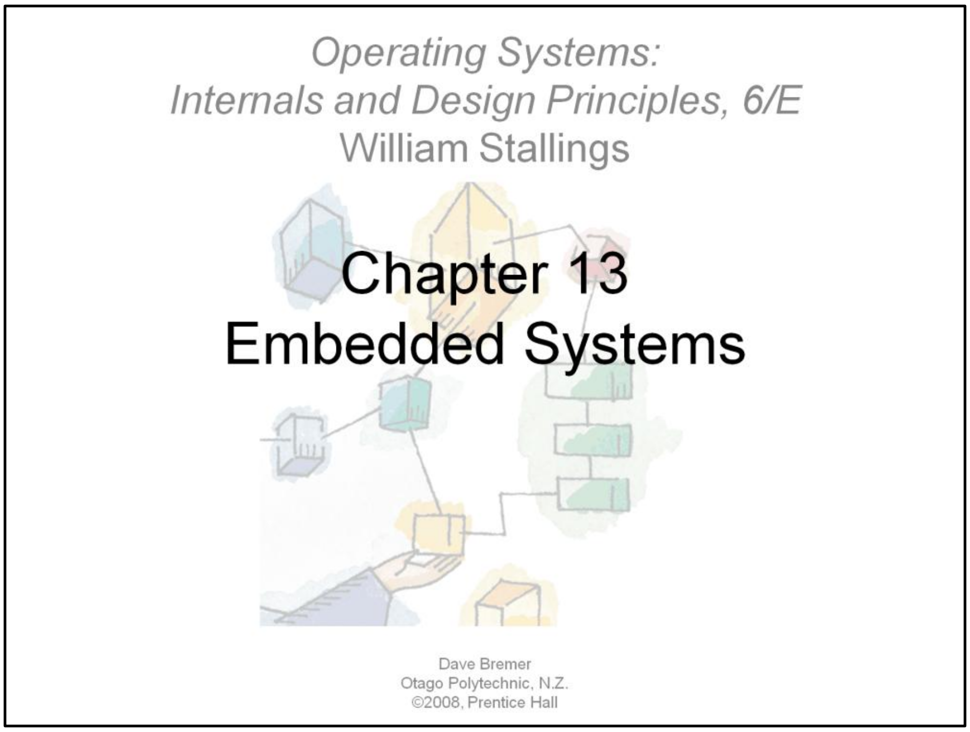
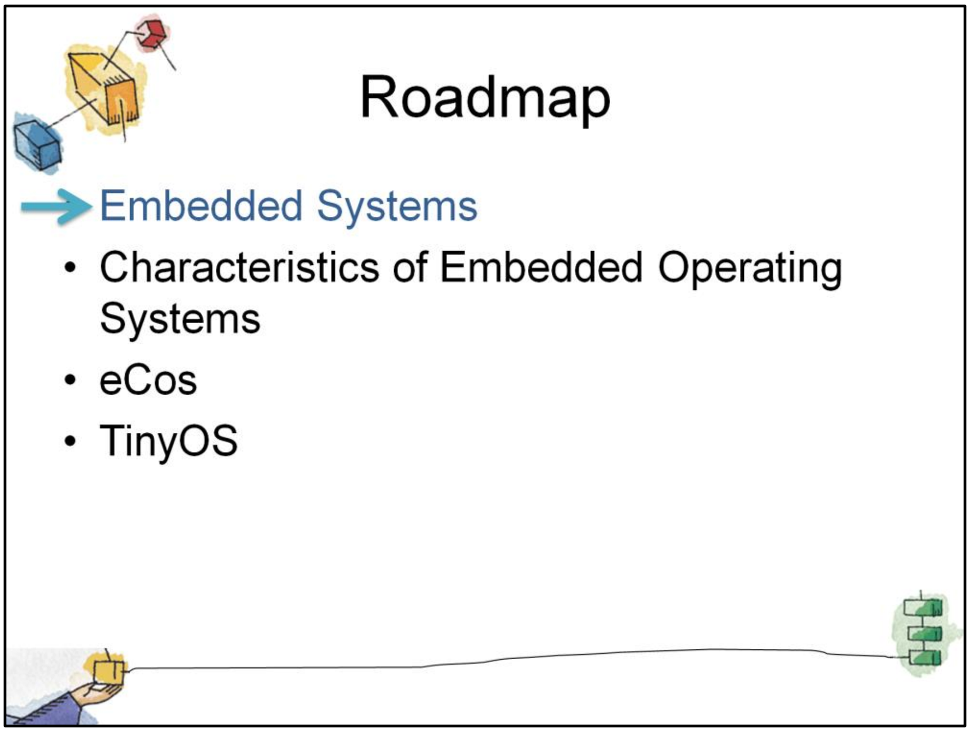


Operating Systems:
Internals and Design Principles, 6/E
William Stallings

Chapter 13 Embedded Systems

Dave Bremer
Otago Polytechnic, N.Z.

These slides are intended to help a teacher develop a presentation. This PowerPoint covers the entire chapter and includes too many slides for a single delivery. Professors are encouraged to adapt this presentation in ways which are best suited for their students and environment.



Roadmap

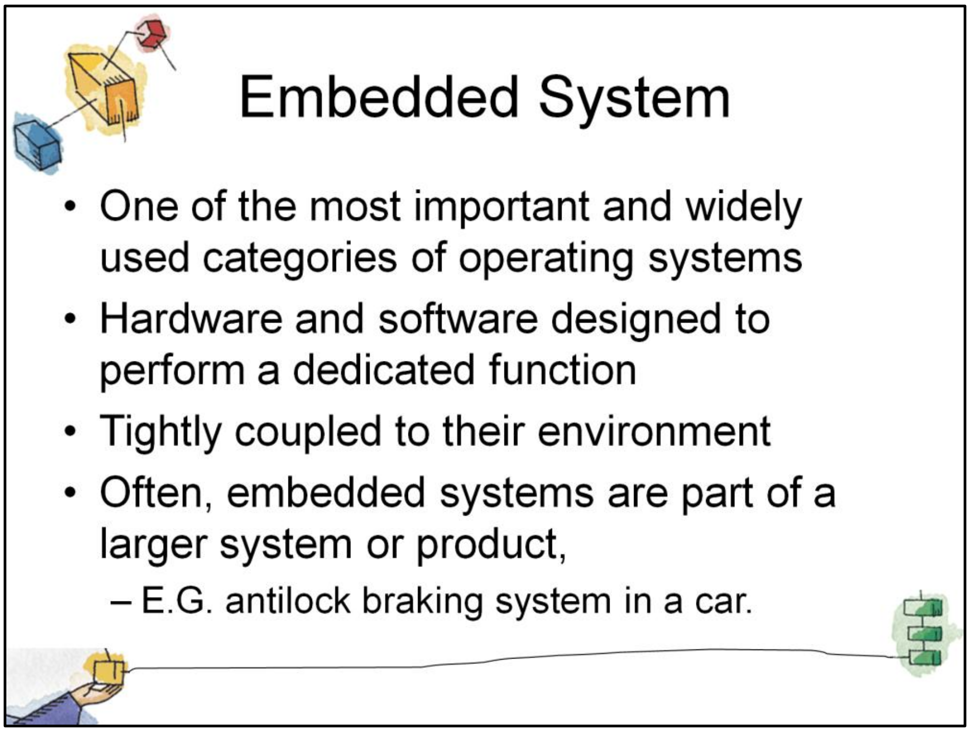
→ Embedded Systems

- Characteristics of Embedded Operating Systems
- eCos
- TinyOS

We begin with an overview, followed by a look at various file organization schemes.

Although file organization is generally beyond the scope of the operating system, it is essential to have a general understanding of the common alternatives to appreciate some of the design tradeoffs involved in file management.

The remainder of this chapter looks at other topics in file management.



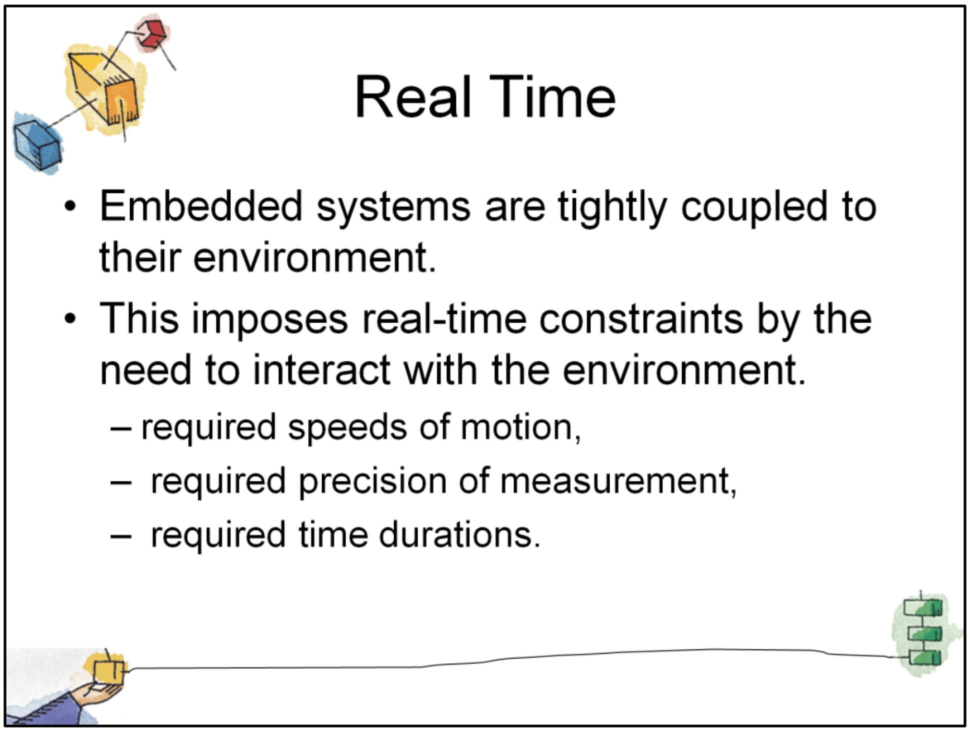
Embedded System

- One of the most important and widely used categories of operating systems
- Hardware and software designed to perform a dedicated function
- Tightly coupled to their environment
- Often, embedded systems are part of a larger system or product,
  - E.G. antilock braking system in a car.

One of the most important and widely used categories of operating systems:

- embedded operating systems.

"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."



Real Time

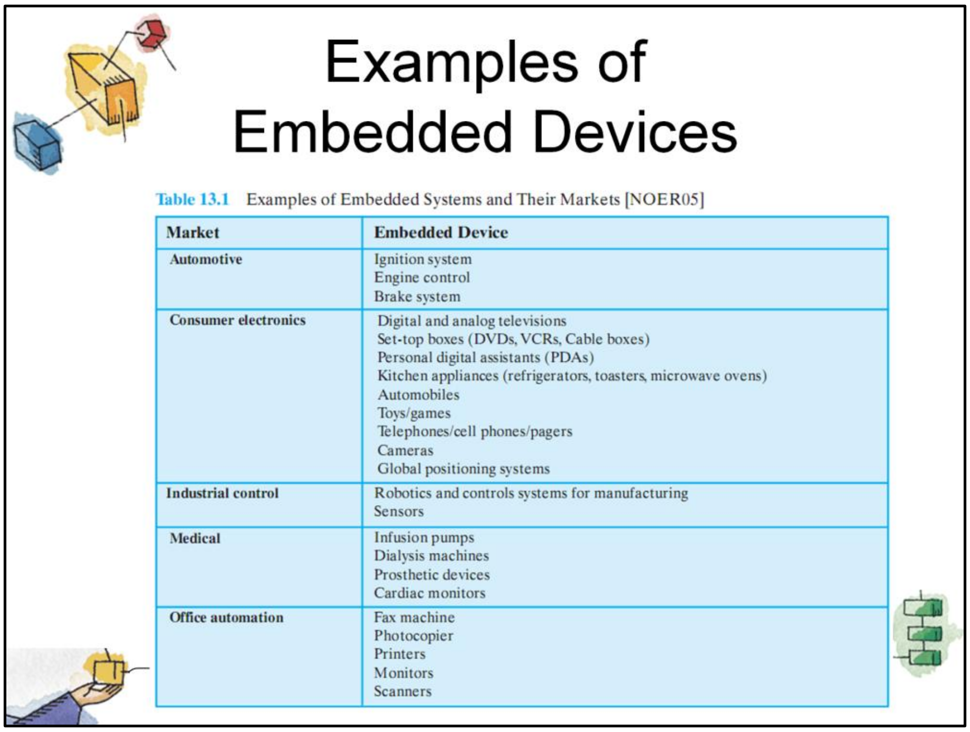
- Embedded systems are tightly coupled to their environment.
- This imposes real-time constraints by the need to interact with the environment.
  - required speeds of motion,
  - required precision of measurement,
  - required time durations.

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

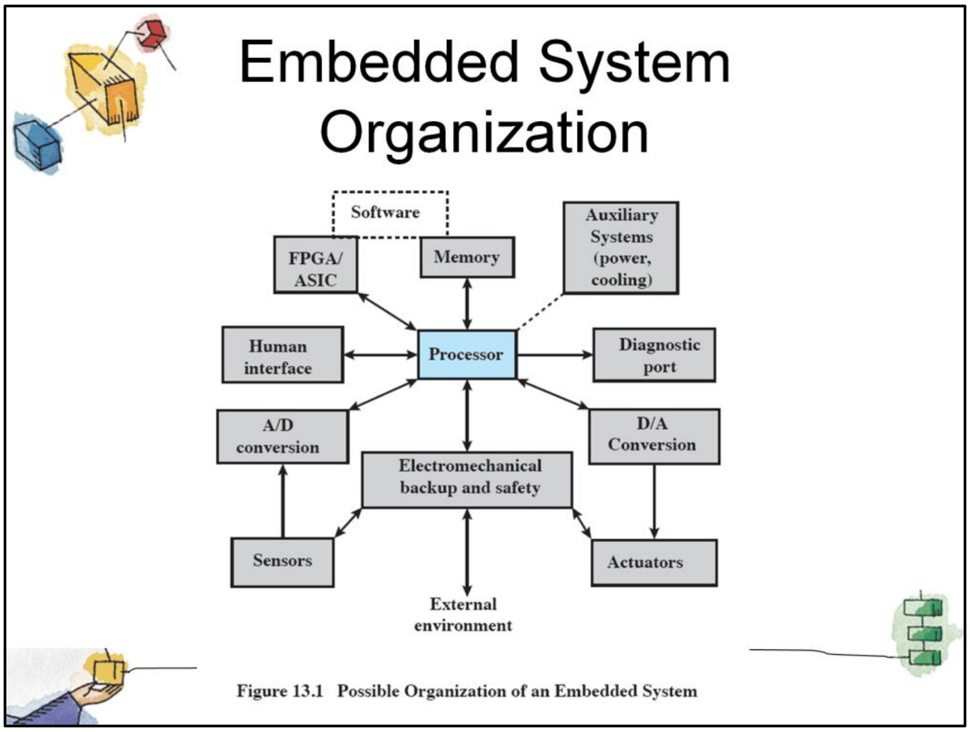
- required speeds of motion,
- required precision of measurement,
- required time durations, dictate the timing of software operations.
- If multiple activities must be managed simultaneously, this imposes more complex real-time constraints.



