TiROS Reference Manual

July 20, 2007

Contact:
Ratish J. Punnoose

Contents

1 TiROS Overview **1**

2 TiROS Usage **5**

2.1 Getting Started with TiROS 5

3 TiROS Programming Guide **11**

3.1 TiROS Configuration 11

3.2 Application Programming Interface (API) 14

3.3 Function Options 15

3.4 Critical Sections 17

3.5 Usage with Interrupt Service Routines 19

3.6 OS initialization and running 22

3.7 Task creation and management 23

3.8 Time services provided by the OS 28

3.9 Utility functions for time operations 30

3.10 Mutexes 33

3.11 Counting semaphores 36

3.12 Message Queues 40

3.13 Event Flags 44

3.14 Advanced Features and Debugging 47

3.15 Return Codes. 51

4 Mutual Exclusion (Mutex) **55**

4.1 What is a Mutex? 55

4.2 Mutexes vs. Critical Sections . . . 57
4.3 Pitfalls of mutexes . . . 59
4.4 Priority Inheritance Protocol . . . 61
4.5 Priority Ceiling Protocol . . . 67
4.6 Rules for using mutexes in TiROS . . . 71

5 Porting to other hardware **73**
5.1 Porting Guide . . . 73
5.2 Define basic types and OS configuration for your port . . . 74
5.3 Critical Sections and Interrupts . . . 77
5.4 Context Switching Functions . . . 79
5.5 Port specific time functions . . . 82
5.6 Setup for the kernel trap . . . 84

6 Port Documentation **85**
6.1 HAL for MSP430 family with mspgcc compiler . . . 85
6.2 HAL for MSP430 family with IAR compiler . . . 87
6.3 Posix HAL . . . 88

7 License and Usage **91**
7.1 License . . . 91

Chapter 1

TiROS Overview

Overview

TiROS (Tickless Real-Time Operating System) is a pre-emptive priority based real-time task-scheduler. It can be used by developers to facilitate multi-tasking on microcontroller based embedded systems that have very limited resources.

It is available under a modified GPL license (eCos open source license) that allows it to be used for commercial purposes at no cost. It requires less than 200 bytes of RAM for typical uses (Exact number is dependent upon number of tasks and hardware). Thus it can be used where sophisticated free OSes like Linux or eCOS cannot be used. It is more closely comparable in features and resource usage to FreeRTOS. At the same time, it provides real-time capabilities and other features that are useful in an embedded system.

- **Hard Priorities:** TiROS schedules tasks in a **deterministically predictable** manner. Every task has a unique priority. The highest priority ready task that is active is always the one running. It stays active until it explicitly sleeps, waits, or is blocked for a resource.
- **Tickless Scheduling:** TiROS avoids most context-switching overhead costs by eliminating periodic ticks. In most embedded real-time OSes, there is a trade off between high time-resolution (by increasing tick frequency) and overhead. TiROS does not use ticks. TiROS supports high resolution time (depending on the hardware capabilities) and only wakes when needed. This reduces overhead. To accomplish this, it takes advantage of the compare features of timers in modern micro-controllers. Even in processors where there is no hardware support for this, software based virtual compare can be performed.
- **Low Memory Usage:** TiROS was designed to have an extremely low RAM footprint. Internally, all data structures use static memory allocation. There is no memory-manager provided with TiROS. With the appropriate mutexes or critical sections, the user can still use other memory managing code.
- **Multi-tasking Primitives:** Tasks have access to synchronization primitives such as mutexes, counting semaphores, message queues, and event flags.

- **Deadlock Prevention:** TiROS has been designed to provide reliable real-time capabilities. TiROS has two mechanisms for preventing priority inversion.

  1. Immediate Priority Ceiling Protocol.
  2. Priority Inheritance Protocol.

  The implementation of these mechanisms is described in the TiROS documentation.

- **Low overhead interrupts:** On hardware that supports software initiated interrupts (or software traps) interrupt service routine can be written with low overhead because they do not have to save and load the context upon every interrupt invocation.

- **Nested interrupts** are supported as a configuration option, but they are discouraged. They are problematic in that, in extreme cases, they can lead to a stack overflow. The size of a task stack is determined by the stack usage of the task, the stack usage of OS calls, and stack usage of any interrupt handlers that occur during the task. If interrupts are nested, the stack can potentially expand in an unbounded manner depending on the nesting level. On most processors, when an interrupts occurs and the program counter vectors to the interrupt, global interrupts are disabled. They can be explicitly re-enabled within the ISR to nest interrupts. However, with some processors like the 8051, global interrupts are not disabled. This means that an nested interrupt can occur just as the first ISR is being started. On such processors, there is no good way to infer the nesting level correctly. The best that can be done is to have the first instruction within the ISR disable global interrupts. But there is still a small period of contention (one instruction time) where a nested interrupt can occure before the first instruction within the ISR is executed.

