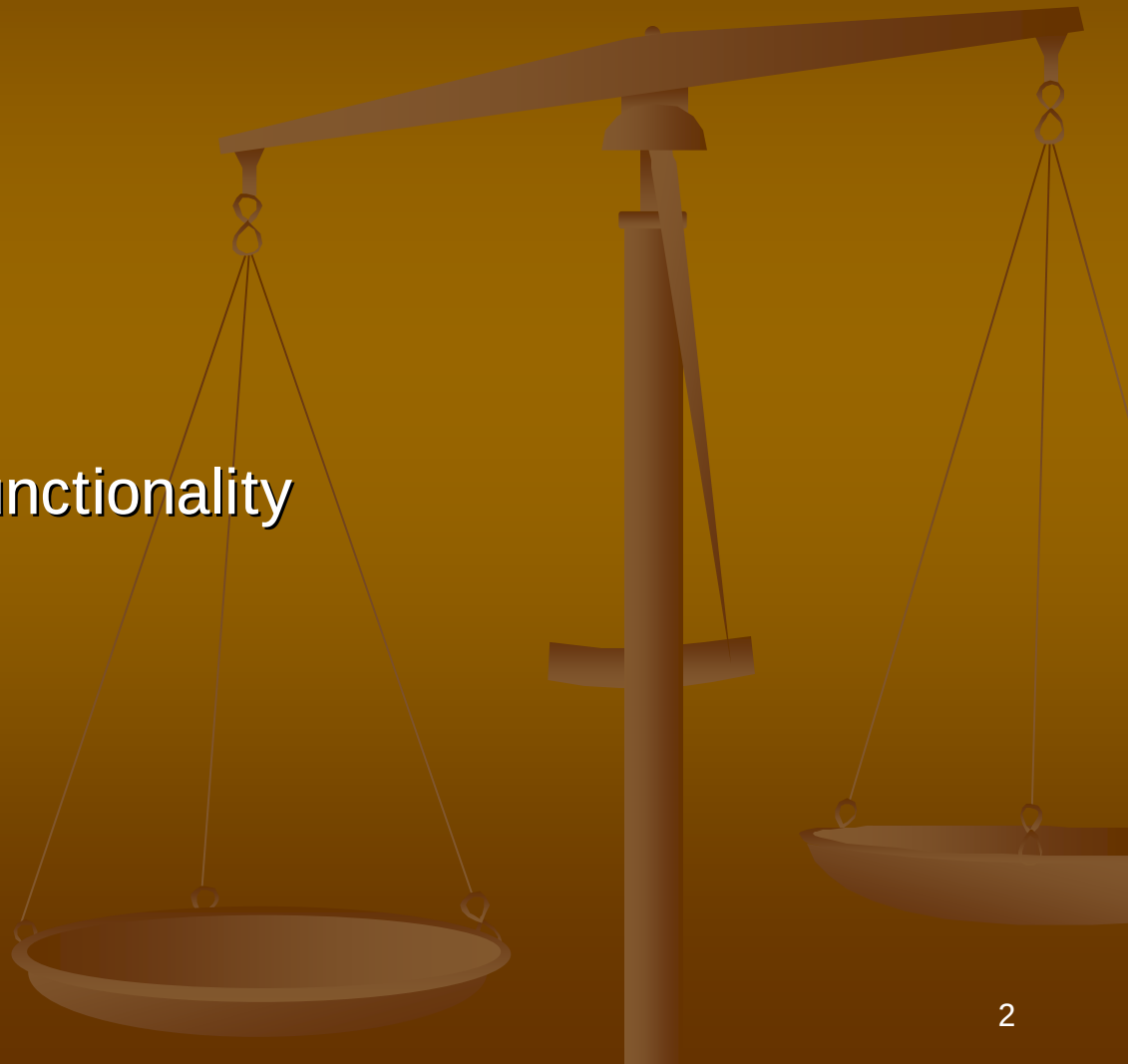
EMBEDDED SOFTWARE DEVELOPMENT WITH ECOS

Chapter 7　Other eCos Architecture Components

指導教授:張軒彬
學生:余勇麟

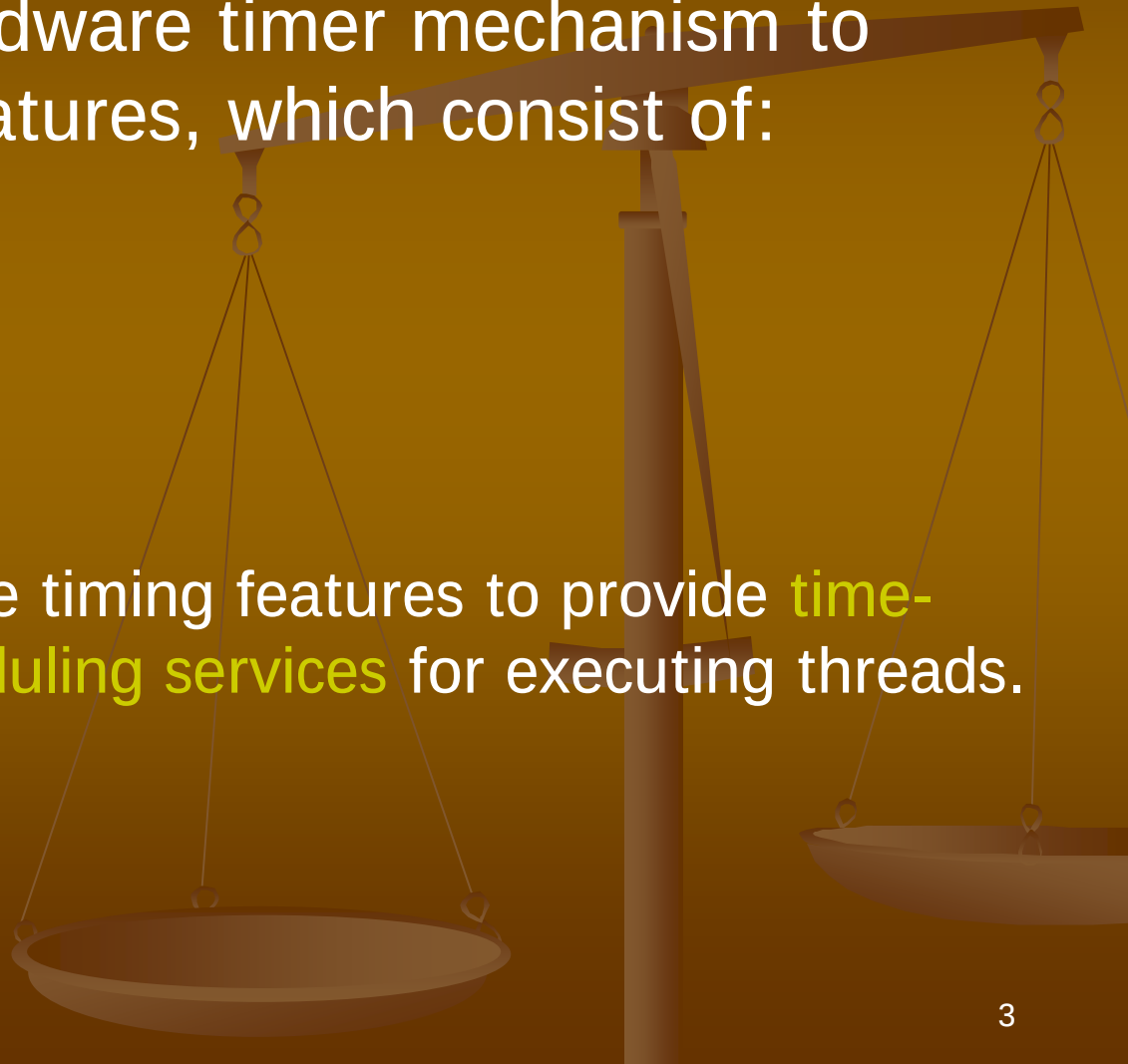
Outline

- Timing components
  - Counters
  - Clocks
  - Alarms
  - Timers
- Assert and Tracing functionality
- I/O control System



Timing components

- eCos uses the hardware timer mechanism to drive its timing features, which consist of:
  - Counters
  - Clocks
  - Alarms
  - Timers
- The kernel uses these timing features to provide time-out, delay, and scheduling services for executing threads.



Timing components(cont.)

- The HAL provides macros to initialize, reset, and read the hardware device used for the kernel timing features.
- Syntax: **HAL_CLOCK_INITIALIZE**(_period_)
  - Parameters: _period_—initial value to set the timing device to achieve the desired interrupt rate.
  - Description: Set the timing device to interrupt at the specified period.
- Syntax: **HAL_CLOCK_RESET**(_vector_,_period_)
  - Parameters:
    vector—timing device interrupt vector. On most HAL packages, this parameter is not used.
    period—initial value to set the timing device to achieve the desired interrupt rate.
  - Description: Reset the timing device with the specified period. This is only necessary for devices that require a reset after the interrupt occurs.
- Syntax: **HAL_CLOCK_READ**(_pvalue_)
  - Parameters: _pvalue_—pointer to counter value read from the timing device.
  - Description: Reads the current value of the timing device counter since the last interrupt. The hardware counter value is returned in the location pointed to by _pvalue_. This macro is hardware dependent and the definition here is the case for most hardware platforms.

Timing components(cont.)

- The HAL architecture-specific configuration components contain a read-only configuration option describing the *real-time clock constants*

- Real-Time Clock Constants' suboption:
  - Real-Time Clock Numerator
  - Real-Time Clock Denominator
  - Real-Time Clock Period
- Period=Numerator / Denominator

- The *Real-Time Clock Period* is the value that is programmed into the processors' hardware timer such that the timer overflows once per kernel tick.

Timing components(cont.)

- When using an eCos-supported target platform, it is usually not necessary to modify these values.
- If want to modify these values, using the *Override Default Clock Settings* configuration option(under the kernel package)
- The *Real-Time Clock Numerator* and *Real-Time Clock Denominator* are also modified to reflect the new timer resolution.

Timing components(cont.)

- **1.** Determine the delay in nanoseconds. In our example, we want a delay of 60 milliseconds,which is the same as 60,000,000 nanoseconds.

- **2.** Next, we need the clock frequency. In this case, we assume a clock running at 100Hz,which corresponds to 1 tick every 10 milliseconds, or 1 tick every 10,000,000 nanoseconds.This corresponds to a numerator of 100 and a denominator of 1,000,000,000.