Examples of Embedded Devices

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



Embedded System Organization

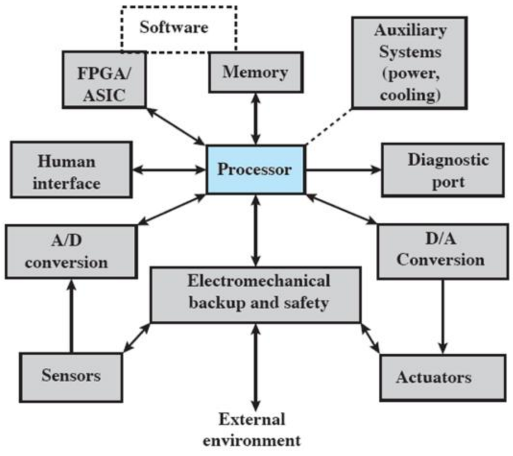
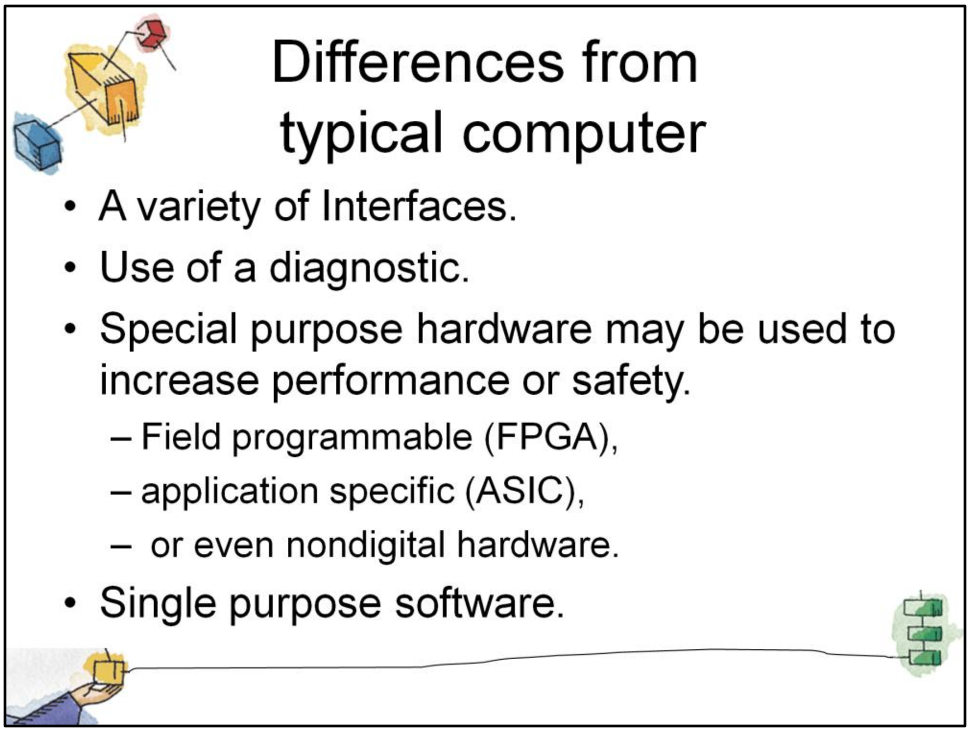


Figure 13.1 Possible Organization of an Embedded System

This figure shows in general terms an embedded system organization.

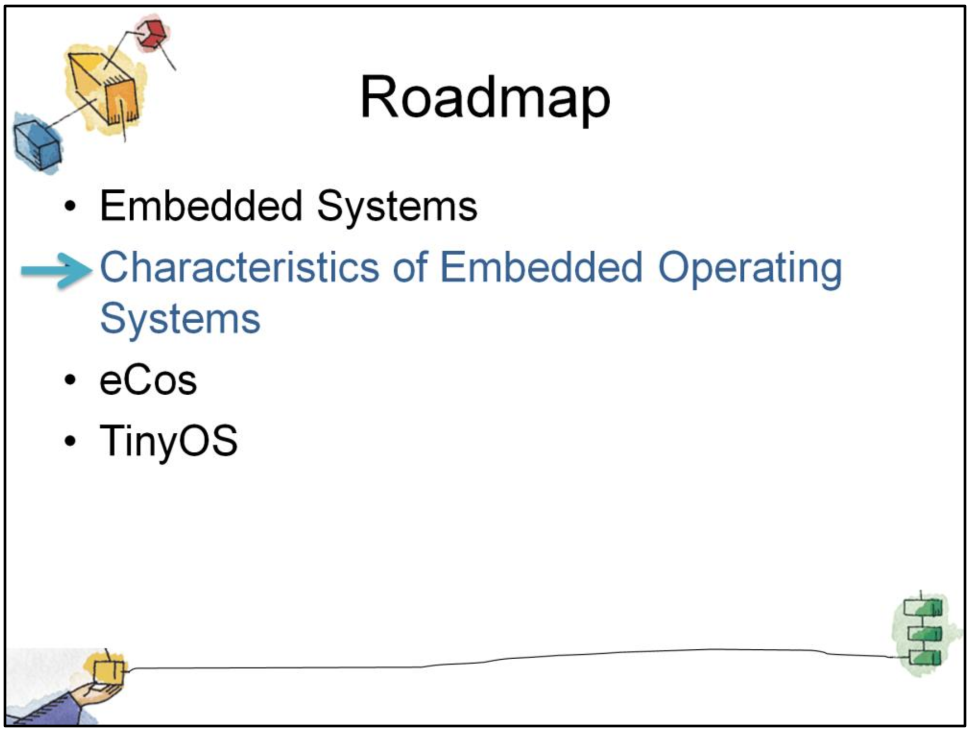


Differences from typical computer

- A variety of Interfaces.
- Use of a diagnostic.
- Special purpose hardware may be used to increase performance or safety.
  - Field programmable (FPGA),
  - application specific (ASIC),
  - or even nondigital hardware.
- Single purpose software.

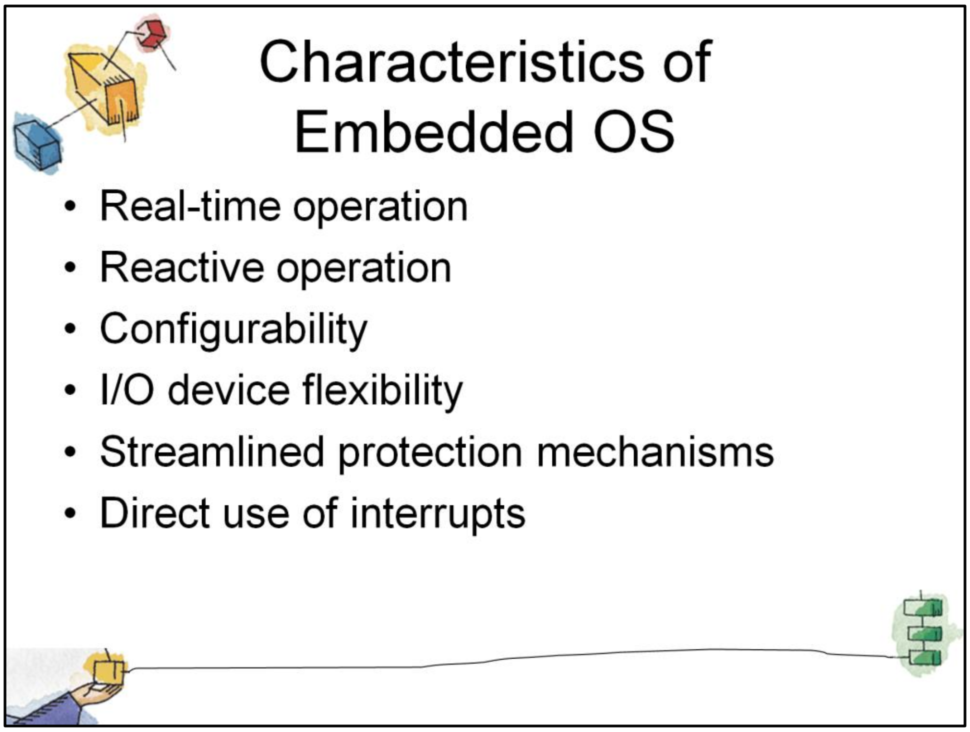
In addition to the processor and memory, there are a number of elements that differ from the typical desktop or laptop computer.

- 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
- 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.



Roadmap

- Embedded Systems
- → Characteristics of Embedded Operating Systems
- eCos
- TinyOS



Characteristics of Embedded OS

- Real-time operation
- Reactive operation
- Configurability
- I/O device flexibility
- Streamlined protection mechanisms
- Direct use of interrupts

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.

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.

Streamlined protection mechanisms:

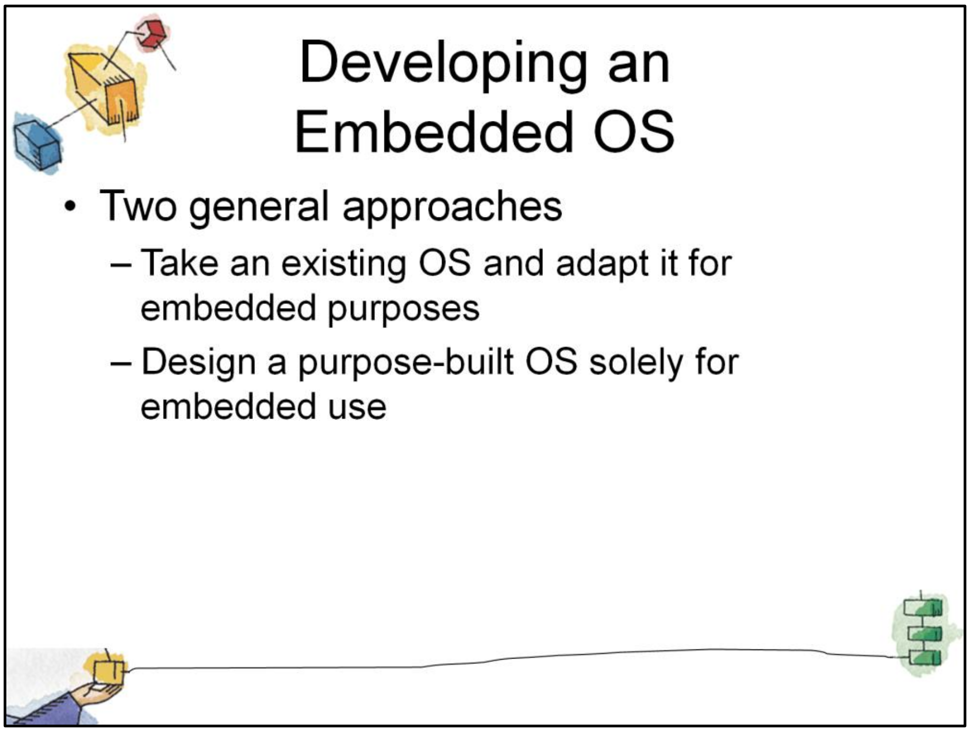
• Embedded systems are typically designed for a limited, well-defined functionality. Untested programs are rarely added to the software.

• 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.

Direct use of interrupts:

• General-purpose operating systems typically do not permit any user process to use interrupts directly.

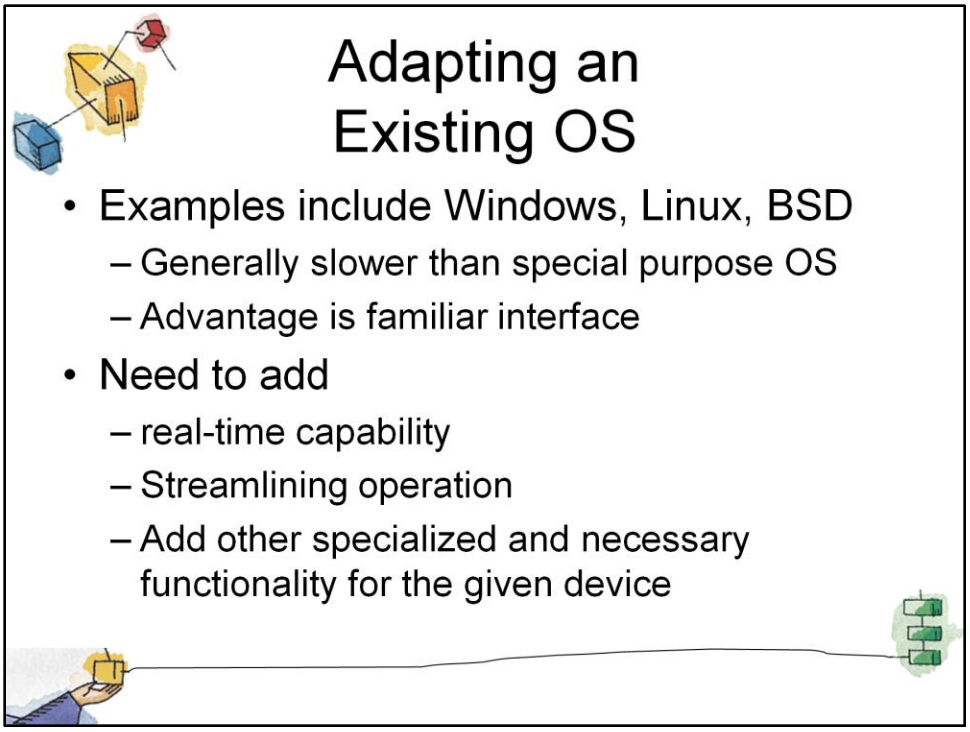


Developing an Embedded OS

- Two general approaches
  - Take an existing OS and adapt it for embedded purposes
  - Design a purpose-built OS solely for embedded use

