PART SIX

Embedded Systems

The number of deployed operating systems in embedded systems dwarfs the number deployed in personal computers and desktop computers. By that measure, embedded OSs constitute the most important category of OS. Embedded operating systems have their own unique requirements and design issues, which are explored in this part.

ROAD MAP FOR PART SIX

Chapter 13 Embedded Operating Systems

Chapter 13 begins with an introduction to the nature of embedded systems and then goes on to look at key characteristics of embedded operating systems. The chapter then looks at two very different approaches to the design of embedded operating systems using two of the most widely used systems: eCos and TinyOS.

CHAPTER 13

EMBEDDED OPERATING SYSTEMS

In this chapter, we examine one of the most important and widely used categories of operating systems: embedded operating systems. The embedded system environment places unique and demanding requirements on the OS and calls for design strategies quite different than that found in ordinary operating systems.

We begin with an overview of the concept of embedded systems and then turn to an examination of the principles of embedded operating systems. Finally, this chapter surveys two very different approaches to embedded OS design.

13.1 EMBEDDED SYSTEMS

The term *embedded system* refers to the use of electronics and software within a product, as opposed to a general-purpose computer, such as a laptop or desktop system. The following is a good general definition:[1]

Embedded system. A combination of computer hardware and software, and perhaps additional mechanical or other parts, designed to perform a dedicated function. In many cases, embedded systems are part of a larger system or product, as in the case of an antilock braking system in a car.

Embedded systems far outnumber general-purpose computer systems, encompassing a broad range of applications (Table 13.1). These systems have widely varying requirements and constraints, such as the following [GRIM05]:

- Small to large systems, implying very different cost constraints, thus different needs for optimization and reuse
- Relaxed to very strict requirements and combinations of different quality requirements, for example, with respect to safety, reliability, real-time, flexibility, and legislation
- Short to long life times
- Different environmental conditions in terms of, for example, radiation, vibrations, and humidity
- Different application characteristics resulting in static versus dynamic loads, slow to fast speed, compute versus interface intensive tasks, and/or combinations thereof
- Different models of computation ranging from discrete-event systems to those involving continuous time dynamics (usually referred to as hybrid systems)

Often, embedded systems are tightly coupled to their environment. This can give rise to real-time constraints imposed by the need to interact with the environment. Constraints, such as required speeds of motion, required precision of measurement, and required time durations, dictate the timing of software operations. If multiple activities must be managed simultaneously, this imposes more complex real-time constraints.

[1]Michael Barr, *Embedded Systems Glossary.* Netrino Technical Library. http://www.netrino.com/Publications/Glossary/index.php

Table 13.1 Examples of Embedded Systems and Their Markets [NOER05]

Market	Embedded Device
Automotive	Ignition system Engine control Brake system
Consumer electronics	Digital and analog televisions Set-top boxes (DVDs, VCRs, Cable boxes) Personal digital assistants (PDAs) Kitchen appliances (refrigerators, toasters, microwave ovens) Automobiles Toys/games Telephones/cell phones/pagers Cameras Global positioning systems
Industrial control	Robotics and controls systems for manufacturing Sensors
Medical	Infusion pumps Dialysis machines Prosthetic devices Cardiac monitors
Office automation	Fax machine Photocopier Printers Monitors Scanners

Figure 13.1, based on [KOOP96], shows in general terms an embedded system organization. In addition to the processor and memory, there are a number of elements that differ from the typical desktop or laptop computer:

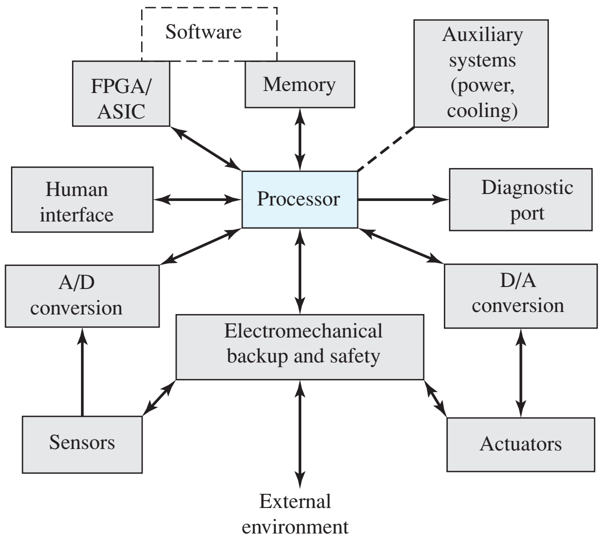


Figure 13.1 **Possible Organization of an Embedded System**

- There may be a variety of interfaces that enable the system to measure, manipulate, and otherwise interact with the external environment.
- The human interface may be as simple as a flashing light or as complicated as real-time robotic vision.
- The diagnostic port may be used for diagnosing the system that is being controlled—not just for diagnosing the computer.
- Special-purpose field programmable (FPGA), application specific (ASIC), or even nondigital hardware may be used to increase performance or safety.
- Software often has a fixed function and is specific to the application.

13.2 CHARACTERISTICS OF EMBEDDED OPERATING SYSTEMS

A simple embedded system, with simple functionality, may be controlled by a special-purpose program or set of programs with no other software. Typically, more complex embedded systems include an OS. Although it is possible in principle to use a general-purpose OS, such as Linux, for an embedded system, constraints of memory space, power consumption, and real-time requirements typically dictate the use of a special-purpose OS designed for the embedded system environment.

The following are some of the unique characteristics and design requirements for embedded operating systems:

- **Real-time operation:** In many embedded systems, the correctness of a computation depends, in part, on the time at which it is delivered. Often, real-time constraints are dictated by external I/O and control stability requirements.
- **Reactive operation:** Embedded software may execute in response to external events. If these events do not occur periodically or at predictable intervals, the embedded software may need to take into account worst-case conditions and set priorities for execution of routines.
- **Configurability:** Because of the large variety of embedded systems, there is a large variation in the requirements, both qualitative and quantitative, for embedded OS functionality. Thus, an embedded OS intended for use on a variety of embedded systems must lend itself to flexible configuration so that only the functionality needed for a specific application and hardware suite is provided. [MARW06] gives the following examples. The linking and loading functions can be used to select only the necessary OS modules to load. Conditional compilation can be used. If an object-oriented structure is used, proper subclasses can be defined. However, verification is a potential problem for designs with a large number of derived tailored operating systems. Takada cites this as a potential problem for eCos [TAKA01].
- **I/O device flexibility:** There is virtually no device that needs to be supported by all versions of the OS, and the range of I/O devices is large. [MARW06] suggests that it makes sense to handle relatively slow devices such as disks and network interfaces by using special tasks instead of integrating their drives into the OS kernel.

- **Streamlined protection mechanisms:** Embedded systems are typically designed for a limited, well-defined functionality. Untested programs are rarely added to the software. After the software has been configured and tested, it can be assumed to be reliable. Thus, apart from security measures, embedded systems have limited protection mechanisms. For example, I/O instructions need not be privileged instructions that trap to the OS; tasks can directly perform their own I/O. Similarly, memory protection mechanisms can be minimized. [MARW06] provides the following example. Let `switch` correspond to the memory-mapped I/O address of a value that needs to be checked as part of an I/O operation. We can allow the I/O program to use an instruction such as `load register, switch` to determine the current value. This approach is preferable to the use of an OS service call, which would generate overhead for saving and restoring the task context.
- **Direct use of interrupts:** General-purpose operating systems typically do not permit any user process to use interrupts directly. [MARW06] lists three reasons why it is possible to let interrupts directly start or stop tasks (e.g., by storing the task's start address in the interrupt vector address table) rather than going through OS interrupt service routines: (1) Embedded systems can be considered to be thoroughly tested, with infrequent modifications to the OS or application code; (2) protection is not necessary, as discussed in the preceding bullet item; and (3) efficient control over a variety of devices is required.

There are two general approaches to developing an embedded OS. The first approach is to take an existing OS and adapt it for the embedded application. The other approach is to design and implement an OS intended solely for embedded use.[2]

Adapting an Existing Commercial Operating System

An existing commercial OS can be used for an embedded system by adding real-time capability, streamlining operation, and adding necessary functionality. This approach typically makes use of Linux, but FreeBSD, Windows, and other general-purpose operating systems have also been used. Such operating systems are typically slower and less predictable than a special-purpose embedded OS. An advantage of this approach is that the embedded OS derived from a commercial general-purpose OS is based on a set of familiar interfaces, which facilitates portability.

The disadvantage of using a general-purpose OS is that it is not optimized for real-time and embedded applications. Thus, considerable modification may be required to achieve adequate performance. In particular, a typical OS optimizes for the average case rather than the worst case for scheduling, usually assigns resources on demand, and ignores most if not all semantic information about an application.

[2]Much of the discussion in Section 13.2 is based on course notes on embedded systems from Prof. Rajesh Gupta, University of California at San Diego. http://mesl.ucsd.edu/gupta/cse237bw07.html

Purpose-Built Embedded Operating System

A significant number of operating systems have been designed from the ground up for embedded applications. Two prominent examples of this latter approach are eCos and TinyOS, both of which are discussed in this chapter.

Typical characteristics of a specialized embedded OS include the following:

- Has a fast and lightweight process or thread switch
- Scheduling policy is real time and dispatcher module is part of scheduler instead of separate component.
- Has a small size
- Responds to external interrupts quickly; typical requirement is response time of less than 10 µs
- Minimizes intervals during which interrupts are disabled
- Provides fixed or variable sized partitions for memory management as well as the ability to lock code and data in memory
- Provides special sequential files that can accumulate data at a fast rate

To deal with timing constraints, the kernel

- Provides bounded execution time for most primitives
- Maintains a real-time clock
- Provides for special alarms and timeouts
- Supports real-time queuing disciplines such as earliest deadline first and primitives for jamming a message into the front of a queue
- Provides primitives to delay processing by a fixed amount of time and to suspend/resume execution

The characteristics just listed are common in embedded operating systems with real-time requirements. However, for complex embedded systems, the requirement may emphasize predictable operation over fast operation, necessitating different design decisions, particularly in the area of task scheduling.