- **3.** Finally, we can calculate the tick value we need to use in the call using the equation

  $$\frac{\text{Delay (in nanoseconds)} \times \text{Numerator}}{\text{Denominator}} = \text{Clock ticks}$$

  Therefore, in our example we plug in the values and get

  $$\frac{60000000 \times 100}{1000000000} = 6 \text{ (Clock ticks)}$$

- **4.** We then call the clock-related kernel function and pass it the parameter 6 for our 60-millisecond delay.

Timing components(cont.)

- The kernel can be configured to provide a *Real-Time Clock* (RTC) for the system.

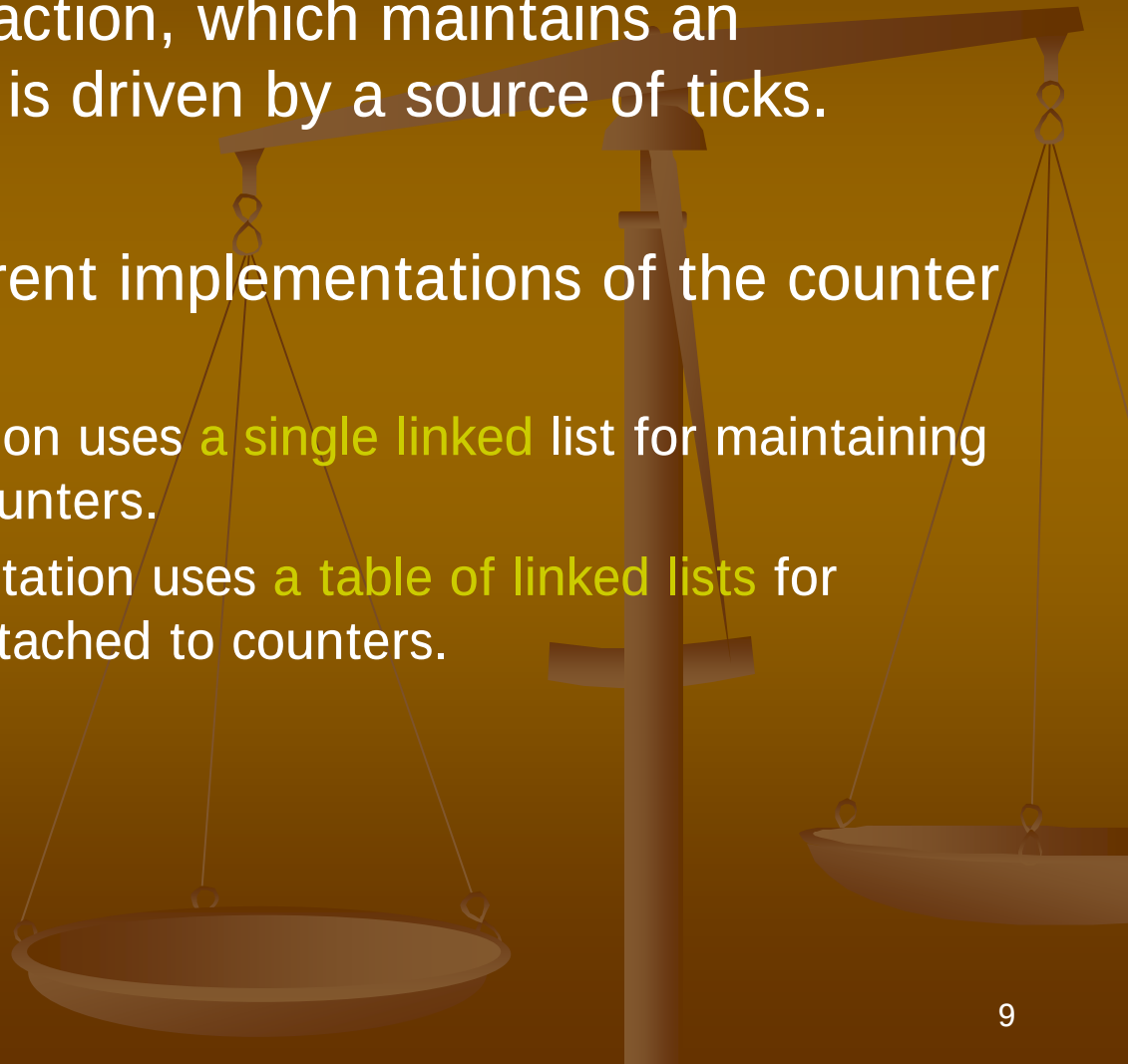
Item List 7.2 Kernel Clock Configuration Options

- Option Name **Provide Real-Time Clock**
- Option Name **Override Default Clock Settings**
  - **Clock Hardware Initialization Value**
  - **Clock Resolution Numerator**
  - **Clock Resolution Denominator**
- Option Name **Measure Real-Time Clock Interrupt Latency**
- Option Name **Measure Real-Time Clock DSR Latency**

Counters

- A counter is an abstraction, which maintains an increasing value that is driven by a source of ticks.

- eCos offers two different implementations of the counter object.
  - The first implementation uses a single linked list for maintaining alarms attached to counters.
  - The second implementation uses a table of linked lists for maintaining alarms attached to counters.



Counters

Item List 7.3 Counter Configuration Options

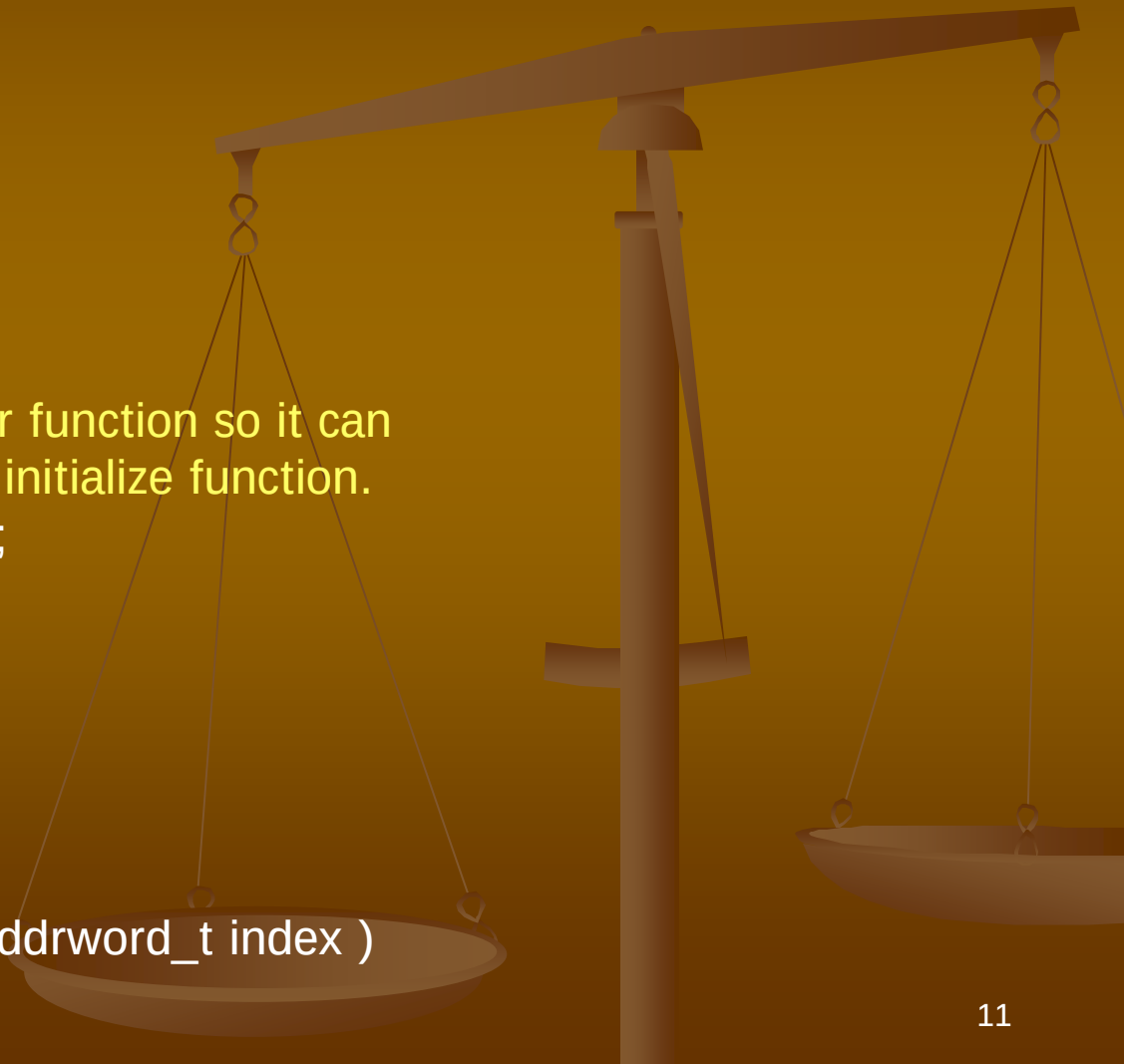
- Option Name: **Implement Counters Using a Single List**
  CDL Name: CYGIMP_KERNEL_COUNTERS_SINGLE_LIST
  Description: This is a more efficient use of memory when a small number of alarms are used in the system. The default setting for this option is enabled.

- Option Name: **Implement Counters Using a Table of Lists**
  CDL Name: CYGIMP_KERNEL_COUNTERS_MULTI_LIST
  Description: This option reduces the amount of computation necessary when a timer triggers, which is useful when many alarms are used in the system. This option is disabled by default.

- Option Name: **Sort the Counter List**
  CDL Name: CYGIMP_KERNEL_COUNTERS_SORT_LIST
  Description: Allows the list of alarms that are attached to counters to be sorted so that the next alarm to trigger is at the front of the list. This reduces the amount of work that needs to be done when a counter tick is processed. This option causes the operation of adding alarms to the list more expensive because the list must be sorted. The default setting for this option is disabled.

