file implements the serial driver needed by the **GDB** stub. The minimal stub only requires **`init`**, **`putc`**, and **`getc`** functions. A stub which supports asynchronous breakpoints also requires functions to handle serial interrupts. For example implementations see `cma_ser.c` or the `plf_stub.c` file for the MN10300 stdeval1 board.

Building the stub

1. Prepare a build directory, configuring **eCos** for `stubs` startup.
2. Disable all packages except ***eCos*** *common HAL*, *infrastructure*, *<arch> common HAL*, and *<arch> <platform> board support*.
3. Disable the HAL common options *CYGFUN_HAL_COMMON_KERNEL_SUPPORT* and *CYGDBG_HAL_DEBUG_GDB_THREAD_SUPPORT*.

Enable the HAL common option *CYGDBG_HAL_DEBUG_GDB_INCLUDE_STUBS*.

4. Build libtarget.
5. Change to the directory `hal/common/current/src/stubrom` and type `make`. This should result in an **eCos GDB** stub image file called `stubrom`. This can be converted to SRECord or binary format (using `objcopy`) which can be used by EPROM burner or PROM emulator software.

Filling in the blanks

When a **GDB** stub has been written and is working, finishing the HAL port is pretty much a question of completing the header files and writing the functions that were not needed for the stub.

8

eCos Interrupt Model

This chapter describes the **eCos** interrupt model in detail.

Interrupt handling is an important part of most real-time systems. Timely handling of interrupt sources is important. This can be severely impacted by certain activities that must be considered atomic (i.e. uninterruptible). Typically these activities are executed with interrupts disabled. In order to keep such activities to a minimum and allow for the smallest possible interrupt latencies, **eCos** uses a split interrupt handling scheme. In this scheme, interrupt handling is separated into two parts. The first part is known as the Interrupt Service Routine or ISR. The second part is the Deferred Service Routine or DSR. This separation explicitly allows for the DSRs to be run with interrupts enabled, thus allowing other potentially higher priority interrupts to occur and be processed while processing a lower priority interrupt.

In order for this model to work, the ISR should run quickly. If the service requirements for the interrupt are small, the interrupt can be completely handled by the ISR and no DSR is required. However, if servicing the interrupt is more complex, a DSR should be used. The DSR will be run at some later time, at the point when thread scheduling is allowed. Postponing the execution of DSRs until this time allows for simple synchronization methods to be used by the kernel.

Further, this controlled calling — when thread scheduling is allowed — means that DSRs can interact with the kernel, for example by signalling that an asynchronous operation has completed.

In order to allow DSRs to run with interrupts enabled, the ISR for a particular interrupt

source (or the hardware) must arrange that that interrupt will not recur until the DSR has completed. In some cases, this is how the hardware works. Once an interrupt is delivered another interrupt will not occur until re-enabled. In the general case, however, it is up to the ISR to enforce this behavior. Typically the ISR will "mask" the interrupt source, thus preventing its recurrence. The DSR will then unmask the interrupt when it has been serviced thus allowing new occurrences of the interrupt to be delivered when they happen.

Alternatively, if an ISR is doing very little per interrupt, for example transferring one byte from memory to an IO device, it may only be necessary to interact with the rest of the system when a "transfer" is complete. In such a case an ISR could execute many times and only when it reaches the end of a buffer does it need to request execution of its DSR.

If the interrupt source is "bursty", it may be OK for several interrupts and calls to the ISR to occur before a requested DSR has been executed; the kernel maintains counts for posted DSRs, and in such a case the DSR will eventually be called with a parameter that tells it how many ISRs requested that the DSR be called. Care is needed to get the interrupt code right for such a situation, for one call to the DSR is required to do the work of several.

As mentioned above, the DSR will execute at some later time. Depending on the state of the system, it may be executed at a much later time. There are periods during certain kernel operations where thread scheduling is disabled, and hence DSRs are not allowed to operate. These periods have been purposefully made as limited as possible in the **eCos** kernel, but they still exist. In addition, user threads have the ability to suspend scheduling as well, thus affecting the possible DSR execution latency. If a DSR cannot be executed sufficiently quickly, the interrupt source may actually overrun. This would be considered a system failure.

One of the problems system designers face is how much stack space to allow each thread in the system. **eCos** does not dictate the size of thread stacks, it is left to the user when the thread is created. The size of the stack depends on the thread requirements as well as some fixed overhead required by the system. In this case, the overhead is enough stack space to hold a complete thread state (the actual amount depends on the CPU architecture). Guidelines for the minimum stack requirements are provided by the HAL using the symbol *CYGNUM_HAL_STACK_SIZE_MINIMUM*.

A potential problem with this scheme is with nested interrupts. Since interrupts are reenabled during the DSR portion of servicing an interrupt, there is the possibility of a new interrupt (hopefully from a separate source) arriving while this processing takes place. When this new interrupt is serviced some state information about the interrupted processing will be saved on the stack. The amount of this information again depends on the CPU architecture and in some cases it is substantial. This implies that any given stack would need enough space to potentially hold "N" interrupt

frames. In a realtime system with many threads this is an untenable situation. To solve this problem, **eCos** allows for a separate interrupt stack to be used while processing interrupts. This stack needs to be large enough to support "N" nested interrupts, but each individual thread stack only needs the overhead of a single interrupt state. This is because the thread state is kept on the thread's own stack, including information about any interrupt that caused the thread to be scheduled. This is a much better situation in the end, however, since only the interrupt stack need be large enough to handle the potential interrupt servicing needs.

eCos allows for the use of the interrupt stack to be totally configurable. The user can elect to not use a separate interrupt stack. This requires making all thread stacks large enough but does reduce the overhead of switching stacks while processing interrupts. On the other hand, if memory is tight, then choosing a separate interrupt stack would be warranted at the cost of a few machine cycles during the processing of each interrupt.

Not all target HALs support this feature from day one anyway; however common configuration features such as this may still be presented in the config tool, and present in include files, even if the actual target selected does not support the feature at this time.

The following problem with the interrupt system has been observed. On the mn10300 simulator, interrupts were occurring immediately after they were re-enabled in the DSR. This should really be considered a case of interrupt overrun since there is no possibility of useful [or any] processing between the time an interrupt has been serviced and an subsequent interrupt occurs, hence the system is totally saturated. The problem came about because the stack was overflowing. It was a user [thread] stack that overflowed because DSR processing was taking place on the thread stack. Analysis of this problem led to a rework of how interrupts are processed, in particular the use of a separate interrupt stack during interrupt processing (both ISR and DSR parts). The overflow can still happen, but now it is restricted to only the interrupt stack. The system designer can make accommodations for this by making a suitably large interrupt stack if it is known that the "overrun" is finite, e.g. in the case of a serial device, this could be the depth of some FIFO. In any case, overrun should be avoided, but having only a single stack that needs to suffer multiple interrupt frames allows for this failure to be detected simply.

Of course, it is only worthwhile having a separate interrupt stack if you are using an **eCos** configuration that has a scheduler and multiple threads. If there is no kernel, then the C library arranges to call **`main()`**, or your application may be entered from **`cyg_user_start()`**, on the startup stack. It runs on the only stack there is in the system. Depending on the design of the particular HAL for your target platform, it is natural to re-use the startup stack as the interrupt stack as soon as the scheduler is running. Since this is only sensible if there is a kernel, HALs typically only implement

the separate interrupt stack if the kernel is present.

Part III: PCI Library

9

The eCos PCI Library

The PCI library is an optional part of eCos, and is only applicable to some packages.

PCI Library

The eCos PCI library provides the following functionality:

1) Scan the PCI bus for specific devices or devices of a certain class.
2) Read and change generic PCI information.
3) Read and change device-specific PCI information.
4) Allocate PCI memory and IO space to devices.
5) Translate a device's PCI interrupts to equivalent HAL vectors.

Example code fragments are from the pci1 test (see

io/pci/<release>/tests/pci1.c).

All of the functions described below are declared in the header file <cyg/io/pci.h> which all clients of the PCI library should include.

Initialising the bus

The PCI bus needs to be initialized before it can be used. This only needs to be done once - some HALs may do it as part of the platform initialization procedure, other HALs may leave it to the application to do it. The following function will do the initialization only once, so it's safe to call from multiple drivers:

```
void cyg_pci_init( void );
```

Scanning for devices

After the bus has been initialized, it is possible to scan it for devices. This is done using the function:

```
cyg_bool cyg_pci_find_next( cyg_pci_device_id cur_devid,
                     cyg_pci_device_id *next_devid );
```

It will scan the bus for devices starting at cur_devid. If a device is found, its devid is stored in next_devid and the function returns true.

The pci1 test's outer loop looks like:

```
cyg_pci_init();
    if (cyg_pci_find_next(CYG_PCI_NULL_DEVID, &devid)) {
        do {
             <use devid>
        } while (cyg_pci_find_next(devid, &devid));
    }
```

What happens is that the bus gets initialized and a scan is started. CYG_PCI_NULL_DEVID causes cyg_pci_find_next() to restart its scan. If the bus does not contain any devices, the first call to cyg_pci_find_next() will return false.

If the call returns true, a loop is entered where the found devid is used. After devid processing has completed, the next device on the bus is searched for; cyg_pci_find_next() continues its scan from the current devid. The loop terminates when no more devices are found on the bus.

This is the generic way of scanning the bus, enumerating all the devices on the bus. But if the application is looking for a device of a given device class (e.g., a SCSI controller), or a specific vendor device, these functions simplify the task a bit:

```
cyg_bool cyg_pci_find_class( cyg_uint32 dev_class,
                             cyg_pci_device_id *devid );
cyg_bool cyg_pci_find_device( cyg_uint16 vendor, cyg_uint16 device,
                              cyg_pci_device_id *devid );
```

They work just like cyg_pci_find_next(), but only return true when the, dev_class or vendor/device qualifiers match those of a device on the bus. The devid serves as both an input and an output operand: the scan starts at the given device, and if a device is found devid is updated with the value for the found device.

The <cyg/io/pci_cfg.h> header file (included by pci.h) contains definitions for PCI class, vendor and device codes which can be used as arguments to the find functions. The list of vendor and device codes is not complete: add new codes as necessary. If possible also register the codes at the PCI Code List (http://www.yourvote.com/pci) which is where the eCos definitions are generated from.