13.3 ECOS

The Embedded Configurable Operating System (eCos) is an open source, royalty-free, real-time OS intended for embedded applications. The system is targeted at high-performance small embedded systems. For such systems, an embedded form of Linux or other commercial OS would not provide the streamlined software required. The eCos software has been implemented on a wide variety of processor platforms, including Intel IA32, PowerPC, SPARC, ARM, CalmRISC, MIPS, and NEC V8xx. It is one of the most widely used embedded operating systems.

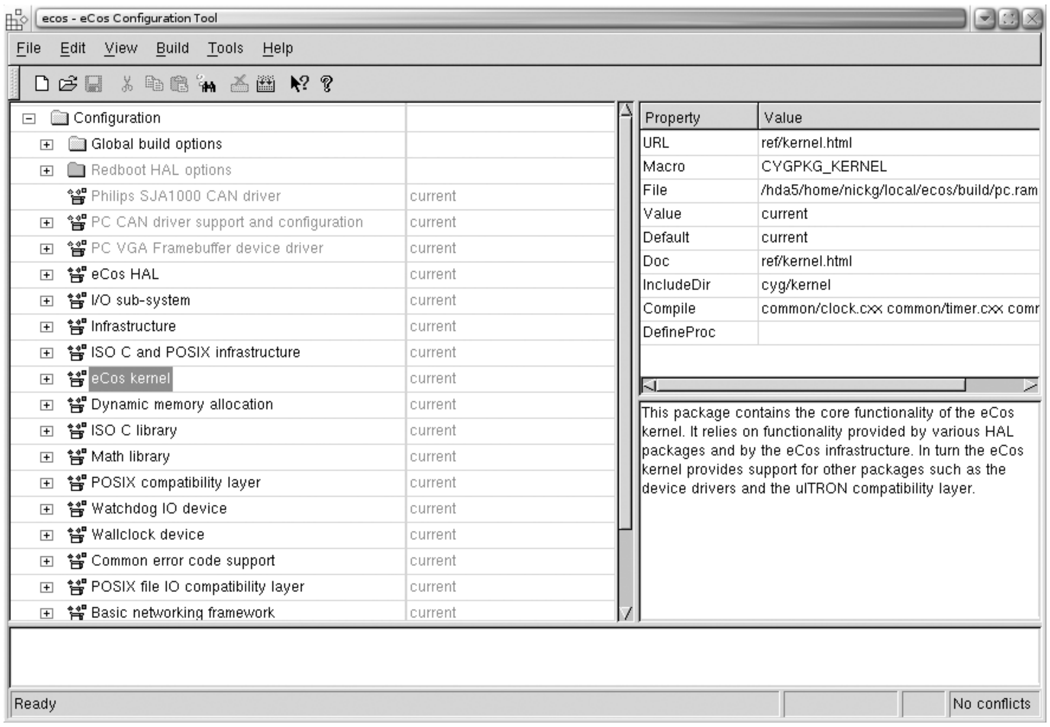
Configurability

An embedded OS that is flexible enough to be used in a wide variety of embedded applications and on a wide variety of embedded platforms must provide more functionality than will be needed for any particular application and platform. For example, many real-time operating systems support task switching, concurrency controls, and a variety of priority scheduling mechanisms. A relatively simple embedded system would not need all these features.

The challenge is to provide an efficient, user-friendly mechanism for configuring selected components and for enabling and disabling particular features within components. The eCos configuration tool, which runs on Windows or Linux, is used to configure an eCos package to run on a target embedded system. The complete eCos package is structured hierarchically, making it easy, using the configuration tool, to assemble a target configuration. At a top level, eCos consists of a number of components, and the configuration user may select only those components needed for the target application. For example, a system might have a particular serial I/O device. The configuration user would select serial I/O for this configuration, then select one or more specific I/O devices to be supported. The configuration tool would include the minimum necessary software for that support. The configuration user can also select specific parameters, such as default data rate and the size of I/O buffers to be used.

This configuration process can be extended down to finer levels of detail, even to the level of individual lines of code. For example, the configuration tool provides the option of including or omitting a priority inheritance protocol.

Figure 13.2 shows the top level of the eCos configuration tool as seen by the tool user. Each of the items on the list in the left-hand window can be selected or



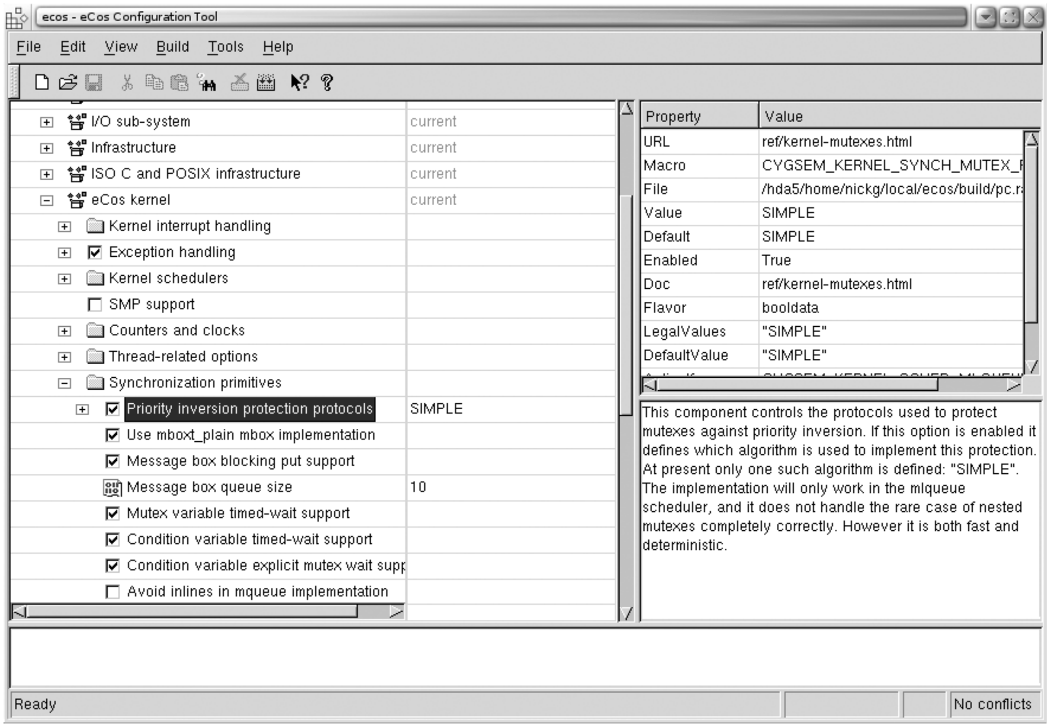
Configuration	
Global build options	
Redboot HAL options	
Philips SJA1000 CAN driver	current
PC CAN driver support and configuration	current
PC VGA Framebuffer device driver	current
eCos HAL	current
I/O sub-system	current
Infrastructure	current
ISO C and POSIX infrastructure	current
eCos kernel	current
Dynamic memory allocation	current
ISO C library	current
Math library	current
POSIX compatibility layer	current
Watchdog IO device	current
Wallclock device	current
Common error code support	current
POSIX file IO compatibility layer	current
Basic networking framework	current

Property	Value
URL	ref/kernel.html
Macro	CYGPKG_KERNEL
File	/hda5/home/nickg/local/ecos/build/pc.ram
Value	current
Default	current
Doc	ref/kernel.html
IncludeDir	cyg/kernel
Compile	common/clock.cxx common/timer.cxx comr
DefineProc	

This package contains the core functionality of the eCos kernel. It relies on functionality provided by various HAL packages and by the eCos infrastructure. In turn the eCos kernel provides support for other packages such as the device drivers and the uITRON compatibility layer.

Ready | No conflicts

Figure 13.2 eCos Configuration Tool—Top Level


ecos - eCos Configuration Tool
File Edit View Build Tools Help

I/O sub-system	current
Infrastructure	current
ISO C and POSIX infrastructure	current
eCos kernel	current
Kernel interrupt handling	
☑ Exception handling	
Kernel schedulers	
☐ SMP support	
Counters and clocks	
Thread-related options	
Synchronization primitives	
☑ Priority inversion protection protocols	SIMPLE
☑ Use mbox_plain mbox implementation	
☑ Message box blocking put support	
Message box queue size	10
☑ Mutex variable timed-wait support	
☑ Condition variable timed-wait support	
☑ Condition variable explicit mutex wait supp	
☐ Avoid inlines in mqueue implementation	

Property	Value
URL	ref/kernel-mutexes.html
Macro	CYGSEM_KERNEL_SYNCH_MUTEX_F
File	/hda5/home/nickg/local/ecos/build/pc.r
Value	SIMPLE
Default	SIMPLE
Enabled	True
Doc	ref/kernel-mutexes.html
Flavor	booldata
LegalValues	"SIMPLE"
DefaultValue	"SIMPLE"

This component controls the protocols used to protect mutexes against priority inversion. If this option is enabled it defines which algorithm is used to implement this protection. At present only one such algorithm is defined: "SIMPLE". The implementation will only work in the mlqueue scheduler, and it does not handle the rare case of nested mutexes completely correctly. However it is both fast and deterministic.