Hardware Support

TiROS is written in "C" and is designed to be easy to port to different hardware. The hardware-specific portion of the operating system is separated into a HAL (Hardware Abstraction Layer). The current version of TiROS is available with three HALs;

- MSP430 family with GCC Compiler.
- MSP430 family with IAR Compiler.
- Posix HAL that allows for the use of TiROS on any Posix compliant operating system such as Linux. This allows for easy simulation and testing of software.

Detailed Description

1. Getting Started with TiROS.
2. Programming Guide
   - TiROS Configuration.
   - Application Programming Interface (API).

3. Mutual Exclusion (Mutex) Implementation.
4. Hardware Support
    - Porting Guide.
    - HAL for MSP430 family with mspgcc compiler.
    - HAL for MSP430 family with IAR compiler.
    - Posix HAL.
5. License.

Chapter 2

TiROS Usage

2.1 Getting Started with TiROS

Description of files in TiROS distribution

The distributed directory tree is as shown below. You can move things around as needed and reflect the changes in your Makefile or other build tool.

```
<root>
  +-- inc
  |   +-- tiros
  |   |   |-- tiros.h
  |   |   |-- tr_types.h
  |   |   |-- tr_util.h
  |   |   `-- tr_time.h
  |   |
  |   `-- util.h
  |
  |
  +-- src
      +-- tiros
      |   |-- tiros.c
      |   |-- tr_ll.h
      |   |-- tr_int.h
      |   |-- tr_util.c
      |   |-- tr_llmgr.h
      |   |-- tr_debug.h
      |   |-- tr_debug.c
      |   |
      |   +-- port
      |       +-- template
      |       |       |-- porttime.h
      |       |       `-- tr_port.h
      |       |
      |       +-- msp430_gcc
      |       |       |-- porttime.h
```

```
|       |       |-- tr_port.h
|       |       `-- tr_port.c
|       |
|       +-- msp430_iar
|       |       |-- porttime.h
|       |       |-- tr_port.h
|       |       `-- tr_port.c
|       |
|       +-- posix
|       |       |-- porttime.h
|       |       |-- tr_port.h
|       |       `-- tr_port.c
|       |
|       `-- other_hardware_ports
|
|
|
+-- projects
|   |-- os_porting_help
|   |
|   +-- os_examples
|   |
|   +-- tiros_parse
|   |
|   `-- common.mk
|
|
+- your project_directory
|   |-- Makefile
|   |-- proj_config.h
|   `-- main.c
|
|
`-- util
```

A typical build session will use the inc/tiros directory, the src/tiros directory and ONE (and only one at a time) of the ports as needed. The files relevant to the user are listed below.

tiros.h: This contains the application programming interface for TiROS. This file will be included via an include statement within your "c" files. This is the only file that needs to be included from your user files for normal use.

tr_types.h: This contains definitions of the data types used by TiROS. It also contains defaults for certain TiROS values. The defaults can be overridden.

tiros.c: This contains the entire implementation of the rtos. This will be compiled and linked with your project.

tr_util.h, tiros_util.c: These provide some utility functions that may be used for debugging. These are not normally needed.

tr_time.h: This contains the declaration of time addition and subtraction routines. This is included through tiros.h and thus does not have to be included through your program.

tr_int.h: This provides definitions of internal data structures used by TiROS and declarations of TiROS functions that can be called from the hardware specific port.

tr_debug.h: Internal state information can be obtained from TiROS using this debug interface. This is only needed for detail debugging. This has to be explicitly included. It is not automatically included by tiros.h

tr_debug.c: This contains the implementation of the debug interface.

tr_ll.h: Inline implementation of a static memory list used by TiROS for queue management.

tr_llmgr.h: Abstraction layer for accessing the ready and wait queues.

tr_port.h: This contains the platform specific definitions for the hardware abstraction layer. This does not need to be included explicitly as tiros.h already includes this.

tr_port.c: There may be one or more files in the port directory that have to be compiled and linked with your project. The port instructions provide details for your specific port.

porttime.[ch]: The port will implement the time addition and subtraction routines that have been declared in tr_time.h. The location of this code may be port specific.

proj_config.h: This is your per project configuration file. TiROS configuration directives can be placed here. These will override the TiROS defaults. This way, no change has to be made to the TiROS source itself.

projects/os_porting_help: This directory contains two projects that can be aid in betting your own custom hardware port working.

projects/os_examples: This directory contains several examples that demonstrate the use of TiROS.

projects/common.mk: Common component of the makefiles used by all the examples.

projects/tiros_parse: This is a program that can decode and display the debug output from TiROS. This runs on the host, not the embedded target. It can take the debug input stream from a file (or serial port) or standard input.

inc/util : Definitions for some utility functions that are used by the examples (not essential to TiROS).

src/util : Implementation of utility functions (not essential to TiROS).

The order of header file inclusion is given in the following figure.