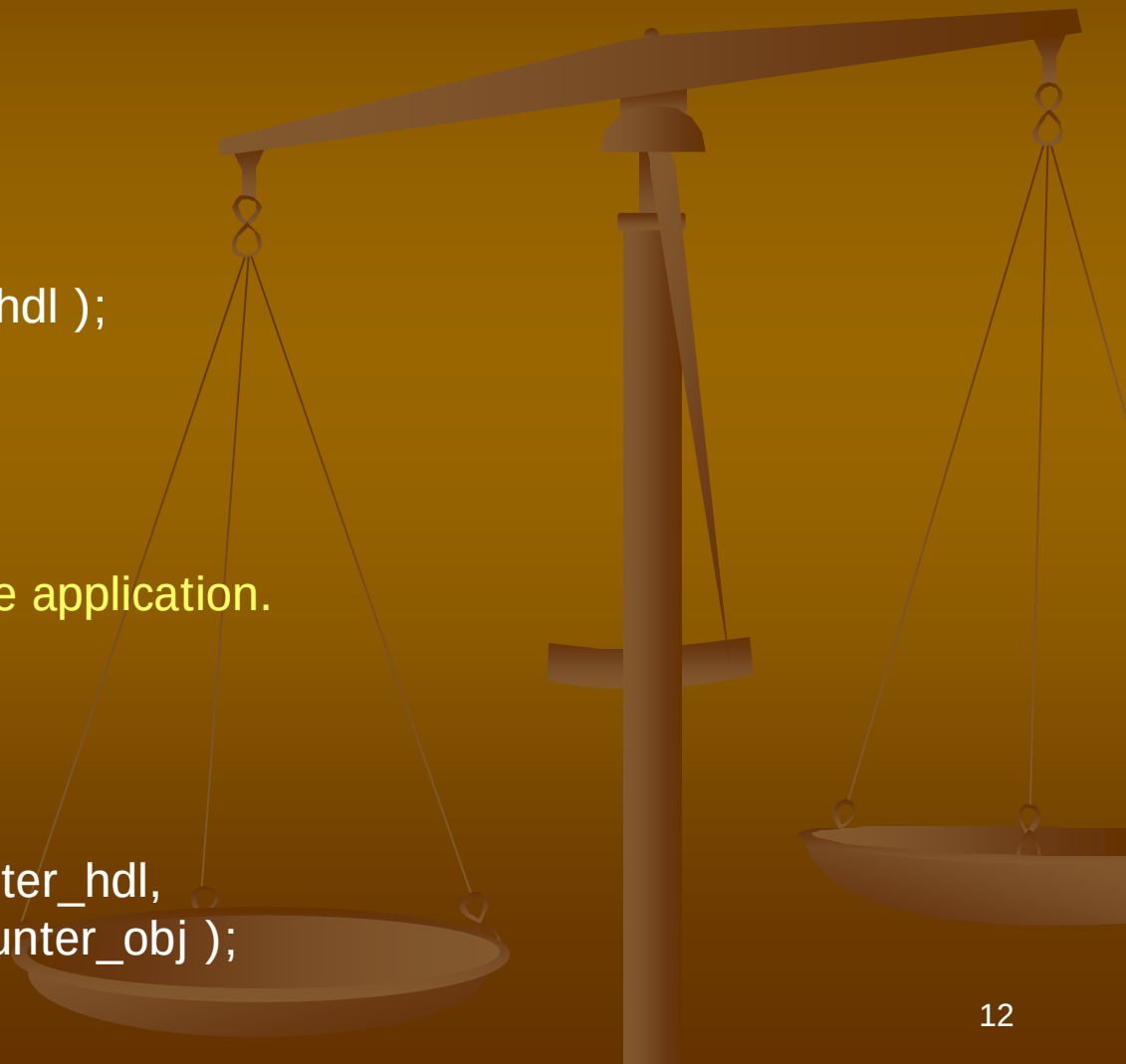
Counters – Causes an alarm to trigger example

```
1   #include <cyg/kernel/kapi.h>
2
3
4   cyg_counter counter_obj;
5   cyg_handle_t counter_hdl;
6
7   cyg_handle_t alarm_hdl;
8   cyg_alarm alarm_obj;
9
10   // Declare the alarm handler function so it can
11   // be passed into the alarm initialize function.
12   cyg_alarm_t alarm_handler;
13
14  unsigned long index = 0;
15
16   //
17   // Counter thread.
18   //
19   void counter_thread( cyg_addrword_t index )
20  {
```



Counters

```
22  // Run forever.
23  while ( 1 )
24  {
25  // Delay for 10 ticks.
26  cyg_thread_delay( 10 );
27
28  // Increment the counter.
29  cyg_counter_tick( counter_hdl );
30  }
31  }
32
33  //
34  // Main starting point for the application.
35  //
36  void cyg_user_start( void )
37  {
38  // Create the counter.
39  cyg_counter_create( &counter_hdl,
40                                  &counter_obj );
```



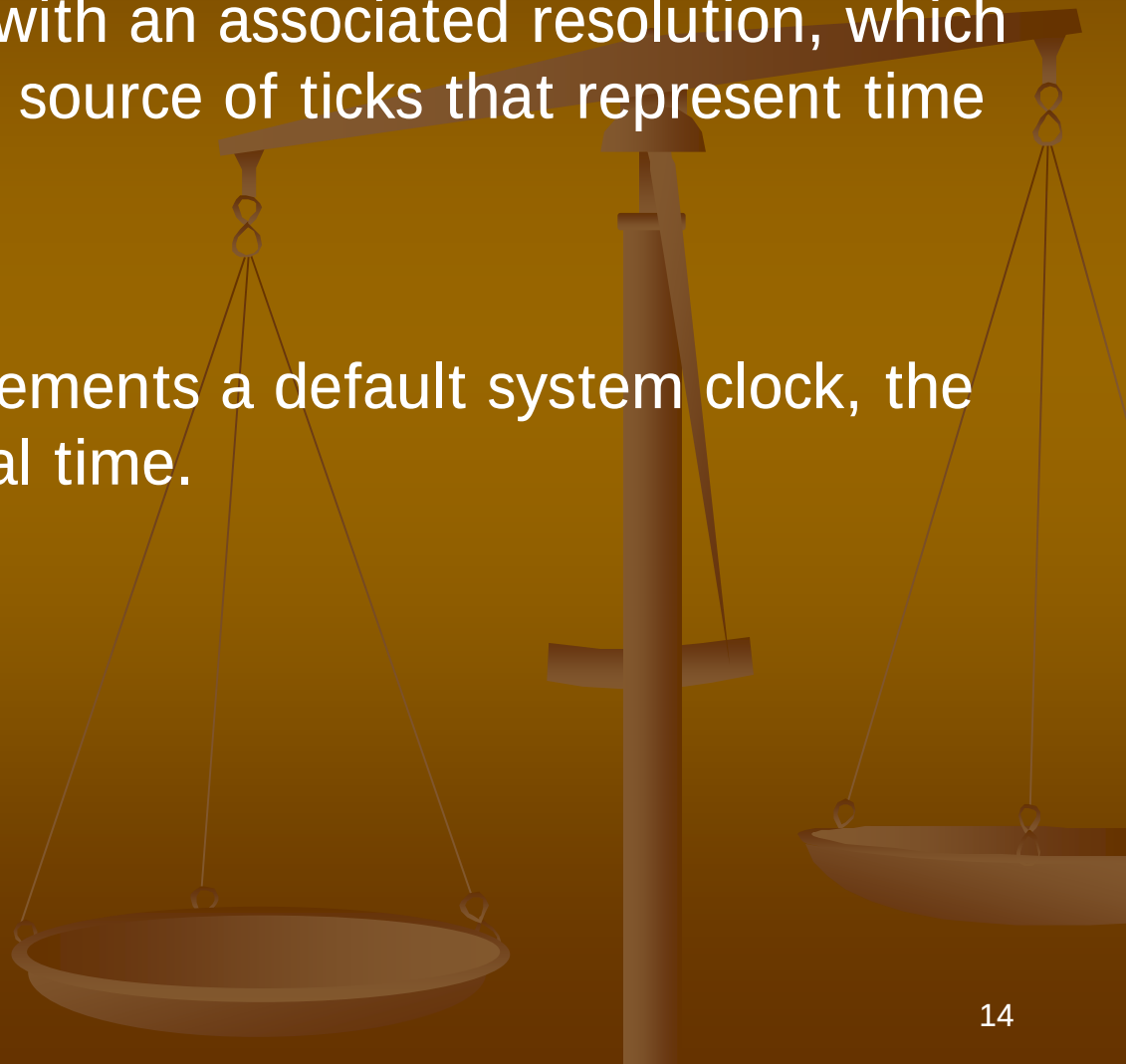
Counters

```
42  // Create the alarm.
43  cyg_alarm_create(      counter_hdl,
44                         alarm_handler,
45                         (cyg_addrword_t)index,
46                         &alarm_hdl,
47                         &alarm_obj );
49  // Initialize the alarm.
50  cyg_alarm_initialize( alarm_hdl,  //initialized using the alarm handle
51                         12,        // counter_hdl value
52                         6 );       // alarm trigger again interval
53
54  // Create and run the counter thread.
55  }
58  // Alarm handler.
60  void alarm_handler(
61                         cyg_handle_t alarm_handle,
62                         cyg_addrword_t data )
63  {
64        (unsigned long)data++;
65  }
```

Clocks

- A *clock* is a counter, with an associated resolution, which is driven by a regular source of ticks that represent time periods.

- The eCos kernel implements a default system clock, the RTC, which tracks real time.



Clocks

Item List 7.5 Kernel Clock API Functions

- Syntax: void **cyg_clock_create**(
  cyg_resolution_t resolution,
  cyg_handle_t *handle,
  cyg_clock *clock
  );

  Description: Construct a new clock with the specified resolution.

- Syntax: void **cyg_clock_delete**(
  cyg_handle_t clock
  );

  Description: Remove the specified clock.