Generic config information

When a valid device ID (devid) is found using one of the above functions, the associated device can be queried and controlled using the functions:

```
void cyg_pci_get_device_info ( cyg_pci_device_id devid,
                               cyg_pci_device *dev_info );
void cyg_pci_set_device_info ( cyg_pci_device_id devid,
                               cyg_pci_device *dev_info );
```

The cyg_pci_device structure (defined in pci.h) primarily holds information as described by the PCI specification [1]. The pci1 test prints out some of this information:

```
// Get device info
 cyg_pci_get_device_info(devid, &dev_info);
 diag_printf("\n Command   0x%04x, Status 0x%04x\n",
             dev_info.command, dev_info.status);
```

The command register can also be written to, controlling (among other things) whether the device responds to IO and memory access from the bus.

Specific config information

The above functions only allow access to generic PCI config registers. A device can have extra config registers not specified by the PCI specification. These can be accessed with these functions:

```
void cyg_pci_read_config_uint8( cyg_pci_device_id devid,
                                cyg_uint8 offset, cyg_uint8 *val);
void cyg_pci_read_config_uint16( cyg_pci_device_id devid,
                                 cyg_uint8 offset, cyg_uint16 *val);
void cyg_pci_read_config_uint32( cyg_pci_device_id devid,
                                 cyg_uint8 offset, cyg_uint32 *val);
void cyg_pci_write_config_uint8( cyg_pci_device_id devid,
                                 cyg_uint8 offset, cyg_uint8 val);
void cyg_pci_write_config_uint16( cyg_pci_device_id devid,
                                  cyg_uint8 offset, cyg_uint16 val);
void cyg_pci_write_config_uint32( cyg_pci_device_id devid,
                                  cyg_uint8 offset, cyg_uint32 val);
```

The write functions should only be used for device-specific config registers since using them on generic registers may invalidate the contents of a previously fetched cyg_pci_device structure.

Allocating memory

A PCI device ignores all IO and memory access from the PCI bus until it has been activated. Activation cannot happen until after device configuration. Configuration means telling the device where it should map its IO and memory resources. This is

done with this function:

```
cyg_bool cyg_pci_configure_device( cyg_pci_device *dev_info );
```

This function handles all IO and memory regions that need configuration on the device. Each region is represented in the PCI device's config space by one of six BARs (Base Address Registers) and is handled individually according to type using these functions:

```
cyg_bool cyg_pci_allocate_memory( cyg_pci_device *dev_info,
                                  cyg_uint32 bar,
                                  CYG_PCI_ADDRESS64 *base );
 cyg_bool cyg_pci_allocate_io( cyg_pci_device *dev_info,
                               cyg_uint32 bar,
                               CYG_PCI_ADDRESS32 *base );
```

The memory bases (in two distinct address spaces) are increased as memory regions are allocated to devices. Allocation will fail (the function returns false) if the base exceeds the limits of the address space (IO is 1MB, memory is 2^32 or 2^64 bytes).

These functions can also be called directly be the application/driver if necessary, but this should not be necessary.

The bases are initialized with default values provided by the HAL. It is possible for an application to override these using the following functions:

```
void cyg_pci_set_memory_base( CYG_PCI_ADDRESS64 base );
void cyg_pci_set_io_base( CYG_PCI_ADDRESS32 base );
```

When a device has been configured, the cyg_pci_device structure will contain the physical address in the CPU's address space where the device's memory regions can be accessed.

This information is provided in base_map[] - there is a 32 bit word for each of the device's BARs. For 32 bit PCI memory regions, each 32 bit word will be an actual pointer that can be used immediately by the driver: the memory space will normally be linearly addressable by the CPU.

However, for 64 bit PCI memory regions, some (or all) of the region may be outside of the CPUs address space. In this case the driver will need to know how to access the region in segments. This functionality may be adopted by the eCos HAL if deemed useful in the future. The 2GB available on many systems should suffice though.

Interrupts

A device may generate interrupts. The HAL vector associated with a given device on the bus is platform specific. This function allows a driver to find the actual interrupt vector for a given device:

```
cyg_bool cyg_pci_translate_interrupt( cyg_pci_device *dev_info,
                                      CYG_ADDRWORD *vec );
```

If the function returns false, no interrupts will be generated by the device. If it returns

true, the CYG_ADDRWORD pointed to by vec is updated with the HAL interrupt vector the device will be using. This is how the function is used in the pci1 test:

```
            if (cyg_pci_translate_interrupt(&dev_info, &irq))
                diag_printf(" Wired to HAL vector %d\n", irq);
            else
                diag_printf(" Does not generate interrupts.\n");
```

The application/drive should attach an interrupt handler to a device's interrupt before activating the device.

Activating a device

When the device has been allocated memory space it can be activated. This is not done by the library since a driver may have to initialize more state on the device before it can be safely activated.

Activating the device is done by enabling flags in its command word. As an example, see the pci1 test which can be configured to enable the devices it finds. This allows these to be accessed from GDB (if a breakpoint is set on cyg_test_exit):

```
#ifdef ENABLE_PCI_DEVICES
                {
                    cyg_uint16 cmd;

                  // Don't use cyg_pci_set_device_info since it clears
                    // some of the fields we want to print out below.
                    cyg_pci_read_config_uint16(dev_info.devid,
                                               CYG_PCI_CFG_COMMAND, &cmd);
                    cmd |=
CYG_PCI_CFG_COMMAND_IO|CYG_PCI_CFG_COMMAND_MEMORY;
                    cyg_pci_write_config_uint16(dev_info.devid,
                                                CYG_PCI_CFG_COMMAND, cmd);
                }
                diag_printf(" **** Device IO and MEM access
enabled\n");
#endif
```

Note: that the best way to activate a device is actually through cyg_pci_set_device_info(), but in this particular case the cyg_pci_device structure contents from before the activation is required for printout further down in the code.

Links

See these links for more information about PCI.

1) See http://www.pcisig.com (information on the PCI specifications)
2) See http://www.yourvote.com/pci (list of vendor and device IDs)
3) See http://www.picmg.org (PCI Industrial Computer Manufacturers Group)

PCI Library reference

This document defines the PCI Support Library for eCos.

The PCI support library provides a set of routines for accessing the PCI bus configuration space in a portable manner. This is provided by two APIs. The high level API is used by device drivers, or other code, to access the PCI configuration space portably. The low level API is used by the PCI library itself to access the hardware in a platform-specific manner, and may also be used by device drivers to access the PCI configuration space directly.

Underlying the low-level API is HAL support for the basic Configuration space operations. These should not generally be used by any code other than the PCI library, and are present in the HAL to allow low level initialization of the PCI bus and devices to take place if necessary.

PCI Library API

The PCI library provides the following routines and types for accessing the PCI configuration space.

The API for the PCI library is found in the header file <cyg/io/pci.h>.

Definitions

The header file contains definitions for the common configuration structure offsets and specimin values for device, vendor and class code.

Types and data structures

The following types are defined:

```
typedef CYG_WORD32 cyg_pci_device_id;
```

This is comprised of the bus number, device number and functional unit number packed into a single word. The macro CYG_PCI_DEV_MAKE_ID() may be used to construct a device id from the bus, device and functional unit numbers of a device. Similarly the macros CYG_PCI_DEV_GET_BUS() and CYG_PCI_DEV_GET_DEVFN() may be used to extract them. It should not be necessary to use these macros under normal circumstances.

```
typedef struct cyg_pci_device;
```

This structure is used to contain data read from a PCI device’s configuration header by cyg_pci_get_device_info(). It is also used to record the resource allocations made to the device.

```
typedef CYG_WORD64 CYG_PCI_ADDRESS64;
typedef CYG_WORD32 CYG_PCI_ADDRESS32;
```

Pointers in the PCI address space are 32 bit (IO space) or 32/64 bit (memory space). In most platform and device configurations all of PCI memory will be linearly addressable using only 32 bit pointers as read from base_map[].

The 64 bit type is used to allow handling 64 bit devices in the future, should it be necessary, without changing the library's API.

Functions

```
void cyg_pci_init(void);
```

Initialize the PCI library and establish contact with the hardware. This function is idempotent and can be called either by all drivers in the system, or just from an application initialization function.

```
cyg_bool cyg_pci_find_device( cyg_uint16 vendor,
      cyg_uint16 device,
      cyg_pci_device_id *devid );
```

Searches the PCI bus configuration space for a device with the given vendor and device ids. The search starts at the device pointed to by devid, or at the first slot if it contains CYG_PCI_NULL_DEVID. *devid will be updated with the ID of the next device found. Returns true if one is found and false if not.

```
cyg_bool cyg_pci_find_class( cyg_uint32 dev_class,
     cyg_pci_device_id *devid );
```

Searches the PCI bus configuration space for a device with the given class code. The search starts at the device pointed to by devid, or at the first slot if it contains CYG_PCI_NULL_DEVID.

*devid will be updated with the ID of the next device found. Returns true if one is found and false if not.

```
cyg_bool cyg_pci_find_next( cyg_pci_device_id cur_devid,
    cyg_pci_device_id *next_devid );
```

Searches the PCI configuration space for the next valid device after cur_devid. If cur_devid is given the value CYG_PCI_NULL_DEVID, then the search starts at the first slot. It is permitted for next_devid to point to cur_devid. Returns true if another device is found and false if not.

```
void cyg_pci_get_device_info ( cyg_pci_device_id devid,
       cyg_pci_device *dev_info );
```

This function gets the PCI configuration information for the device indicated in devid. The common fields of the cyg_pci_device structure, and the appropriate fields of the relevant header union member are filled in from the device's configuration space. If the device has not been enabled, then this function will also fetch the size and type information from the base address registers and place it in the base_size[] array.

```
void cyg_pci_set_device_info ( cyg_pci_device_id devid,
       cyg_pci_device *dev_info );
```

This function sets the PCI configuration information for the device indicated in devid. Only the configuration space registers that are writable are actually written. Once all the fields have been written, the device info will be read back into *dev_info, so that it reflects the true state of the hardware.

```
void cyg_pci_read_config_uint8( cyg_pci_device_id devid, cyg_uint8 offset,
cyg_uint8 *val );
void cyg_pci_read_config_uint16( cyg_pci_device_id devid, cyg_uint8 offset,
cyg_uint16 *val );
void cyg_pci_read_config_uint32( cyg_pci_device_id devid, cyg_uint8 offset,
cyg_uint32 *val );
```

These functions read registers of the appropriate size from the configuration space of the given device. They should mainly be used to access registers that are device specific. General PCI registers are best accessed through cyg_pci_get_device_info().

```
void cyg_pci_write_config_uint8( cyg_pci_device_id devid, cyg_uint8 offset,
cyg_uint8 val );
void cyg_pci_write_config_uint16( cyg_pci_device_id devid, cyg_uint8 offset,
cyg_uint16 val );
void cyg_pci_write_config_uint32( cyg_pci_device_id devid, cyg_uint8 offset,
cyg_uint32 val );
```

These functions write registers of the appropriate size to the configuration space of the given device. They should mainly be used to access registers that are device specific. General PCI registers are best accessed through cyg_pci_get_device_info(). Writing the general registers this way may render the contents of a cyg_pci_device structure invalid.