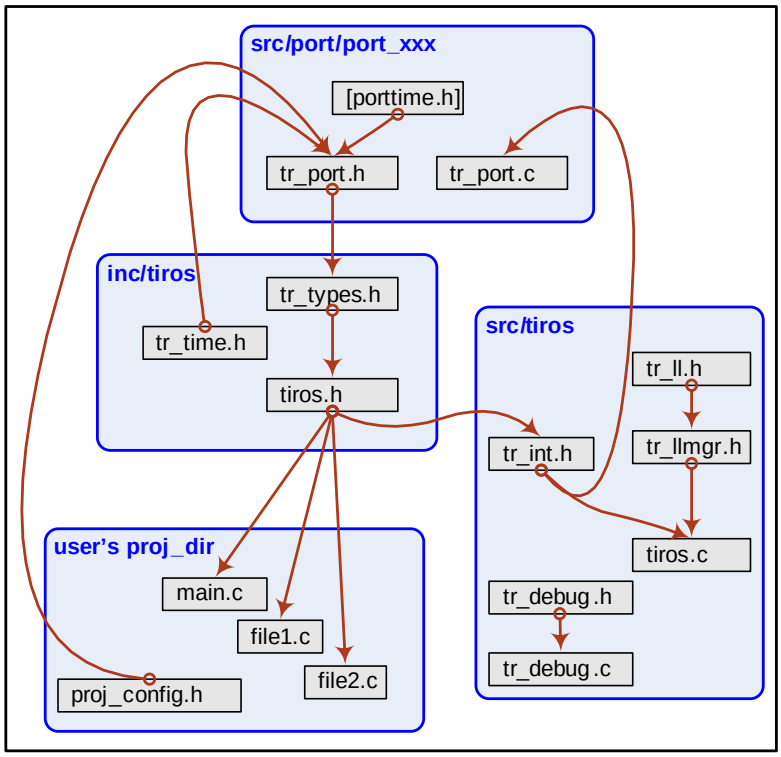

Figure 2.1: Include File Hierarchy

The example projects provided have Makefiles setup the directories to be included. That can be used as a template to set it up for your directory tree. Some build tools/compilers have trouble referencing include files from different directories. In this case, you can copy all the relevant files (src/tiros/∗, inc/tiros/∗ , src/tiros/port/port_xxx/∗) to a single directory.

Running Examples

The easiest way to experiment with TiROS is to run it on a desktop host running Linux or running Cygwin in a Windows environment. The Posix hardware port supplied with TiROS allows you to run TiROS within a posix process.

The provided examples in src/projects/os_examples can be compiled using the default Makefiles. The target compilation platform can be set in common.mk (look for DEFAULT_PLATFORM=xxx and set that to posix). Alternatively, set PLATFORM=posix in the Makefile in each individual directory.

There are several examples provided in the os_examples directory. Go to any one of them using a command shell and type make clean; make. This should create a subdirectory called "objects-release" and the executable should be found there. Here is a sequence of steps to run the example in os_sleep_example:

```
{prompt:} cd src/projects/os_examples/os_sleep_example;
{prompt:} make clean;
{prompt:} make;
{prompt:} ./objects-release/os
```

This example should now run.

Chapter 3

TiROS Programming Guide

3.1 TiROS Configuration

Main Configuration Options

These configuration options can be placed in the proj_config.h file.

TIROS_MAX_ISR_NESTING [1]: Define the maximum nesting level for ISRs. TiROS supports nested interrupts. By default, TIROS_MAX_ISR_NESTING is set to 1. This specifies that only one ISR will run at a time (interrupts should not be enabled within ISRs.) TiROS cannot enforce this, so the user should ensure that interrupts are not reenabled within an ISR beyond this defined level.

See also:

Usage with Interrupt Service Routines.

TIROS_MAX_PROCS: (Integer value) The maximum number of tasks. This should also include the idle task.

TIROS_ENABLE_MUTEX [1 | 0] : By default mutexes are enabled. If they should be disabled, then define TIROS_ENABLE_MUTEX to be 0 in the proj_config.h file.

TIROS_MAX_MUTEXES [255]: Maximum number or mutexes that can be held by a single task. This can be overridden by defining TIROS_MAX_MUTEXES in proj_config.h

TIROS_ALLOW_SLEEP_W_MUTEX [0 | 1] : By default TiROS does not allow tasks that are holding a mutex to sleep. An error code is returned. This is done to prevent unbounded locking. This behaviour can be overridden by defining this directive to 1 in the user's configuration file.

TIROS_PRIO_INHERIT_PROTOCOL: Define this in the user configuration file, if priority inheritance will be used. By default, priority ceiling will be used.

TIROS_ENABLE_CSEM [1 | 0] : By default counting semaphores are enabled. If they should be disabled, then define TIROS_ENABLE_CSEM to be 0 in the proj_config.h file.