- Syntax: void **cyg_clock_to_counter**(
  cyg_handle_t clock,
  cyg_handle_t *counter
  );

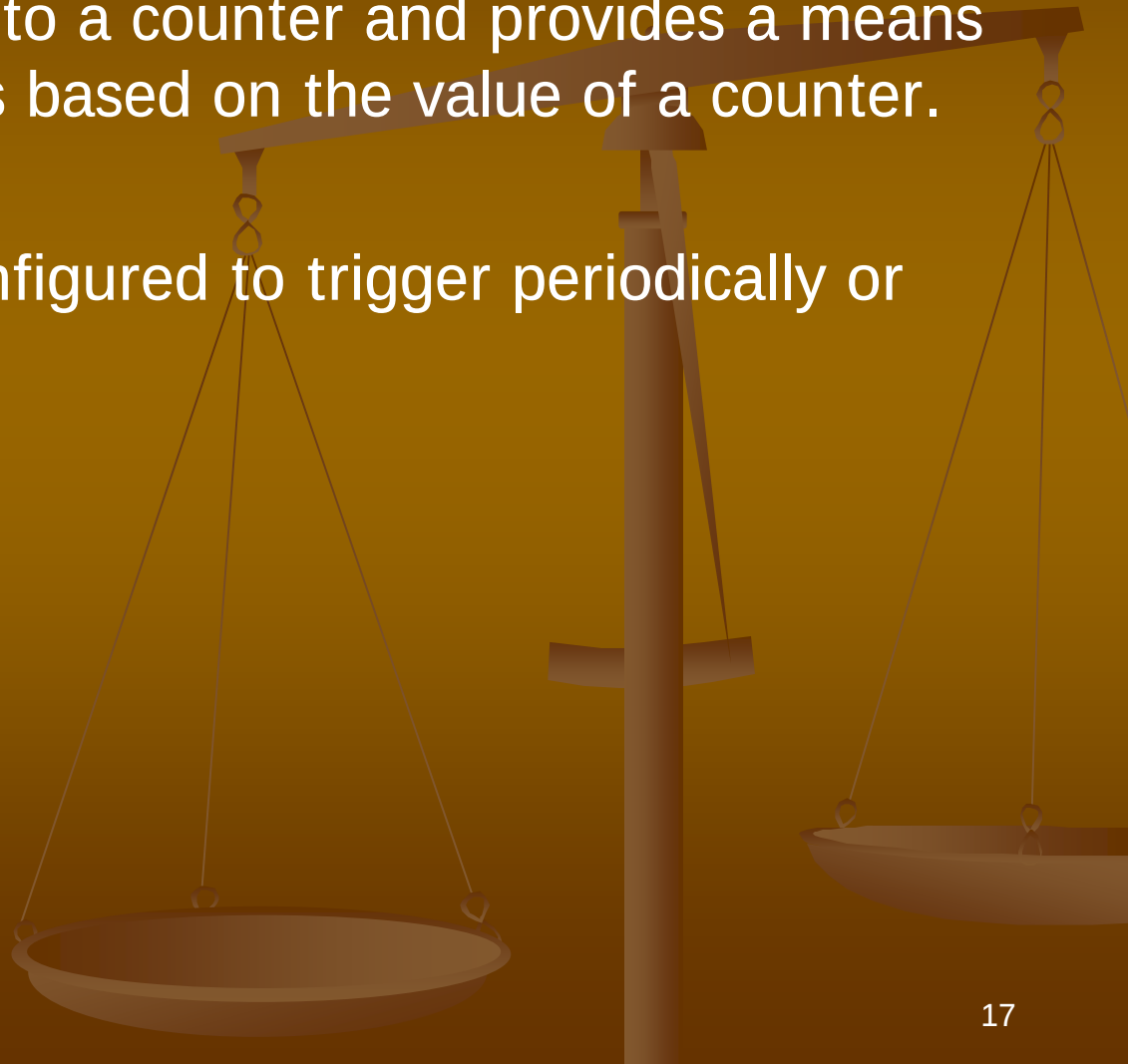
  Description: Converts a clock handle to counter handle allowing the use of kernel counter API functions.This gives access to the clock's attached counter.

Clocks

- Syntax: void **cyg_clock_set_resolution**(
  cyg_handle_t clock, cyg_resolution_t resolution
  );
  Description: Changes the resolution of the specified clock object. This function does not actually changethe behavior of the hardware driving the clock. Instead, cyg_clock_set_resolution synchronizes the kernel clock object to match resolution of the underlying hardware clock providing the ticks.
- Syntax: cyg_resolution_t **cyg_clock_get_resolution**(
  cyg_handle_t clock
  );
  Description: Returns the current resolution of the specified clock.
- Syntax: cyg_handle_t **cyg_real_time_clock**(
  void
  );
  Description: Returns a handle to the system real-time clock.
- Syntax: cyg_tick_count_t **cyg_current_time**(
  void
  );
  Description: Returns the real-time clock counter value in ticks.

Alarm

- An alarm is attached to a counter and provides a means for generating events based on the value of a counter.
- The event can be configured to trigger periodically or once.



Alarm

Item List 7.6 Kernel Alarm API Functions

- Syntax: void **cyg_alarm_create**(

```
cyg_handle_t counter,        //handle to counter which alarm is attached.
cyg_alarm_t *alarm_fn,       //pointer to alarm handler function.
cyg_addrword_t data,         //parameter passed into alarm handler.
cyg_handle_t *handle,        //pointer to the new alarm handle.
cyg_alarm *alarm             //pointer to the new alarm object.
);
```

  Description: Construct an alarm object that is attached to the specified counter. The alarm handler is called when the alarm triggers and executes in the context of the function that incremented the counter. The alarm does not run until after the call to cyg_alarm_initialize.

- Syntax: void **cyg_alarm_delete**(

```
cyg_handle_t alarm           //handle to the alarm
);
```

  Description: Disables the specified alarm and detaches it from the counter.

- Syntax: void **cyg_alarm_initialize**(

```
cyg_handle_t alarm,          //handle to the alarm
cyg_tick_count_t trigger,    //tick value that causes alarm to occur
cyg_tick_count_t interval    //tick value that causes alarm to reoccur
);
```

  Description: Initializes the specified alarm to trigger when the tick value is equal to the trigger parameter. If the interval parameter is set to zero, the alarm is disabled after it occurs once. Otherwise, the alarm reoccurs according to the interval parameter setting.

Alarm

- Syntax: void **cyg_alarm_enable**(

```
externC void cyg_alarm_enable( cyg_handle_t alarm )
{
    ((Cyg_Alarm *)alarm)->enable();
}

externC void cyg_alarm_disable( cyg_handle_t alarm )
{
    ((Cyg_Alarm *)alarm)->disable();
}
```

Des ase with the
n.

- Syn

Des ing. The
alarm can be re-enabled using the cyg_alarm_initialize or
cyg_alarm_enable functions.

Alarm – Using clocks and alarm

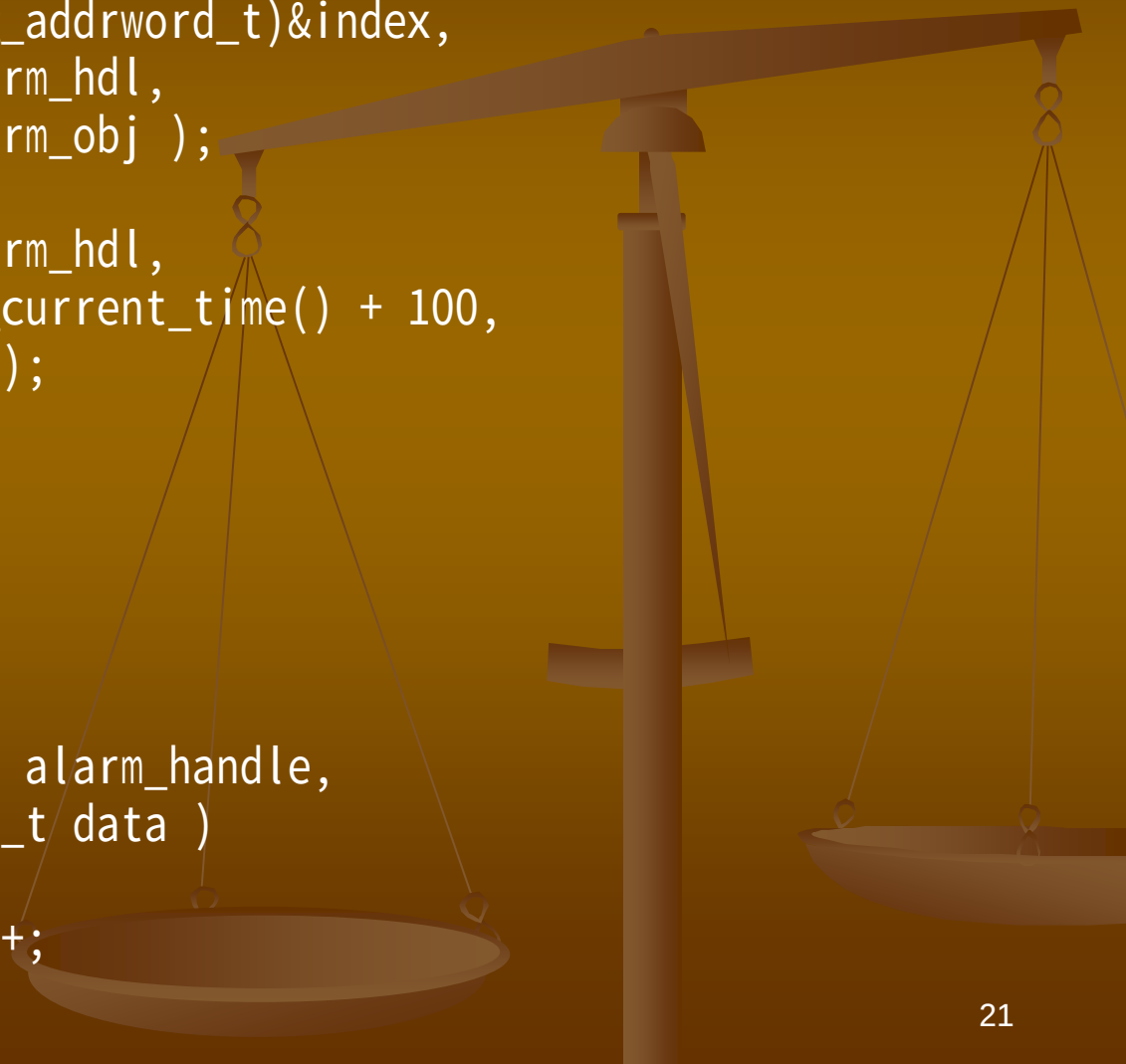
```
1  #include <cyg/kernel/kapi.h>
2
3  cyg_handle_t counter_hdl;
4  cyg_handle_t sys_clk;
5  cyg_handle_t alarm_hdl;
6  cyg_tick_count_t ticks;
7  cyg_alarm_t alarm_handler;
8  cyg_alarm alarm_obj;
9
10  unsigned long index;
```

```
externC void cyg_clock_to_counter(
    cyg_handle_t        clock,
    cyg_handle_t        *counter
)
{
    CYG_CHECK_DATA_PTR( counter, "Bad counter handle
pointer" );
    *counter = (cyg_handle_t)(Cyg_Counter *)clock;
}
```

```
19  cyg_clock_to_counter( sys_clk,
20                        &counter_hdl );
```

Alarm

```
22  cyg_alarm_create(   counter_hdl,
23                      alarm_handler,
24                      (cyg_addrword_t)&index,
25                      &alarm_hdl,
26                      &alarm_obj );
27
28  cyg_alarm_initialize(alarm_hdl,
29                      cyg_current_time() + 100,
30                      100 );
31  }
32
33  //
34  // Alarm handler.
35  //
36  void alarm_handler(
37              cyg_handle_t alarm_handle,
38              cyg_addrword_t data )
39  {
40      (unsigned long)data++;
41  }
```



Timers

- A *timer* is an alarm that is attached to a clock. There is a timer object defined by the kernel.
- eCos does not provide a formal implementation, or kernel API functions, for the timer object.
- Timers in the eCos system are used within the μ ITRON compatibility layer package.
- The μ ITRON package uses the timer object attached to the real-time clock for performing its needed timing related functions.