Resource allocation

These routines allocate memory and IO space to PCI devices.

```
cyg_bool cyg_pci_configure_device( cyg_pci_device *dev_info )
```

Allocate memory and IO space to all base address registers using the current memory and IO base addresses in the library. The allocated base addresses, translated into directly usable values, will be put into the matching base_map[] entries in *dev_info. If *dev_info does not contain valid base_size[] entries, then the result is false. This function will also call cyg_pci_translate_interrupt() to put the interrupt vector into the hal_vector entry.

```
cyg_bool cyg_pci_translate_interrupt( cyg_pci_device *dev_info, CYG_ADDRWORD
*vec );
```

Translate the device's PCI interrupt (INTA#-INTD#) to the associated HAL vector. This may also depend on which slot the device occupies. If the device may generate interrupts, the translated vector number will be stored in vec and the result is true. Otherwise the result is false.

```
cyg_bool cyg_pci_allocate_memory( cyg_pci_device *dev_info,
```

```
                                        cyg_uint32 bar,
                                        CYG_PCI_ADDRESS64 *base );
cyg_bool cyg_pci_allocate_io( cyg_pci_device *dev_info,
                                        cyg_uint32 bar,
                                    CYG_PCI_ADDRESS32 *base );
```

These routines allocate memory or IO space to the base address register indicated by bar. The base address in *base will be correctly aligned and the address of the next free location will be written back into it if the allocation succeeds. If the base address register is of the wrong type for this allocation, or dev_info does not contain valid base_size[] entries, the result is false. These functions allow a device driver to set up its own mappings if it wants. Most devices should probably use cyg_pci_configure_device().

```
void cyg_pci_set_memory_base( CYG_PCI_ADDRESS64 base );
void cyg_pci_set_io_base( CYG_PCI_ADDRESS32 base );
```

These routines set the base addresses for memory and IO mappings to be used by the memory allocation routines. Normally these base addresses will be set to default values based on the platform. These routines allow these to be changed by application code if necessary.

PCI Library Hardware API

This API is used by the PCI library to access the PCI bus configuration space. Although it should not normally be necessary, this API may also be used by device driver or application code to perform PCI bus operations not supported by the PCI library.

```
void cyg_pcihw_init(void);
```

Initialize the PCI hardware so that the configuration space may be accessed.

```
void cyg_pcihw_read_config_uint8( cyg_uint8 bus, cyg_uint8 devfn, cyg_uint8
offset, cyg_uint8 *val);
void cyg_pcihw_read_config_uint16( cyg_uint8 bus, cyg_uint8 devfn, cyg_uint8
offset, cyg_uint16 *val);
void cyg_pcihw_read_config_uint32( cyg_uint8 bus, cyg_uint8 devfn, cyg_uint8
offset, cyg_uint32 *val);
```

These functions read a register of the appropriate size from the PCI configuration space at an address composed from the bus, devfn and offset arguments.

```
void cyg_pcihw_write_config_uint8( cyg_uint8 bus, cyg_uint8 devfn, cyg_uint8
offset, cyg_uint8 val);
void cyg_pcihw_write_config_uint16( cyg_uint8 bus, cyg_uint8 devfn, cyg_uint8
offset, cyg_uint16 val);
void cyg_pcihw_write_config_uint32( cyg_uint8 bus, cyg_uint8 devfn, cyg_uint8
offset, cyg_uint32 val);
```

These functions write a register of the appropriate size to the PCI configuration space at an address composed from the bus, devfn and offset arguments.

```
cyg_bool cyg_pcihw_translate_interrupt( cyg_uint8 bus, cyg_uint8 devfn,
CYG_ADDRWORD *vec);
```

This function interrogates the device and determines which HAL interrupt vector it is connected to.

HAL PCI support

HAL support consists of a set of C macros that provide the implementation of the low level PCI API.

```
HAL_PCI_INIT()
```

Initialize the PCI bus.

```
HAL_PCI_READ_UINT8( bus, devfn, offset, val )
HAL_PCI_READ_UINT16( bus, devfn, offset, val )
HAL_PCI_READ_UINT32( bus, devfn, offset, val )
```

Read a value from the PCI configuration space of the appropriate size at an address composed from the bus, devfn and offset.

```
HAL_PCI_WRITE_UINT8( bus, devfn, offset, val )
HAL_PCI_WRITE_UINT16( bus, devfn, offset, val )
HAL_PCI_WRITE_UINT32( bus, devfn, offset, val )
```

Write a value to the PCI configuration space of the appropriate size at an address composed from the bus, devfn and offset.

```
HAL_PCI_TRANSLATE_INTERRUPT( bus, devfn, *vec, valid )
```

Translate the device's interrupt line into a HAL interrupt vector.

```
HAL_PCI_ALLOC_BASE_MEMORY
HAL_PCI_ALLOC_BASE_IO
```

These macros define the default base addresses used to initialize the memory and IO allocation pointers.

```
HAL_PCI_PHYSICAL_MEMORY_BASE
HAL_PCI_PHYSICAL_IO_BASE
```

PCI memory and IO range do not always correspond directly to physical memory or IO addresses. Frequently the PCI address spaces are windowed into the processor's address range at some offset. These macros define offsets to be added to the PCI base addresses to translate PCI bus addresses into physical memory addresses that can be used to access the allocated memory or IO space.

NOTE The chunk of PCI memory space directly addressable though the window by the CPU may be smaller than the amount of PCI memory actually provided. In that case drivers will have to access PCI memory space in segments. Doing this will be platform specific and is currently beyond the scope of the HAL.

Part IV: I/O Package (Device Drivers)

10

Introduction

The I/O package is designed as a general purpose framework for supporting device drivers. This includes all classes of drivers from simple serial to networking stacks and beyond.

Components of the I/O package, such as device drivers, are configured into the system just like all other components. Additionally, end users may add their own drivers to this set.

While the set of drivers (and the devices they represent) may be considered static, they must be accessed via an opaque “handle”. Each device in the system has a unique name and the **`cyg_io_lookup()`** function is used to map that name onto the handle for the device. This “hiding” of the device implementation allows for generic, named devices, as well as more flexibility. Also, the **`cyg_io_lookup()`** function provides drivers the opportunity to initialize the device when usage actually starts.

All devices have a name. The standard provided devices use names such as “/dev/console” and “/dev/serial0”, where the “/dev/” prefix indicates that this is the name of a device.

The entire I/O package API, as well as the standard set of provided drivers, is written in C.

Basic functions are provided to send data to and receive data from a device. The details of how this is done is left to the device [class] itself. For example, writing data to a block device like a disk drive may have different semantics than writing to a serial port.

Additional functions are provided to manipulate the state of the driver and/or the actual device. These functions are, by design, quite specific to the actual driver.

This driver model supports layering; in other words, a device may actually be created “on top of” another device. For example, the “tty” (terminal-like) devices are built on top of simple serial devices. The upper layer then has the flexibility to add features and functions not found at the lower layers. In this case the “tty” device provides for line buffering and editing not available from the simple serial drivers.

Some drivers will support visibility of the layers they depend upon. The “tty” driver allows information about the actual serial device to be manipulated by passing get/set config calls that use a serial driver “key” down to the serial driver itself.

11

User API

All functions, except **cyg_io_lookup()** require an I/O “handle”.

All functions return a value of the type Cyg_ErrNo. If an error condition is detected, this value will be negative and the absolute value indicates the actual error, as specified in cyg/error/codes.h. The only other legal return value will be ENOERR. All other function arguments are pointers (references). This allows the drivers to pass information efficiently, both into and out of the driver. The most striking example of this is the “length” value passed to the read and write functions. This parameter contains the desired length of data on input to the function and the actual transferred length on return.

```
// Lookup a device and return its handle
Cyg_ErrNo cyg_io_lookup(
    const char *name,
    cyg_io_handle_t *handle )
```

This function maps a device name onto an appropriate handle. If the named device is not in the system, then the error -ENOENT is returned. If the device is found, then the handle for the device is returned by way of the handle pointer *handle.

```
 // Write data to a device
Cyg_ErrNo cyg_io_write(
    cyg_io_handle_t handle,
    const void *buf,
    cyg_uint32 *len )
```

This function sends data to a device. The size of data to send is contained in **len* and the actual size sent will be returned in the same place.

```
 // Read data from a device
Cyg_ErrNo cyg_io_read(
    cyg_io_handle_t handle,
    void *buf,
    cyg_uint32 *len )
```

This function receives data from a device. The desired size of data to receive is contained in **len* and the actual size obtained will be returned in the same place.

```
 // Get the configuration of a device
Cyg_ErrNo cyg_io_get_config(
    cyg_io_handle_t handle,
    cyg_uint32 key,
    void *buf,
    cyg_uint32 *len )
```

This function is used to obtain run-time configuration about a device. The type of information retrieved is specified by the key. The data will be returned in the given buffer. The value of **len* should contain the amount of data requested, which must be at least as large as the size appropriate to the selected key. The actual size of data retrieved is placed in **len*. The appropriate key values differ for each driver and are all listed in the file `cyg/io/config_keys.h`.

```
 // Change the configuration of a device
Cyg_ErrNo cyg_io_set_config(
    cyg_io_handle_t handle,
    cyg_uint32 key,
    const void *buf,
    cyg_uint32 *len )
```

This function is used to manipulate or change the run-time configuration of a device. The type of information is specified by the key. The data will be obtained from the given buffer. The value of **len* should contain the amount of data provided, which must match the size appropriate to the selected key. The appropriate key values differ for each driver and are all listed in the file `cyg/io/config_keys.h`.

12

Serial driver details

Two different classes of serial drivers are provided as a standard part of the eCos system. These are described as “simple serial” (serial) and “tty-like” (tty).

“simple serial” driver

Use the include file `cyg/io/serialio.h` for this driver.