TIROS_ENABLE_MSGQ [1 | 0] : By default message queues are enabled. If they should be disabled, then define TIROS_ENABLE_MSGQ to be 0 in the proj_config.h file.

TIROS_ENABLE_EFLAG [1 | 0] : By default event flags are enabled. If they should be disabled, then define TIROS_ENABLE_EFLAG to be 0 in the proj_config.h file.

TIROS_USER_CTXT_UPCALL: This can be used to perform user defined functions when tasks are switched. This is an upcall made by the scheduler upon task switch. If this is defined, the user_-ctxt_upcall function defined in the API MUST be implemented by the user. This feature can be used in imaginative ways to set task-specific settings. For example, a user can implement per task power control.

See also:

Advanced Features and Debugging.

TIROS_DEBUG_LEVEL 0-4 [0]: Debugging messages can be output from TiROS by setting TIROS_-DEBUG_LEVEL.

```
0 - No debugging messages.
1 - Critical error messages.
2 - Adds warning mesages.
3 - Adds informational messages.
4 - Adds trace messages.
```

The debugging messages are output by calling the putchar() function. This function may be adapted by the user to direct the output to a serial port, or to a file depending on the hardware platform.

See also:

Advanced Features and Debugging.

TIROSINT_DATA_DEBUG : If this is defined, the functions to access internal TiROS data structures is enabled. This is useful for debugging.

See also:

Advanced Features and Debugging.

TIROS_STK_CHECK : If this is defined, TiROS keeps track of additional information that can be used for getting useful information about stack usage.

See also:

Advanced Features and Debugging.

TIROS_C90_COMPATIBLE [0 | 1]: Use code downward compatible with an ISO C90 standard compiler. The code of concern is the definition of the message queue structure. Some older compilers or strict C++ compilers don't support zero-length array/flexible arrays/variable-length array/. These are useful as the last element of a structure which is really a header for a variable length object. TiROS uses such a construct to save memory when initializing message queues. The default behaviour is to work with GCC or any ISO C99 standard compiler. If your compiler complains that struct msgQ or msgQ_t has incomplete type or that the sizeof operator cannot be applied to this struct, then define this to be 1 in your proj_config.h file.

TiROS Data Type configuration: Some of the data types used within TiROS data structures can be modified by the user. Normally, this is not necessary. In special cases, the user may want to override the defaults. For example, by default, the counting semaphores use an 8 bit integer (signed) to represent count. This limits the maximum count to 127. The user can override this default and force TiROS to use a 16 bit integer for the count. This vastly increases the count range. For a comprehensive list of these options, see tr_types.h.

Other options: There can be other hardware port specific configuration options. For a description of this, look in the tr_port.h file in the appropriate port directory.

3.2 Application Programming Interface (API)

The Application Programming Interface (API) is described in the following major sections.

1. Critical Sections.
2. Usage with Interrupt Service Routines.
3. OS initialization and running.
4. Task creation and management.
5. Time services provided by the OS.
6. Utility functions for time operations.
7. Task cooperation and synchronization primitives.
8. Advanced Features and Debugging.

3.3 Function Options

3.3.1 Detailed Description

These option flags can be used with the various API functions.

Defines

- #define O_NONBLOCKING 0x01
- #define O_EFFECTIVE_PRIO 0x01
- #define O_RELATIVE_TIME 0x02
- #define O_EFLAG_CLEAR 0x04
- #define O_EFLAG_TRIGGER 0x08
- #define O_EFLAG_AND 0x04
- #define O_EFLAG_OR 0x00

3.3.2 Define Documentation

3.3.2.1 #define O_NONBLOCKING 0x01

Do not block while performing this command.

Definition at line 161 of file tiros.h.

3.3.2.2 #define O_EFFECTIVE_PRIO 0x01

Use effective priority instead of real priority for this command.

Definition at line 165 of file tiros.h.

3.3.2.3 #define O_RELATIVE_TIME 0x02

Use relative time instead of absolute time.

Definition at line 168 of file tiros.h.

3.3.2.4 #define O_EFLAG_CLEAR 0x04

Clear an event flag instead of setting it.

This is only used while setting an event flag.

Definition at line 174 of file tiros.h.

3.3.2.5 #define O_EFLAG_TRIGGER 0x08

Trigger an event flag, rather than permanently setting it This is only used while setting an event flag.

Definition at line 178 of file tiros.h.

3.3.2.6 #define O_EFLAG_AND 0x04

Wait for all the events corresponding to requested bits.

Only applicable to event flags.

Definition at line 182 of file tiros.h.

3.3.2.7 #define O_EFLAG_OR 0x00

Wait for any of the events correspoind to the requested bits.

Definition at line 185 of file tiros.h.

3.4 Critical Sections

3.4.1 Detailed Description

Critical Sections can be used to prevent preemption.