Ready   No conflicts

Figure 13.3 **eCos Configuration Tool—Kernel Details**

deselected. When an item is highlighted, the lower right-hand window provides a description and the upper right-hand window provides a link to further documentation plus additional information about the highlighted item. Items on the list can be expanded to provide a finer-grained menu of options. Figure 13.3 illustrates an expansion of the eCos kernel option. In this figure, note that exception handling has been selected for inclusion, but SMP (symmetric multiprocessing) has been omitted. In general, components and individual options can be selected or omitted. In some cases, individual values can be set; for example, a minimum acceptable stack size is an integer value that can be set or left to a default value.

Figure 13.4 shows a typical example of the overall process of creating the binary image to execute in the embedded system. This process is run on a source system, such as a Windows or Linux platform, and the executable image is destined to execute on a target embedded system, such as a sensor in an industrial environment. At the highest software level is the application source code for the particular embedded application. This code is independent of eCos but makes use of application programming interfaces (API) to sit on top of the eCos software. There may be only one version of the application source code, or there may be variations for different versions of the target embedded platform. In this example, the GNU make utility is used to selectively determine which pieces of a program need to be compiled or recompiled (in the case of a modified version of the source code) and issues the commands to recompile them. The GNU cross compiler, executing on the source

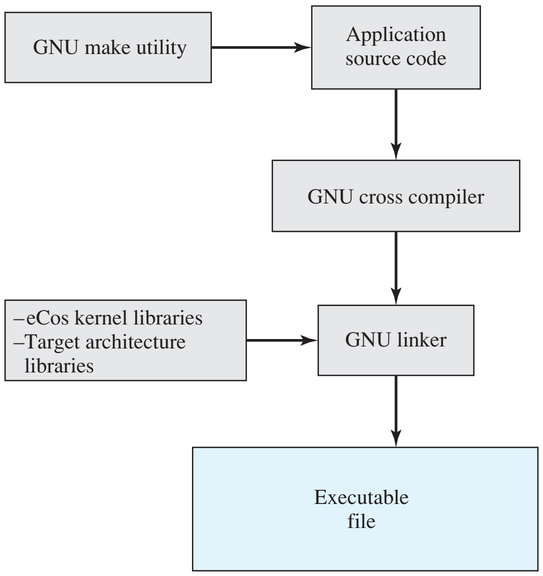


Figure 13.4 Loading an eCos Configuration

platform, then generates the binary executable code for the target embedded platform. The GNU linker links the application object code with the code generated by the eCos configuration tool. This latter set of software includes selected portions of the eCos kernel plus selected software for the target embedded system. The result can then be loaded into the target system.

eCos Components

A key design requirement for eCos is portability to different architectures and platforms with minimal effort. To met this requirement, eCos consists of a layered set of components (Figure 13.5).

Hardware Abstraction Layer (HAL) At the bottom is the hardware abstraction layer (HAL). The HAL is software that presents a consistent API to the upper

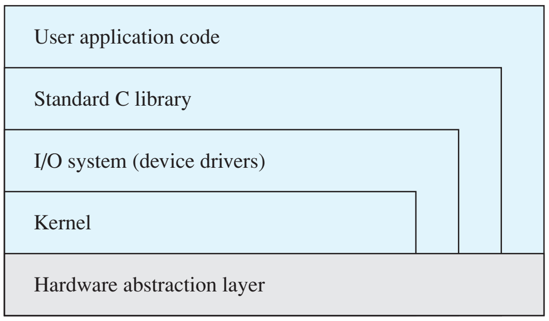


Figure 13.5 eCos Layered Structure

```
#define HAL_ENABLE_INTERRUPTS()                         \
 asm volatile (                                         \
   "mrs r3, cpsr;"                                      \
   "bic r3, r3, #0xC0;"                                 \
   "mrs cpsr, r3;"                                      \
   :                                                    \
   :                                                    \
   : "r3"                                               \
   );                                                   \
```

(a) ARM architecture

```
#define HAL_ENABLE_INTERRUPTS()                         \
 CYG_MACRO_START                                        \
 cyg_uint32 tmp1, tmp2                                  \
 asm volatile (                                         \
   "mfmsr %0;"                                          \
   "ori %1,%1,0x800;"                                   \
   "r1wimi %0,%1,0,16,16;"                              \
   "mtmsr %0;"                                          \
   : "=r" (tmp1), "=r" (tmp2));                         \
  CYG_MACRO_END                                         \
```

(b) PowerPC architecture

Figure 13.6 Two Implementations of `Hal_Enable_Interrupts()` Macro

layers and maps upper-layer operations onto a specific hardware platform. Thus, the HAL is different for each hardware platform. Figure 13.6 is an example that demonstrates how the HAL abstracts hardware-specific implementations for the same API call on two different platforms. As this example shows, the call from an upper layer to enable interrupts is the same on both platforms, but the C code implementation of the function is specific to each platform.

The HAL is implemented as three separate modules:

- **Architecture:** Defines the processor family type. This module contains the code necessary for processor startup, interrupt delivery, context switching, and other functionality specific to the instruction set architecture of that processor family.
- **Variant:** Supports the features of the specific processor in the family. An example of a supported feature is an on-chip module such as a memory management unit (MMU).
- **Platform:** Extends the HAL support to tightly coupled peripherals like interrupt controllers and timer devices. This module defines the platform or board that includes the selected processor architecture and variant. It includes code for startup, chip selection configuration, interrupt controllers, and timer devices.

Note that the HAL interface can be directly used by any of the upper layers, promoting efficient code.

eCos Kernel The eCos kernel was designed to satisfy four main objectives:

- **Low interrupt latency:** The time it takes to respond to an interrupt and begin executing an ISR.

- **Low task switching latency:** The time it takes from when a thread becomes available to when actual execution begins.
- **Small memory footprint:** Memory resources for both program and data are kept to a minimum by allowing all components to configure memory as needed.
- **Deterministic behavior:** Throughout all aspect of execution, the kernels performance must be predictable and bounded to meet real-time application requirements.

The eCos kernel provides the core functionality needed for developing multi-threaded applications:

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

Some functionality that is typically included in the kernel of an OS is not included in the eCos kernel. For example, memory allocation is handled by a separate package. Similarly, each device driver is a separate package. Various packages are combined and configured using the eCos configuration technology to meet the requirements of the application. This makes for a lean kernel. Further, the minimal nature of the kernel means that for some embedded platforms, the eCos kernel is not used at all. Simple single-threaded applications can be run directly on HAL. Such configurations can incorporate needed C library functions and device drivers but avoid the space and time overhead of the kernel.

I/O System The eCos I/O system is a framework for supporting device drivers. A variety of drivers for a variety of platforms are provided in the eCos configuration package. These include drivers for serial devices, Ethernet, flash memory interfaces, and various I/O interconnects such as PCI (peripheral component interconnect) and USB (universal serial bus). In addition, users can develop their own device drivers.

The principal objective for the I/O system is efficiency, with no unnecessary software layering or extraneous functionality. Device drivers provide the necessary functions for input, output, buffering, and device control.

As mentioned, device drivers and other higher-layer software may be implemented directly on the HAL if this is appropriate. If specialized kernel type functions are needed, then the device driver is implemented using kernel APIs. The kernel provides a three-level interrupt model [ECOS07]:

- **Interrupt service routines (ISRs):** Invoked in response to a hardware interrupt. 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.

Table 13.2 Device Driver Interface to the eCos Kernel: Concurrency

`cyg_drv_spinlock_init`	Initialize a spinlock in a locked or unlocked state.
`cyg_drv_spinlock_destroy`	Destroy a spinlock that is no longer of use.
`cyg_drv_spinlock_spin`	Claim a spinlock, waiting in a busy loop until it is available.
`cyg_drv_spinlock_clear`	Clear a spinlock. This clears the spinlock and allows another CPU to claim it. If there is more than one CPU waiting in `cyg_drv_spinlock_spin`, then just one of them will be allowed to proceed.
`cyg_drv_spinlock_test`	Inspect the state of the spinlock. If the spinlock is not locked, then the result is TRUE. If it is locked then the result will be FALSE.
`cyg_drv_spinlock_spin_intsave`	This function behaves exactly like
`cyg_drv_spinlock_spin`	except that it also disables interrupts before attempting to claim the lock. The current interrupt enable state is saved in *istate. Interrupts remain disabled once the spinlock has been claimed and must be restored by calling `cyg_drv_spinlock_clear_intsave`.
`cyg_drv_mutex_init`	Initialize a mutex.
`cyg_drv_mutex_destroy`	Destroy a mutex.
`cyg_drv_mutex_lock`	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`	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`	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`	Release all threads waiting on the mutex.
`cyg_drv_cond_init`	Initialize a condition variable associated with a mutex with. 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`	Destroy the condition variable.
`cyg_drv_cond_wait`	Wait for a signal on a condition variable.
`cyg_drv_cond_signal`	Signal a condition variable. If there are any threads waiting on this variable, at least one of them will all be awakened.
`cyg_drv_cond_broadcast`	Signal a condition variable. If there are any threads waiting on this variable, they will all be awakened.

- **Deferred service routines (DSRs):** Invoked in response to a request by an ISR. 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.
- **Threads:** The clients of the driver. Threads are able to make all API calls and in particular are allowed to wait on mutexes and condition variables.