The simple serial driver is capable of sending and receiving blocks of raw data to a serial device. Controls are provided to configure the actual hardware, but there is no manipulation of the data by this driver.

There may be many instances of this driver in a given system, one for each serial channel. Each channel corresponds to a physical device and there will typically be a device module created for this purpose. The device modules themselves are configurable, allowing specification of the actual hardware details, as well as such details as whether the channel should be buffered by the serial driver, etc.

Runtime configuration

```
typedef struct {
 cyg_serial_baud_rate_t baud;
 cyg_serial_stop_bits_t stop;
 cyg_serial_parity_t parity;
 cyg_serial_word_length_t word_length;
```

```
  cyg_uint32 flags;
} cyg_serial_info_t;
```

The field "word_length" contains the number of data bits per word (character). This must be one of the values:

```
CYGNUM_SERIAL_WORD_LENGTH_5
CYGNUM_SERIAL_WORD_LENGTH_6
CYGNUM_SERIAL_WORD_LENGTH_7
CYGNUM_SERIAL_WORD_LENGTH_8
```

The field "baud" contains a baud rate selection. This must be one of the values:

```
CYGNUM_SERIAL_BAUD_50
CYGNUM_SERIAL_BAUD_75
CYGNUM_SERIAL_BAUD_110
CYGNUM_SERIAL_BAUD_134_5
CYGNUM_SERIAL_BAUD_150
CYGNUM_SERIAL_BAUD_200
CYGNUM_SERIAL_BAUD_300
CYGNUM_SERIAL_BAUD_600
CYGNUM_SERIAL_BAUD_1200
CYGNUM_SERIAL_BAUD_1800
CYGNUM_SERIAL_BAUD_2400
CYGNUM_SERIAL_BAUD_3600
CYGNUM_SERIAL_BAUD_4800
CYGNUM_SERIAL_BAUD_7200
CYGNUM_SERIAL_BAUD_9600
CYGNUM_SERIAL_BAUD_14400
CYGNUM_SERIAL_BAUD_19200
CYGNUM_SERIAL_BAUD_38400
CYGNUM_SERIAL_BAUD_57600
CYGNUM_SERIAL_BAUD_115200
CYGNUM_SERIAL_BAUD_234000
```

The field "stop" contains the number of stop bits. This must be one of the values:

```
CYGNUM_SERIAL_STOP_1
CYGNUM_SERIAL_STOP_1_5
CYGNUM_SERIAL_STOP_2
```

NOTE On most hardware, a selection of 1.5 stop bits is only valid if the word (character) length is 5.

The field "parity" contains the parity mode. This must be one of the values:

```
CYGNUM_SERIAL_PARITY_NONE
CYGNUM_SERIAL_PARITY_EVEN
CYGNUM_SERIAL_PARITY_ODD
CYGNUM_SERIAL_PARITY_MARK
```

```
CYGNUM_SERIAL_PARITY_SPACE
```

The field "flags" is a bitmask which controls the behavior of the serial device driver. It should be built from the values **CYG_SERIAL_FLAGS_xxx** defined below:

```
#define CYG_SERIAL_FLAGS_RTSCTS 0x0001
```

If this bit is set then the port is placed in “hardware handshake” mode. In this mode, the CTS and RTS pins control when data is allowed to be sent/received at the port. This bit is ignored if the hardware does not support this level of handshake.

```
typedef struct {
  cyg_int32 rx_bufsize;
  cyg_int32 rx_count;
  cyg_int32 tx_bufsize;
  cyg_int32 tx_count;
}
cyg_serial_buf_info_t;
```

The field 'rx_bufsize' contains the total size of the incoming data buffer. This is set to 0 on devices that do not support buffering (i.e. polled devices).

The field 'rx_count' contains the number of bytes currently occupied in the incoming data buffer. This is set to 0 on devices that do not support buffering (i.e. polled devices).

The field 'tx_bufsize' contains the total size of the transmit data buffer. This is set to 0 on devices that do not support buffering (i.e. polled devices).

The field 'tx_count' contains the number of bytes currently occrupied in the transmit data buffer. This is set to 0 on devices that do not support buffering (i.e. polled devices).

API details

```
cyg_io_write(handle, buf, len)
```

Send the data from "buf" to the device. The driver maintains a buffer to hold the data. The size of the intermediate buffer is configurable within the interface module. The data is not modified at all while it is being buffered. On return, *len contains the amount of characters actually consumed .

It is possible to configure the write call to be blocking (default) or non-blocking. Non-blocking mode requires both the configuration option CYGOPT_IO_SERIAL_SUPPORT_NONBLOCKING to be enabled, and the specific device to be set to non-blocking mode for writes (see cyg_io_set_config). In blocking mode, the call will not return until there is space in the buffer and the entire contents of "buf" have been consumed.

In non-blocking mode, as much as possible gets consumed from "buf". If everything was consumed, the call returns ENOERR. If only part of the "buf" contents was consumed, -EAGAIN is returned and the caller must try again. On return, *len contains the amount of characters actually consumed .

The call can also return -EINTR if interrupted via the cyg_io_get_config/ABORT key. On return, *len contains the amount of characters actually consumed .

cyg_io_read(handle, buf, len)

Receive data into the specified buffer from the device. No manipulation of the data is performed before being transferred. An interrupt driven interface module will support data arriving when no read is pending by buffering the data in the serial driver. Again, this buffering is completely configurable. On return, *len contains the amount of characters actually received.

It is possible to configure the read call to be blocking (default) or non-blocking. Non-blocking mode requires both the configuration option CYGOPT_IO_SERIAL_SUPPORT_NONBLOCKING to be enabled, and the specific device to be set to non-blocking mode for reads (see cyg_io_set_config).

In blocking mode, the call will not return until the requested amount of data has been read.

In non-blocking mode, data waiting in the device buffer is copied to "buf", and the call returns immediately. If there was enough data in the buffer to fulfill the request, ENOERR is returned. If only part of the request could be fulfilled, -EAGAIN is returned and the caller must try again. On return, *len contains the amount of characters actually received.

The call can also return -EINTR if interrupted via the cyg_io_get_config/ABORT key. On return, *len contains the amount of characters actually received.

cyg_io_get_config(handle, key, buf, len)

This function returns current [runtime] information about the device and/or driver.

Key:

CYG_IO_GET_CONFIG_SERIAL_INFO

Buf type:

cyg_serial_info_t

Function:

This function retrieves the current state of the driver and hardware. This information contains fields for hardware baud rate, number of stop bits, and parity mode. It also includes a set of flags that control the port, such as hardware flow control.

Key:

CYG_IO_GET_CONFIG_SERIAL_BUFFER_INFO

Buf type:

cyg_serial_buf_info_t

Function:

This function retrieves the current state of the software buffers in the serial drivers. For both receive and transmit buffers it returns the total buffer size and the current number of bytes occupied in the buffer. It does not take into account any buffering such as FIFOs or holding registers that the serial device itself may have.

Key:

CYG_IO_GET_CONFIG_SERIAL_OUTPUT_DRAIN

Buf type:

void *

Function:

This function waits for any buffered output to complete. This function only completes when there is no more data remaining to be sent to the device.

Key:

CYG_IO_GET_CONFIG_SERIAL_OUTPUT_FLUSH

Buf type:

void *

Function:

This function discards any buffered output for the device.

Key:

CYG_IO_GET_CONFIG_SERIAL_INPUT_DRAIN

Buf type:

void *

Function:

This function discards any buffered input for the device.

Key:

CYG_IO_GET_CONFIG_SERIAL_ABORT

Buf type:

void*

Function:

This function will cause any pending read or write calls on this device to return with -EABORT.

Key:

CYG_IO_GET_CONFIG_SERIAL_READ_BLOCKING

Buf type:

cyg_uint32 (values 0 or 1)

Function:

This function will read back the blocking-mode setting for read calls on this device. This call is only available if the configuration option CYGOPT_IO_SERIAL_SUPPORT_NONBLOCKING is enabled.

Key:

CYG_IO_GET_CONFIG_SERIAL_WRITE_BLOCKING

Buf type:

cyg_uint32 (values 0 or 1)

Function:

This function will read back the blocking-mode setting for write calls on this device. This call is only available if the configuration option CYGOPT_IO_SERIAL_SUPPORT_NONBLOCKING is enabled.

```
cyg_io_set_config(handle, key, buf, len)
```

This function is used to update or change runtime configuration of a port.

Key:

CYG_IO_SET_CONFIG_SERIAL_INFO

Buf type:

cyg_serial_info_t

Function:

This function updates the information for the driver and hardware. The information contains fields for hardware baud rate, number of stop bits, and parity mode. It also includes a set of flags that control the port, such as hardware flow control.

Key:

CYG_IO_SET_CONFIG_SERIAL_READ_BLOCKING

Buf type:

cyg_uint32 (values 0 or 1)

Function:

This function will set the blocking-mode for read calls on this device. This call is only available if the configuration option CYGOPT_IO_SERIAL_SUPPORT_NONBLOCKING is enabled.

Key:

CYG_IO_SET_CONFIG_SERIAL_WRITE_BLOCKING

Buf type:

cyg_uint32 (values 0 or 1)

Function:

This function will set the blocking-mode for write calls on this device. This call is only available if the configuration option CYGOPT_IO_SERIAL_SUPPORT_NONBLOCKING is enabled.

“tty” driver

Use the include file `cyg/io/ttyio.h` for this driver.

This driver is built on top of the simple serial driver and is typically used for a device that interfaces with humans such as a terminal. It provides some minimal formatting of data on output and allows for line-oriented editing on input.

Runtime configuration

```
typedef struct {
 cyg_uint32 tty_out_flags;
 cyg_uint32 tty_in_flags;
} cyg_tty_info_t;
```

The field "tty_out_flags" is used to control what happens to data as it is send to the serial port. It contains a bitmap comprised of the bits as defined by the **`CYG_TTY_OUT_FLAGS_xxx`** values below.

```
#define CYG_TTY_OUT_FLAGS_CRLF 0x0001 // Map '\n' => '\n\r' on output
```

If this bit is set in 'tty_out_flags', any occurrence of the character '\n' will be replaced by the sequence '\n\r' before sending to the device.