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



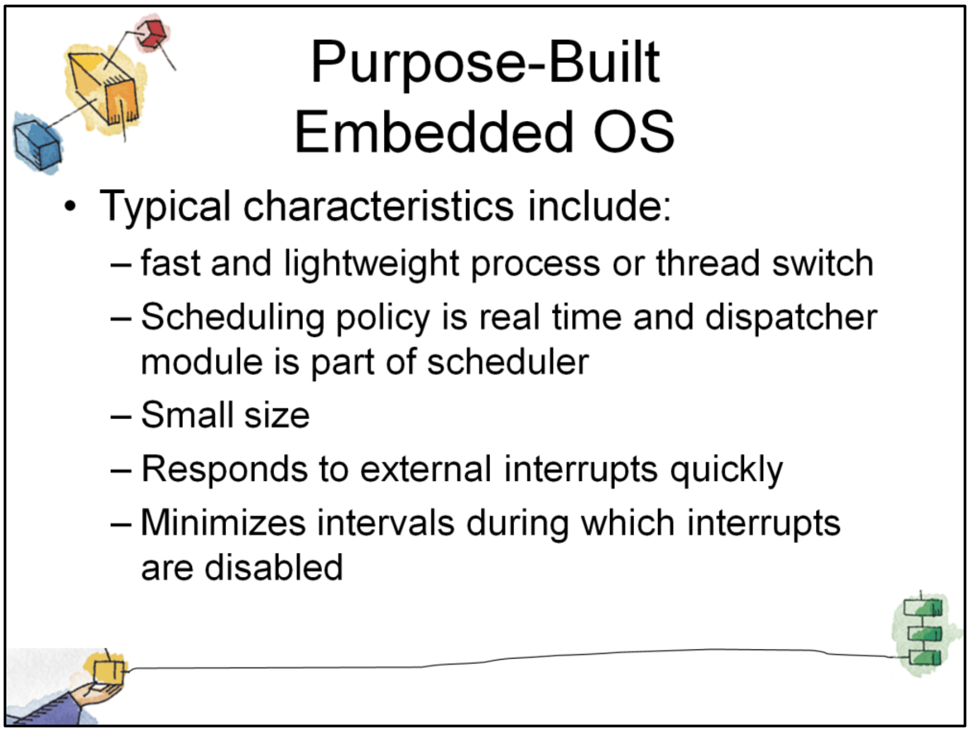
Adapting an Existing OS

- Examples include Windows, Linux, BSD
  - Generally slower than special purpose OS
  - Advantage is familiar interface
- Need to add
  - real-time capability
  - Streamlining operation
  - Add other specialized and necessary functionality for the given device

Typically use Linux, FreeBSD, Windows, and other general-purpose operating systems.

- 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.



Purpose-Built Embedded OS

- Typical characteristics include:
  - fast and lightweight process or thread switch
  - Scheduling policy is real time and dispatcher module is part of scheduler
  - Small size
  - Responds to external interrupts quickly
  - Minimizes intervals during which interrupts are disabled

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.

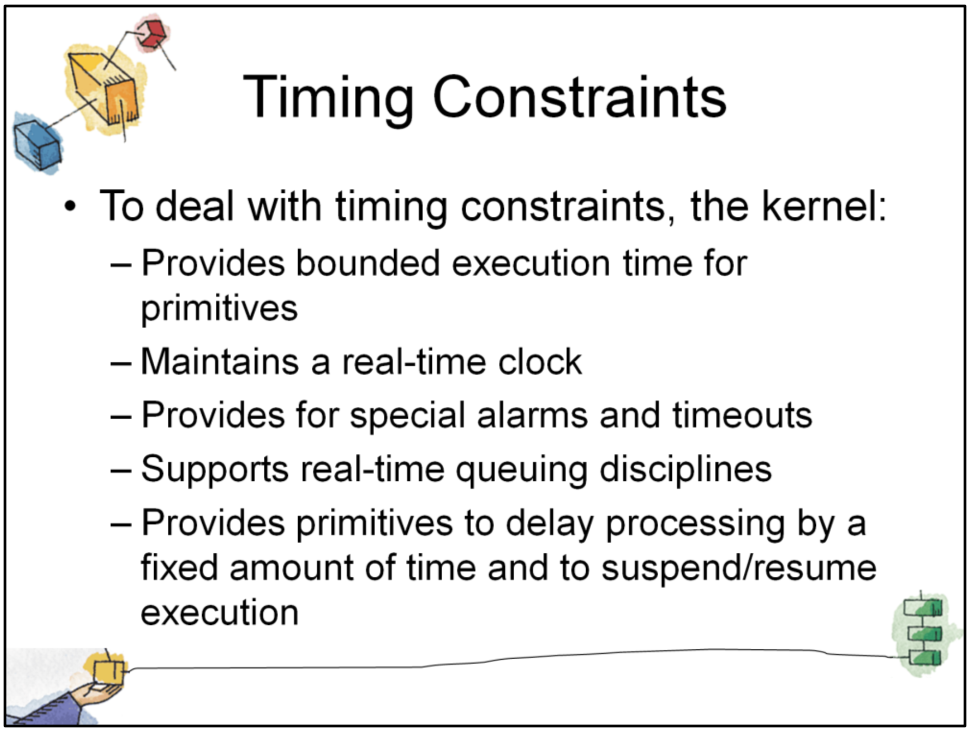
• Has a small size

• 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

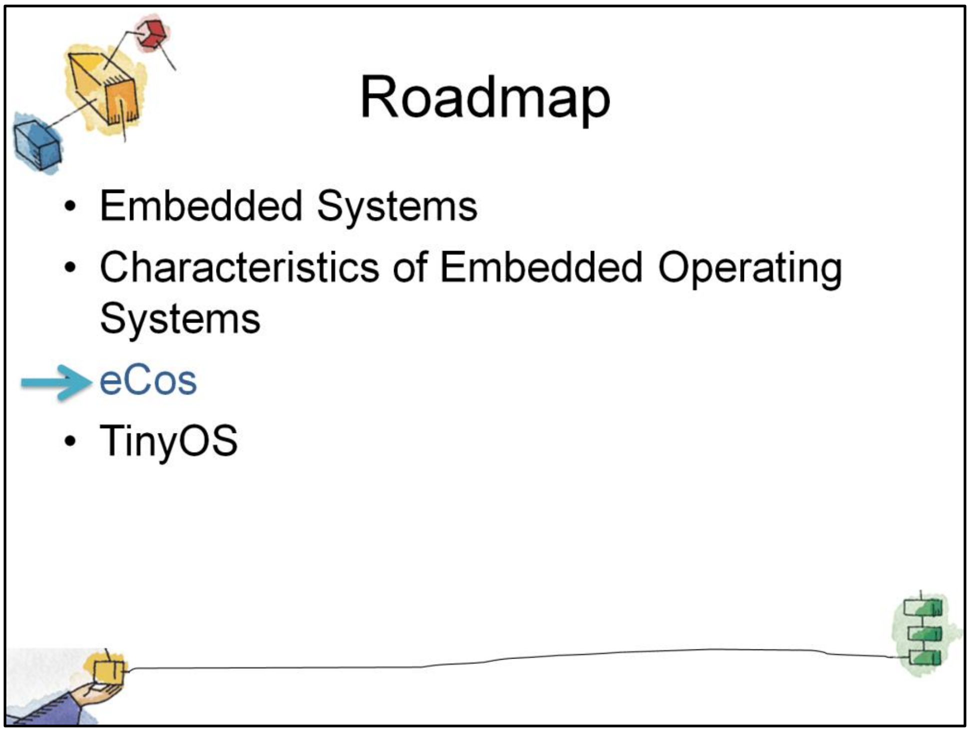


Timing Constraints

- To deal with timing constraints, the kernel:
  - Provides bounded execution time for primitives
  - Maintains a real-time clock
  - Provides for special alarms and timeouts
  - Supports real-time queuing disciplines
  - Provides primitives to delay processing by a fixed amount of time and to suspend/resume execution

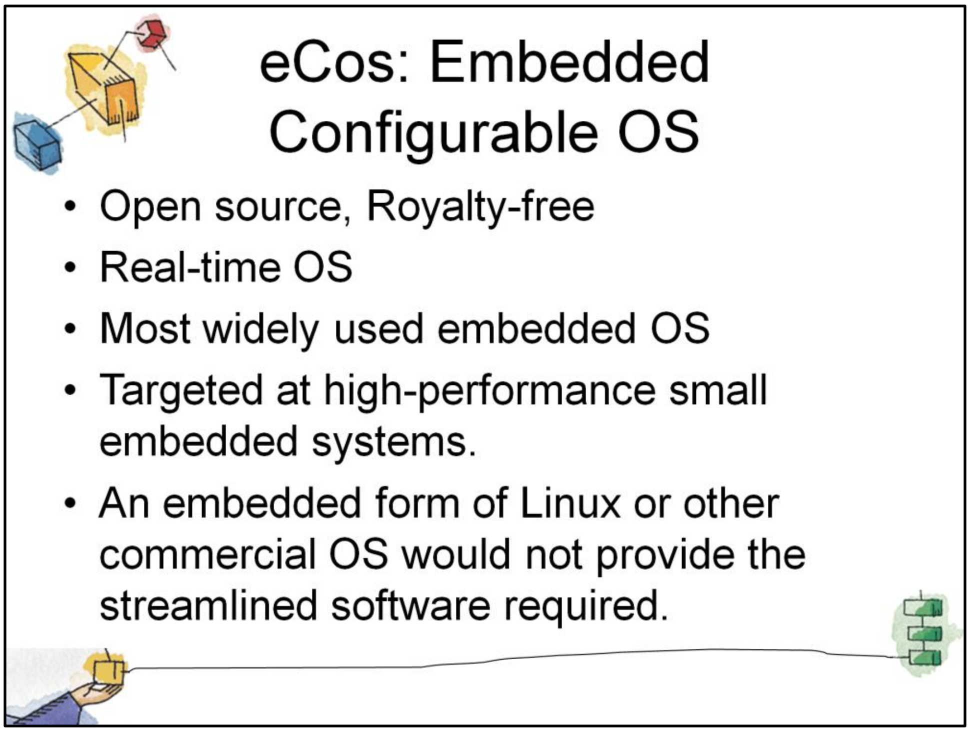
The characteristics just listed are common in embedded operating systems with real-time requirements.

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.



Roadmap

- Embedded Systems
- Characteristics of Embedded Operating Systems
- → eCos
- TinyOS



eCos: Embedded Configurable OS

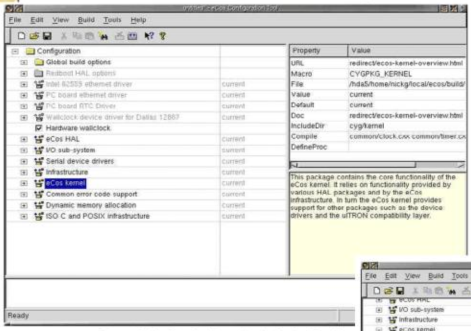
- Open source, Royalty-free
- Real-time OS
- Most widely used embedded OS
- Targeted at high-performance small embedded systems.
- An embedded form of Linux or other commercial OS would not provide the streamlined software required.

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.

eCos Configuration Tool

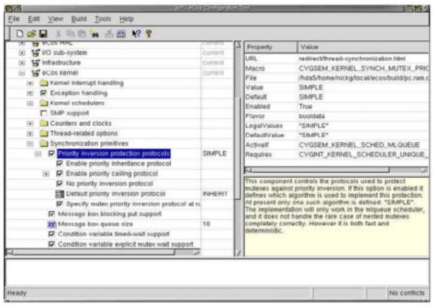


Configuration	
Global build options	
Redboot HAL options	
Intel 82559 ethernet driver	current
PC board ethernet driver	current
PC board RTC Driver	current
Wallclock device driver for Dallas 12887	current
Hardware wallclock	
eCos HAL	current
I/O sub-system	current
Serial device drivers	current
Infrastructure	current
eCos kernel	current
Common error code support	current
Dynamic memory allocation	current
ISO C and POSIX infrastructure	current

Property	Value
URL	redirect/ecos-kernel-overview.html
Macro	CYGPKG_KERNEL
File	/hda5/home/nickg/local/ecos/build/
Value	current
Default	current
Doc	redirect/ecos-kernel-overview.html
IncludeDir	cyg/kernel
Compile	common/clock.cxx common/timer.cx
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.