Tables 13.2 and 13.3 show the device driver interface to the kernel. These tables give a good feel for the type of functionality available in the kernel to support

Table 13.3 Device Driver Interface to the eCos Kernel: Interrupts

`cyg_drv_isr_lock`	Disable delivery of interrupts, preventing all ISRs running. This function maintains a counter of the number of times it is called.
`cyg_drv_isr_unlock`	Reenable delivery of interrupts, allowing ISRs to run. This function decrements the counter maintained by `cyg_drv_isr_lock`, and only reallows interrupts when it goes to zero.
`cyg_ISR_t`	Define ISR.
`cyg_drv_dsr_lock`	Disable scheduling of DSRs. This function maintains a counter of the number of times it has been called.
`cyg_drv_dsr_unlock`	Reenable 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_DSR_t`	Define DSR prototype.
`cyg_drv_interrupt_create`	Create an interrupt object and returns a handle to it.
`cyg_drv_interrupt_delete`	Detach the interrupt from the vector and free the memory for reuse.
`cyg_drv_interrupt_attach`	Attach an interrupt to a vector so that interrupts will be delivered to the ISR when the interrupt occurs.
`cyg_drv_interrupt_detach`	Detach the interrupt from the vector so that interrupts will no longer be delivered to the ISR.
`cyg_drv_interrupt_mask`	Program the interrupt controller to stop delivery of interrupts on the given vector.
`cyg_drv_interrupt_mask_intunsafe`	Program the interrupt controller to stop delivery of interrupts on the given vector. This version differs from `cyg_drv_interrupt_mask` in not being interrupt safe. So in situations where, for example, interrupts are already known to be disabled, this may be called to avoid the extra overhead.
`cyg_drv_interrupt_unmask, cyg_drv_interrupt_unmask_intunsafe`	Program the interrupt controller to reallow delivery of interrupts on the given vector.
`cyg_drv_interrupt_acknowledge`	Perform any processing required at the interrupt controller and in the CPU to cancel the current interrupt request.
`cyg_drv_interrupt_configure`	Program the interrupt controller with the characteristics of the interrupt source.
`cyg_drv_interrupt_level`	Program the interrupt controller to deliver the given interrupt at the supplied priority level.
`cyg_drv_interrupt_set_cpu`	On multiprocessor systems, this function causes all interrupts on the given vector to be routed to the specified CPU. Subsequently, all such interrupts will be handled by that CPU.
`cyg_drv_interrupt_get_cpu`	On multiprocessor systems, this function returns the ID of the CPU to which interrupts on the given vector are currently being delivered.

device drivers. Note that the device driver interface can be configured for one or more of the following concurrency mechanisms: spinlocks, condition variables, and mutexes. These are described in a subsequent portion of this discussion.

Standard C Libraries A complete Standard C run-time library is provided. Also included is a complete math run time library for high-level mathematics functions, including a complete IEEE-754 floating-point library for those platforms without hardware floating points.

eCos Scheduler

The eCos kernel can be configured to provide one of two scheduler designs: the bitmap scheduler and a multilevel queue scheduler. The configuration user selects the appropriate scheduler for the environment and the application. The bitmap scheduler provides efficient scheduling for a system with a small number of threads that may be active at any point in time. The multiqueue scheduler is appropriate if the number of threads is dynamic or if it is desirable to have multiple threads at the same priority level. The multilevel scheduler is also needed if time slicing is desired.

Bitmap Scheduler A bitmap scheduler supports multiple priority levels, but only one thread can exist at each priority level at any given time. Scheduling decisions are quite simple with this scheduler (Figure 13.7a). When a blocked thread become ready to run, it may preempt a thread of lower priority. When a running thread suspends, the ready thread with the highest priority is dispatched. A thread can be suspended because it is blocked on a synchronization primitive, because it is interrupted, or because it relinquishes control. Because there is only one thread, at most, at each priority level, the scheduler does not have to make a decision as to which thread at a given priority level should be dispatched next.

The bitmap scheduler is configured with 8, 16, or 32 priority levels. A simple bitmap is kept of the threads that are ready to execute. The scheduler need only determine the position of the most significant one bit in the bitmap to make a scheduling decision.

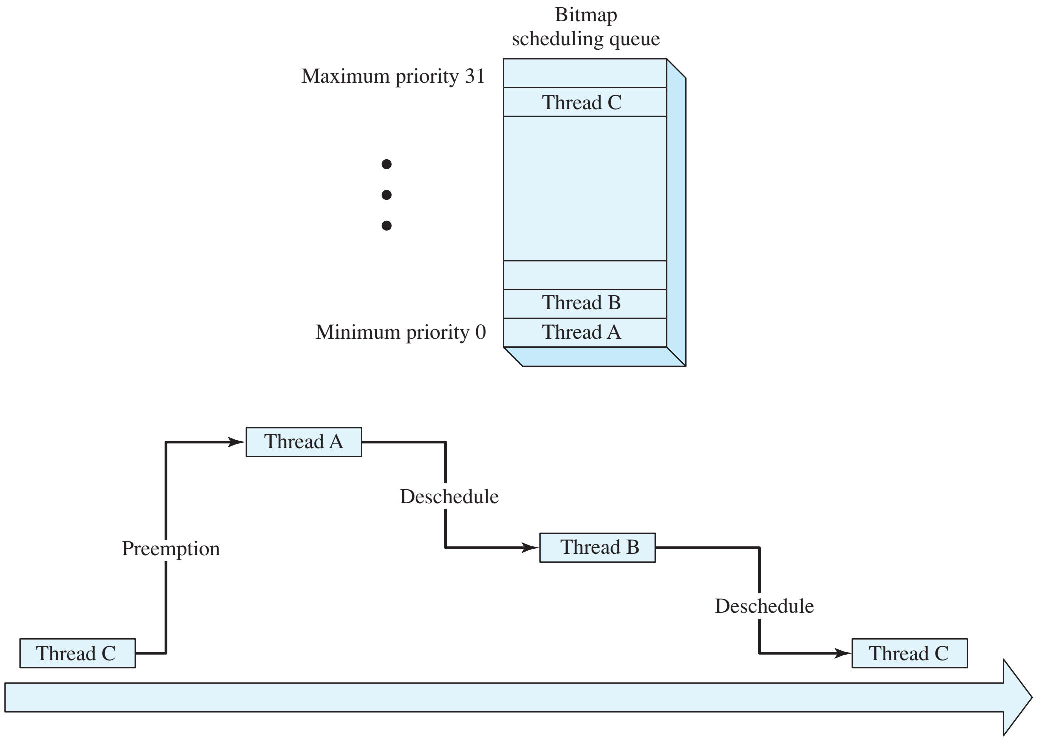
Multilevel Queue Scheduler As with the bitmap scheduler, the multilevel queue scheduler supports up to 32 priority levels. The multilevel queue scheduler allows for multiple active threads at each priority level, limited only by system resources.

Figure 13.7b illustrates the nature of the multilevel queue scheduler. A data structure represents the number of ready threads at each priority level. When a blocked thread become ready to run, it may preempt a thread of lower priority. As with the bitmap scheduler, a running thread may be blocked on a synchronization primitive, because it is interrupted, or because it relinquishes control. When a thread is blocked, the scheduler must first determine if one or more threads at the same priority level as the blocked thread is ready. If so, the scheduler chooses the one at the front of the queue. Otherwise, the scheduler looks for the next highest priority level with one or more ready threads and dispatches one of these threads.

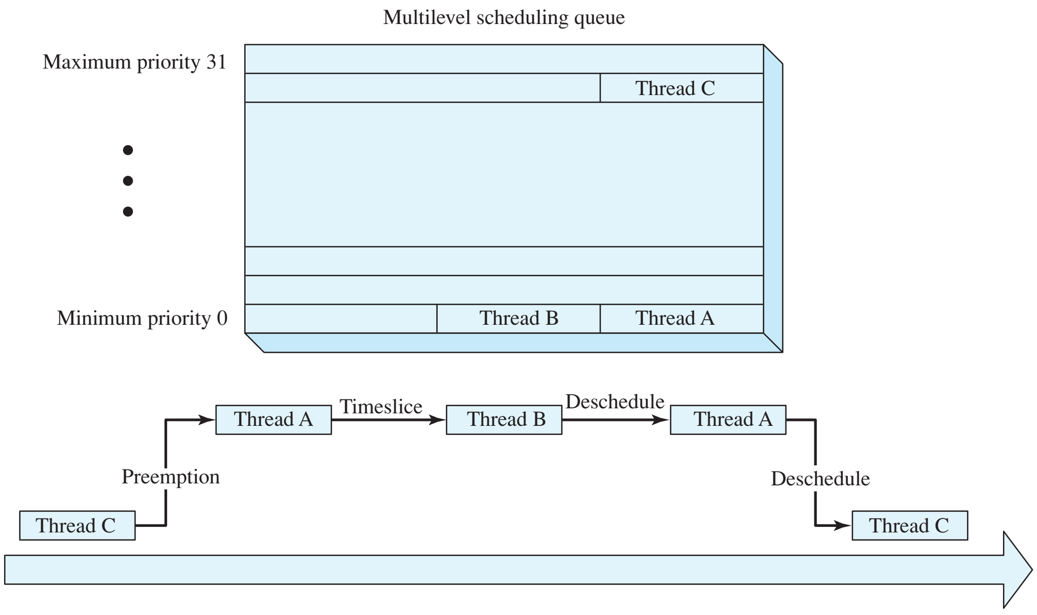
In addition, the multilevel queue scheduler can be configured for time slicing. Thus, if a thread is running and there is one or more ready threads at the same priority level, the scheduler will suspend the running thread after one time slice and choose the next thread in the queue at that priority level. This is a round-robin policy within one priority level. Not all applications require time slicing. For example, an application may contain only threads that block regularly for some other reason. For these applications, the user can disable time slicing, which reduces the overhead associated with timer interrupts.

eCos Thread Synchronization

The eCos kernel can be configured to include one or more of six different thread synchronization mechanisms. These include the classic synchronization mechanisms: mutexes, semaphores, and condition variables. In addition, eCos supports two syn-



(a) Bitmap scheduler thread operation



(b) Multilevel queue scheduler thread operation

Figure 13.7 eCos Scheduler Options

chronization/communication mechanisms that are common in real-time systems, namely event flags and mailboxes. Finally, the eCos kernel supports spinlocks, which are useful in SMP (symmetric multiprocessing) systems.