The field "tty_in_flags" is used to control how data is handled as it comes from the serial port. It contains a bitmap comprised of the bits as defined by the **`CYG_TTY_IN_FLAGS_xxx`** values below.

```
#define CYG_TTY_IN_FLAGS_CR 0x0001 // Map '\r' => '\n' on input
```

If this bit is set in "tty_in_flags", the character "\r" (“return” or “enter” on most

keyboards) will be mapped to "\n".

```
#define CYG_TTY_IN_FLAGS_CRLF 0x0002 // Map '\n\r' => '\n' on input
```

If this bit is set in "tty_in_flags", the character sequence "\n\r" (often sent by DOS/Windows based terminals) will be mapped to "\n".

```
#define CYG_TTY_IN_FLAGS_BINARY 0x0004 // No input processing
```

If this bit is set in "tty_in_flags", the input will not be manipulated in any way before being placed in the user's buffer.

```
#define CYG_TTY_IN_FLAGS_ECHO 0x0008 // Echo characters as processed
```

If this bit is set in "tty_in_flags", characters will be echoed back to the serial port as they are processed.

API details

cyg_io_read(handle, buf, len)

This function is used to read data from the device. In the default case, data is read until an end-of-line character ("\n" or "\r") is read. Additionally, the characters are echoed back to the [terminal] device. Minimal editing of the input is also supported.

NOTE When connecting to a remote target via GDB it is not possible to provide console input while GDB is connected. The GDB remote protocol does not support input. Users must disconnect from GDB if this functionality is required.

cyg_io_write(handle, buf, len)

This function is used to send data to the device. In the default case, the end-of-line character "\n" is replaced by the sequence "\n\r".

cyg_io_get_config(handle, key, buf, len)

This function is used to get information about the channel's configuration at runtime.

Key:

CYG_IO_GET_CONFIG_TTY_INFO

Buf type:

cyg_tty_info_t

Function:

This function retrieves the current state of the driver.

The key must be "CYG_IO_GET_CONFIG_TTY_INFO" which returns the control

flags for the channel. The buffer "buf" must be of type "cyg_tty_info_t" and the length should match.

Serial driver keys (see above) may also be specified in which case the call is passed directly to the serial driver.

```
cyg_io_set_config(handle, key, buf, len)
```

This function is used to modify the channel's configuration at runtime.

Key:

CYG_IO_SET_CONFIG_TTY_INFO

Buf type:

cyg_tty_info_t

Function:

This function changes the current state of the driver.

The key must be **`CYG_IO_SET_CONFIG_TTY_INFO`** which returns the control flags for the channel. The buffer "buf" must be of type `cyg_tty_info_t` and the length should match.

Serial driver keys (see above) may also be specified in which case the call is passed directly to the serial driver.

13

How to write a driver

A device driver is nothing more than a named entity that supports the basic I/O functions - read, write, get config, and set config. Typically a device driver also uses and manages interrupts from the device as well. While the interface is generic and device driver independent, the actual driver implementation is completely up to the device driver designer.

That said, the reason for using a device driver is to provide access to a device from application code in as general purpose a fashion as reasonable. Most driver writers are also concerned with making this access as simple as possible while being as efficient as possible.

Most device drivers are concerned with the movement of information, for example data bytes along a serial interface, or packets in a network. In order to make the most efficient use of system resources, interrupts are used. This can allow for other application processing to take place while the data transfers are underway, with interrupts used to indicate when various events have occurred. For example, a serial port typically generates an interrupt after a character has been sent “down the wire” and the interface is ready for another. It makes sense to allow further application processing while the data is being sent since this can take quite a long time. The interrupt can be used to allow the driver to send a character as soon as the current one is complete, without any active participation by the application code.

The main building blocks for device drivers are found in the include file:
`cyg/io/devtab.h`

All device drivers in **eCos** are described by a device table entry, using the “cyg_devtab_entry_t” type. The entry should be created using the **DEVTAB_ENTRY** macro, like this:

```
DEVTAB_ENTRY(l,name,dep_name,handlers,init,lookup,priv)
```

Arguments:
l - The "C" label for this device table entry.
name - The "C" string name for the device.
dep_name - For a layered device, the "C" string name of the device this device is built upon.
handlers - A pointer to the I/O function "handlers" (see below).
init - A function called when eCos is initialized. This function can query the device, setup hardware, etc.
lookup - A function called when "cyg_io_lookup()" is called for this device.
priv - A placeholder for any device specific data required by the driver.

The interface to the driver is through the “handlers” field. This is a pointer to a set of functions which implement the various **cyg_io_XXX()** routines. This table is defined by the macro:

```
DEVIO_TABLE(l,write,read,get_config,set_config)
```

Arguments:
l - The "C" label for this table of handlers.
write - The function called as a result of "cyg_io_write()".
read - The function called as a result of "cyg_io_read()".
get_config - The function called as a result of "cyg_io_get_config()".
set_config - The function called as a result of "cyg_io_set_config()".

When **eCos** is initialized (sometimes called “boot” time), the “init” function is called for all devices in the system. The “init” function is allowed to return an error in which case the device will be placed “off line” and all I/O requests to that device will be considered in error.

The “lookup” function is called whenever the **cyg_io_lookup()** function is called with this device name. The lookup function may cause the device to come “on line” which would then allow I/O operations to proceed. Future versions of the I/O system will allow for other states, including power saving modes, etc.

How to write a serial hardware interface module

The standard serial driver supplied with **eCos** is structured as a hardware independent portion and a hardware dependent interface module. To add support for a new serial port, the user should be able to use the existing hardware independent portion and just add their own interface module which handles the details of the actual device. The user should have no need to change the hardware independent portion.

The interfaces used by the serial driver and serial implementation modules are contained in the file `cyg/io/serial.h`

NOTE In the text below we use the notation <<xx>> to mean a module specific value, referred to as “xx” below.

The interface module contains the devtab entry (or entries if a single module supports more than one interface). This entry should have the form:

```
DEVTAB_ENTRY(<<module_name>>,
<<device_name>>,
0,
&serial_devio,
<<module_init>>,
<<module_lookup>>,
&<<serial_channel>>
);
```

Where:

module_name - The "C" label for this devtab entry

device_name - The "C" string for the device. E.g. "/dev/serial0".

serial_devio - The table of I/O functions. This set is defined in the hardware independent serial driver and should be used.

module_init - The module initialization function.

module_lookup - The device lookup function. This function typically sets up the device for actual use, turning on interrupts, configuring the port, etc.

serial_channel - This table (defined below) contains the interface between the interface module and the serial driver proper.

Each serial device must have a “serial channel”. This is a set of data which describes all operations on the device. It also contains buffers, etc., if the device is to be buffered. The serial channel is created by the macro:

```
SERIAL_CHANNEL_USING_INTERRUPTS(l,funs,dev_priv,baud,stop,parity,word
_length,
 flags,out_buf,out_buflen,in_buf,in_buflen)
```

Arguments:
l - The "C" label for this structure.
funs - The set of interface functions (see below).
dev_priv - A placeholder for any device specific data for this channel.
baud - The initial baud rate value (cyg_serial_baud_t).
stop - The initial stop bits value (cyg_serial_stop_bits_t)
parity - The initial parity mode value (cyg_serial_parity_t)
word_length - The initial word length value (cyg_serial_word_length_t)
flags - The initial driver flags value
out_buf - Pointer to the output buffer. NULL if none required.
out_buflen - The length of the output buffer.
in_buf - Pointer to the input buffer. NULL if none required.
in_buflen - The length of the input buffer.

If either buffer length is zero, no buffering will take place in that direction and only polled mode functions will be used.

The interface from the hardware independent driver into the hardware interface module is contained in the "funs" table above. This is defined by the macro:

```
SERIAL_FUNS(l,putc,getc,set_config,start_xmit,stop_xmit)
```

Arguments:
l - The "C" label for this structure.
putc - bool (*putc)(serial_channel *priv, unsigned char c)
This function sends one character to the interface. It should return 'true' if the character is actually consumed. It should return 'false' if there is no space in the interface
getc - unsigned char (*getc)(serial_channel *priv)
This function fetches one character from the interface. It will be only called in a non-interrupt driven mode, thus it should wait for a character by polling the device until ready.
set_config - bool (*set_config)(serial_channel *priv, cyg_serial_info_t *config)
This function is used to configure the port. It should return 'true' if the hardware is updated to match the desired configuration. It should return 'false' if the port cannot support some parameter specified by the given configuration. E.g. selecting 1.5 stop bits and 8 data bits is invalid for most serial devices and should not be allowed.
start_xmit - void (*start_xmit)(serial_channel *priv)
In interrupt mode, turn on the transmitter and allow for transmit interrupts.
stop_xmit - void (*stop_xmit)(serial_channel *priv)
In interrupt mode, turn off the transmitter.

The device interface module can execute functions in the hardware independent driver

via “chan->callbacks”. These functions are available:

`void` **`(*serial_init)`**`(`

`serial_channel *`*`chan )`*

This function is used to initialize the serial channel. It is only required if the channel is being used in interrupt mode.

`void` **`(*xmt_char)`**`(`

`serial_channel *`*`chan )`*

This function would be called from an interrupt handler after a transmit interrupt indicating that additional characters may be sent. The upper driver will call the "putc" function as appropriate to send more data to the device.

`void` **`(*rcv_char)`**`(`

`serial_channel *`*`chan`*`,`

`unsigned char` *`c )`*

This function is used to tell the driver that a character has arrived at the interface. This function is typically called from the interrupt handler.

Furthermore, if the device has a FIFO it should require the hardware independent driver to provide block transfer functionality (driver CDL should include 'implements CYGINT_IO_SERIAL_BLOCK_TRANSFER). In that case, the following functions are available as well:

```
bool (*data_xmt_req)(serial_channel *chan, int space,
                     int* chars_avail, unsigned char** chars)
void (*data_xmt_done)(serial_channel *chan)
```

Instead of calling xmt_char to get a single character for transmission at a time, the driver should call data_xmt_req in a loop, requesting character blocks for transfer. Call with 'space' argument of how much space there is available in the FIFO.

If the call returns true, the driver can read 'chars_avail' characters from 'chars' and copy them into the FIFO.

If the call returns false, there are no more buffered characters and the driver should continue without filling up the FIFO.

When all data has been unloaded, the driver must call data_rcv_done.

```
bool (*data_rcv_req)(serial_channel *chan, int avail,
                     int* space_avail, unsigned char** space)
void (*data_rcv_done)(serial_channel *chan)
```

Instead of calling rcv_char with a single character at a time, the driver should call data_rcv_req in a loop, requesting space to unload the FIFO to. 'avail' is the number of characters the driver wishes to unload.

If the call returns true, the driver can copy 'space_avail' characters to 'space'.