Figure 13.2 eCos Configuration Tool - Top Level



eCos HAL	current
I/O sub-system	current
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
☑ Enable priority inheritance protocol	
☑ Enable priority ceiling protocol	
☑ No priority inversion protocol	
Default priority inversion protocol	INHERIT
☑ Specify mutex priority inversion protocol at ru	
☑ Message box blocking put support	
Message box queue size	10
☑ Condition variable timed-wait support	
☑ Condition variable explicit mutex wait support	

Property	Value
URL	redirect/thread-synchronization.html
Macro	CYGSEM_KERNEL_SYNCH_MUTEX_PRIC
File	/hda5/home/nickg/local/ecos/build/pc.ram.c
Value	SIMPLE
Default	SIMPLE
Enabled	True
Flavor	booldata
LegalValues	"SIMPLE"
DefaultValue	"SIMPLE"
ActiveIf	CYGSEM_KERNEL_SCHED_MLQUEUE
Requires	CYGINT_KERNEL_SCHEDULER_UNIQUE_

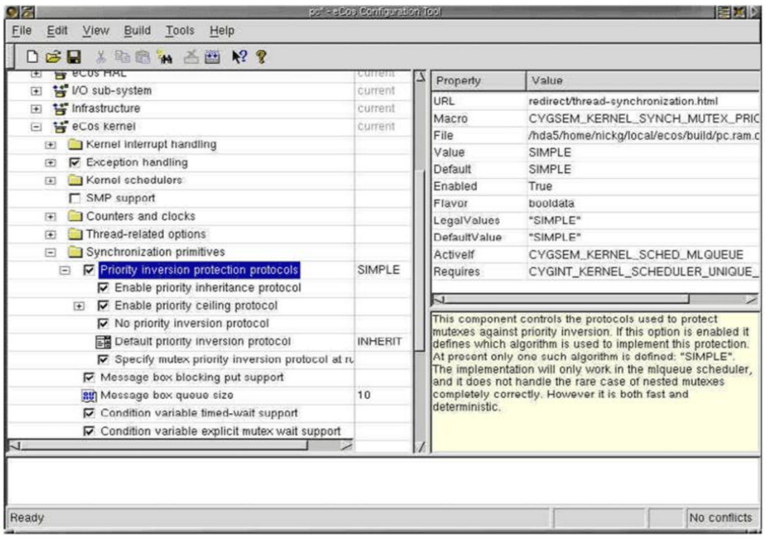
This component controls the protocols used to protect mutexes against priority inversion. If this option is enabled it defines which algorithm is used to implement this protection. At present only one such algorithm is defined: "SIMPLE". The implementation will only work in the mlqueue scheduler, and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

Figure 13.3 eCos Configuration Tool - Kernel Details

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 deselected

eCos Configuration Tool


pc* - eCos Configuration Tool
File Edit View Build Tools Help

```
⊞ eCos HAL                                        current
⊞ I/O sub-system                                  current
⊞ 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
              ☑ Enable priority inheritance protocol
          ⊞ ☑ Enable priority ceiling protocol
              ☑ No priority inversion protocol
                Default priority inversion protocol   INHERIT
              ☑ Specify mutex priority inversion protocol at ru
          ☑ Message box blocking put support
            Message box queue size                    10
          ☑ Condition variable timed-wait support
          ☑ Condition variable explicit mutex wait support
```

Property	Value
URL	redirect/thread-synchronization.html
Macro	CYGSEM_KERNEL_SYNCH_MUTEX_PRIC
File	/hda5/home/nickg/local/ecos/build/pc.ram.c
Value	SIMPLE
Default	SIMPLE
Enabled	True
Flavor	booldata
LegalValues	"SIMPLE"
DefaultValue	"SIMPLE"
ActiveIf	CYGSEM_KERNEL_SCHED_MLQUEUE
Requires	CYGINT_KERNEL_SCHEDULER_UNIQUE_

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

Loading an eCos Configuration

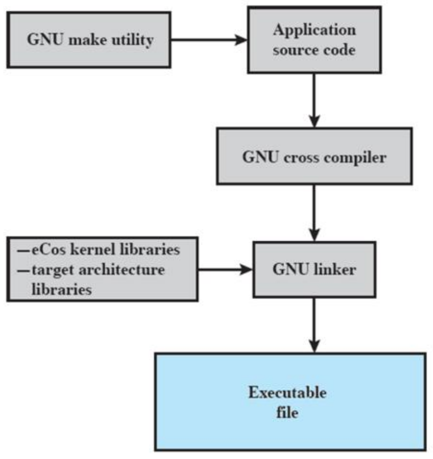


Figure 13.4 Loading an eCos Configuration

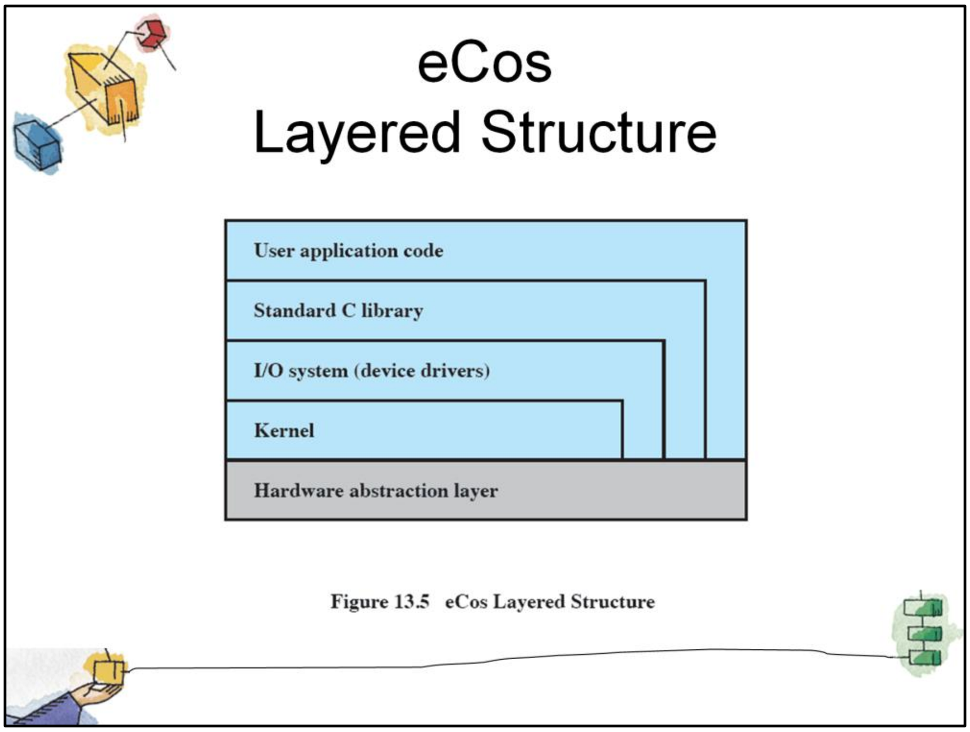
This figure shows a typical example of the overall process of creating the binary image to execute in the embedded system.

This process is run on a source system, e.g. 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.



eCos Layered Structure

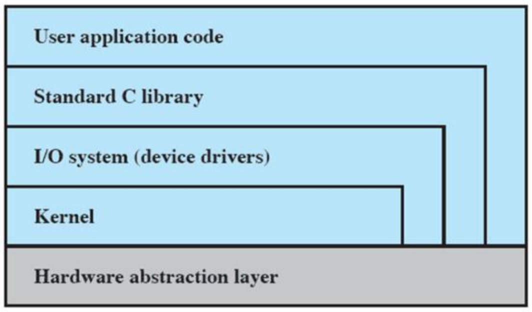
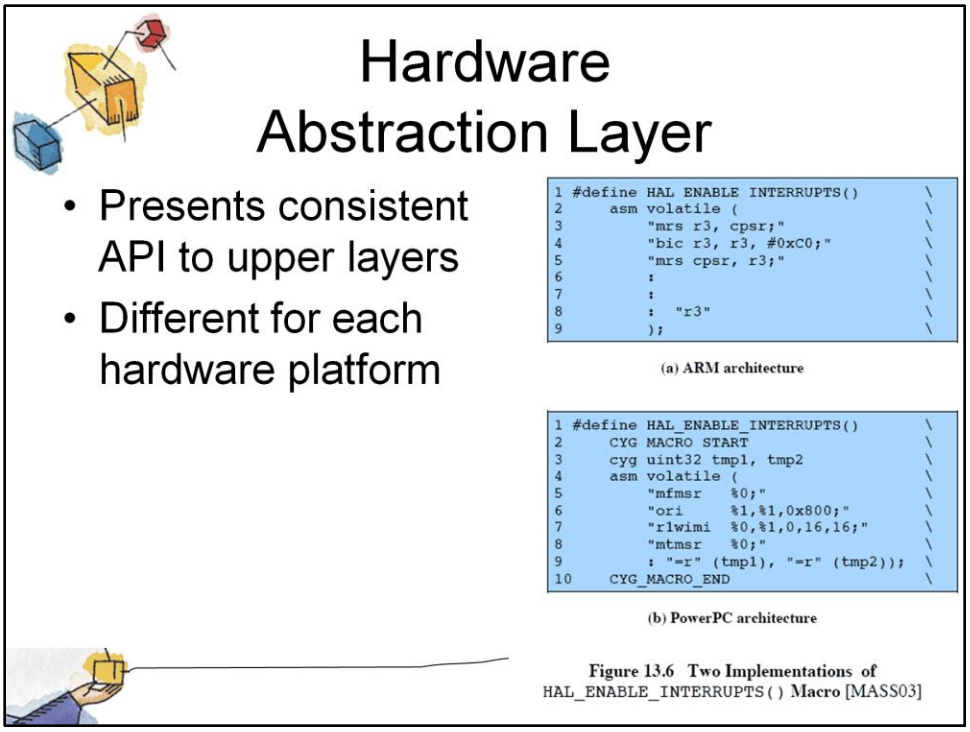


Figure 13.5 eCos Layered Structure

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



Hardware Abstraction Layer

- Presents consistent API to upper layers
- Different for each hardware platform

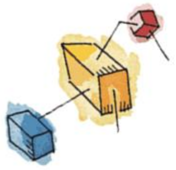
```
#define HAL ENABLE INTERRUPTS()             \
    asm volatile (                          \
        "mrs r3, cpsr;"                     \
        "bic r3, r3, #0xC0;"                \
        "mrs cpsr, r3;"                     \
        :                                   \
        :           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

(a) ARM architecture

```
#define HAL_ENABLE_INTERRUPTS()             \
    CYG MACRO START                         \
    cyg uint32 tmp1, tmp2                   \
    asm volatile (                          \
        "mfmsr   %0;"                       \
        "ori     %1,%1,0x800;"              \
        "rlwimi  %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** [MASS03]