Within a critical section, a task will not be interrupted. The following code structure is used to implement critical sections:

Eg.

```
unsigned char port_status;  // This is global, may be accessed
                            // through an interrupt and its
                            // access must be protected with
                            // critical sections

void tst(void) {
 int a;
 char x;
 char txt[20];
 OS_CRITICAL_ENABLE();  // This should be declared immediately after
                       // other variable declarations.

 tst[0] = 3;

 OS_CRITICAL_BEGIN();      // Protect port_status using critical
 if (port_status) {        // sections
     x = port_status;
     port_status = 0;
 }
 OS_CRITICAL_END();
}
```

Defines

- #define OS_CRITICAL_ENABLE() OS_PORT_CRITICAL_ENABLE()
- #define OS_CRITICAL_BEGIN() OS_PORT_CRITICAL_BEGIN()
- #define OS_CRITICAL_END() OS_PORT_CRITICAL_END()
- #define OS_INT_ENABLE() OS_PORT_INT_ENABLE()
- #define OS_INT_DISABLE() OS_PORT_INT_DISABLE()

3.4.2 Define Documentation

3.4.2.1 #define OS_CRITICAL_ENABLE() OS_PORT_CRITICAL_ENABLE()

Enable a critical section within a function.

This must be the first call in the functions after the variable declarations. This has to be called before OS_BEGIN_CRITICAL() or OS_CRITICAL_END() can be called. See example for critical sections.

Definition at line 229 of file tiros.h.

3.4.2.2 #define OS_CRITICAL_BEGIN() OS_PORT_CRITICAL_BEGIN()

Begin a critical section.

This should always be matched with an OS_CRITICAL_END() in the same function. Critical sections should be enabled in the function using OS_CRITICAL_ENABLE()

Definition at line 236 of file tiros.h.

3.4.2.3 #define OS_CRITICAL_END() OS_PORT_CRITICAL_END()

End a critical section.

This should be matched to a corresponding OS_BEGIN_CRITICAL() call.

Definition at line 240 of file tiros.h.

3.4.2.4 #define OS_INT_ENABLE() OS_PORT_INT_ENABLE()

Explicitly enable interrupts.

Ideally, there should be no need to use this call. The OS_CRITICAL_BEGIN() and OS_CRITICAL_END() calls are much safer, since they will store the interrupt state at the beginning of a critical section and restore the state at the end of a critical section. There might be rare occassions where this call may be needed (for example to explicitly re-enable interrupts within an ISR)

Definition at line 250 of file tiros.h.

3.4.2.5 #define OS_INT_DISABLE() OS_PORT_INT_DISABLE()

Explicitly disable interrupts.

Ideally, there should be no need to use this call. The OS_CRITICAL_BEGIN() and OS_CRITICAL_END() calls are much safer, since they will store the interrupt state at the beginning of a critical section and restore the state at the end of a critical section. There might be rare occassions where this call may be needed to explicitly disable interrupts.

Definition at line 260 of file tiros.h.

3.5 Usage with Interrupt Service Routines

3.5.1 Detailed Description

TiROS API calls can be used within ISRs.

Mark the beginning and end of an ISR, so that TiROS can appropriately perform task scheduling if needed. Interrupt service routines are notoriusly hardware and compiler dependent, so use the instructions for your hardware/compiler port.

TiROS supports three ways of servicing interrupts:

1. Simple Usage: An ISR can be written as you would normally write one for your hardware only if all of the following are true:
   - Interrupts will stay disabled while the ISR is being invoked.
   - The ISR will not call any TiROS api calls.
   - The ISR will not perform any action that will cause any task to become unblocked, or in general, will not change the state of any task.
2. Saving context per interrupt response: This is the standard way to service interrupts. Due to the presence of an OS, the sequence of steps to be taken are as follows:

```
function ISR( ) {
   stkptr = save_context_of_task();  // See port instructions.
    // stkptr = pointer to stack of saved task.

    perform_functions();
   newstkptr = osint_taskswitcher(stkptr);
   load_context_of_newtask(newstkptr);
}
```

   The exact implementation depends on the hardware port. See the port documentation for more details.

3. Using Kernel Traps: TiROS can take advantage of hardware architectures that can support a software initiated interrupt (kernel trap). This is the most preferable option because interrupt service routines have very low overhead and can yet use all of the TiROS calls and interact with tasks. If your port supports this, use this. It also makes writing the ISRs very simple. The way this works is that the context of a task is not saved on entry to the ISR. Instead, at the end of the ISR, OS_ISR_END() determines if the task has to be switched out. If so, it invokes a kernel trap. The kernel trap is entered as soon as the ISR ends. The kernel trap performs the context switch. Thus, the context switch overhead is only invoked when necessary, not at every ISR. Eg.

```
void ISR_function(void)
{
  int x;
  OS_CRITICAL_ENABLE();    // Only needed if any critical sections
                           // will be used and interrupt nesting is enabled.

  OS_ISR_BEGIN();          // Mark the beginning of an ISR. Note
```

```
                                // that if OS_CRITICAL_ENABLE() is also
                                // present, then this comes after
                                // that.

       x = 3;
         Do more stuff ....

       OS_ISR_END();
      }
```

TiROS supports nested interrupts. However, for reliability, try not to use nested interrupts unless absolutely necessary. Interrupt service routines use the stack of the task that was running when the interrupt occured. Nested interrupts can cause the stack to grow unpredictably and overflow.