If the call returns false, the input buffer is full. It is up to the driver to decide what to

do in that case (callback functions for registering overflow are being planned for later versions of the serial driver).

When all data has been unloaded, the driver must call data_rcv_done.

14

Device Driver Interface to the Kernel

This chapter describes the API that device drivers may use to interact with the kernel and HAL. It is primarily concerned with the control and management of interrupts.

The same API will be present in configurations where the kernel is not present. In this case the functions will be supplied by code acting directly on the HAL.

Interrupt Model

eCos presents a three level interrupt model to device drivers. This consists of Interrupt Service Routines (ISRs) that are invoked in response to a hardware interrupt; Deferred Service Routines (DSRs) that are invoked in response to a request by an ISR; and threads that are the clients of the driver.

Hardware interrupts are delivered with minimal intervention to an ISR. The HAL decodes the hardware source of the interrupt and calls the ISR of the attached interrupt object. This ISR may manipulate the hardware but is only allowed to make a restricted set of calls on the driver API. When it returns, an ISR may request that its DSR should be scheduled to run.

A DSR will be run when it is safe to do so without interfering with the scheduler. Most of the time the DSR will run immediately after the ISR, but if the current thread is in

the scheduler, it will be delayed until the thread is finished. A DSR is allowed to make a larger set of driver API calls, including, in particular, being able to call **cyg_drv_cond_signal()** to wake up waiting threads.

Finally, threads are able to make all API calls and in particular are allowed to wait on mutexes and condition variables.

For a device driver to receive interrupts it must first define ISR and DSR routines as shown below, and then call **cyg_drv_interrupt_create()**. Using the handle returned, the driver must then call **cyg_drv_interrupt_attach()** to actually attach the interrupt to the hardware vector.

Synchronization

There are three levels of synchronization supported:

1. Synchronization with ISRs. This normally means disabling interrupts to prevent the ISR running during a critical section. On a multiprocessor this will also require a spinlock. This is implemented by the **cyg_drv_isr_lock()** and **cyg_drv_isr_unlock()** functions. This mechanism should be used sparingly and for short periods only.
2. Synchronization with DSRs. This will be implemented in the kernel by taking the scheduler lock to prevent DSRs running during critical sections. In non-kernel configurations it will be implemented by non-kernel code. This is implemented by the **cyg_drv_dsr_lock()** and **cyg_drv_dsr_unlock()** functions. As with ISR synchronization, this mechanism should be used sparingly.
3. Synchronization with threads. This is implemented with mutexes and condition variables. Only threads may lock the mutexes and wait on the condition variables, although DSRs may signal condition variables.

ISRs are run with interrupts disabled, so it is not necessary to call **cyg_drv_isr_lock()** in an ISR. Similarly DSRs are run with the scheduler lock taken, so it is not necessary to call **cyg_drv_dsr_lock()** in DSRs.

Device Driver Models

There are several ways in which device drivers may be built. The exact model chosen will depend on the properties of the device and the behavior desired. There are three basic models that may be adopted.

The first model is to do all device processing in the ISR. When it is invoked the ISR programs the device hardware directly and accesses data to be transferred directly in

memory. The ISR should also call **cyg_drv_interrupt_acknowledge()**. When it is finished it may optionally request that its DSR be invoked. The DSR does nothing but call **cyg_drv_cond_signal()** to cause a thread to be woken up. Thread level code must call **cyg_drv_isr_lock()**, or **cyg_drv_interrupt_mask()** to prevent ISRs running while it manipulates shared memory.

The second model is to defer device processing to the DSR. The ISR simply prevents further delivery of interrupts by either programming the device, or by calling **cyg_drv_interrupt_mask()**. It may then call **cyg_drv_interrupt_acknowledge()** to allow other interrupts to be delivered and request that its DSR be called. When the DSR runs it does the majority of the device handling, optionally signals a condition variable to wake a thread, and finishes by calling **cyg_drv_interrupt_unmask()** to re-allow device interrupts. Thread level code uses **cyg_drv_dsr_lock()** to prevent DSRs running while it manipulates shared memory.

The third model is to defer device processing even further to a thread. The ISR behaves exactly as in the previous model and simply blocks and acknowledges the interrupt before request that the DSR run. The DSR itself only calls **cyg_drv_cond_signal()** to wake the thread. When the thread awakens it performs all device processing, and has full access to all kernel facilities while it does so. It should finish by calling **cyg_drv_interrupt_unmask()** to re-allow device interrupts.

The first model is good for devices that need immediate processing and interact infrequently with thread level. The second model trades a little latency in dealing with the device for a less intrusive synchronization mechanism. The last model allows device processing to be scheduled with other threads and permits more complex device handling.

Synchronization Levels

Since it would be dangerous for an ISR or DSR to make a call that might reschedule the current thread (by trying to lock a mutex for example) all functions in this API have an associated synchronization level. These levels are:

Thread

This function may only be called from within threads. This is usually the client code that makes calls into the device driver. In a non-kernel configuration, this will be code running at the default non-interrupt level.

DSR

This function may be called by either DSR or thread code.

ISR

This function may be called from ISR, DSR or thread code.

The following table shows, for each API function, the levels at which is may be called:

Function	Callable from: ISR	DSR	Thread
cyg_drv_isr_lock		X	X
cyg_drv_isr_unlock		X	X
cyg_drv_dsr_lock			X
cyg_drv_dsr_unlock			X
cyg_drv_mutex_init			X
cyg_drv_mutex_destroy			X
cyg_drv_mutex_lock			X
cyg_drv_mutex_trylock			X
cyg_drv_mutex_unlock			X
cyg_drv_mutex_release			X
cyg_drv_cond_init			X
cyg_drv_cond_destroy			X
cyg_drv_cond_wait			X
cyg_drv_cond_signal		X	X
cyg_drv_cond_broadcast		X	X
cyg_drv_interrupt_create			X
cyg_drv_interrupt_delete			X
cyg_drv_interrupt_attach	X	X	X
cyg_drv_interrupt_detach	X	X	X
cyg_drv_interrupt_mask	X	X	X
cyg_drv_interrupt_unmask	X	X	X
cyg_drv_interrupt_acknowledge	X	X	X
cyg_drv_interrupt_configure	X	X	X
cyg_drv_interrupt_level	X	X	X

The API

This section details the Driver Kernel Interface. Note that most of these functions are identical to Kernel C API calls, and will in most configurations be wrappers for them. In non-kernel configurations they will be supported directly by the HAL, or by code to emulate the required behavior.

This API is defined in the header file `cyg/hal/drv_api.h`.

cyg_drv_isr_lock

Function:

```
void cyg_drv_isr_lock()
```

Arguments:

None

Result:

None

Level:

DSR

Description:

Disables delivery of interrupts, preventing all ISRs running. This function maintains a counter of the number of times it is called.

cyg_drv_isr_unlock

Function:

```
void cyg_drv_isr_unlock()
```

Arguments:

None

Result:

None

Level:

DSR

Description:

Re-enables delivery of interrupts, allowing ISRs to run. This function decrements the counter maintained by **`cyg_drv_isr_lock()`**, and only re-allows interrupts when it goes to zero.

cyg_drv_dsr_lock

Function:

```
void cyg_drv_dsr_lock()
```

Arguments:

None

Result:

None

Level:

Thread

Description:

Disables scheduling of DSRs. This function maintains a counter of the number of times it has been called.

cyg_drv_dsr_unlock

Function:

```
void cyg_drv_dsr_unlock()
```

Arguments:

None

Result:

None

Level:

Thread

Description:

Re-enables scheduling of DSRs. This function decrements the counter incremented by **`cyg_drv_dsr_lock()`**. DSRs are only allowed to be delivered when the counter goes to zero.

cyg_drv_mutex_init

Function:

```
void cyg_drv_mutex_init(cyg_drv_mutex *mutex)
```

Arguments:

mutex - pointer to mutex to initialize

Result:

None

Level:

Thread

Description:

Initialize the mutex pointer to by the mutex argument.

cyg_drv_mutex_destroy

Function:

```
void cyg_drv_mutex_destroy( cyg_drv_mutex *mutex )
```

Arguments:

mutex - pointer to mutex to destroy

Result:

None

Level:

Thread

Description:

Destroy the mutex pointed to by the mutex argument.

cyg_drv_mutex_lock

Function:

```
cyg_bool cyg_drv_mutex_lock( cyg_drv_mutex *mutex )
```

Arguments:

mutex - pointer to mutex to lock

Result:

TRUE it the thread has claimed the lock, FALSE otherwise.

Level:

Thread

Description:

Attempt to lock the mutex pointed to by the mutex argument. If the mutex is already locked by another thread then this thread will wait until that thread is finished. If the result from this function is FALSE then the thread was broken out of its wait by some other thread. In this case the mutex will not have been locked.

cyg_drv_mutex_trylock

Function:

```
cyg_bool cyg_drv_mutex_trylock( cyg_drv_mutex *mutex )
```

Arguments:

mutex - pointer to mutex to lock

Result:

TRUE if the mutex has been locked, FALSE otherwise.