Asserts and Tracing

- ***Assert*** is a piece of code that checks, at run time, whether a condition is expected. If the condition is not expected,an error message can be output and the application is halted.

- ***Tracing*** allows the output of text messages at various points in the application’ s execution.

  This output enables you to follow the execution flow of a program or check a particular status

  when certain events occur.

Asserts and Tracing

- Both asserts and traces are defined as macros. The three basic assert macros are:

- **CYG_FAIL**—does not accept a condition as its first parameter. Instead, this macro outputs the standard message along with a possible user-defined message regardless of any conditions being met.
  - // unconditional failure; use like panic(), coredump() &c.
  - # define CYG_FAIL(_msg_)
  - CYG_MACRO_START
  - CYG_ASSERT_DOCALL(_msg_);
  - CYG_MACRO_END

Asserts and Tracing

- **CYG_ASSERT**—accepts a condition as its first parameter. Depending on the value of the condition, this macro outputs the standard message along with a possible user defined message.

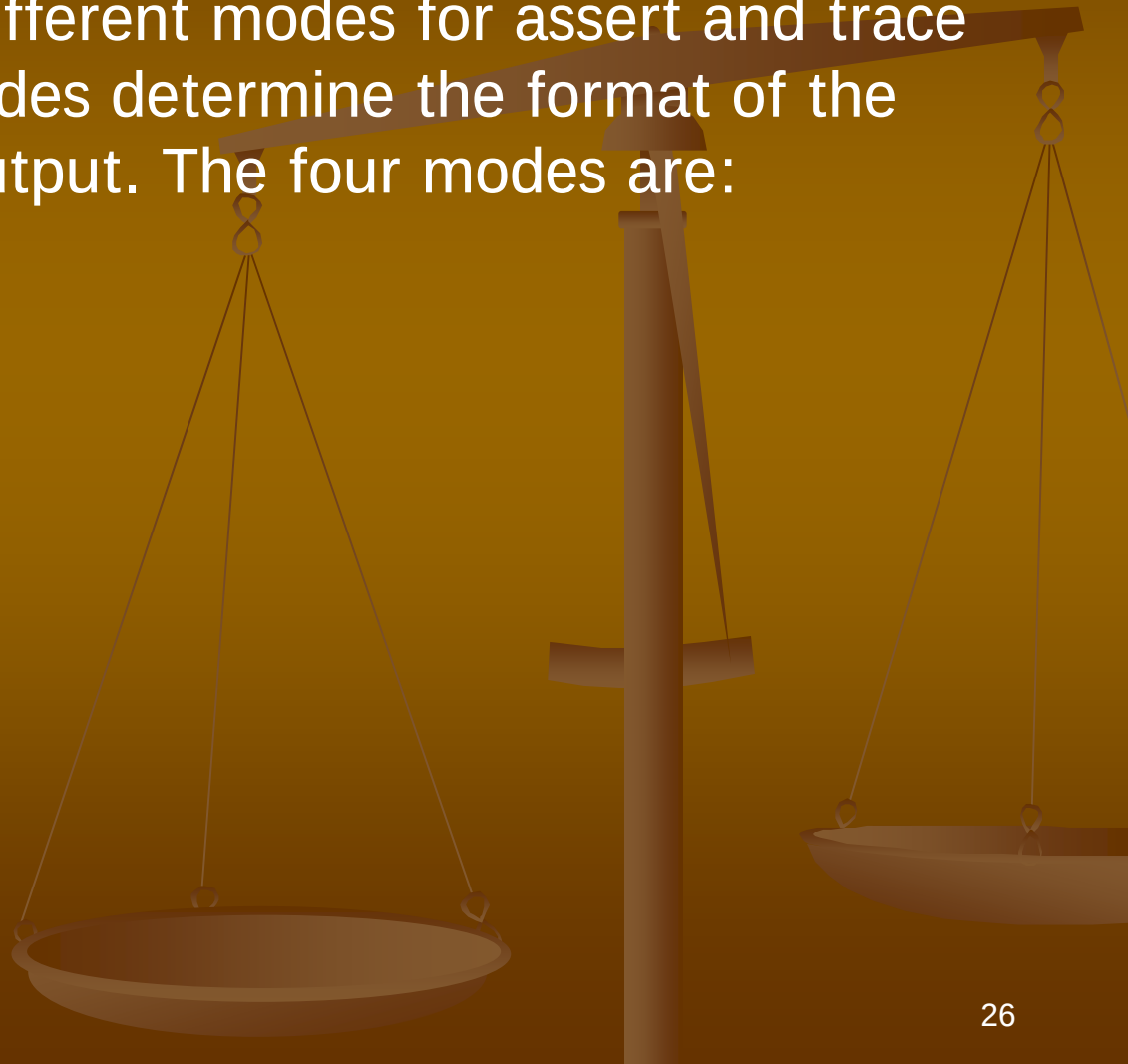
```
// conditioned assert; if the condition is false, fail.
# define CYG_ASSERT( _bool_, _msg_ )
        CYG_MACRO_START
        if ( ! ( _bool_ ) )
            CYG_ASSERT_DOCALL( _msg_ );
        CYG_MACRO_END
```

- **CYG_ASSERTC**—compact version of the assertion macro that outputs the standard message along with the resulting value of the first parameter.

```
# define CYG_ASSERTC( _bool_ )
        CYG_MACRO_START
        if ( ! ( _bool_ ) )
            CYG_ASSERT_DOCALL( #_bool_ );
        CYG_MACRO_END
```

Asserts and Tracing

- eCos supports four different modes for assert and trace messages. These modes determine the format of the information that is output. The four modes are:
  - Null
  - Simple
  - Fancy
  - Buffered



Asserts and Tracing

Item List 7.7 Assertion and Tracing Configuration Options

- Option Name: **Use Asserts**
  CDL Name: CYGDBG_USE_ASSERTS
  Description: Enables assertion code checking and output messages.
  SubOption:
    - Preconditions
    - Postconditions
    - Loop Invariants
    - Use Assert Text
- Option Name: **Use Tracing**
  CDL Name: CYGDBG_USE_TRACING
  Description: Enables trace code output messages.
  SubOption:
    - Trace Function Reports
    - Use Trace Text
- Option Name: **Null Output**
  CDL Name: CYGDBG_INFRA_DEBUG_TRACE_ASSERT_NULL
  Description: Disables output messages for tracing and assertion functions. This enables breakpoints to be placed in the trace and assert routines during a debug session instead of interpreting output messages.

Asserts and Tracing

- Option Name: **Simple Output**

  CDL Name: CYGDBG_INFRA_DEBUG_TRACE_ASSERT_SIMPLE

  Description: Specifies the message format for assert and trace output. This format includes the thread identification number, the filename, line number, routine name, and any additional text message.

- Option Name: **Fancy Output**

  CDL Name: CYGDBG_INFRA_DEBUG_TRACE_ASSERT_FANCY

  Description: Specifies the message format for assert and trace output. This format includes the thread identification number, the filename, line number, routine name, and any additional text message.

Asserts and Tracing

- Option Name: **Buffered Tracing**

  CDL Name: CYGDBG_INFRA_DEBUG_TRACE_ASSERT_BUFFER

  Description: Allows tracing and assertion messages to be stored in a buffer. These messages are output when CYG_TRACE_PRINT is called. Suboptions define the buffer size and whether the buffer wraps, halts, or outputs when it is full. The trace buffer can also be configured to output when an assertion occurs.

- Option Name: **Use Function Names**

  CDL Name: CYGDBG_INFRA_DEBUG_FUNCTION_PSEUDOMACRO

  Description: Allows trace and assert macros to include the function name in output messages. Although this is helpful to read during debug,this option increases the code size.

Asserts and Tracing

Table 7.1 Trace Macros

Macro Name	Description
CYG_TRACE0 through CYG_TRACE8	First parameter is a Boolean that determines whether the trace message is output. The other possible arguments are output using printf-style formatting.
CYG_TRACE1X through CYG_TRACE8X CYG_TRACE1Y through CYG_TRACE8Y CYG_TRACE1D through CYG_TRACE8D	First parameter is a Boolean that determines whether the trace message is output. X—outputs arguments using `%08x` format. Y—outputs arguments using `%x` format. D—outputs arguments using `%d` format.

CYG_TRACE1XV through CYG_TRACE8XV
CYG_TRACE1YV through CYG_TRACE8YV
CYG_TRACE1DV through CYG_TRACE8DV
	First parameter is a Boolean that determines whether the trace message is output. X, Y, and D have the same formats as defined previously. V causes the argument name to be output in the trace message. For example:
	`CYG_TRACE1XV(var);`
	would output the following trace message:
	`TRACE:file.c[8]rout(): var=12`

Asserts and Tracing

Table 7.1 Trace Macros *(Continued)*

Macro Name	Description
CYG_TRACE1XB through CYG_TRACE8XB CYG_TRACE1YB through CYG_TRACE8YB CYG_TRACE1DB through CYG_TRACE8DB	B means that there is no first parameter Boolean; therefore, using this trace macro always results in a message output. X, Y, and D have the same formats as defined previously.
CYG_TRACE1XVB through CYG_TRACE8XVB CYG_TRACE1YVB through CYG_TRACE8YVB CYG_TRACE1DVB through CYG_TRACE8DVB	B means that there is no first parameter Boolean; therefore, using this trace macro always results in a message output. X, Y, and D have the same formats as defined previously. V causes the argument name to be output in the trace message.

Asserts and Tracing

- The trace macros provide a means for tailoring the level of trace messages within an application.This allows control of the amount of output messages during runtime. The trace level can be controlled by the first parameter passed into the trace macro.

```
 1  #include <cyg/infra/cyg_trac.h>
 2
 3  static int trace_level = 1;
 4
 5  #define TL1 ( 0 < trace_level )
 6  #define TL2 ( 1 < trace_level )
 7
 8  void my_routine(
 9          unsigned long index)
10  {
11     unsigned char v1, v2, v3;
12
13     index++;
14
15     // Processing using local variables v1, v2, and v3.
16
17     CYG_TRACE1( TL1, "Index: %d", index );
18
19     CYG_TRACE3( TL2, "Locals: %d %d %d", v1, v2, v3 );
20  }
```

Code Listing 7.3 Trace output runtime control example.