To use nested interrupts, write code as below:

```
extern uint8_t os_isr_nesting; // Internal TiROS variable that keeps track
                               // of nesting level.
void ISR_function(void)
{
  int x;
  OS_CRITICAL_ENABLE();    // Only needed if any critical sections
                           // will be used and interrupt nesting is enabled.

  OS_ISR_BEGIN();          // Mark the beginning of an ISR. Note
                           // that if OS_CRITICAL_ENABLE() is also
                           // present, then this comes after
                           // that.

 if (os_isr_nesting < TIROS_MAX_ISR_NESTING) {
    // Interrupts should only be reenabled if the current nesting level
    // is less than or equal to the maximum configured nesting level.
    renable_interrupts();
 }
 x = 3;
   Do more stuff ....

 OS_ISR_END();
}
```

Defines

- #define OS_ISR_BEGIN() OS_PORT_ISR_BEGIN()

Functions

- void OS_ISR_END (void)

3.5.2 Define Documentation

3.5.2.1 #define OS_ISR_BEGIN() OS_PORT_ISR_BEGIN()

Enable this within ISRs immediately after any variable declarations, and after any OS_CRITICAL_-ENABLE().

Definition at line 379 of file tiros.h.

3.5.3 Function Documentation

3.5.3.1 void OS_ISR_END (void)

Mark the end of an ISR.

This should be called at the end of an ISR. There must exist a corresponding OS_ISR_BEGIN() statement.

3.6 OS initialization and running

3.6.1 Detailed Description

Routines to initialize and start the OS.

Functions

- void os_init (void)
- void os_start (void)

3.6.2 Function Documentation

3.6.2.1 void os_init (void)

Initialize the OS.

This initializes the OS data structures. This should be the first OS API called.

3.6.2.2 void os_start (void)

Start the OS by running tasks in the ready list.

Note: Tasks should have been created and made ready to run before this function is called.

3.7 Task creation and management

3.7.1 Detailed Description

This group of functions provides routines to create, delete, and manipulate the state of tasks.

Defines

- #define TIROS_MIN_CTXT_SZ TRPORT_MIN_CTXT_SZ

Typedefs

- typedef void(∗) taskfunc_t (void ∗)

Functions

- tid_t os_task_create (taskfunc_t func, osptr_t param, osword_t ∗stack, osword_t stacksize, tid_t priority)
- tid_t os_self_tid (void)
- int8_t os_prio_set (tid_t task, tid_t prio)
- tid_t os_prio_get (tid_t task, uint8_t options)
- int8_t os_task_del (tid_t task)
- int8_t os_task_suspend (tid_t task)
- int8_t os_task_resume (tid_t task)

3.7.2 Define Documentation

3.7.2.1 #define TIROS_MIN_CTXT_SZ TRPORT_MIN_CTXT_SZ

Minimum amount of space is oswwords (not bytes).

Definition at line 423 of file tiros.h.

3.7.3 Typedef Documentation

3.7.3.1 typedef void(∗) taskfunc_t(void ∗)

Task Prototype.

This specifies the form of the function that begins a task.

Definition at line 428 of file tiros.h.

3.7.4 Function Documentation

3.7.4.1 tid_t os_task_create (taskfunc_t *func*, osptr_t *param*, osword_t * *stack*, osword_t *stacksize*, tid_t *priority*)

Create a task.

Calling Context:

1. Before os_start().
2. From within a task.
3. From an ISR.

```
#define TASK1_STKSZ   (TIROS_MIN_CTXT_SZ  + 64)  // 64 words more than min
osword_t task1_stk[TASK1_STKSZ];
void task1(void *arg)
{
//  Task does something. never exits
}

void main(void)
{
  tid_t t1_tid;
  tid_t idle_tid;
  int arg_to_pass_to_task1
  os_init();
  t1_tid = os_task_create(task1, (osptr_t) arg_to_pass_to_task1,
                          task1_stk, TASK1_STKSZ, TASK1_PRIO);

  if (t1_tid == ILLEGAL_ELEM) {
    error("task1 error");
  }
            .
            .
            .


  os_start();
}
```

Parameters:

func Function pointer to the task.

param The parameter to be passed to the task.

stack Pointer to the stack to be used.

stacksize Size of the stack. NOTE: This is specified in osword_t not in bytes. This is so that the stack is not misaligned. Use the TIROS_MIN_CTXT_SZ value to help in sizing the stack appropriately.

priority Task priority.