Mutexes The mutex (mutual exclusion lock) was introduced in Chapter 6. Recall that a mutex is used to enforce mutually exclusive access to a resource, allowing only one thread at a time to gain access. The mutex has only two states: locked and unlocked. This is similar to a binary semaphore: When a mutex is locked by one thread, any other thread attempting to lock the mutex is blocked; when the mutex is unlocked, then one of the threads blocked on this mutex is unblocked and allowed to lock the mutex and gain access to the resource.

The mutex differs from a binary semaphore in two respects. First, the thread that locks the mutex must be the one to unlock it. In contrast, it is possible for one thread to lock a binary semaphore and for another to unlock it. The other difference is that a mutex provides protection against priority inversion, whereas a semaphore does not.

The eCos kernel can be configured to support either a priority inheritance protocol or a priority ceiling protocol. These are described in Chapter 10.

Semaphores The eCos kernel provides support for a counting semaphore. Recall from Chapter 5 that a counting semaphore is an integer value used for signaling among threads. The `cyg_semaphore_post` command increments the semaphore count. If the new count is less than or equal to zero, then a thread is waiting on this semaphore and is awakened. The `cyg_semaphore_wait` function checks the value of a semaphore count. If the count is zero, the thread calling this function will wait for the semaphore. If the count is nonzero, the count is decremented and the thread continues.

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

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

Condition Variables A condition variable is used to block a thread until a particular condition is true. Condition variables are used with mutexes to allow multiple thread to access shared data. They can be used to implement monitors of the type discussed in Chapter 6 (e.g., Figure 6.14). The basic commands are as follows:

`cyg_cond_wait`	Causes the current threat to wait on the specified condition variable and simultaneously unlocks the mutex attached to the condition variable
`cyg_cond_signal`	Wakes up one of the threads waiting on this condition variable, causing that thread to become the owner of the mutex
`cyg_cond_broadcast`	Wakes up one of the threads waiting on this condition variable, causing that thread to become the owner of the mutex

In eCos, condition variables are typically used in conjunction with mutexes to implement long-term waits for some condition to become true. We use an example from [ECOS07] to illustrate. Figure 13.8 defines a set of functions to control access to a pool of resources using mutexes. The mutex is used to make the allocation and

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

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

res_t res_allocate(void)
{
      res_t res;

      cyg_mutex_lock(&res_lock);                    //   lock the mutex

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

      cyg_mutex_unlock(&res_lock);                  //   unlock the mutex

      return res;
}

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

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

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

Figure 13.8 **Controlling Access to a Pool of Resources Using Mutexes**

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

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

res_t res_allocate(void)
{
      res_t res;

      cyg_mutex_lock(&res_lock);                   //     lock the mutex

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

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

      cyg_mutex_unlock(&res_lock);                 //     unlock the mutex

      return res;
}

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

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

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

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

Figure 13.9 Controlling Access to a Pool of Resources Using Mutexes and Condition Variables

freeing of resources from a pool atomic. The function `res_t res_allocate` checks to see if one or more units of a resource are available and, if so, takes one unit. This operation is protected by a mutex so that no other thread can check or alter the resource pool while this thread has control of the mutex. The function `res_free(res_t res)` enables a thread to release one unit of a resource that it had previously acquired. Again, this operation is made atomic by a mutex.

In this example, if a thread attempts to access a resource and none are available, the function returns `RES_NONE`. Suppose, however, that we want the thread to be blocked and wait for a resource to become available, rather than retuning `RES_NONE`. Figure 13.9 accomplishes this with the use of a condition variable associated with the mutex. When `res_allocate` detects that there are no resources, it calls `cyg_cond_wait`. This latter function unlocks the mutex and puts the calling thread to sleep on the condition variable. When `res_free` is eventually called, it puts a resource back into the pool and calls `cyg_cond_signal` to wake up any thread waiting on the condition variable. When the waiting thread eventually gets to run again, it will relock the mutex before returning from `cyg_cond_wait`.

[ECOS07] points out two significant features of this example, and of the use of condition variables in general. First, the mutex unlock and wait in `cyg_cond_wait` are atomic: No other thread can run between the unlock and the wait. If this were not the case, then a call to `res_free` by some other thread would release the resource, but the call to `cyg_cond_signal` would be lost, and the first thread would end up waiting when there were resources available.

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

Event Flags An event flag is a 32-bit word used as a synchronization mechanism. Application code may associate a different event with each bit in a flag. A thread can wait for either a single event or a combination of events by checking one or multiple bits in the corresponding flag. The thread is blocked until all of the required bits are set (AND) or until at least one of the bits is set (OR). A signaling thread can set or reset bits based on specific conditions or events so that the appropriate thread is unblocked. For example, bit 0 could represent completion of a specific I/O operation, making data available, and bit 1 could indicate that the user has pressed a start button. A producer thread or DSR could set these two bits, and a consumer thread waiting on these two events will be woken up.

A thread can wait on one or more events using the `cyg_flag_wait` command, which takes three arguments: a particular event flag, a combination of bit positions in the flag, and a mode parameter. The mode parameter specifies whether the thread will block until all the bits are set (AND) or until at least one of the bits is set (OR). The mode parameter may also specify that when the wait succeeds, the entire event flag is cleared (set to all zeros).

Mailboxes Mailboxes, also called message boxes, are an eCos synchronization mechanism that provides a means for two threads to exchange information. Section 5.5 provides a general discussion of message-passing synchronization. Here, we look at the specifics of the eCos version.

The eCos mailbox mechanism can be configured for blocking or nonblocking on both the send and receive side. The maximum size of the message queue associated with a given mailbox can also be configured.

The send message primitive, called put, includes two arguments: a handle to the mailbox, and a pointer for the message itself. There are three variants to this primitive:

`cyg_mbox_put`	If there is a spare slot in the mailbox, then the new message is placed there; if there is a waiting thread, it will be woken up so that it can receive

	the message. If the mailbox is currently full, `cyg_mbox_put` blocks until there has been a corresponding get operation and a slot is available.
`cyg_mbox_timed_put`	Same as `cyg_mbox_put` if there is a spare slot. Otherwise, the function will wait a specified time limit and place the message if a slot becomes available. If the time limit expires, the operation returns `false`. Thus, `cyg_mbox_timed_put` is blocking only for less than or equal to a specified time interval.
`cyg_mbox_tryput`	This is a nonblocking version, which returns `true` if the message is sent successfully and `false` if the mailbox is full.

Similarly, there are three variants to the get primitive.

`cyg_mbox_get`	If there is a pending message in the specified mailbox, `cyg_mbox_get` returns with the message that was put into the mailbox. Otherwise, this function blocks until there is a put operation.
`cyg_mbox_timed_get`	Immediately returns a message if one is available. Otherwise, the function will wait until either a message is available or until a number of clock ticks have occurrred.. If the time limit expires, the operation returns a null pointer. Thus, `cyg_mbox_timed_get` is blocking only for less than or equal to a specified time interval.
`cyg_mbox_tryget`	This is a nonblocking version, which returns', message if one is available and a null pointer if the mailbox is empty.

Spinlocks A spinlock is a flag that a thread can check before executing a particular piece of code. Recall from our discussion of Linux spinlocks in Chapter 6 the basic operation of the spinlock: Only one thread at a time can acquire a spinlock. Any other thread attempting to acquire the same lock will keep trying (spinning) until it can acquire the lock. In essence, a spinlock is built on an integer location in memory that is checked by each thread before it enters its critical section. If the value is 0, the thread sets the value to 1 and enters its critical section. If the value is nonzero, the thread continually checks the value until it is zero.

A spinlock is should not be used on a single-processor system, which is why it is compiled away on Linux. As an example of the danger, consider a uniprocessor system with preemptive scheduling, in which a higher-priority thread attempts to acquire a spinlock already held by a lower priority thread. The lower-priority thread cannot execute so as to finish its work and release the spinlock, because the higher-priority thread preempts it. The higher-priority thread can execute but is stuck checking the spinlock. On an SMP system, the current owner of a spinlock can continue running on a different processor.

13.4 TINYOS

The eCos system provides a more streamlined approach for an embedded OS than one based on a commercial general-purpose OS, such as an embedded version of Linux. Thus, eCos and similar systems are better suited for small embedded systems with tight requirements on memory, processing time, real-time response, power consumption, and so on. TinyOS takes the process of streamlining to a much further point, resulting in a very minimal OS for embedded systems. The core OS requires 400 bytes of code and data memory, combined.

TinyOS represents a significant departure from other embedded operating systems. One striking difference is that TinyOS is not a real-time OS. The reason for this is the expected workload, which is in the context of a wireless sensor network, as described in the next subsection. Because of power consumption, these devices are off most of the time. Applications tend to be simple, with processor contention not much of an issue.

Additionally, in TinyOS there is no kernel, as there is no memory protection and it is a component-based OS; there are no processes; the OS itself does not have a memory allocation system (although some rarely used components do introduce one); interrupt and exception handling is dependent on the peripheral; and it is completely nonblocking, so there are few explicit synchronization primitives.

TinyOS has become a popular approach to implementing wireless sensor network software. Currently, over 500 organizations are developing and contributing to an open source standard for Tiny OS.