HAL

```
#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

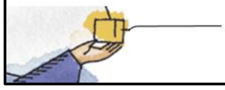
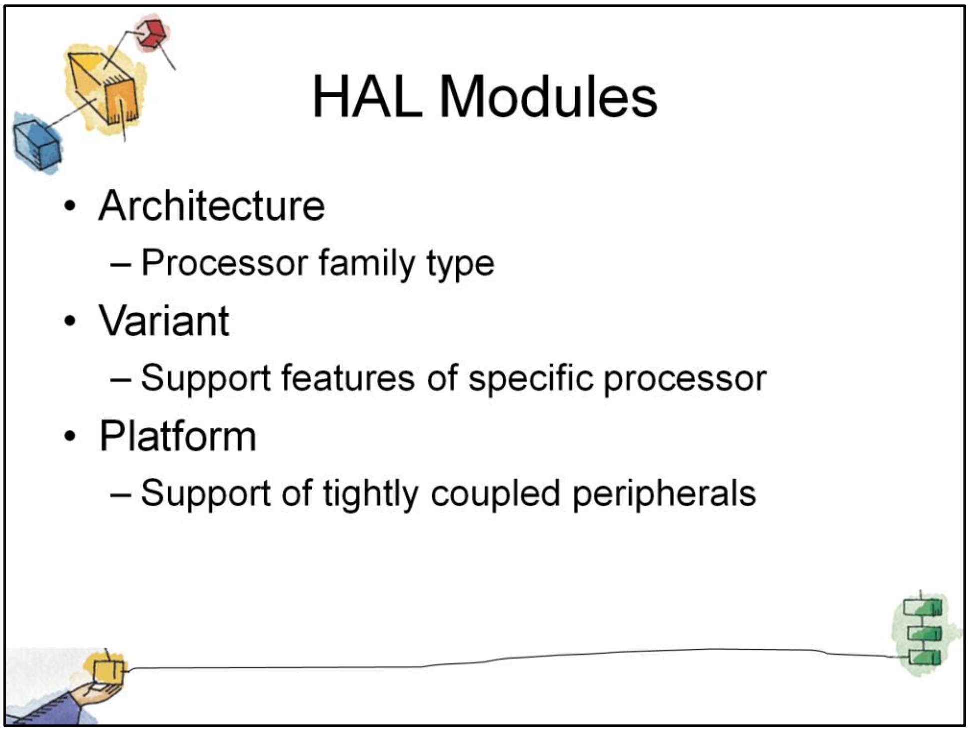


Figure 13.6 Two Implementations of `HAL_ENABLE_INTERRUPTS()` **Macro** [MASS03]



HAL Modules

- Architecture
  - Processor family type
- Variant
  - Support features of specific processor
- Platform
  - Support of tightly coupled peripherals

The HAL is implemented as three separate modules:

Architecture:

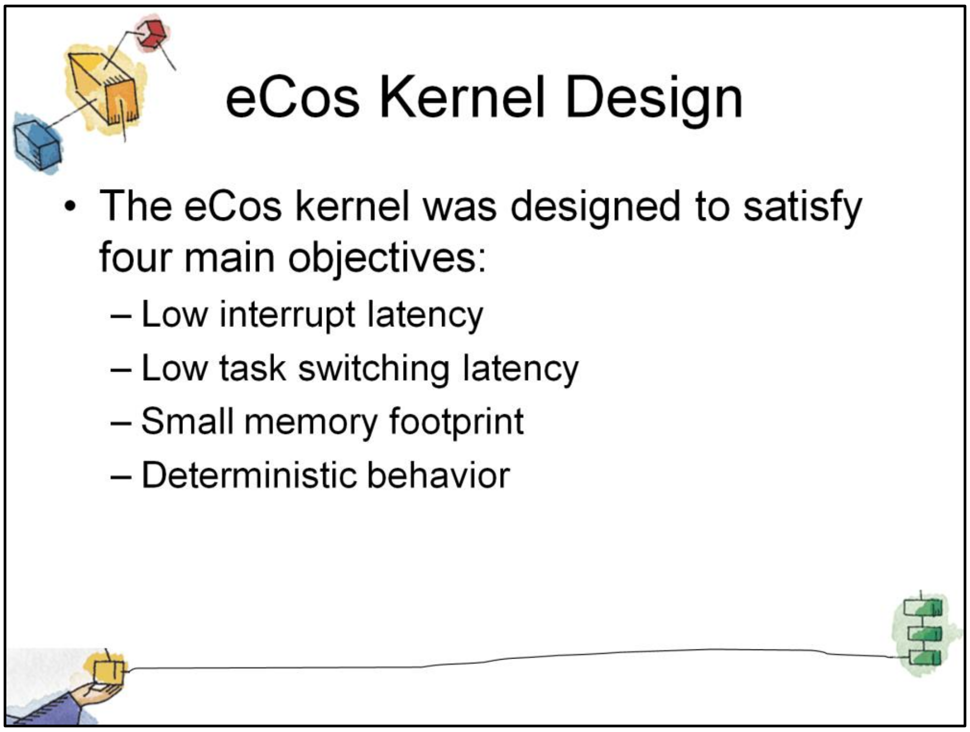
- Defines the processor family type.
- This module contains the code necessary for processor startup, interrupt delivery, context switching, and other functionality specific to the instruction set architecture of that processor family.

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.



eCos Kernel Design

- The eCos kernel was designed to satisfy four main objectives:
  - Low interrupt latency
  - Low task switching latency
  - Small memory footprint
  - Deterministic behavior

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:

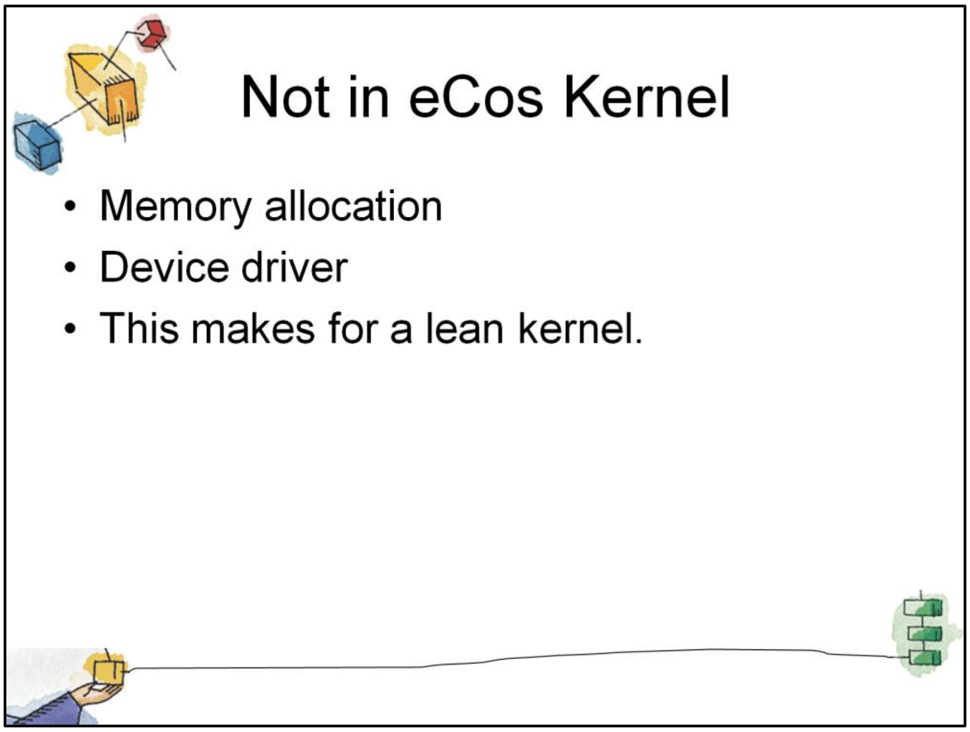
• The time it takes from when a thread becomes available to when actual execution begins.

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.

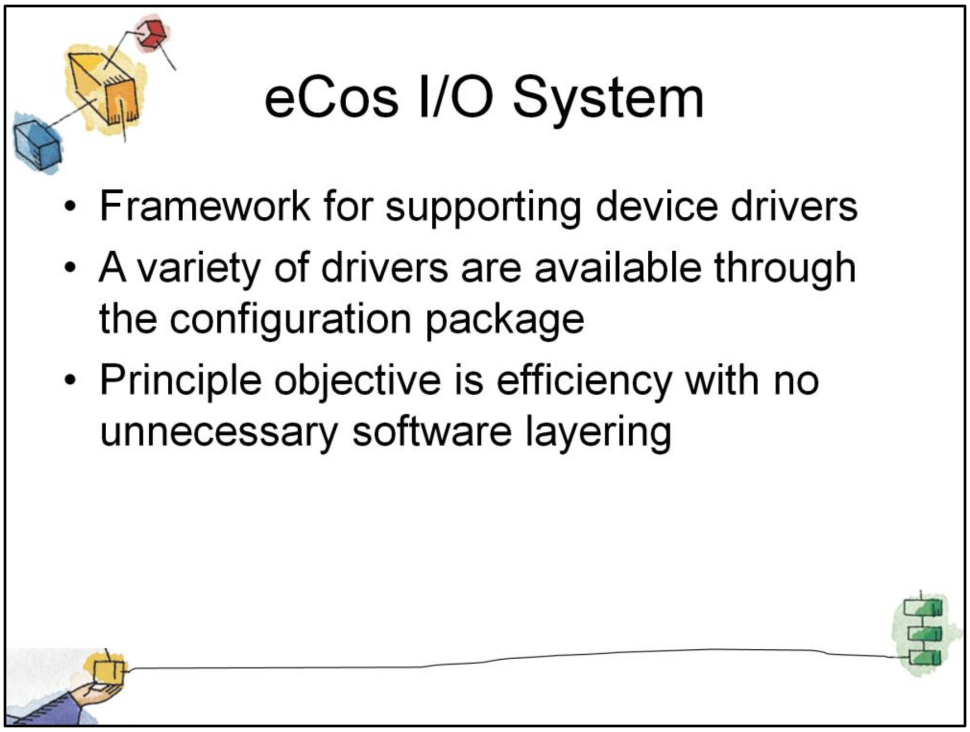


Not in eCos Kernel

- Memory allocation
- Device driver
- This makes for a lean kernel.

Some functionality that is typically included in the kernel of an OS is not included in the eCos kernel.

- E.g. memory allocation is handled by a separate package.
- Similarly, each device driver is a separate package.



eCos I/O System

- Framework for supporting device drivers
- A variety of drivers are available through the configuration package
- Principle objective is efficiency with no unnecessary software layering

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.

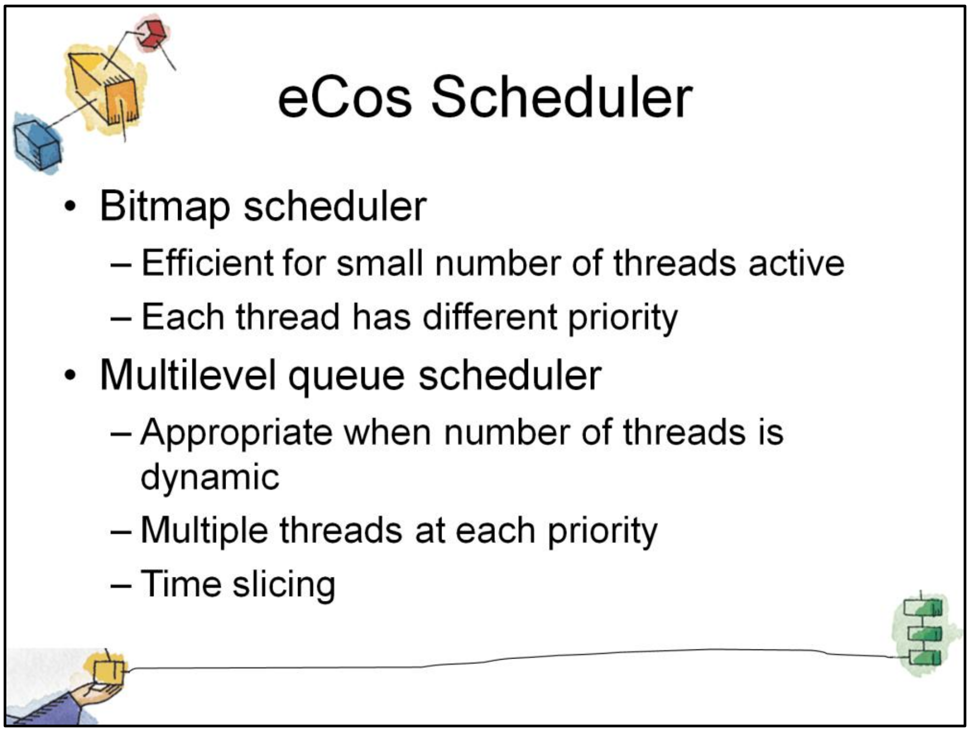
- These include drivers for serial devices, Ethernet, flash memory interfaces, and various I/O interconnects such as PCI (peripheral component interconnect) and USB (universal serial bus).
- In addition, users can develop their own device drivers.

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.

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.



eCos Scheduler

- Bitmap scheduler
  - Efficient for small number of threads active
  - Each thread has different priority
- Multilevel queue scheduler
  - Appropriate when number of threads is dynamic
  - Multiple threads at each priority
  - Time slicing

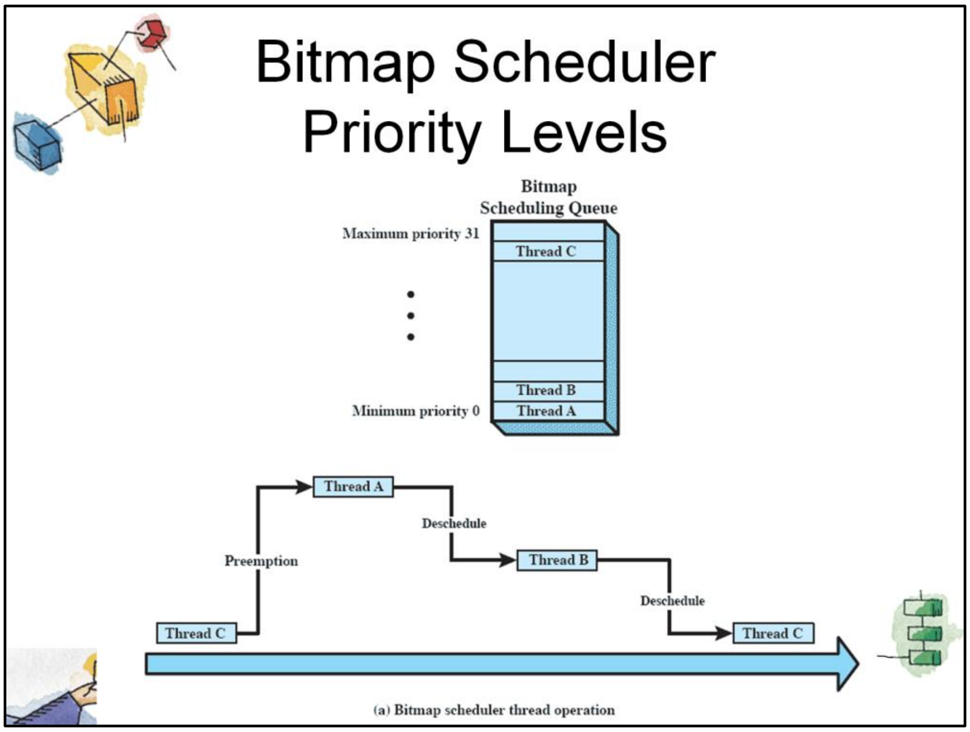
The eCos kernel can be configured to provide one of two scheduler designs:

- the bitmap scheduler and
- a multilevel queue scheduler.

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 Priority Levels

(a) Bitmap scheduler thread operation

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.

When a blocked thread become ready to run, it may preempt a thread of lower priority.

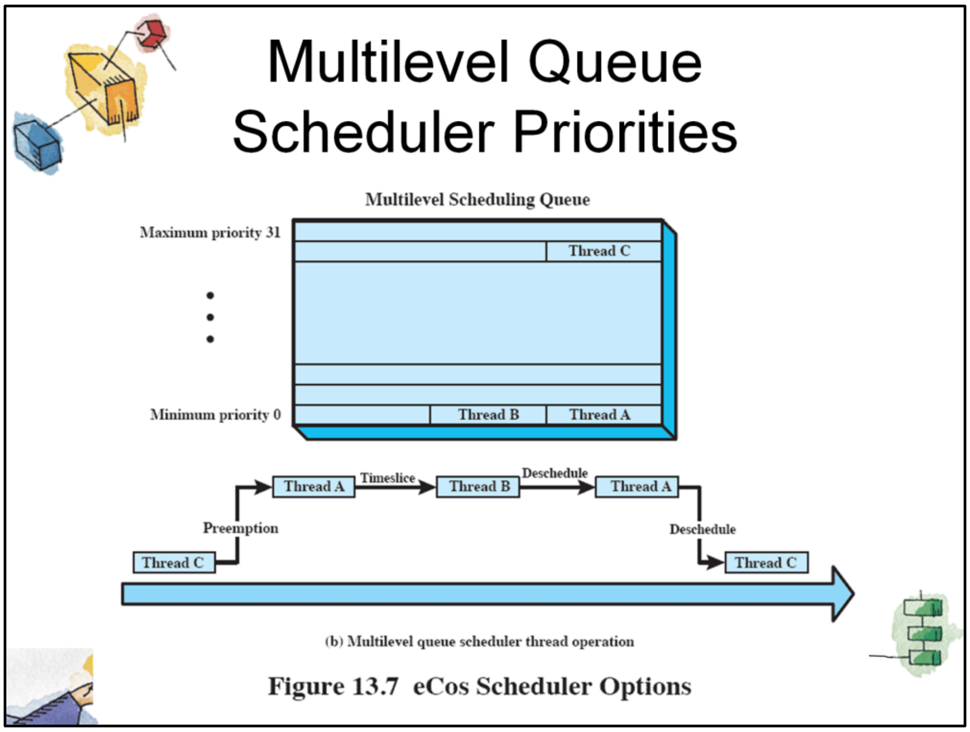
When a running thread suspends, the ready thread with the highest priority is dispatched.

A thread can besuspended 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 Priorities

(b) Multilevel queue scheduler thread operation

Figure 13.7 eCos Scheduler Options

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.

This Figure 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.

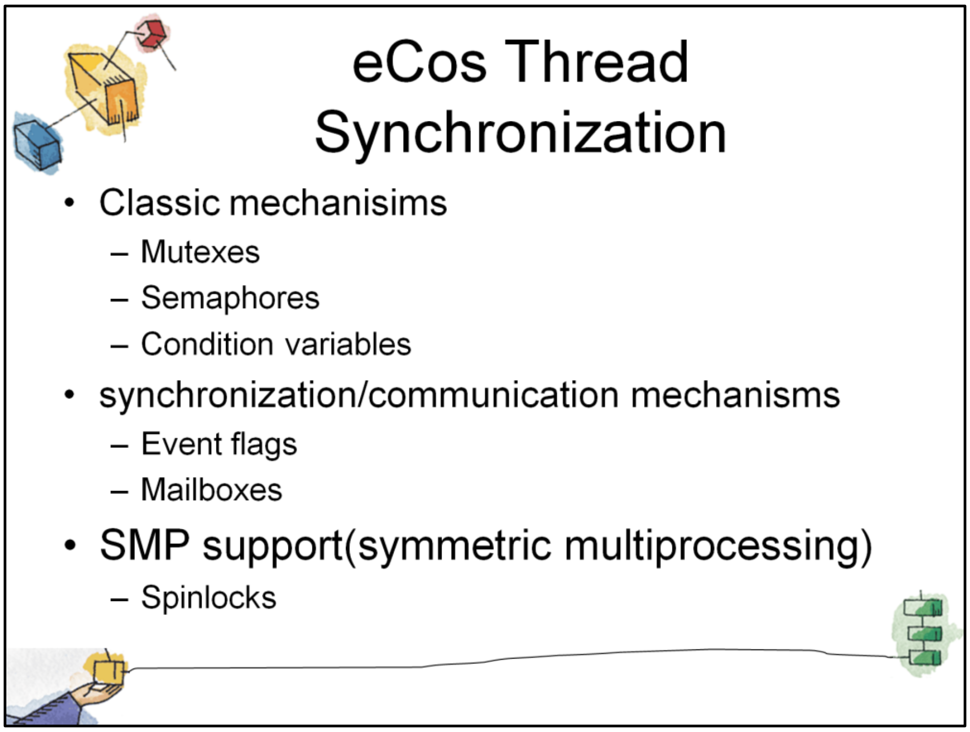
When a thread is blocked, the scheduler must first determine if one or more threads at the same priority level as the blocked thread is ready.

- 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.

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.



eCos Thread Synchronization

- Classic mechanisims
  - Mutexes
  - Semaphores
  - Condition variables
- synchronization/communication mechanisms
  - Event flags
  - Mailboxes
- SMP support(symmetric multiprocessing)
  - Spinlocks

The eCos kernel can be configured to include one or more of six different thread synchronization mechanisms.

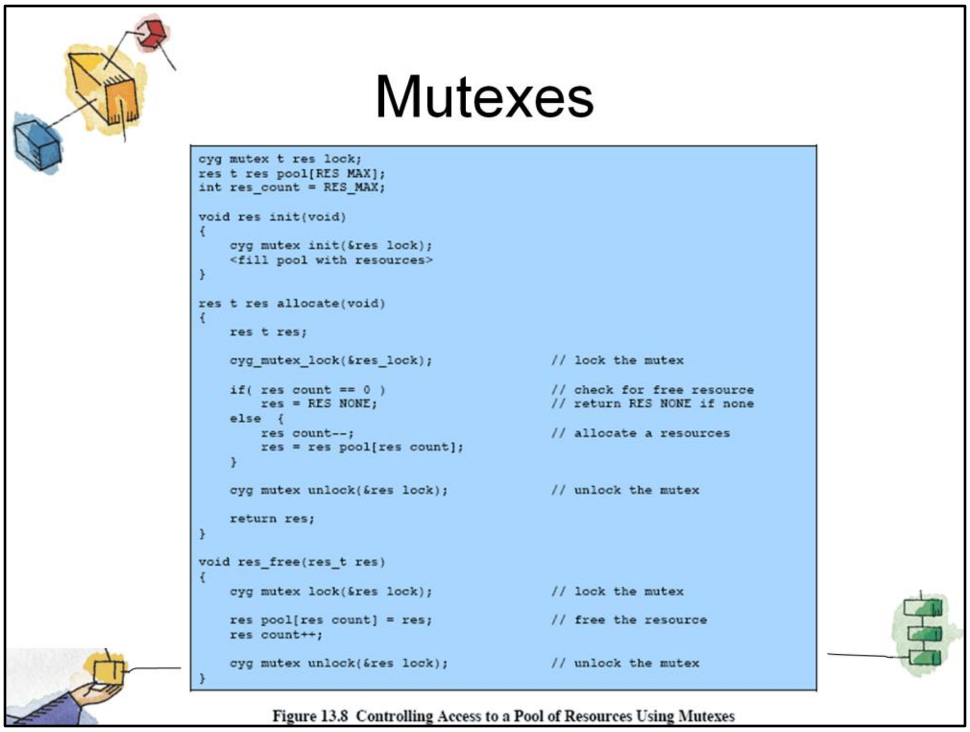
These include the classic synchronization mechanisms:

- mutexes,
- semaphores, and
- condition variables.

In addition, eCos supports two synchronization/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