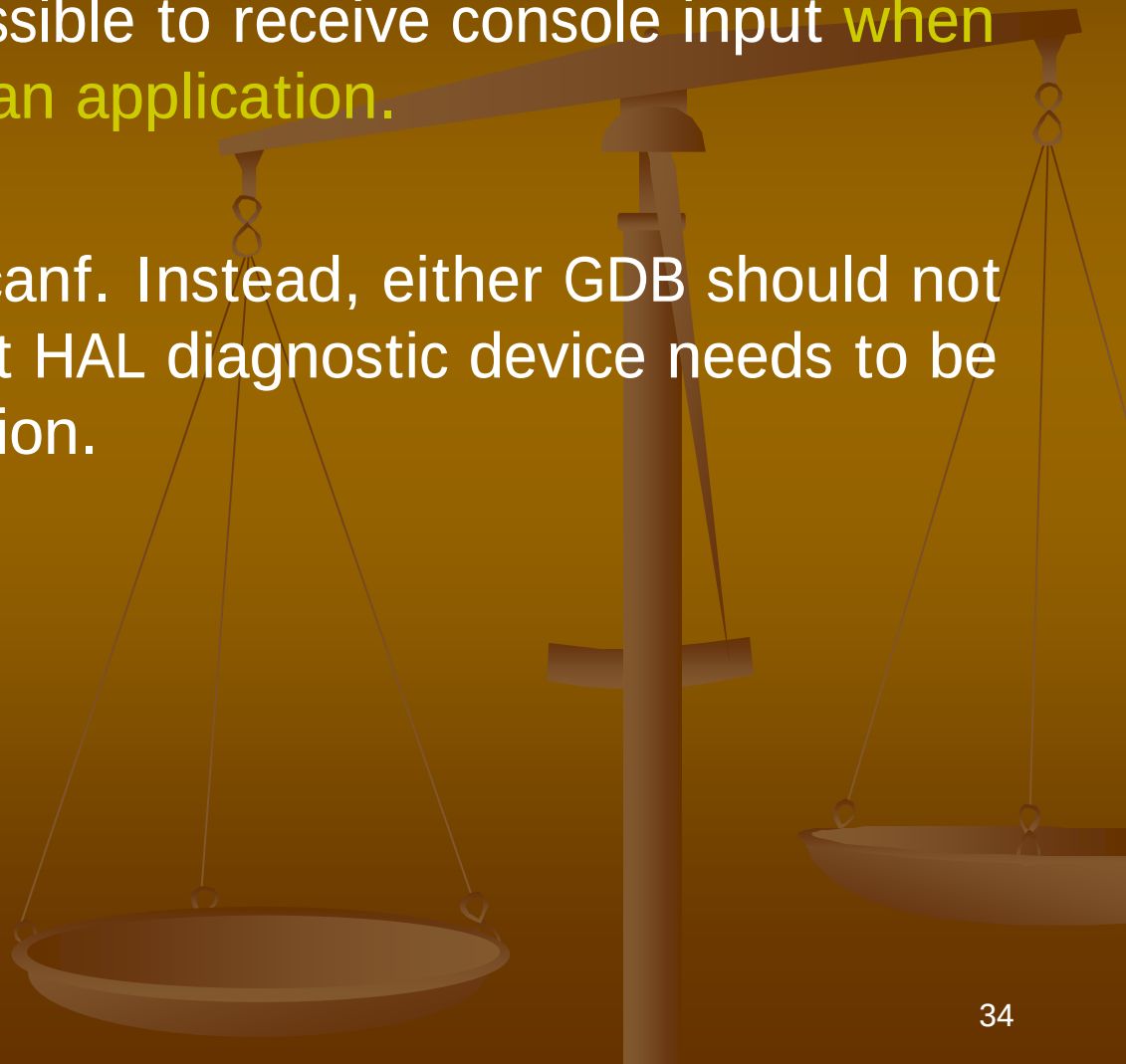
ISO C and Math Libraries

- The ISO C standard output uses the diagnostic console device provided by the HAL. This is controlled by the *Default Console Device* configuration option.

- The HAL diagnostic device uses a polling mode for communication.
  - This means that output can be slow especially when communicating with a GDB host, which involves utilizing the GDB remote protocol. Input can be processing intensive, meaning that other threads might not have an opportunity to run.

ISO C and Math Libraries

- It is currently not possible to receive console input when using GDB to debug an application.
- for example, using scanf. Instead, either GDB should not be used or a different HAL diagnostic device needs to be used for communication.



ISO C and Math Libraries

Item List 7.8 ISO C Library Configuration Option Packages

- Option Name: **ISO C Library Internationalization Functions**

  CDL Name: CYGPKG_LIBC_I18N

  Description: Allows configuration of ISO C internationalization functions such as ctype.h and locale-related functions.

- Option Name: **ISO C Library setjmp/longjmp Functions**

  CDL Name: CYGPKG_LIBC_SETJMP

  Description: Allows configuration of the build options for the setjmp.h functions.

- Option Name: **ISO C Library Signal Functions**

  CDL Name: CYGPKG_LIBC_SIGNALS

  Description: Specifies the configuration of the signal functionality within the ISO C library, such as the signal and raise functions.

- Option Name: **ISO C Environment Startup/Termination Functions**

  CDL Name: CYGPKG_LIBC_STARTUP

  Description: Controls the configuration of startup, such as the main entry point, and termination, such as exit, for full ISO C ompatibility.

ISO C and Math Libraries

- Option Name: **ISO C Library Standard Input/Output Functions**
  CDL Name: CYGPKG_LIBC_STDIO
  Description: Allows configuration of the input/output functions found in the stdio.h library file.

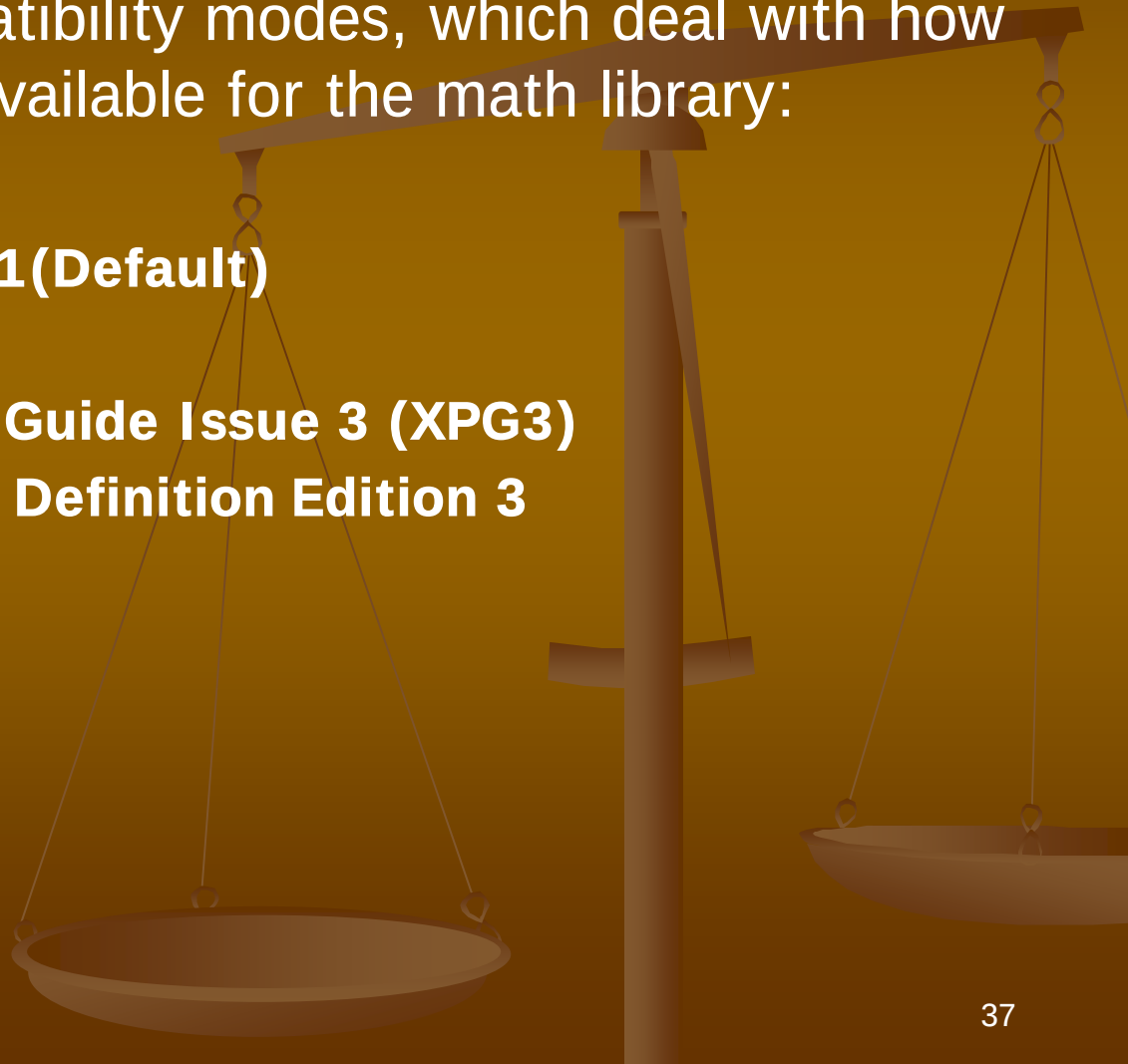
- Option Name: **ISO C Library General Utility Functions**
  CDL Name: CYGPKG_LIBC_STDLIB
  Description: Specifies the configuration options for the utility functions found in the stdlib.h library file.

- Option Name: **ISO C Library String Functions**
  CDL Name: CYGPKG_LIBC_STRING
  Description: Controls the configuration options for the string functions found in the string.h library file.

- Option Name: **ISO C Library Date and Time Functions**
  CDL Name: CYGPKG_LIBC_TIME
  Description: Configures the ISO C date and time functions.