Returns:

ID of the task created or ILLEGAL_ELEM if call failed. This happens if there are no TCBs or if a task is already allocated to the priority level.

3.7.4.2 tid_t os_self_tid (void)

A task's self identification.

Calling Context:

1. From within a task.
2. From an ISR.

Returns:

Return the task id or ILLEGAL_ELEM if called from an ISR.

3.7.4.3 int8_t os_prio_set (tid_t *task*, tid_t *prio*)

Set the priority of a task.

This modifies the base priority of a task. The function returns SUCCESS if the priority change was successful OR if the current priority of the task is the same as the desired priority. When the default priority ceiling algorithm is used for mutual exclusion, the priority of a task holding a mutex cannot be changed. Attempting this returns ERR_WOULDBLOCK_MUTEX

Calling Context:

1. From within a task.
2. From an ISR.

Parameters:

task ID of the task to have a priority change.

prio New priority.

Returns:

{SUCCESS, ERR_NOSUCHTASK, ERR_PRIO_IN_USE, ERR_WOULDBLOCK_MUTEX }.

3.7.4.4 tid_t os_prio_get (tid_t *task*, uint8_t *options*)

Get the priority of a task.

Parameters:

task ID of the task whose priority is queried.

options Get either the real or effective priority. If O_EFFECTIVE_PRIO is specified, then the effective priority is returned.

Calling Context:

1. Before os_start().
2. From within a task.
3. From an ISR.

Parameters:

task ID of the task

options [O_EFFECTIVE_PRIO]

Returns:

ILLEGAL_ELEM if failed or the task priority

3.7.4.5 int8_t os_task_del (tid_t *task*)

Delete a task.

A task can be deleted if it is not in possession of any mutexes. A task can also delete itself.

Calling Context:

1. From within a task.
2. From an ISR.

Parameters:

task Task to be deleted. This can be the current task.

Returns:

{SUCCESS, ERR_NOSUCHTASK, ERR_TASKBLOCKED}.

3.7.4.6 int8_t os_task_suspend (tid_t *task*)

Suspend a task.

A task can only be suspended if it is not holding any mutexes. This is by design: If a task holding a mutex were to be suspended, it could result in a priority inversion or starvation. A task can also suspend itself. In this case, it will only resume running after some other task resumes it (

See also:

os_task_resume). When the task resumes, it returns SUCCESS (not ERR_RESUMED), since this is the intended operation.

Calling Context:

1. From within a task.
2. From an ISR.

Parameters:

task Task to be suspended.

Returns:

{SUCCESS, ERR_NOSUCHTASK, ERR_TASKBLOCKED} .

3.7.4.7 int8_t os_task_resume (tid_t *task*)

Resume a task.

This call can be used to forcibly resume: 1) a task that is waiting on a lock, 2) one that has been suspended, 3) one that is sleeping. The effect of this call is to make the designated task ready for running. Note: a task that is sleeping may be woken up earlier than its sleeptime, explicitly by the resume command. Similarly, a task that is waiting on a lock will be forcibly readied and scheduled even before its timeout expires. If a task is already in the ready queue, then it is not altered.

Calling Context:

1. From within a task.
2. From an ISR.

Parameters:

task Task to be resumed.

Returns:

{SUCCESS, ERR_NOSUCHTASK }.

3.8 Time services provided by the OS

3.8.1 Detailed Description

This set of API calls allows time related functions.

TiROS operates with high-resolution time. Sleep and timeouts are specified in absolute or relative time (using the O_RELATIVE_TIME option). Time is represented using the trtime_t structure. This structure consists of two elements:

1. A subsecond unit (could be milli-seconds, nano-seconds,etc.)
2. A supersecond unit (could be a unit of 1 second, 10 secs, 1 minute etc.

This choice of time is kept deliberately vague, since the application is an embedded system, where these functions have to be fast, take advantage of the hardware, and cannot waste cycles converting time units from the hardware representation to a standard form. The actual representation of the time structure is specified by the platform specific port.

Functions

- void os_time_get (trtime_t *curr_time)
- int8_t os_time_set (const trtime_t *new_time)
- int8_t os_wake_at (const trtime_t *wake_time, uint8_t options)

3.8.2 Function Documentation

3.8.2.1 void os_time_get (trtime_t * *curr_time*)

Get the current time.

Calling Context:

1. From within a task.
2. From an ISR.

Parameters:

→ ***curr_time*** Pointer to structure where the time should be stored.

3.8.2.2 int8_t os_time_set (const trtime_t * *new_time*)

Set the current time.

This sets the current time. This has to be used with care. This can affect tasks that have been sleeping or waiting for a lock. Such tasks will be woken up if the new time set is beyond their wakeup deadline. This system call may not be possible on some hardware ports.

Calling Context:

1. From within a task.
2. From an ISR.

To set the time before the os has started use the hal_time_set function directly, which is supplied by the hardware port.