Wireless Sensor Networks

TinyOS was developed primarily for use with networks of small wireless sensors. A number of trends have enabled the development of extremely compact, low-power sensors. The well-known Moore's law continues to drive down the size of memory and processing logic elements. Smaller size in turn reduces power consumption. Low power and small size trends are also evident in wireless communications hardware, micro-electromechanical sensors (MEMS), and transducers. As a result, it is possible to develop an entire sensor complete with logic in a cubic millimeter. The application and system software must be compact enough that sensing, communication, and computation capabilities can be incorporated into a complete, but tiny, architecture.

Low cost, small size, low-power-consuming wireless sensors make can be used in a host of applications [ROME04]. Figure 13.10 shows a typical configuration. A base station connects the sensor network to a host PC and passes on sensor data from the network to the host PC, which can do data analysis and/or transmit the data over a corporate network or Internet to an analysis server. Individual sensors collect data and transmit these to the base station, either directly or through sensors that act as data relays. Routing functionality is needed to determine how to relay the data through the sensor network to the base station. [BUON01] points out that, in many applications, the user will want to be able to quickly deploy a large number of low-cost devices without having to configure or manage them. This means that they must be capable of assembling themselves into an ad hoc network. The mobility of individual sensors and the presence of RF interference means that the network will have to be capable of reconfiguring itself in a matter of seconds.

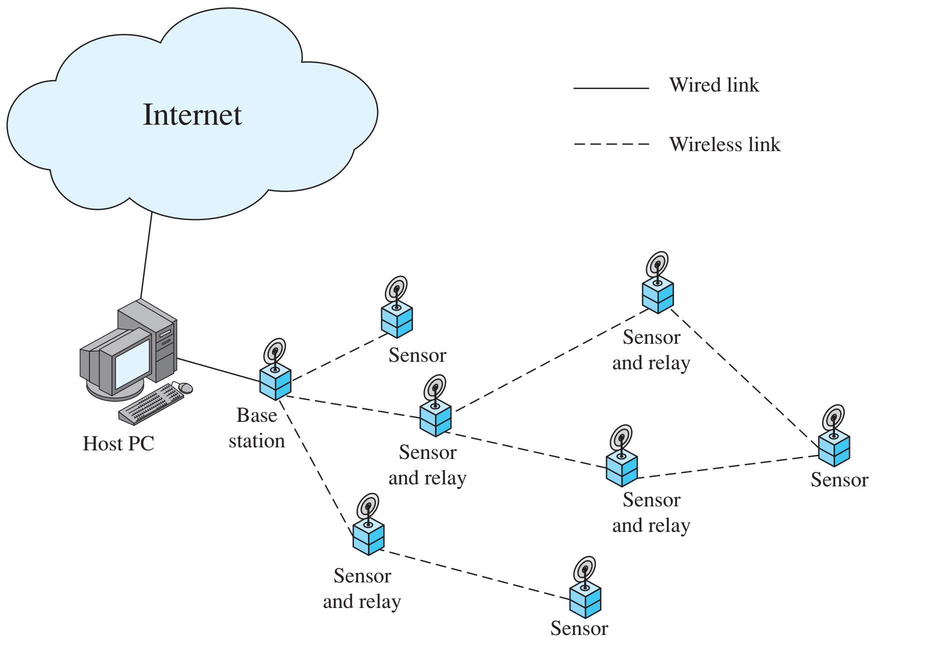


Figure 13.10 **Typical Wireless Sensor Network Topology**

TinyOS Goals

With the tiny, distributed sensor application in mind, a group for researchers from UC Berkeley [HILL00] set the following goals for TinyOS:

- **Allow high concurrency:** In a typical wireless sensor network application, the devices are concurrency intensive. Several different flows of data must be kept moving simultaneously. While sensor data is input in a steady stream, processed results must be transmitted in a steady stream. In addition, external controls from remote sensors or base stations must be managed.
- **Operate with limited resources:** The target platform for TinyOS will have limited memory and computational resources and run on batteries or solar power. A single platform may offer only kilobytes of program memory and hundreds of bytes of RAM. The software must make efficient use of the available processor and memory resources while enabling low-power communication.
- **Adapt to hardware evolution:** Mote hardware is in constant evolution; applications and most system services must be portable across hardware generations. Thus, it should be possible to upgrade the hardware with little or no software change, if the functionality is the same.
- **Support a wide range of applications:** Applications exhibit a wide range of requirements in terms of lifetime, communication, sensing, and so on. A modular, general-purpose embedded OS is desired so that a standardized approach leads to economies of scale in developing applications and support software.
- **Support a diverse set of platforms:** As with the preceding point, a general-purpose embedded OS is desirable.

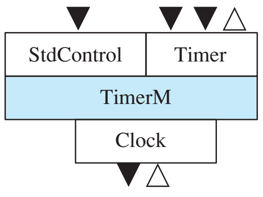
- **Be robust:** Once deployed, a sensor network must run unattended for months or years. Ideally, there should be redundancy both within a single system and across the network of sensors. However, both types of redundancy require additional resources. One software characteristic that can improve robustness is to use highly modular, standardized software components.

It is worth elaborating on the concurrency requirement. In a typical application, there will be dozens, hundreds, or even thousands of sensors networked together. Usually, little buffering is done, because of latency issues. For example, if you are sampling every 5 minutes and want to buffer 4 samples before sending, the average latency is 10 minutes. Thus, information is typically captured, processed, and streamed onto the network in a continuous flow. Further, if the sensor sampling produces a significant amount of data, the limited memory space available limits the number of samples that could be buffered. Even so, in some applications, each of the flows may involve a large number of low-level events interleaved with higher-level processing. Some of the high-level processing will extend over multiple real-time events. Further, sensors in a network, because of the low power of transmission available, typically operate over a short physical range. Thus data from outlying sensors must be relayed to one or more base stations by intermediate nodes.

TinyOS Components

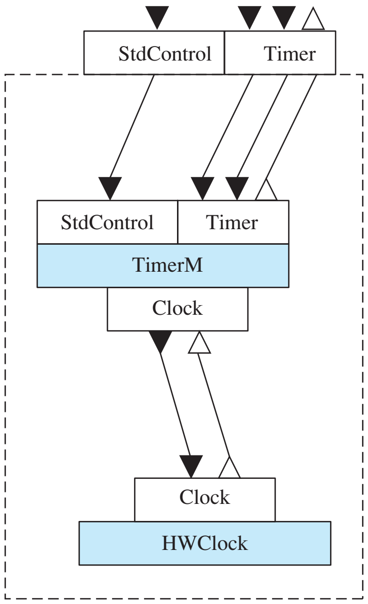
An embedded software system built using TinyOS consists of a set of small modules, called components, each of which performs a simple task or set of tasks and which interface with each other and with hardware in limited and well-defined ways. The only other software module is the scheduler, discussed subsequently. In fact, because there is no kernel, there is no actual OS. But we can take the following view. The application area of interest is the wireless sensor network (WSN). To meet the demanding software requirements of this application, a rigid, simplified software architecture is dictated, consisting of components. The TinyOS development community has implemented a number of open-source components that provide the basic functions needed for the WSN application. Examples of such standardized components include single-hop networking, ad-hoc routing, power management, timers, and nonvolatile storage control. For specific configurations and applications, users build additional special-purpose components and link and load all of the components needed for the user's application. TinyOS, then, consists of a suite of standardized components. Some but not all of these components are used, together with application-specific user-written components, for any given implementation. The OS for that implementation is simply the set of standardized components from the TinyOS suite.

All components in a TinyOS configuration have the same structure, an example of which is shown in Figure 13.11a. The shaded box in the diagram indicates the component, which is treated as an object that can only be accessed by defined interfaces, indicated by white boxes. A component may be hardware or software. Software components are implemented in nesC, which is an extension of C with two distinguishing features: a programming model where components interact via interfaces, and an event-based concurrency model with run-to-completion task and interrupt handlers, explained subsequently.



```
module TimerM {
   provides {
      interface StdControl;
      interface Timer;
   }
   uses interface Clock as Clk;
} ...
```

(a) TimerM component



```
configuration TimerC {
   provides {
      interface StdControl;
      interface Timer;
   }
}

implementation {
   components TimerM, HWClock;
   StdControl = TimerM.StdControl;
   Timer = TimerM.Timer;
   TimerM.Clk -> HWClock.Clock;
}
```

(b) TimerC configuration

Figure 13.11 Example Component and Configuration

The architecture consists of a layered arrangement of components. Each component can link to only two other components, one below it in the hierarchy and one above it. A component issues commands to its lower-level component and receives event signals from it. Similarly, the component accepts commands from its upper-level component and issues event signals to it. At the bottom of the hierarchy are hardware components and at the top of the hierarchy are application components, which may not be part of the standardized TinyOS suite but which must conform to the TinyOS component structure.

A software component implements one or more tasks. Each **task** in a component is similar to a thread in an ordinary OS, with certain limitations. Within a component, tasks are atomic: Once a task has started, it runs to completion. It cannot be preempted by another task in the same component, and there is no timeslicing. However, a task can be preempted by an event. A task cannot block or spin wait. These limitations greatly simplify the scheduling and management of tasks within a component. There is only a single stack, assigned to the currently running task. Tasks can perform computations, call lower-level components (commands) and signal higher-level events, and schedule other tasks.

Commands are nonblocking requests. That is, a task that issues a command does not block or spin wait for a reply from the lower-level component. A command is typically a request for the lower-level component to perform some service, such as initiating a sensor reading. The effect on the component that receives the command is specific to the command given and the task required to satisfy the command. Generally, when a command is received, a task is scheduled for later execution, because a command cannot preempt the currently running task. The command returns immediately to the calling component; at a later time, an event will signal completion to the calling component. Thus, a command does not cause a preemption in the called component and does not cause blocking in the calling component.