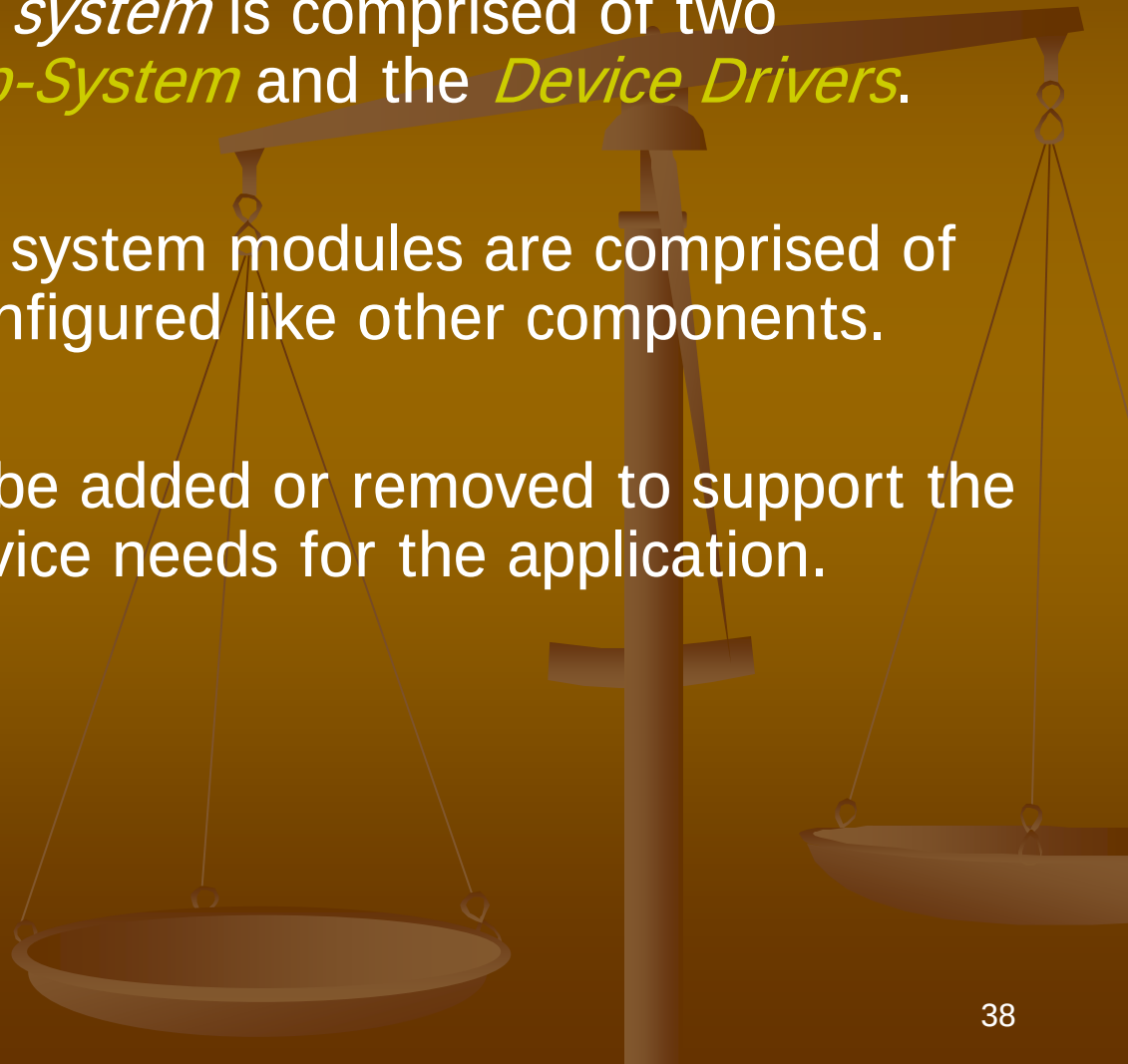
ISO C and Math Libraries

- There are four compatibility modes, which deal with how errors are handled, available for the math library:
  - **ANSI/POSIX 1003.1(Default)**
  - **IEEE-754**
  - **X/Open Portability Guide Issue 3 (XPG3)**
  - **System V Interface Definition Edition 3**



I/O Control System

- The eCos *I/O control system* is comprised of two modules, the *I/O Sub-System* and the *Device Drivers*.
- The eCos I/O control system modules are comprised of packages that are configured like other components.
- These packages can be added or removed to support the specific hardware device needs for the application.



I/O Control System

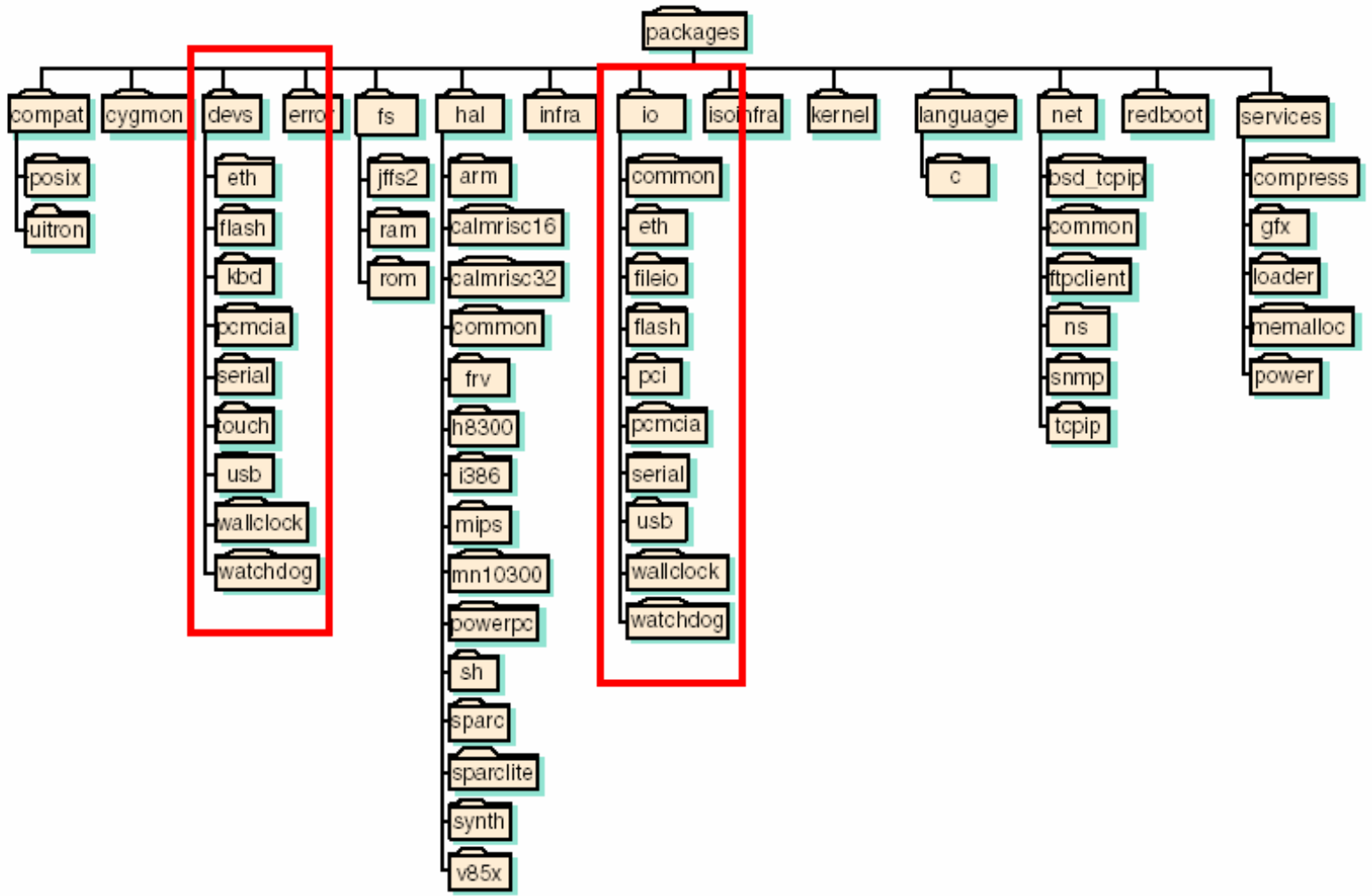


Figure 1.3 High-level component repository directory structure snapshot.

I/O Control System

- The I/O control system design uses a layered approach.
- This example is not of any particular platform; it is intended to show the software layers within the I/O control system.