Level:

Thread

Description:

Attempt to lock the mutex pointed to by the mutex argument without waiting. If the mutex is already locked by some other thread then this function returns FALSE. If the function can lock the mutex without waiting, then TRUE is returned.

cyg_drv_mutex_unlock

Function:

```
void cyg_drv_mutex_unlock( cyg_drv_mutex *mutex )
```

Arguments:

mutex - pointer to mutex to unlock

Result:

None

Level:

Thread

Description:

Unlock the mutex pointed to by the mutex argument. If there are any threads waiting to claim the lock, one of them is woken up to try and claim it.

cyg_drv_mutex_release

Function:

```
void cyg_drv_mutex_release( cyg_drv_mutex *mutex )
```

Arguments:

mutex - pointer to mutex to release

Result:

None

Level:

Thread

Description:

Release all threads waiting on the mutex pointed to by the mutex argument. These threads will return from **`cyg_drv_mutex_lock()`** with a FALSE result and will not have claimed the mutex. This function has no effect on any thread that may have the mutex claimed.

cyg_drv_cond_init

Function:

```
void cyg_drv_cond_init( cyg_drv_cond *cond,
cyg_drv_mutex *mutex )
```

Arguments:

cond—condition variable to initialize

mutex—mutex to associate with this condition variable

Result:

None

Level:

Thread

Description:

Initialize the condition variable pointed to by the cond argument. The mutex argument must point to a mutex with which this condition variable is associated. A thread may only wait on this condition variable when it has already locked the associated mutex. Waiting will cause the mutex to be unlocked, and when the thread is reawakened, it will automatically claim the mutex before continuing.

cyg_drv_cond_destroy

Function:

```
void cyg_drv_cond_destroy( cyg_drv_cond *cond )
```

Arguments:

cond - condition variable to destroy

Result:

None

Level:

Thread

Description:

Destroy the condition variable pointed to by the cond argument.

cyg_drv_cond_wait

Function:

```
void cyg_drv_cond_wait( cyg_drv_cond *cond )
```

Arguments:

cond - condition variable to wait on

Result:

None

Level:

Thread

Description:

Wait for a signal on the condition variable pointed to by the cond argument. The

thread must have locked the associated mutex before waiting on this condition variable. While the thread waits, the mutex will be unlocked, and will be re-locked before this function returns. It is possible for threads waiting on a condition variable to occasionally wake up spuriously. For this reason it is necessary to use this function in a loop that re-tests the condition each time it returns. Note that this function performs an implicit scheduler unlock/relock sequence, so that it may be used within an explicit `cyg_drv_dsr_lock()...cyg_drv_dsr_unlock()` structure.

cyg_drv_cond_signal

Function:

```
void cyg_drv_cond_signal( cyg_drv_cond *cond )
```

Arguments:

cond - condition variable to signal

Result:

None

Level:

DSR

Description:

Signal the condition variable pointed to by the cond argument. If there are any threads waiting on this variable at least one of them will be awakened. Note that in some configurations there may not be any difference between this function and **`cyg_drv_cond_broadcast()`**.

cyg_drv_cond_broadcast

Function:

```
void cyg_drv_cond_broadcast( cyg_drv_cond *cond )
```

Arguments:

cond - condition variable to broadcast to

Result:

None

Level:

DSR

Description:

Signal the condition variable pointed to by the cond argument. If there are any threads waiting on this variable they will all be awakened.

cyg_drv_interrupt_create

Function:

```
void cyg_drv_interrupt_create(
cyg_vector_t vector,
cyg_priority_t priority,
cyg_addrword_t data,
cyg_ISR_t *isr,
cyg_DSR_t *dsr,
cyg_handle_t *handle,
cyg_interrupt *intr
)
```

Arguments:

vector—vector to attach to

priority—queueing priority

data—data pointer

isr—interrupt service routine

dsr—deferred service routine

handle—returned handle

intr—put interrupt object here

Result:

None

Level:

Thread

Description:

Create an interrupt object and returns a handle to it. The object contains information about which interrupt vector to use and the ISR and DSR that will be called after the interrupt object is attached. The interrupt object will be allocated in the memory passed in the intr parameter. The interrupt object is not immediately attached; it must be attached with the **`cyg_interrupt_attach()`** call.

cyg_drv_interrupt_delete

Function:

```
void cyg_drv_interrupt_delete( cyg_handle_t interrupt )
```

Arguments:

interrupt—interrupt to delete

Result:

None

Level:

Thread

Description:

Detach the interrupt from the vector and free the memory passed in the intr argument to **cyg_drv_interrupt_create()** for reuse.

cyg_drv_interrupt_attach

Function:

```
void cyg_drv_interrupt_attach( cyg_handle_t interrupt )
```

Arguments:

interrupt—interrupt to attach

Result:

None

Level:

ISR

Description:

Attach the interrupt to the vector so that interrupts will be delivered to the ISR when the interrupt occurs.

cyg_drv_interrupt_detach

Function:

```
void cyg_drv_interrupt_detach( cyg_handle_t interrupt )
```

Arguments:

interrupt—interrupt to detach

Result:

None

Level:

ISR

Description:

Detach the interrupt from the vector so that interrupts will no longer be delivered to the ISR.

cyg_drv_interrupt_mask

Function:

```
void cyg_drv_interrupt_mask(cyg_vector_t vector )
```

Arguments:

vector—vector to mask

Result:

None

Level:

ISR

Description:

Program the interrupt controller to stop delivery of interrupts on the given vector. On architectures which implement interrupt priority levels this may also disable all lower priority interrupts.

cyg_drv_interrupt_unmask

Function:

```
void cyg_drv_interrupt_unmask(cyg_vector_t vector )
```

Arguments:

vector—vector to unmask

Result:

None

Level:

ISR

Description:

Program the interrupt controller to re-allow delivery of interrupts on the given vector.

cyg_drv_interrupt_acknowledge

Function:

```
void cyg_drv_interrupt_acknowledge( cyg_vector_t vector )
```

Arguments:

vector—vector to acknowledge

Result:

None

Level:

ISR

Description:

Perform any processing required at the interrupt controller and in the CPU to cancel the interrupt request. An ISR may also need to program the hardware of the device to prevent an immediate re-triggering of the interrupt.

cyg_drv_interrupt_configure

Function:

```
void cyg_drv_interrupt_configure(
 cyg_vector_t vector,
 cyg_bool_t level,
 cyg_bool_t up
 )
```

Arguments:

vector—vector to configure

level—level or edge triggered

up—rising/falling edge, high/low level

Result:

None

Level:

ISR

Description:

Program the interrupt controller with the characteristics of the interrupt source. The level argument chooses between level- or edge-triggered interrupts. The up argument chooses between high and low level for level triggered interrupts or rising and falling edges for edge triggered interrupts. This function only works with interrupt controllers that can control these parameters.

cyg_drv_interrupt_level

Function:

```
void cyg_drv_interrupt_level(
 cyg_vector_t vector,
 cyg_priority_t level
 )
```

Arguments:

vector—vector to configure

level—level to set

Result:

None

Level:

ISR

Description:

Program the interrupt controller to deliver the given interrupt at the supplied priority level. This function only works with interrupt controllers that can control this parameter.

cyg_ISR_t

Type:

```
typedef cyg_uint32 cyg_ISR_t(
 cyg_vector_t vector,
 cyg_addrword_t data
 )
```

Fields:

vector—vector being delivered

data—data value supplied by client

Result:

Bit mask indicating whether interrupt was handled and whether the DSR should be called.

Description:

Interrupt Service Routines definition. A pointer to a function with this prototype is passed to **`cyg_interrupt_create()`** when an interrupt object is created. When an interrupt is delivered the function will be called with the vector number and the data value that was passed to **`cyg_interrupt_create()`**.

The return value is a bit mask containing one or both of the following bits:

CYG_ISR_HANDLED

indicates that the interrupt was handled by this ISR. It is a configuration option whether this will prevent further ISR being run.

CYG_ISR_CALL_DSR

causes the DSR that was passed to **`cyg_interrupt_create()`** to be scheduled to

be called.

cyg_DSR_t

Type:

```
typedef void cyg_DSR_t(
cyg_vector_t vector,
cyg_ucount32 count,
cyg_addrword_t data
)
```

Fields:

vector—vector being delivered

count—number of times DSR has been scheduled

data—data value supplied by client

Result:

None

Description:

Deferred Service Routine definition. A pointer to a function with this prototype is passed to **`cyg_interrupt_create()`** when an interrupt object is created. When the ISR request the scheduling of its DSR, this function will be called at some later point. In addition to the vector and data arguments, which will be the same as those passed to the ISR, this routine is also passed a count of the number of times the ISR has requested that this DSR be scheduled. This counter is zeroed each time the DSR actually runs, so it indicates how many interrupts have occurred since it last ran.

Part V: The ISO Standard C and Math Libraries

15

C and math library overview

eCos provides compatibility with the ISO 9899:1990 specification for the standard C library, which is essentially the same as the better-known ANSI C3.159-1989 specification (C-89).

There are three aspects of this compatibility supplied by **eCos**. First there is a *C library* which implements the functions defined by the ISO standard, except for the mathematical functions. This is provided by the **eCos** C library package.

Then **eCos** provides a *math library*, which implements the mathematical functions from the ISO C library. This distinction between C and math libraries is frequently drawn — most standard C library implementations provide separate linkable files for the two, and the math library contains all the functions from the `math.h` header file.

There is a third element to the ISO C library, which is the environment in which applications run when they use the standard C library. This environment is set up by the C library startup procedure (see “C library startup” on page 153) and it provides (among other things) a **`main()`** entry point function, an **`exit()`** function that does the cleanup required by the standard (including handlers registered using the **`atexit()`** function), and an environment that can be read with **`getenv()`**.

The description in this manual focuses on the **eCos**-specific aspects of the C library (mostly related to **eCos**'s configurability) as well as mentioning the omissions from the standard in this release. We do not attempt to define the semantics of each function, since that information can be found in the ISO, ANSI, POSIX and IEEE standards, and the many good books that have been written about the standard C

library, that cover usage of these functions in a more general and useful way.

Omitted functionality

The ISO C functionality that is currently omitted in the C library can be grouped by the header files in which they are declared:

stdio.h

remove()
rename()
tmpfile()
tmpnam()
fseek()
ftell()
rewind()
fgetpos()
fsetpos()

Most of these functions are omitted because they only apply to disk-based file systems. These will be supported in a future version of **eCos**.

stdlib.h

mblen()
mbtowc()
wctomb()
mbstowcs()
wcstombs()
MB_CUR_MAX

All of these functions are related to multibyte and wide character support.

Included non-ISO functions

The following functions from the POSIX specification are included for convenience:

`extern char **`*environ* variable (for setting up the environment for use with **getenv()**)

_exit()

strtok_r()

rand_r()

asctime_r()

ctime_r()

localtime_r()

gmtime_r()

eCos provides the following additional implementation-specific functions within the standard C library to adjust the date and time settings:

```
void cyg_libc_time_setdst(
          cyg_libc_time_dst state );
```

This function sets the state of Daylight Savings Time. The values for *state* are:

```
CYG_LIBC_TIME_DSTNA   unknown
CYG_LIBC_TIME_DSTOFF  off
CYG_LIBC_TIME_DSTON   on
```

```
void cyg_libc_time_setzoneoffsets(
          time_t stdoffset, time_t dstoffset);
```

This function sets the offsets from UTC used when Daylight Savings Time is enabled or disabled. The offsets are in time_t's, which are seconds in the current inplementation.

```
Cyg_libc_time_dst cyg_libc_time_getzoneoffsets(
          time_t *stdoffset, time_t *dstoffset);
```

This function retrieves the current setting for Daylight Savings Time along with the offsets used for both STD and DST. The offsets are both in time_t's, which are seconds in the current implementation.

```
cyg_bool cyg_libc_time_settime(
          time_t utctime);
```

This function sets the current time for the system The time is specified as a **time_t** in UTC. It returns non-zero on error.