Events in TinyOS may be tied either directly or indirectly to hardware events. The lowest level software components interface directly to hardware interrupts, which may be external interrupts, timer events, or counter events. An event handler in a lowest level component may handle the interrupt itself or may propagate event messages up through the component hierarchy. A command can post a task that will signal an event in the future. In this case, there is no tie of any kind to a hardware event.

A task can be viewed as having three phases. A caller posts a command to a module. The module then runs the requested task. The module then notifies the caller, via an event, that the task is complete.

The component depicted in Figure 13.11a, TimerM, is part of the TinyOS timer service. This component **provides** the StdControl and Timer interface and **uses** a Clock interface. Providers implement commands (i.e., the logic in this component). Users implement events (i.e., external to the component). Many TinyOS components use the StdControl interface to be initialized, started, or stopped. TimerM provides the logic that maps from a hardware clock into TinyOS's timer abstraction. The timer abstraction can be used for counting down a given time interval. Figure 13.11a also shows the formal specification of the TimerM interfaces.

The interfaces associated with TimerM are specified as follows:

```
interface StdControl {
    command result_t init();
    command result_t start();
    command result_t stop();
}
interface Timer {
    command result_t start(char type, uint32_t interval);
    command result_t stop();
    event result_t fired();
}
interface Clock {
    command result_t setRate(char interval, char scale);
    event result_t fire();
}
```

Components are organized into configurations by "wiring" the together at their interfaces and equating the interfaces of the configuration with some of the interfaces of the components. A simple example is shown in Figure 13.11b. The uppercase C stands for Component. It is used to distinguish between an interface (e.g., Timer) and a component that provides the interface (e.g., TimerC). The uppercase M stands for Module. This naming convention is used when a single logical component has both a configuration and a module. The TimerC component, providing the Timer interface, is a configuration that links its implementation (TimerM) to Clock and LED providers. Otherwise, any user of TimerC would have to explicitly wire its subcomponents.

TinyOS Scheduler

The TinyOS scheduler operates across all components. Virtually all embedded systems using TinyOS will be uniprocessor systems, so that only one task among all the tasks in all the components may execute at a time. The scheduler is a separate component. It is the one portion of TinyOS that must be present in any system.

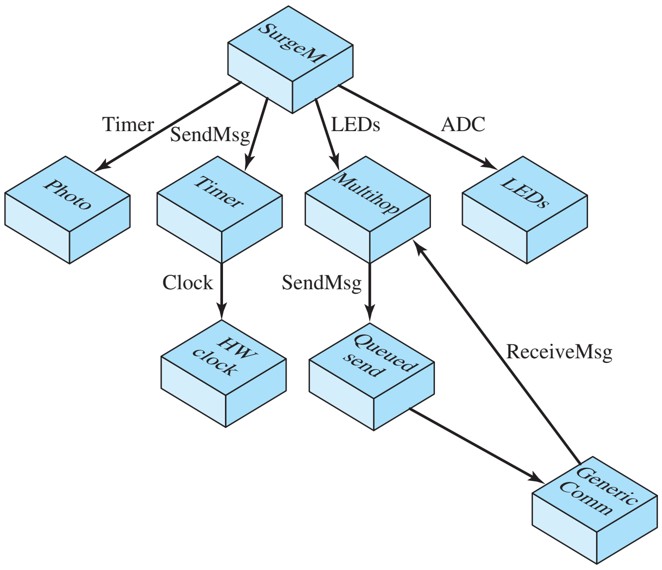
The default scheduler in TinyOS is a simple FIFO (first-in-first-out) queue. A task is posted to the scheduler (place in the queue) either as a result of an event, which triggers the posting, or as a result of a specific request by a running task to schedule another task. The scheduler is power aware. This means that the scheduler puts the processor to sleep when there are no tasks in the queue. The peripherals remain operating, so that one of them can wake up the system by means of a hardware event signaled to a lowest-level component. Once the queue is empty, another task can be scheduled only as a result of a direct hardware event. This behavior enables efficient battery usage.

The scheduler has gone through two generations. In TinyOS 1.x, there is a shared task queue for all tasks, and a component can post a task to the scheduler multiple times. If the task queue is full, the post operation fails. Experience with networking stacks showed this to be problematic, as the task might signal completion of a split-phase operation: If the post fails, the component above might block forever, waiting for the completion event. In TinyOS 2.x, every task has its own reserved slot in the task queue, and a task can only be posted once. A post fails if and only if the task has already been posted. If a component needs to post a task multiple times, it can set an internal state variable so that when the task executes, it reposts itself. This slight change in semantics greatly simplifies a lot of component code. Rather than test to see if a task is posted already before posting it, a component can just post the task. Components do not have to try to recover from failed posts and retry. The cost is one byte of state per task.

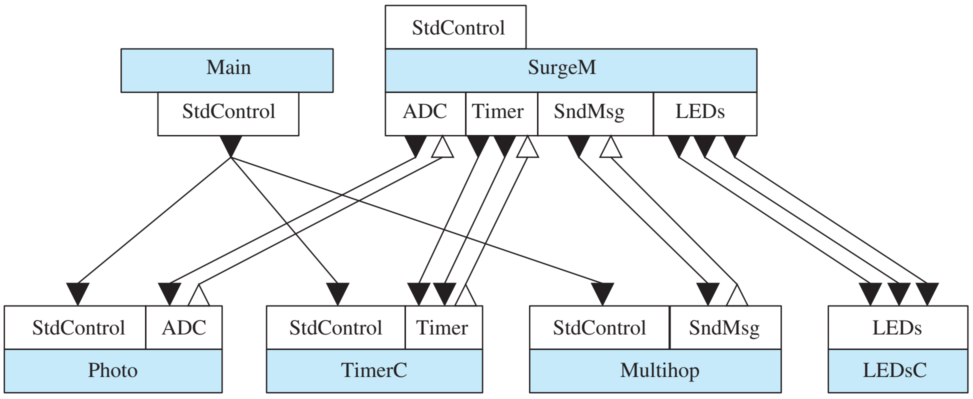
A user can replace the default scheduler with one that uses a different dispatching scheme, such as a priority-based scheme or a deadline scheme. However, preemption and timeslicing should not be used because of the overhead such systems generate. More importantly, they violate the TinyOS concurrency model, which assumes tasks do not preempt each other.

Example Configuration

Figure 13.12 shows a configuration assembled from software and hardware components. This simplified example, called Surge and described in [GAY03], performs periodic sensor sampling and uses ad-hoc multihop routing over the wireless network to deliver samples to the base station. The upper part of the figure shows the components of Surge (represented by boxes) and the interfaces by which they are wired (represented by arrowed lines). The SurgeM component is the application-level component that orchestrates the operation of the configuration.



(a) Simplified view of the Surge Application



(b) Top-level surge configuration

LED = light-emitting diode
ADC = analog-to-digital converter

Figure 13.12 Examples TinyOS Application

Figure 13.12b shows a portion of the configuration for the Surge application. The following is a simplified excerpt from the SurgeM specification.

```
module SurgeM {
    provides interface StdControl;
    uses interface ADC;
    uses interface Timer;
    uses interface SendMsg;
    uses interface LEDs;
}
```

```
implementation {
    uint16_t sensorReading;
    command result_t StdControl.init() {
    return call Timer.start(TIMER_REPEAT, 1000);
    }
    event result_t Timer.fired() {
    call ADC.getData();
    return SUCCESS;
    }
    event result_t ADC.dataReady(uint16_t data) {
    sensorReading = data;
    ... send message with data in it ...
    return SUCCESS;
    }
    ...
}
```

This example illustrates the strength of the TinyOS approach. The software is organized as an interconnected set of simple modules, each of which defines one or a few tasks. Components have simple, standardized interfaces to other components, be they hardware or software. Thus, components can easily be replaced. Components can be hardware or software, with a boundary change not visible to the application programmer.

TinyOS Resource Interface

TinyOS provides a simple but powerful set of conventions for dealing with resources. Three abstractions for resources are used in TinyOS:

- **Dedicated:** A resource that a subsystem needs exclusive access to at all times. In this class of resources, no sharing policy is needed since only a single component ever requires use of the resource. Examples of dedicated abstractions include interrupts and counters.
- **Virtualized:** Every client of a virtualized resource interacts with it as if it were a dedicated resource, with all virtualized instances being multiplexed on top of a single underlying resource. The virtualized abstraction may be used when the underlying resource need not be protected by mutual exclusion. An example is a clock or timer.
- **Shared:** The shared resource abstraction provides access to a dedicated resource through an arbiter component. The arbiter enforces mutual exclusion, allowing only one user (called a client) at a time to have access to a resource and enabling the client to lock the resource.

In the remainder of this subsection, we briefly define the shared resource facility of TinyOS. The arbiter determines which client has access to the resource at which time. While a client holds a resource, it has complete and unfettered control. Arbiters assume that clients are cooperative, only acquiring the resource when needed and holding on to it no longer than necessary. Clients explicitly release resources: There is no way for an arbiter to forcibly reclaim it.