```
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

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.

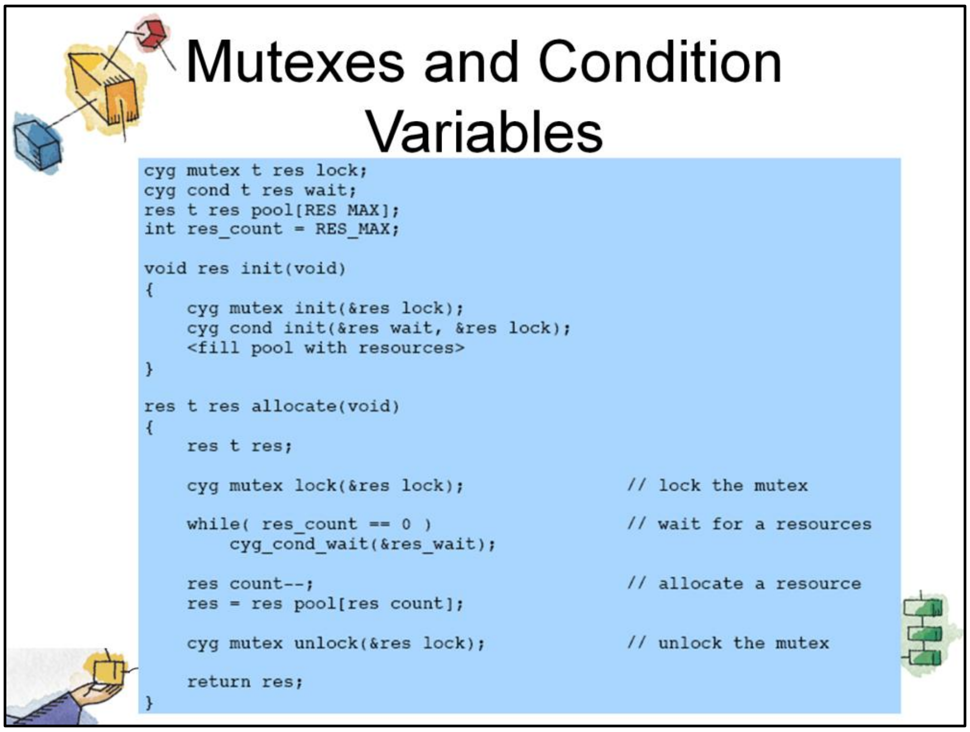
Similar to a binary semaphore:

- 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.

- the thread that locks the mutex must be the one to unlock it.
- But t 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.



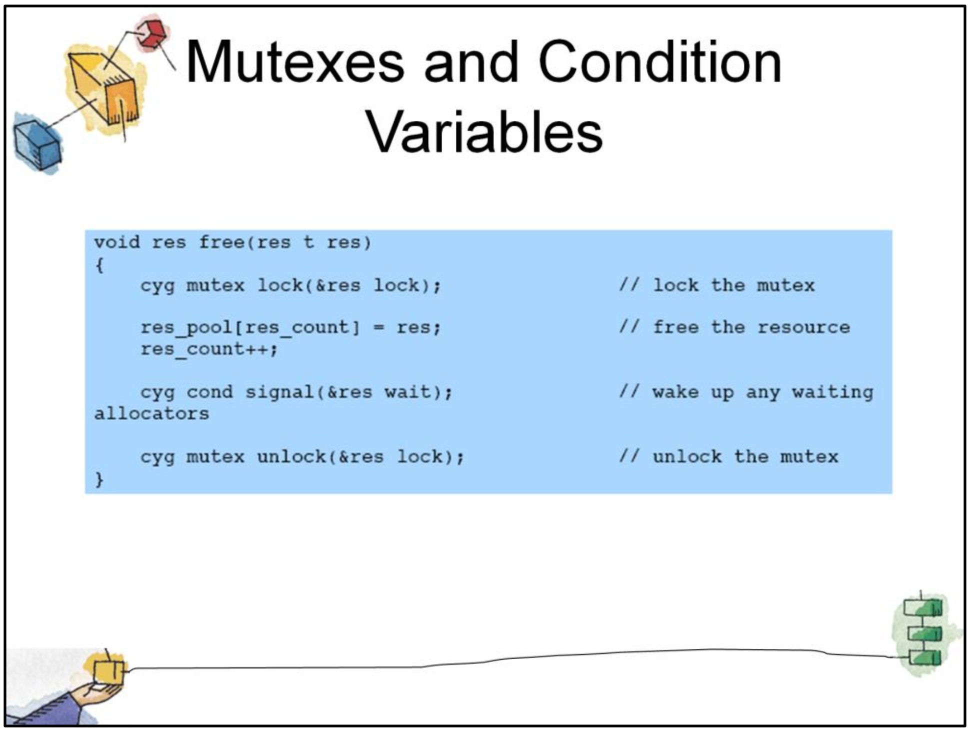
Mutexes and Condition Variables

```
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;
}
```

A condition variable is used to block a thread until a particular condition is true.

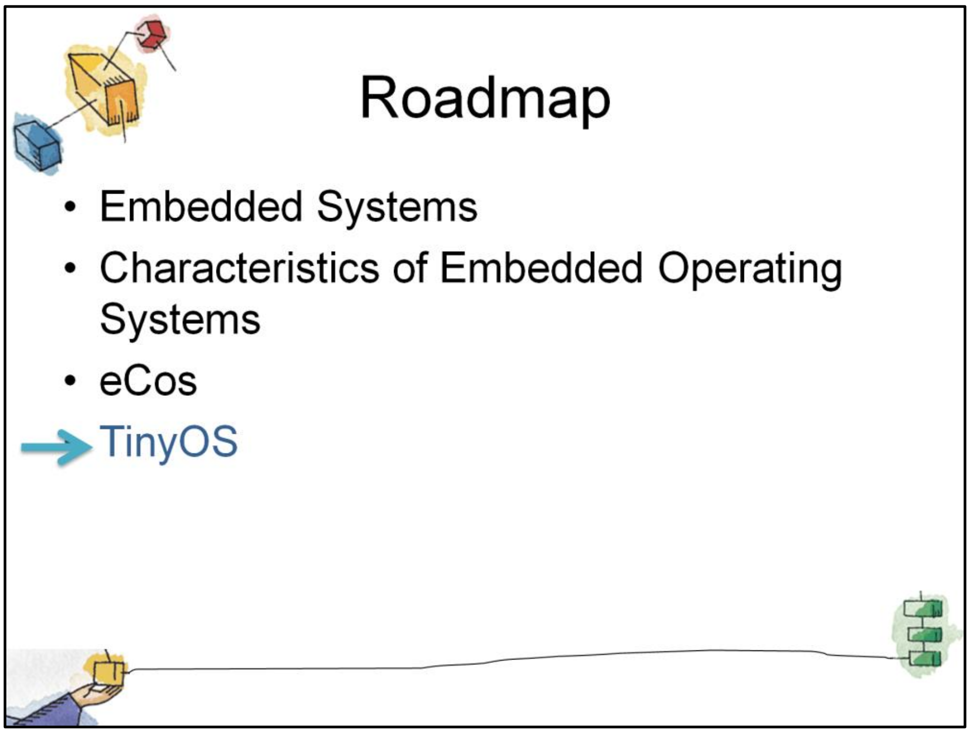
Condition variables are used with mutexes to allow multiple thread to access shared data.

In eCos, condition variables are typically used in conjunction with mutexes to implement long-term waits for some condition to become true.



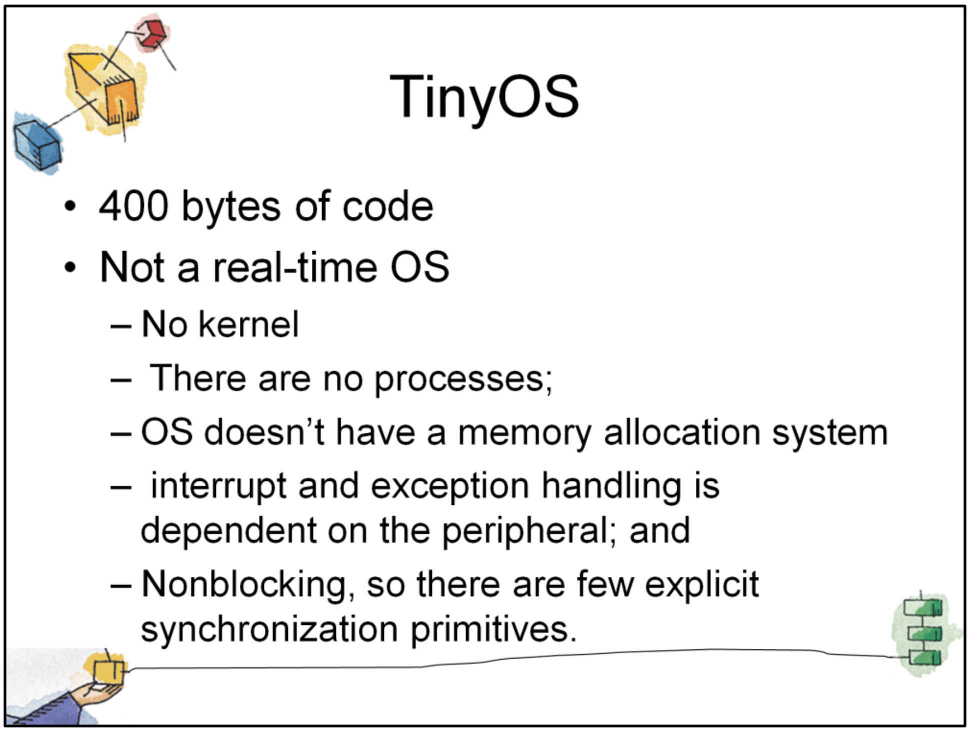
Mutexes and Condition Variables

```
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
}
```



Roadmap

- Embedded Systems
- Characteristics of Embedded Operating Systems
- eCos
- → TinyOS



TinyOS

- 400 bytes of code
- Not a real-time OS
  - No kernel
  - There are no processes;
  - OS doesn't have a memory allocation system
  - interrupt and exception handling is dependent on the peripheral; and
  - Nonblocking, so there are few explicit synchronization primitives.

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.