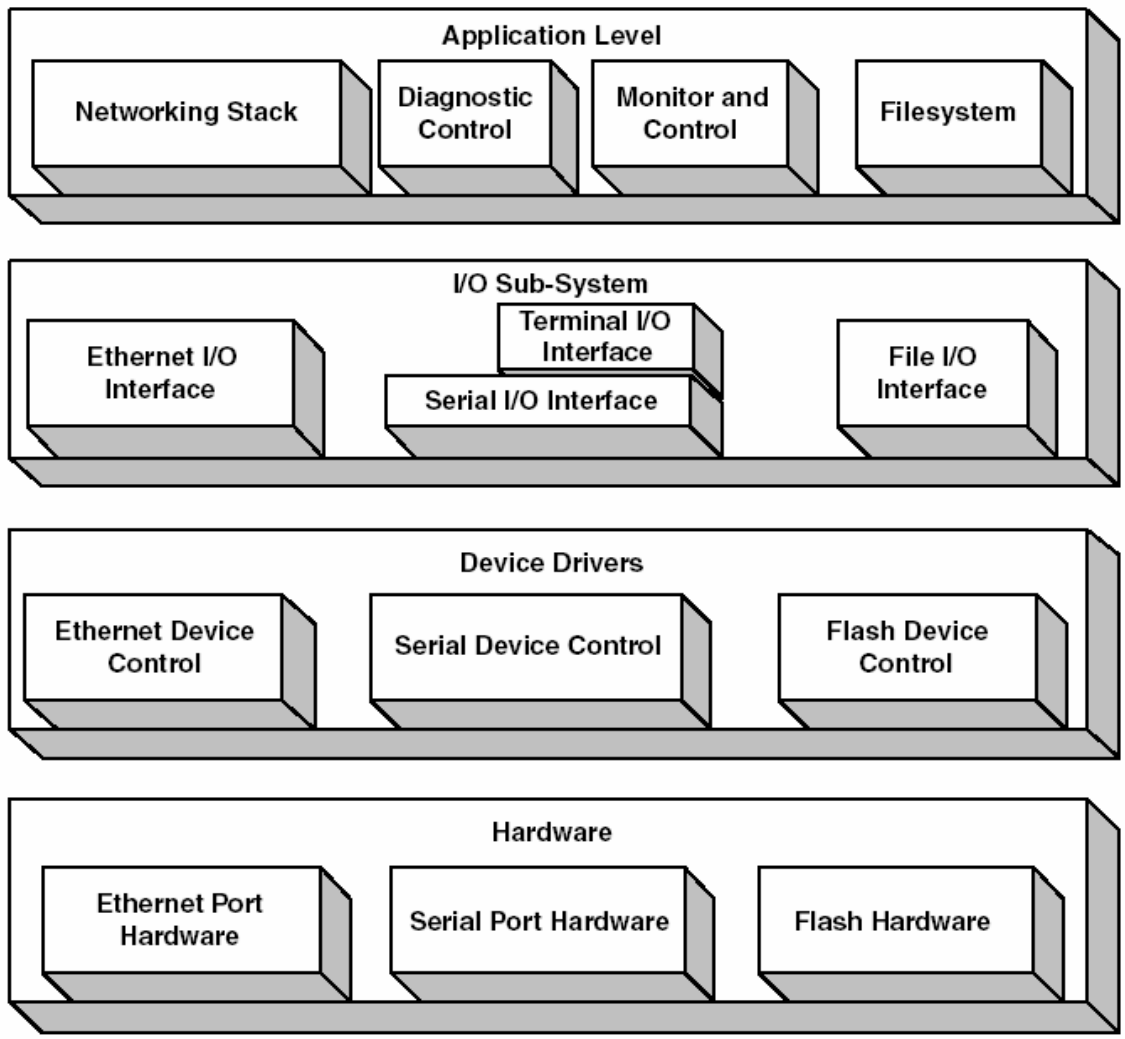
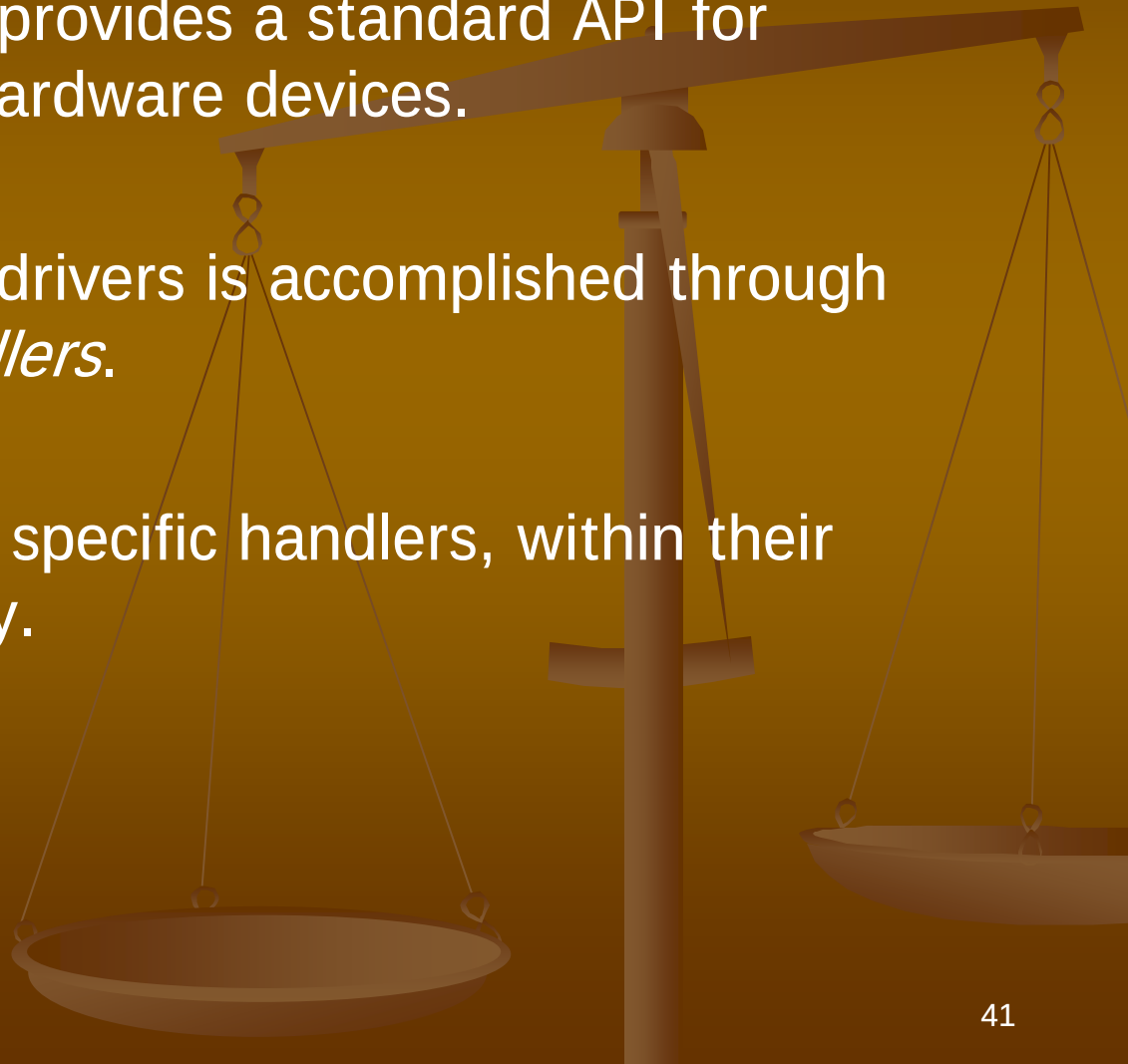


Figure 7.1 Example eCos I/O control system.

I/O Control System_I/O Sub-system

- The I/O Sub-System provides a standard API for accessing low-level hardware devices.
- Access to the device drivers is accomplished through functions called *handlers*.
- Device drivers define specific handlers, within their device I/O table entry.



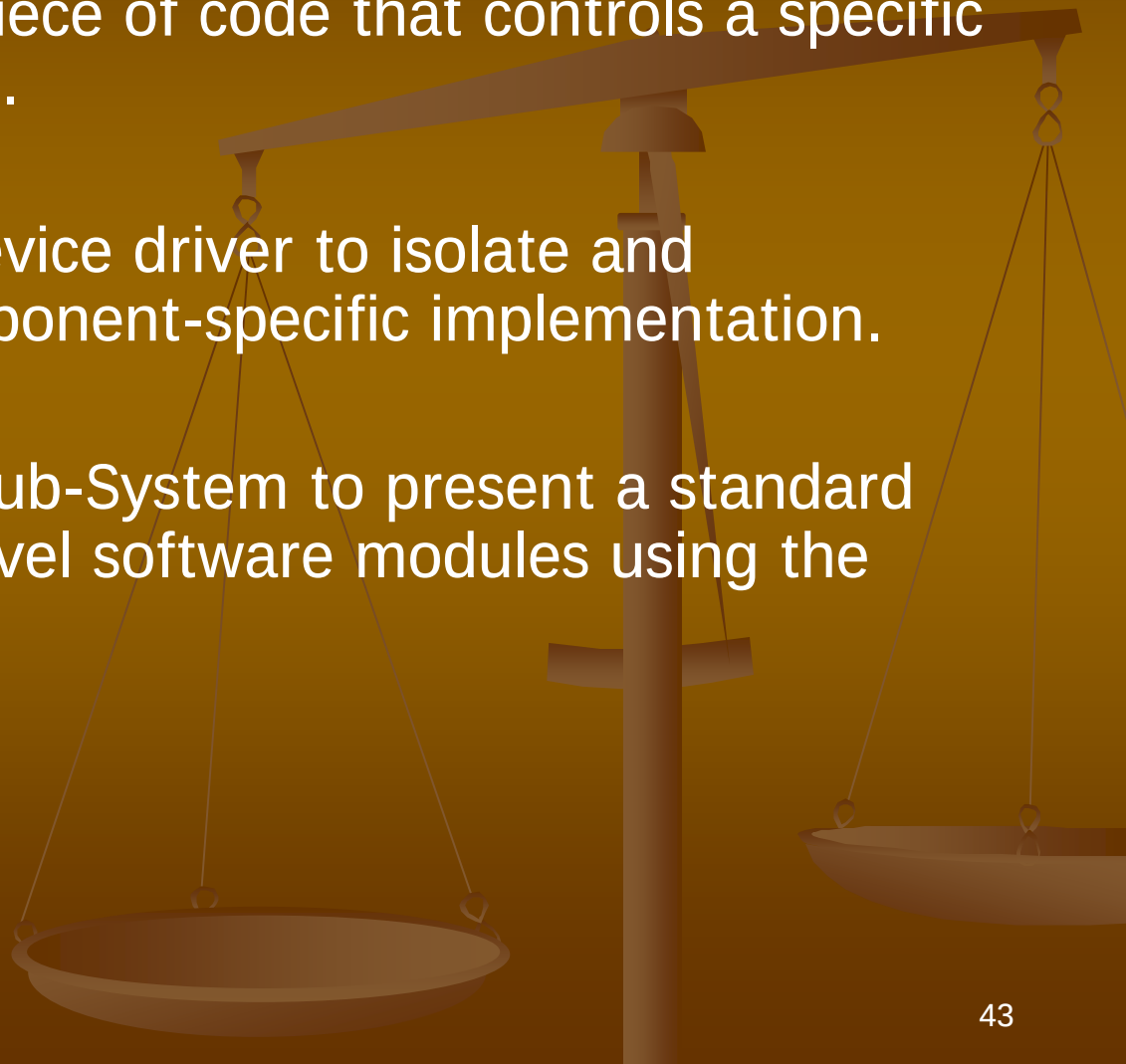
I/O Control System_I/O Sub-system

```
 1   #include <cyg/kernel/kapi.h>
 2   #include <cyg/io/io.h>
 3   #include <cyg/infra/diag.h>
 4
 5   //
 6   // Main starting point for the application.
 7   //
 8   void cyg_user_start( void )
 9   {
10       cyg_io_handle_t tty_hdl;
11       int err;
12       char output_string[] = "Hello There!!!\n";
13       cyg_uint32 output_len = sizeof( output_string );
14
15       err = cyg_io_lookup( "/dev/tty0", &tty_hdl );
16
17       if ( err )
18       {
19           diag_printf( "ERROR opening device tty0.\n" );
20           return;
21       }
22
23       err = cyg_io_write( tty_hdl, output_string, &output_len );
24
25       if ( err )
26       {
27           diag_printf( "ERROR writing to device tty0.\n" );
28           return;
29       }
30   }
```

Code Listing 7.4 I/O Sub-System API example code.

I/O Control System_Device Driver

- A *device driver* is a piece of code that controls a specific hardware component.
- It is the job of the device driver to isolate and encapsulate the component-specific implementation.
- This allows the I/O Sub-System to present a standard interface to higher-level software modules using the device I/O table.



I/O Control System_Device Driver

- Device drivers use an API for interacting with the kernel and HAL.
- Ex.
  - The cyg_drv_isr_lock and cyg_drv_isr_unlock functions are defined the same as the cyg_interrupt_disable and cyg_interrupt_enable functions.
- The difference between using the kernel API and the driver API is that the driver API is guaranteed to be present in configurations where the eCos kernel is not present. This makes the drivers more portable.

END~