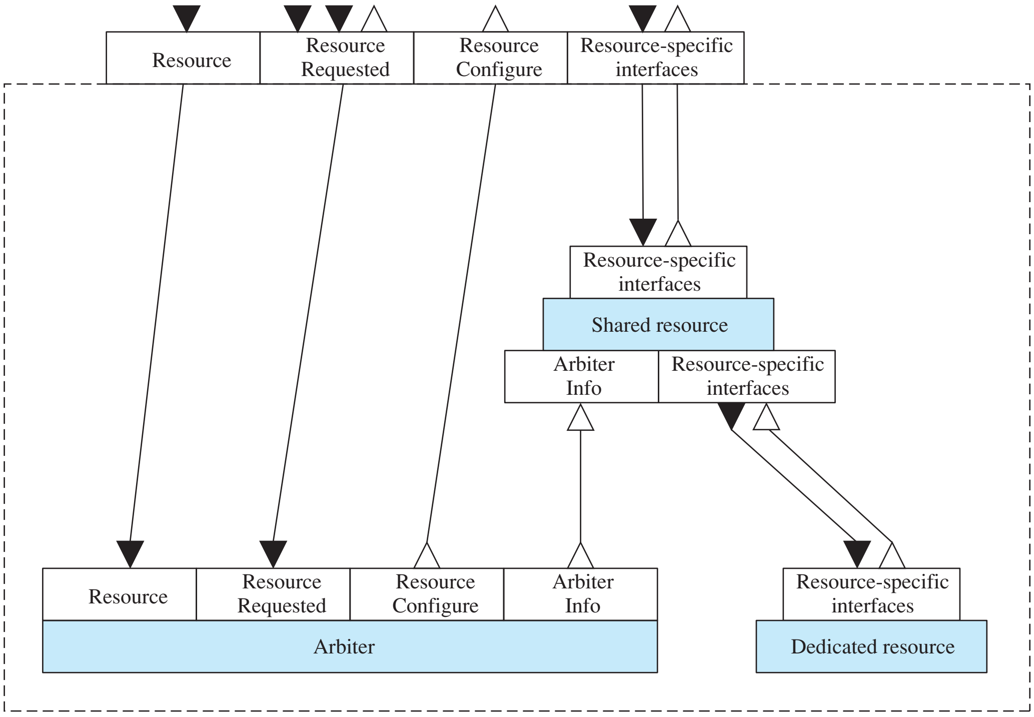


Figure 13.13 **Shared Resource Configuration**

Figure 13.13 shows a simplified view of the shared resource configuration used to provide access to an underlying resource. Associated with each resource to be shared is an arbiter component. The arbiter enforces a policy that enables a client to lock the resource, use it and then release the resource. The shared resource configuration provides the following interfaces to a client:

- **Resource:** The client issues a request at this interface requesting access to the resource. If the resource is currently locked, the arbiter places the request in a queue. When a client is finished with the resource, it issues a release command at this interface.
- **Resource Requested:** This is similar to the Resource interface. In this case, the client is able to hold onto a resource until the client is notified that someone else needs the resource.
- **Resource Configure:** This interface allows a resource to be automatically configured just before a client is granted access to it. Components providing the ResourceConfigure interface use the interfaces provided by an underlying dedicated resource to configure it into one of its desired modes of operation.
- **Resource-specific interfaces:** Once a client has access to a resource, it uses resource-specific interfaces to exchange data and control information with the resource.

In addition to the dedicated resource, the shared resource configuration consists of two components. The Arbiter accepts requests for access and configuration from a client and enforce the lock on the underlying resource. The shared resource component mediates data exchange between the client and the underlying resource.

Arbiter information passed from the arbiter to the shared resource component controls the access of the client to the underlying resource.

13.5 RECOMMENDED READING AND WEB SITES

[KOOP96] provides a systematic discussion of the requirements for embedded systems. [STAN96] is a useful overview of real-time and embedded systems.

[MASS03] and [ECOS07] both provide a detailed description of eCos internals. [THOM01] provides a brief overview with some code examples from the kernel. [LARM05] gives a more detailed description of the eCos configuration process.

[HILL00] gives an overview and design rationale for TinyOS. [GAY05] is an interesting discussion of software design strategies using TinyOS. [BUON01] provides a good example of the use of TinyOS in building a network or wireless sensors. Two excellent references for the current version of TinyOS are [GAY03] and [LEVI05].

BUON01 Buonadonna, P.; Hill, J.; and Culler, D. "Active Message Communication for Tiny Networked Sensors." *Proceedings, IEEE INFOCOM 2001*, April 2001

ECOS07 eCosCentric Limited, and Red Hat, Inc. *eCos Reference Manual.* 2007. http://www.ecoscentric.com/ecospro/doc/html/ref/ecos-ref.html

GAY03 Gay, D., et al. "The nesC Language: A Holistic Approach to Networked Embedded Systems." Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, 2003.

GAY05 Gay, D.; Levis, P.; and Culler, D. "Software Design Patterns for TinyOS." *Proceedings, Conference on Languages, Compilers, and Tools for Embedded Systems*, 2005.

HILL00 Hill, J. , et al. "System Architecture Directions for Networked Sensors." *Proceedings, Architectural Support for Programming Languages and Operating Systems*. 2000.

KOOP96 Koopman, P. "Embedded System Design Issues (the Rest of the Story)." *Proceedings, 1996 International Conference on Computer Design*, 1996.

LARM05 Larmour, J. "How eCos Can Be Shrunk to Fit." *Embedded Systems Europe*, May 2005. www.embedded.com/europe/esemay05.htm

LEVI05 Levis, P., et al. "T2: A Second Generation OS For Embedded Sensor Networks." Technical Report TKN-05-007, Telecommunication Networks Group, Technische Universitat Berlin, 2005. http://csl.stanford.edu/~pal/pubs.html

MASS03 Massa, A. *Embedded Software Development with eCos.* Upper Saddle River, NJ: Prentice Hall, 2003.

STAN96 Stankovic, J., et al. "Strategic Directions in Real-Time and Embedded Systems." *ACM Computing Surveys*, December 1996.

THOM01 Thomas, G. " eCos: An Operating System for Embedded Systems." *Dr. Dobb's Journal*, January 2001.

Recommended Web sites:

- **Embedded.com:** Wide variety of information on embedded systems
- **eCos:** Downloadable software, information, and links on eCos
- **TinyOS Community Forum:** Downloadable software, information, and links on TinyOS

13.6 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

eCos embedded operating system	embedded system	TinyOS

Review Questions

13.1 What is an embedded system?

13.2 What are some typical requirements or constraints on embedded systems?

13.3 What is an embedded OS?

13.4 What are some of the key characteristics of an embedded OS?

13.5 Explain the relative advantages and disadvantages of an embedded OS based on an existing commercial OS compared to a purpose-built embedded OS.

13.6 What are the principal objectives that guided the design of the eCos kernel?

13.7 In eCos, what is the difference between an interrupt service routine and a deferred service routine?

13.8 What concurrency mechanisms are available in eCos?

13.9 What is the target application for TinyOS?

13.10 What are the design goals for TinyOS?

13.11 What is a TinyOS component?

13.12 What software comprises the TinyOS operating system?

13.13 What is the default scheduling discipline for TinyOS?

Problems

13.1 With reference to the device driver interface to the eCos kernel (Table 13.2), it is recommended that device drivers should use the `_intsave()` variants to claim and release spinlocks rather than the non-`_intsave()` variants. Explain why.

13.2 Also in Table 13.2, it is recommended that `cyg_drv_spinlock_spin` should be used sparingly, and in situations where deadlocks/livelocks cannot occur. Explain why.

13.3 In Table 13.2, what should be the limitations on the use of `cyg_drv_spinlock_destroy`? Explain.

13.4 In Table 13.2, what limitations should be placed in the use of `cyg_drv_mutex_destroy`?

13.5 Why does the eCos bitmap scheduler not support timeslicing?

13.6 The implementation of mutexes within the eCos kernel does not support recursive locks. If a thread has locked a mutex and then attempts to lock the mutex again, typically as a result of some recursive call in a complicated call graph, then either an assertion failure will be reported or the thread will deadlock. Suggest a reason for this policy.

13.7 Figure 13.14 is a listing of code intended for use on the eCos kernel.

a. Explain the operation of the code. Assume thread B begins execution first and thread A begins to execute after some event occurs.

b. What would happen if the mutex unlock and wait code execution in the call to cyg_cond_wait, on line 30, were not atomic?

c. Why is the while loop on line 26 needed?

13.8 The discussion of eCos spinlocks included an example showing why spinlocks should not be used on a uniprocessor system if two threads of different priorities can compete for the same spinlock. Explain why the problem still exists even if only threads of the same priority can claim the same spinlock.

13.9 TinyOS's scheduler serves tasks in FIFO order. Many other schedulers for TinyOS have been proposed, but none have caught on. What characteristics of the sensornet domain might cause a lack of need for more complex scheduling?

13.10 **a.** The TinyOS Resource interface does not allow a component that already has a request in the queue for a resource to make a second request. Suggest a reason why.

b. However, the TinyOS Resource interface allows a component holding the resource lock to re-request the lock. This request is enqueued for a later grant. Suggest a reason for this policy. (*Hint:* What might cause there to be latency between one component releasing a lock and the next requester being granted it?)

```
unsigned char buffer_empty = true;
cyg_mutex_t mut_cond_var;
cyg_cond-t cond_var;

void thread_a( cyg_addrword_t index )
{
   while ( 1 )    // run this thread forever
   {
       // acquire data into the buffer ...

       // there is data in the buffer now
       buffer_empty = false;

       cyg_mutex_lock( &mut_cond_var );

       cyg_cond_signal( &cond_var );

       cyg_mutex_unlock( &mut_cond_var );
   }
 }

 void thread_b( cyg_addrword_t index )
 {
    while ( 1 )    // run this thread forever
    {
       cyg_mutex_lock( &mut_cond_var );

       while ( buffer_empty == true )
       {
       cyg_cond_wait( &cond_var );
       }


       // get the buffer data ...

       // set flag to indicate the data in the buffer has been processed
       buffer_empty = true;

       cyg_mutex_unlock( &mut_cond_var );

       // process the data in the buffer
   }
 {
```

Figure 13.14 **Condition Variable Example Code**