Math library compatibility modes

This math library is capable of being operated in several different compatibility modes. These options deal solely with how errors are handled.

There are 4 compatibility modes: ANSI/POSIX 1003.1; IEEE-754; X/Open Portability Guide issue 3 (XPG3); and System V Interface Definition Edition 3.

In IEEE mode, the **matherr()** function (see below) is never called, no warning messages are printed on the stderr output stream, and errno is never set.

In ANSI/POSIX mode, errno is set correctly, but **matherr()** is never called and no warning messages are printed on the stderr output stream.

In X/Open mode, errno is set correctly, **matherr()** is called, but no warning messages are printed on the stderr output stream.

In SVID mode, functions which overflow return a value *HUGE* (defined in `math.h`), which is the maximum single precision floating point value (as opposed to *HUGE_VAL* which is meant to stand for infinity). errno is set correctly and **matherr()** is called. If **matherr()** returns 0, warning messages are printed on the stderr output stream for some errors.

The mode can be compiled-in as IEEE-only, or any one of the above methods settable at run-time.

NOTE This math library assumes that the hardware (or software floating point emulation) supports IEEE-754 style arithmetic, 32-bit 2's complement integer arithmetic, doubles are in 64-bit IEEE-754 format.

matherr()

As mentioned above, in X/Open or SVID modes, the user can supply a function **matherr()** of the form:

```
int matherr(
 struct exception *e )
```

where struct exception is defined as:

```
struct exception {
 int type;
 char *name;
 double arg1, arg2, retval;
};
```

type is the exception type and is one of:

DOMAIN
: argument domain exception

SING
: argument singularity

OVERFLOW
: overflow range exception

UNDERFLOW
: underflow range exception

TLOSS
: total loss of significance

PLOSS

partial loss of significance

name is a string containing the name of the function

arg1 and *arg2* are the arguments passed to the function

retval is the default value that will be returned by the function, and can be changed by **matherr()**

NOTE matherr must have “C” linkage, not “C++” linkage.

If matherr returns zero, or the user doesn't supply their own matherr, then the following *usually* happens in SVID mode:

Table 1: Behavior of math exception handling

Type	*Behavior*
DOMAIN	0.0 returned, errno=EDOM, and a message printed on stderr
SING	HUGE of appropriate sign is returned, errno=EDOM, and a message is printed on stderr
OVERFLOW	HUGE of appropriate sign is returned, and errno=ERANGE
UNDERFLOW	0.0 is returned and errno=ERANGE
TLOSS	0.0 is returned, errno=ERANGE, and a message is printed on stderr
PLOSS	The current implementation doesn't return this type

X/Open mode is similar except that the message is not printed on stderr and HUGE_VAL is used in place of HUGE

Thread-safety and re-entrancy

With the appropriate configuration options set below, the math library is fully thread-safe if:

- Depending on the compatibility mode, the setting of the errno variable from the C library is thread-safe
- Depending on the compatibility mode, sending error messages to the stderr output stream using the C library **fputs()** function is thread-safe
- Depending on the compatibility mode, the user-supplied **matherr()** function and anything it depends on are thread-safe

In addition, with the exception of the **gamma*()** and **lgamma*()** functions, the math library is reentrant (and thus safe to use from interrupt handlers) if the Math library is always in IEEE mode.

Some implementation details

Here are some details about the implementation which might be interesting, although they do not affect the ISO-defined semantics of the library.

- It is possible to configure **eCos** to have the standard C library without the kernel. You might want to do this to use less memory. But if you disable the kernel, you will be unable to use memory allocation, thread-safety and certain stdio functions such as input. Other C library functionality is unaffected.
- The opaque type returned by **`clock()`** is called clock_t, and is implemented as a 64 bit integer. The value returned by **`clock()`** is only correct if the kernel is configured with real-time clock support, as determined by the *`CYGVAR_KERNEL_COUNTERS_CLOCK`* configuration option in `kernel.h`.
- The FILE type is not implemented as a structure, but rather as a CYG_ADDRESS.
- The GNU C compiler will place its own *built-in* implementations instead of some C library functions. This can be turned off with the *-fno-builtin* option. The functions affected by this are **`abs()`**, **`cos()`**, **`fabs()`**, **`labs()`**, **`memcmp()`**, **`memcpy()`**, **`sin()`**, **`sqrt()`**, **`strcmp()`**, **`strcpy()`**, and **`strlen()`**.
- For faster execution speed you should avoid this option and let the compiler use its built-ins. This can be turned off by invoking **GCC** with the *-fno-builtin* option.
- **`memcpy()`** and **`memset()`** are located in the infrastructure package, not in the C library package. This is because the compiler calls these functions, and the kernel needs to resolve them even if the C library is not configured.
- Error codes such as *`EDOM`* and *`ERANGE`*, as well as **`strerror()`**, are implemented in the *error* package. The error package is separate from the rest of the C and math libraries so that the rest of **eCos** can use these error handling facilities even if the C library is not configured.
- When **`free()`** is invoked, heap memory will normally be coalesced. If the *`CYGSEM_KERNEL_MEMORY_COALESCE`* configuration parameter is not set, memory will not be coalesced, which might cause programs to fail.
- Signals, as implemented by `<signal.h>`, are guaranteed to work correctly if raised using the **`raise()`** function from a normal working program context. Using signals from within an ISR or DSR context is not expected to work. Also, it is not guaranteed that if *`CYGSEM_LIBC_SIGNALS_HWEXCEPTIONS`* is set, that handling a signal using **`signal()`** will necessarily catch that form of exception. For example, it may be expected that a divide-by-zero error would be caught by handling `SIGFPE`. However it depends on the underlying HAL implementation to implement the required hardware exception. And indeed the hardware itself may

not be capable of detecting these exceptions so it may not be possible for the HAL implementer to do this in any case. Despite this lack of guarantees in this respect, the signals implementation is still ISO C compliant since ISO C does not offer any such guarantees either.

- The **getenv()** function is implemented (unless the *CYGPKG_LIBC_ENVIRONMENT* configuration option is turned off), but there is no shell or **putenv()** function to set the environment dynamically. The environment is set in a global variable *environ*, declared as:

```
extern char **environ; // Standard environment definition
```

The environment can be statically initialized at startup time using the *CYGDAT_LIBC_DEFAULT_ENVIRONMENT* option. If so, remember that the final entry of the array initializer must be NULL.

Here is a minimal **eCos** program which demonstrates the use of environments (see also the test case in `language/c/libc/current/tests/stdlib/getenv.c`):

```
#include <stdio.h>
#include <stdlib.h> // Main header for stdlib functions

extern char **environ; // Standard environment definition

int
main( int argc, char *argv[] )
{
 char *str;
 char *env[] = { "PATH=/usr/local/bin:/usr/bin",
 "HOME=/home/fred",
 "TEST=1234=5678",
 "home=hatstand",
 NULL };

 printf("Display the current PATH environment variable\n");

 environ = (char **)&env;

 str = getenv("PATH");

 if (str==NULL) {
 printf("The current PATH is unset\n");
 } else {
 printf("The current PATH is \"%s\"\n", str);
 }
 return 0;
}
```

Thread safety

The ISO C library has configuration options that control thread safety, i.e. working behavior if multiple threads call the same function at the same time.

The following functionality has to be configured correctly, or used carefully in a multi-threaded environment:

- **`printf()`** (and all standard I/O functions except for **`sprintf()`** and **`sscanf()`**
- **`strtok()`**
- **`rand()`** and **`srand()`**
- **`signal()`** and **`raise()`**
- **`asctime()`**, **`ctime()`**, **`gmtime()`**, and **`localtime()`**
- the **`errno`** variable
- the **`environ`** variable
- date and time settings

In some cases, to make **eCos** development easier, functions are provided (as specified by POSIX 1003.1) that define re-entrant alternatives, i.e. **`rand_r()`**, **`strtok_r()`**, **`asctime_r()`**, **`ctime_r()`**, **`gmtime_r()`**, and **`localtime_r()`**. In other cases, configuration options are provided that control either locking of functions or their shared data, such as with standard I/O streams, or by using per-thread data, such as with the **`errno`** variable.

In some other cases, like the setting of date and time, no re-entrant or thread-safe alternative or configuration is provided as it is simply not a worthwhile addition (date and time should rarely need to be set.)

C library startup

The C library includes a function declared as:

```
void cyg_iso_c_start( void )
```

This function is used to start an environment in which an ISO C style program can run in the most compatible way.

What this function does is to create a thread which will invoke **`main()`** — normally considered a program's entry point. In particular, it can supply arguments to **`main()`** using the *`CYGDAT_LIBC_ARGUMENTS`* configuration option (see “Option: Arguments to main()”, in Section V), and when returning from **`main()`**, or calling **`exit()`**, pending stdio file output is flushed and any functions registered with **`atexit()`** are invoked. This is all compliant with the ISO C standard in this respect.

This thread starts execution when the **eCos** scheduler is started. If the **eCos** kernel

package is not available (and hence there is no scheduler), then **cyg_iso_c_start()** will invoke the **main()** function directly, i.e. it will not return until the **main()** function returns.

The **main()** function should be defined as the following, and if defined in a C++ file, should have "C" linkage:

```
extern int main(
    int argc,
    char *argv )[]
```

The thread that is started by **cyg_iso_c_start()** can be manipulated directly, if you wish. For example you can suspend it. The kernel C API needs a handle to do this, which is available by including the following in your source code.

```
extern cyg_handle_t cyg_libc_main_thread;
```

Then for example, you can suspend the thread with the line:

```
cyg_thread_suspend( cyg_libc_main_thread );
```

If you call **cyg_iso_c_start()** and do not provide your own **main()** function, the system will provide a **main()** for you which will simply return immediately.

In the default configuration, **cyg_iso_c_start()** is invoked automatically by the **cyg_package_start()** function in the infrastructure configuration. This means that in the simplest case, your program can indeed consist of simply:

```
int main( int argc, char *argv[] )
{
 printf("Hello eCos\n");
}
```

If you override **cyg_package_start()** or **cyg_start()**, or disable the infrastructure configuration option *CYGSEM_START_ISO_C_COMPATIBILITY* then you must ensure that you call **cyg_iso_c_start()** yourself if you want to be able to have your program start at the entry point of **main()** automatically.

Index

A

C

D

E

F

G

H

I

K

L

M

N

P

R

S

T

U

V