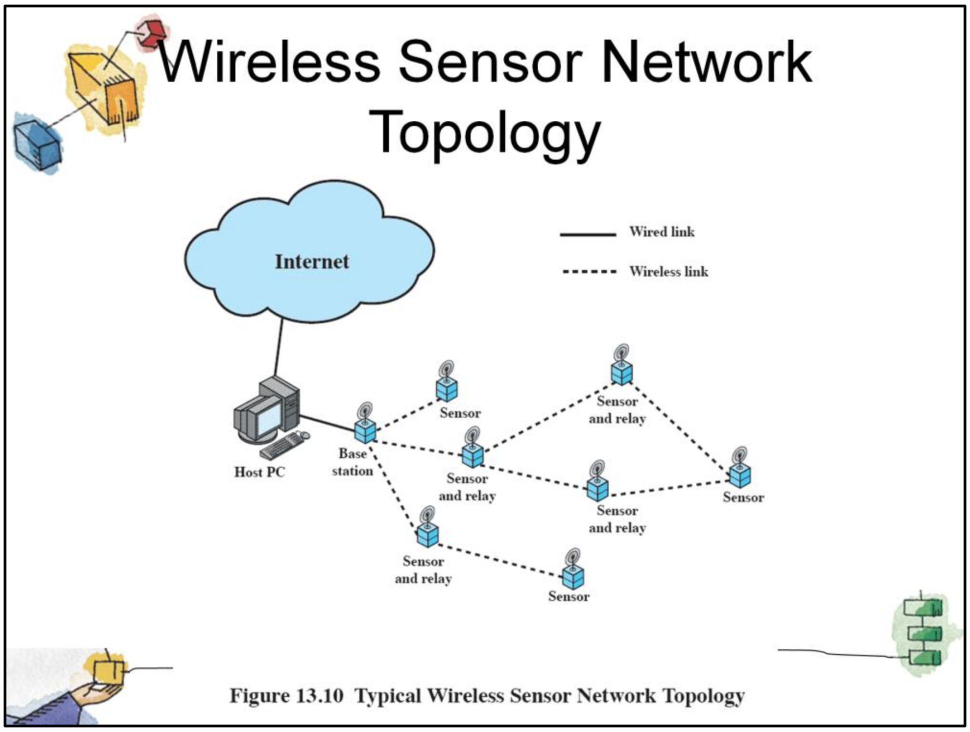
- TinyOS is not a real-time OS.
- The reason for this is the expected workload, which is in the context of a wireless sensor network.

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.

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
- interrupt and exception handling is dependent on the peripheral; and
- it is completely nonblocking, so there are few explicit synchronization primitives.



Wireless Sensor Network Topology

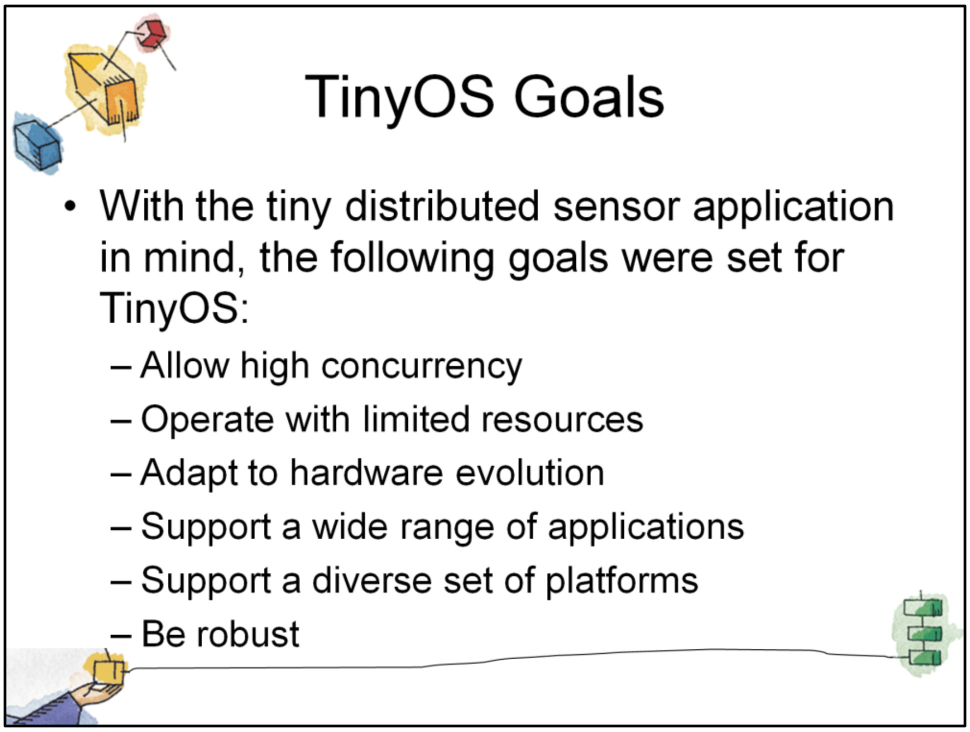
Figure 13.10 Typical Wireless Sensor Network Topology

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.

Low cost, small size, low-power-consuming wireless sensors make can be used in a host of application.

This figure shows a typical configuration.



TinyOS Goals

- With the tiny distributed sensor application in mind, the following goals were set for TinyOS:
  - Allow high concurrency
  - Operate with limited resources
  - Adapt to hardware evolution
  - Support a wide range of applications
  - Support a diverse set of platforms
  - Be robust

With the tiny, distributed sensor application in mind, the following goals were set 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.
- Also, external controls from remote sensors or base stations must be managed.

Operate with limited resources:

- Target platform has 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:

- hardware is in constant evolution; applications and most system services must be portable across hardware generations.
- 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:

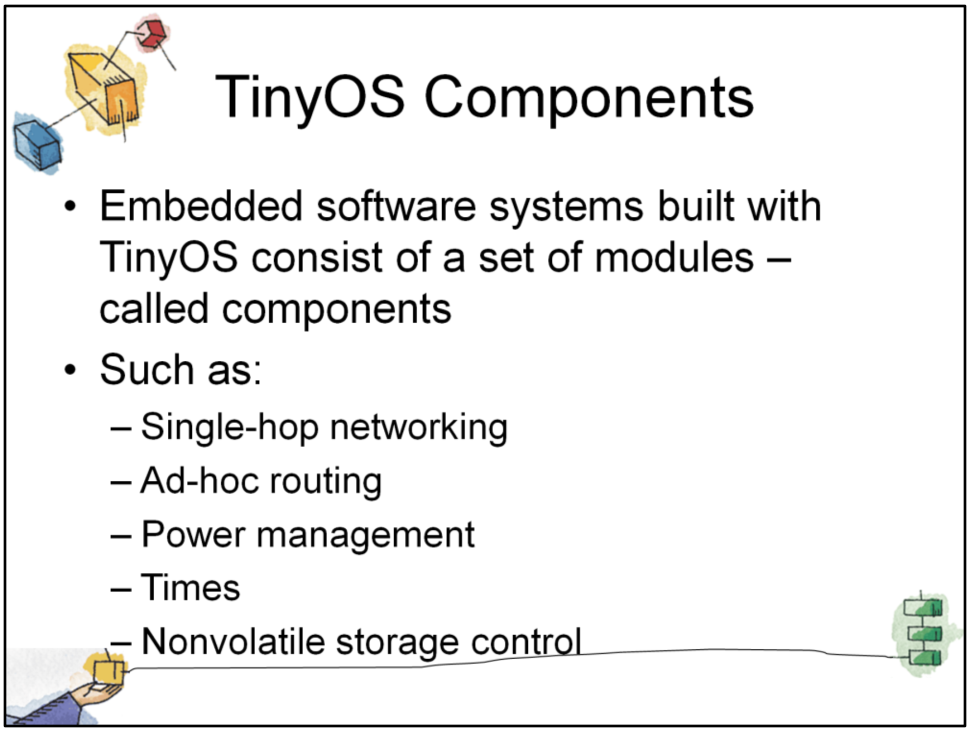
- Applications exhibit a wide range of requirements in terms of lifetime, communication, sensing, and so on.
- A modular, general-purpose embedded OS is desired so that a standardized approach leads to economies of scale in developing applications and support software.

Support a diverse set of platforms:

- 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.

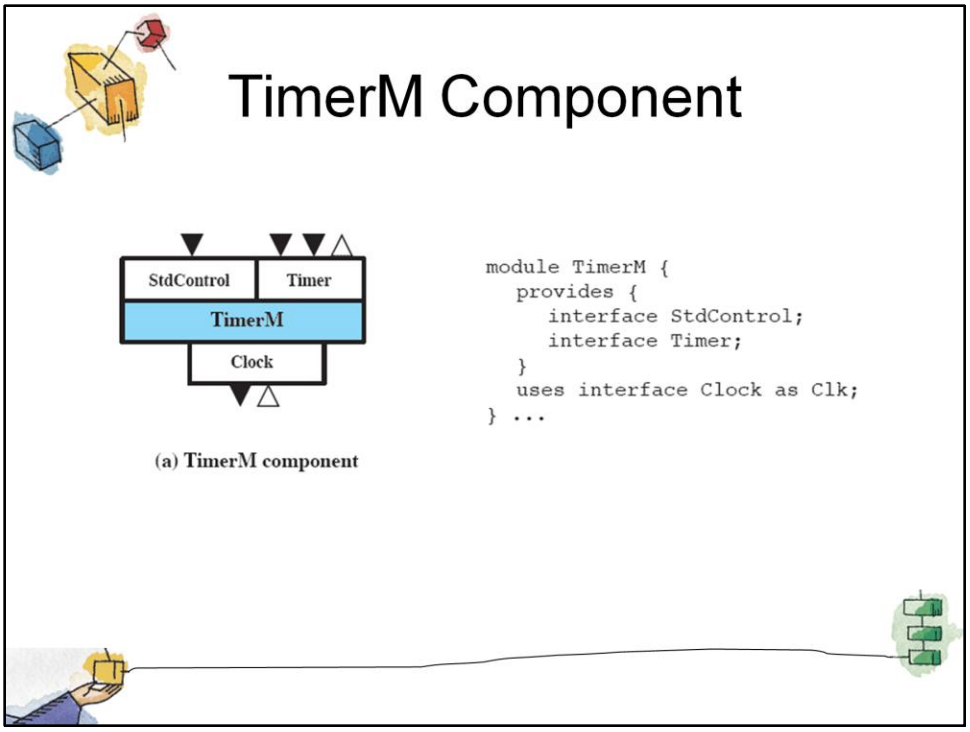


TinyOS Components

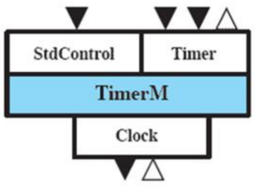
- Embedded software systems built with TinyOS consist of a set of modules – called components
- Such as:
  - Single-hop networking
  - Ad-hoc routing
  - Power management
  - Times
  - Nonvolatile storage control

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.



TimerM Component



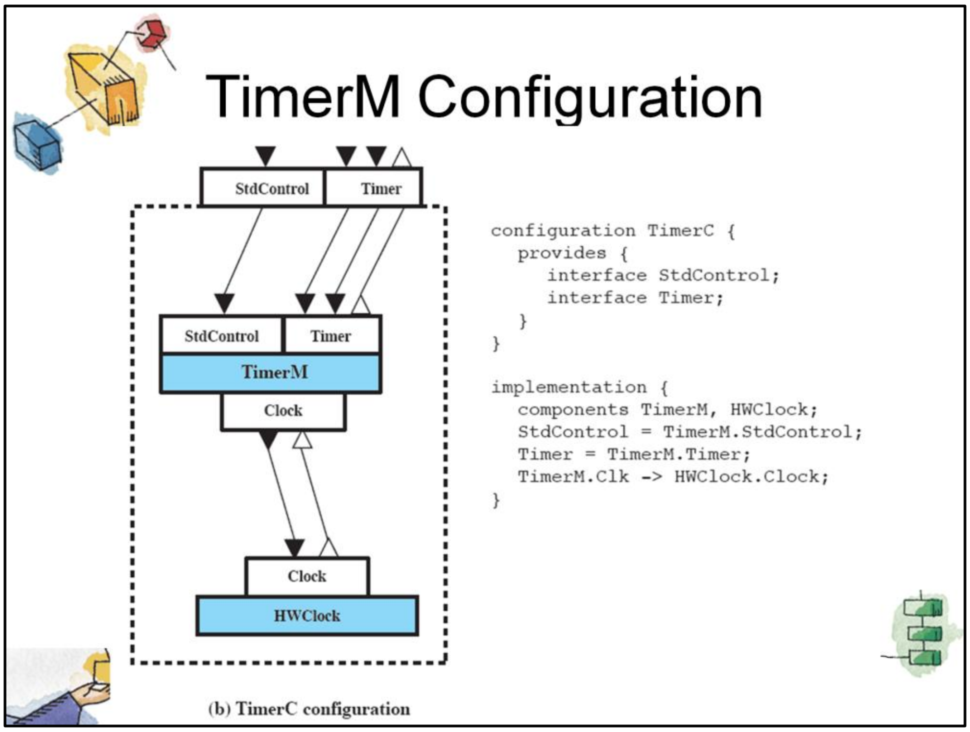
(a) TimerM component

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

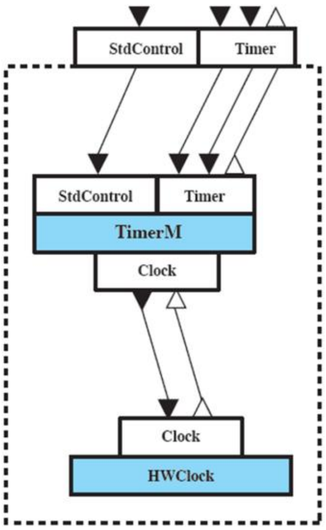
All components in a TinyOS configuration have the same structure, an example of which is shown here.

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.



TimerM Configuration

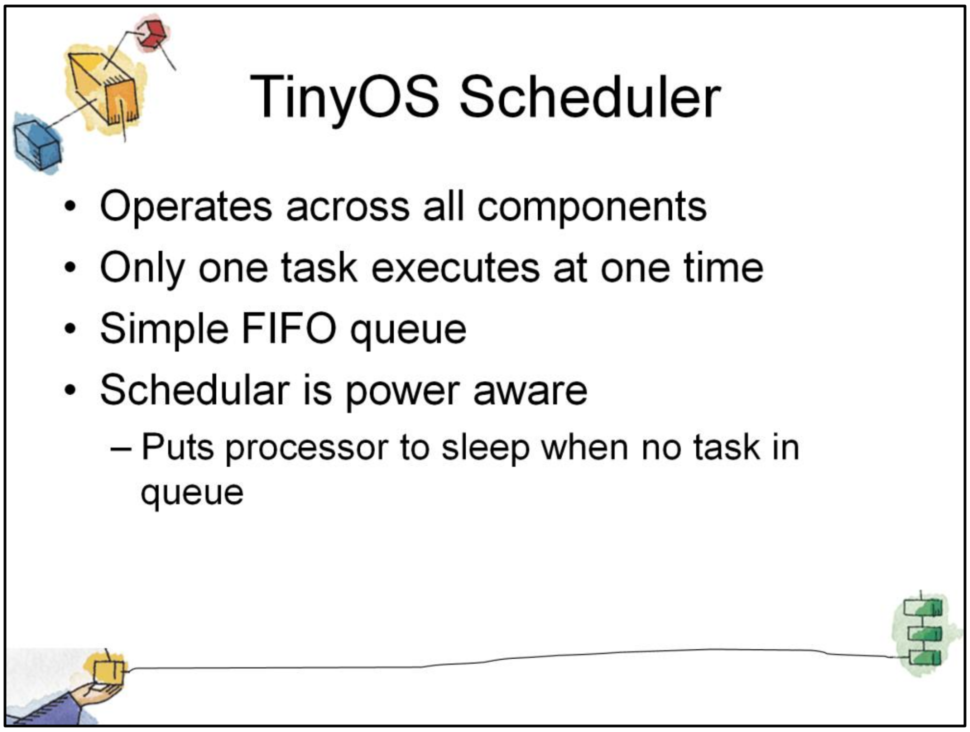


(b) TimerC configuration

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

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

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 here



TinyOS Scheduler

- Operates across all components
- Only one task executes at one time
- Simple FIFO queue
- Schedular is power aware
  - Puts processor to sleep when no task in queue

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.

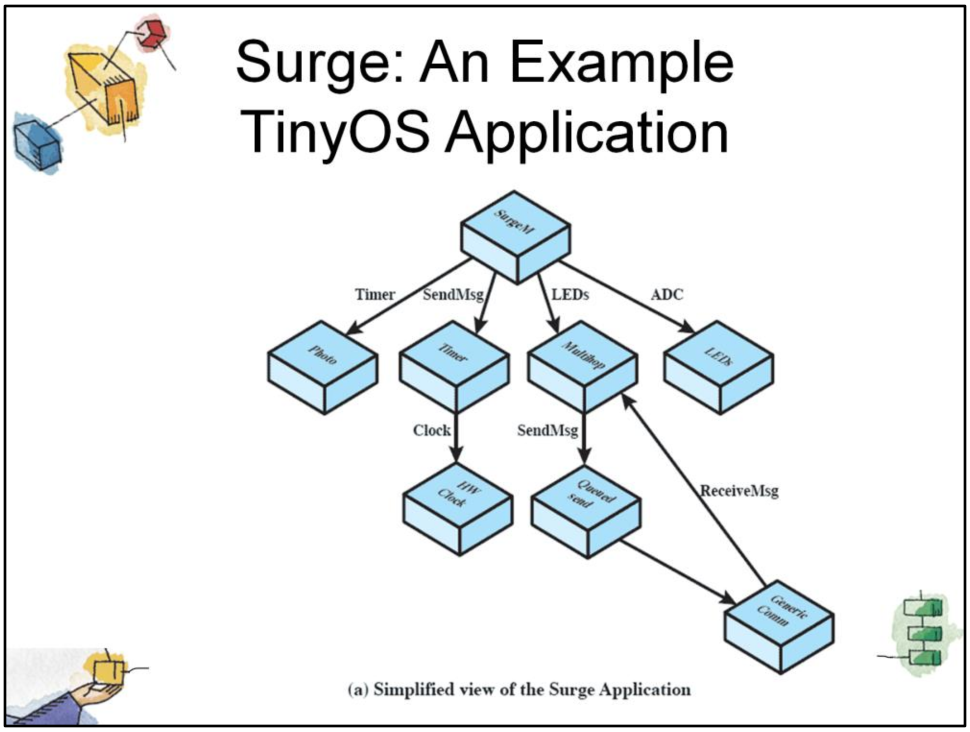
A task is posted to the scheduler (place in the queue) either as a result of an event, which triggers the posting, or as a result of a specific request by a running task to schedule another task.

The scheduler is power aware.

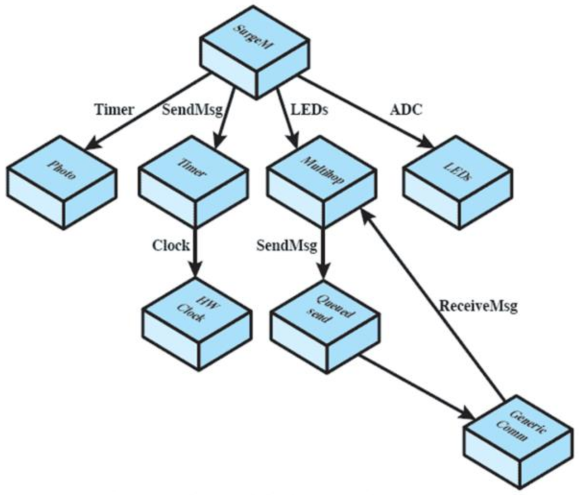
- This means that the scheduler puts the processor to sleep when there are no tasks in the queue.

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



Surge: An Example TinyOS Application

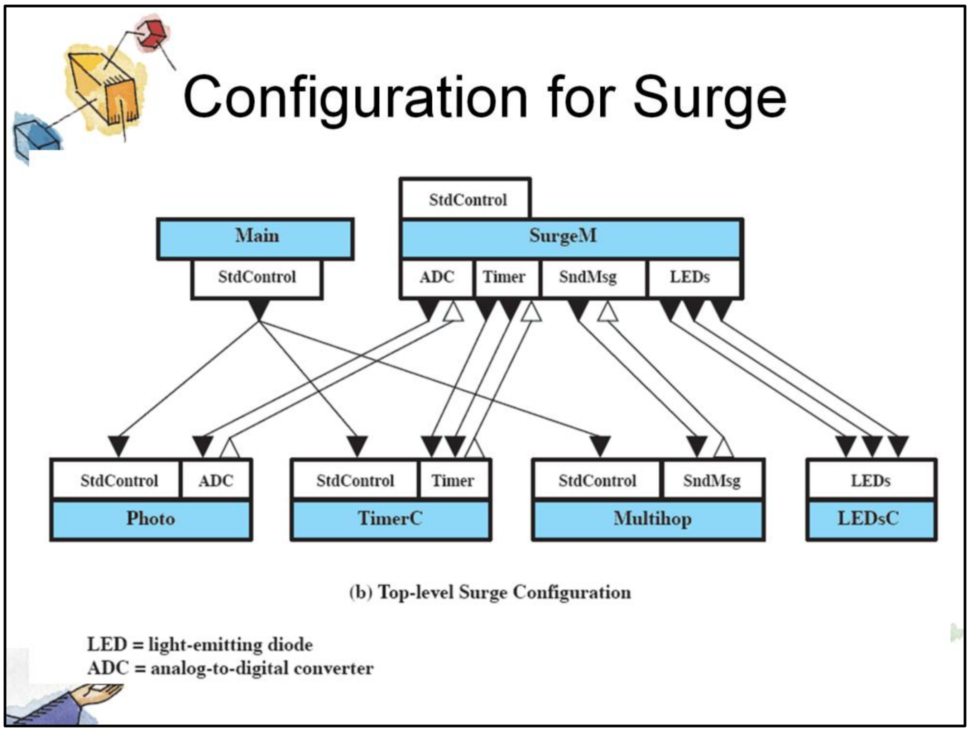


(a) Simplified view of the Surge Application

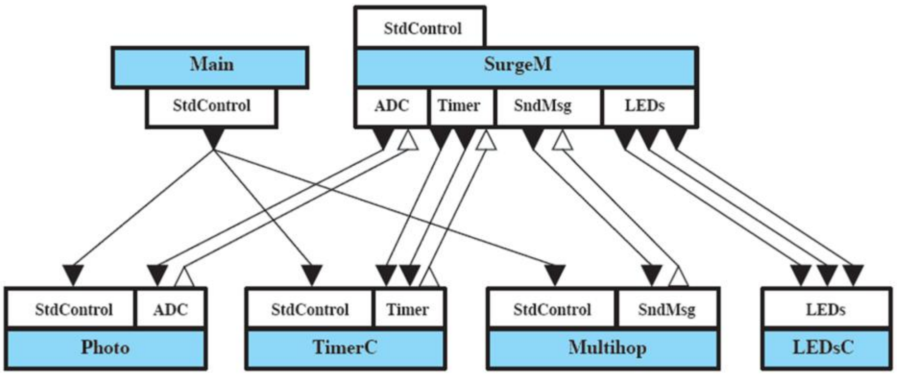
This figure shows a configuration assembled from software and hardware components.

This simplified example, called Surge performs periodic sensor sampling and uses ad-hoc multihop routing over the wireless network

to deliver samples to the base station.



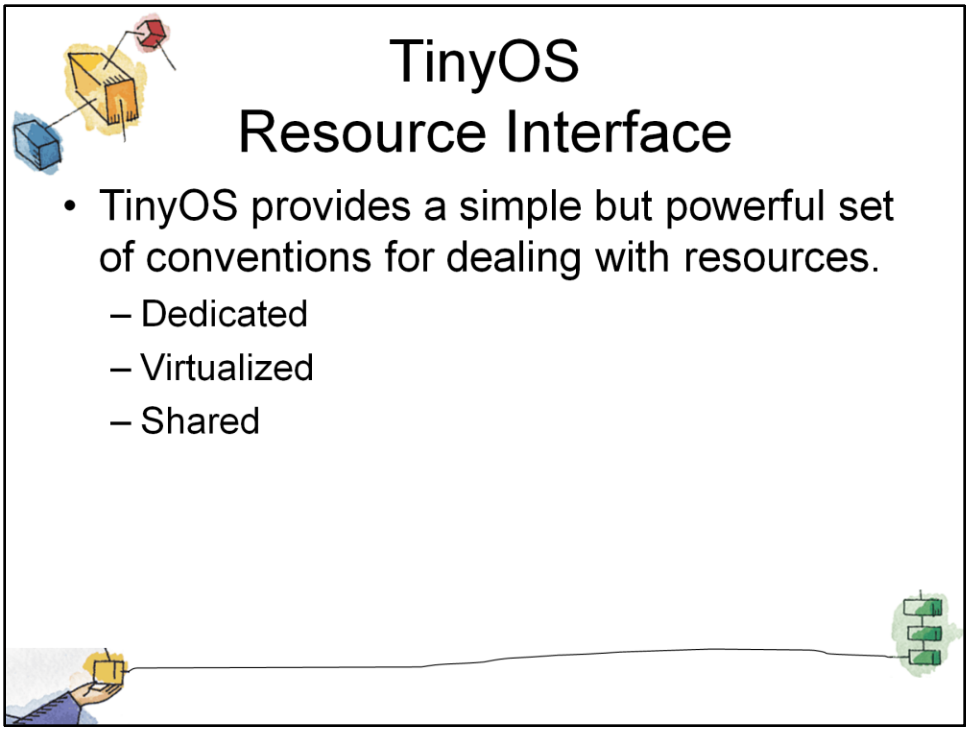
Configuration for Surge



(b) Top-level Surge Configuration

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

This figure shows a portion of the configuration for the Surge application.



TinyOS Resource Interface

- TinyOS provides a simple but powerful set of conventions for dealing with resources.
  - Dedicated
  - Virtualized
  - Shared

TinyOS provides a simple but powerful set of conventions for dealing with resources.

Dedicated:

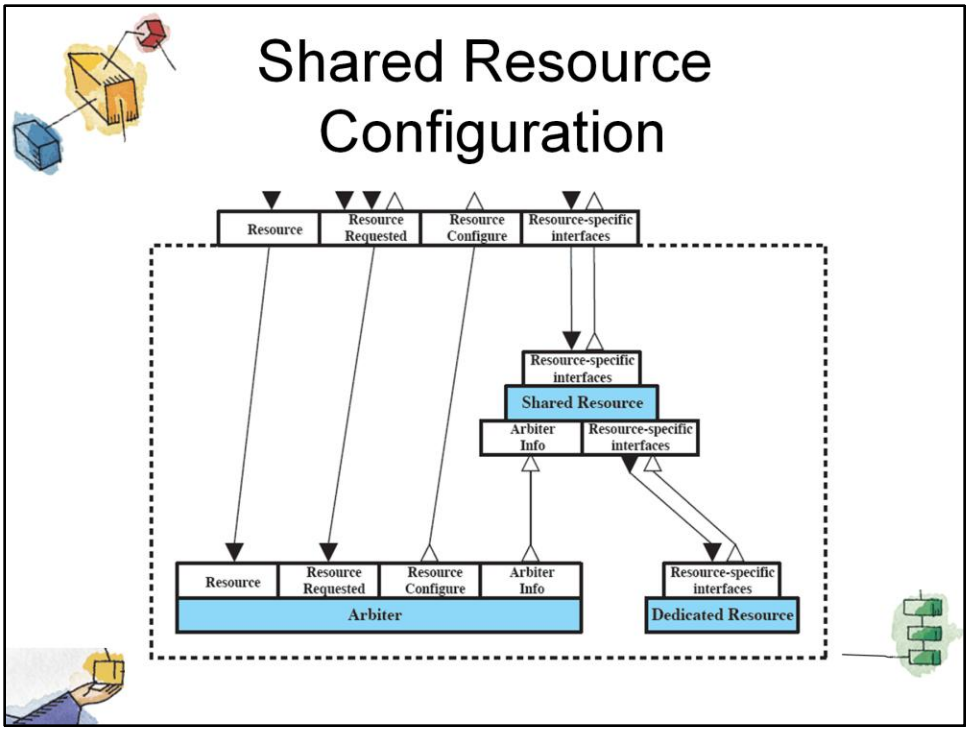
- A resource that a subsystem needs exclusive access to at all times.
- In this class of resources, no sharing policy is needed since only a single component ever requires use of the resource.
- Examples of dedicated abstractions include interrupts and counters.

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.



Shared Resource Configuration

This figure 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.`