Parameters:

new_time Pointer to structure with the new time.

Returns:

{SUCCESS, ERR_FAILED}

3.8.2.3 int8_t os_wake_at (const trtime_t * *wake_time*, uint8_t *options*)

Wake the invoking task at a specific time.

The task can be woken up earlier by an explicit resume. The task is not allowed to sleep if it holds a mutex lock. This call is not allowed from an ISR.

Calling Context:

1. From within a task.
2. Call from ISR will result in an error.

Parameters:

← ***wake_time*** Time at which to be woken up.

options Default is absolute time. For relative time, set O_RELATIVE_TIME.

Returns:

{SUCCESS, ERR_RESUMED, ERR_WOULDBLOCK_MUTEX, ERR_WOULDBLOCK_ISR}

3.9 Utility functions for time operations

Data Structures

- struct trtime

  Time structure.

Typedefs

- typedef trtime trtime_t

Functions

- LT_INLINE uint32_t trtime_to_secs (const trtime_t *const lt)
- LT_INLINE void secs_to_trtime (uint32_t seconds, trtime_t *lt)
- LT_INLINE void time_sub (const trtime_t *x, const trtime_t *y, trtime_t *res)
- LT_INLINE void time_add (const trtime_t *x, const trtime_t *y, trtime_t *res)
- LT_INLINE int time_compare (const trtime_t *t1, const trtime_t *t2)
- LT_INLINE int time_lessthan (const trtime_t *t1, const trtime_t *t2)

3.9.1 Typedef Documentation

3.9.1.1 typedef struct trtime trtime_t

Time structure.

Different hardware architectures may represent time in formats that are most efficient and suitable for that hardware. We impose no requirements except for two functions that can convert this hardware dependent time structure into units of seconds and vice-versa The trtime_t structure consists of two elements:

1. A subsecond unit (could be milli-seconds, nano-seconds,etc.)
2. A supersecond unit (could be a unit of 1 second, 10 secs, 1 minute etc.

This choice of time is kept deliberately vague, since the application is an embedded system, where these functions have to be fast, take advantage of the hardware, and cannot waste cycles converting time units from the hardware representation to a standard form.

3.9.2 Function Documentation

3.9.2.1 LT_INLINE uint32_t trtime_to_secs (const trtime_t *const *lt*)

Convert time represented in trtime to seconds.

Parameters:

← ***lt*** Pointer to trtime structure.

Returns:

Equivalent time in seconds

3.9.2.2 LT_INLINE void secs_to_trtime (uint32_t *seconds*, trtime_t ∗ *lt*)

Convert time represented in seconds to trtime.

Parameters:

seconds Time in seconds to be converted.

→ ***lt*** Pointer to trtime structure.

3.9.2.3 LT_INLINE void time_sub (const trtime_t ∗ *x*, const trtime_t ∗ *y*, trtime_t ∗ *res*)

Time subtraction.

Note: Time1 >= Time2, check using time_compare if needed.

Parameters:

x Time1

y Time2

→ ***res*** Time1 - Time 2

3.9.2.4 LT_INLINE void time_add (const trtime_t ∗ *x*, const trtime_t ∗ *y*, trtime_t ∗ *res*)

Time addition.

Parameters:

x Time1

y Time2

→ ***res*** Time1 + Time 2

3.9.2.5 LT_INLINE int time_compare (const trtime_t * *t1*, const trtime_t * *t2*)

Time comparison.

Parameters:

t1 Time1

t2 Time2

Returns:

(-1,0,1) depending on whether t1 $<$ t2, t1==t2, t1 $>$ t2

3.9.2.6 LT_INLINE int time_lessthan (const trtime_t * *t1*, const trtime_t * *t2*)

Time less than.

Parameters:

t1 Time1

t2 Time2

Returns:

(1,0) depending on whether t1 $<$ t2. This can easily be implemented using time_compare but is provided as an optimization.

3.10 Mutexes

3.10.1 Detailed Description

This page contains the API for mutex management.

Mutexes are used to synchronize access by cooperating tasks to shared resources. For every shared resource, create an associated shared mutex. Mutexes should be locked by a task before beginning usage of the shared resource and then unlocked after the usage is over. A task cannot lock a mutex that is already locked by another mutex. It will either block for the mutex to unlock or will return an error code (depending on the options used). Mutexes must always be used in a lock/unlock sequence.

```
mutex_t resource_mutex; // Previously initialized (possibly in main)
void task1(void *dummy)
{
    while(1) {
      mutex_lock(&resource_mutex, 0, 0);
      do_stuff();
      mutex_unlock(&resource_mutex);
      do_other_stuff();
    }
}
```

A detailed description of the design and implementation of mutexes within TiROS are provided in the Mutual Exclusion (Mutex) section.

NOTE: Always initialize before use.

Functions

- void mutex_init (mutex_t *m, tid_t prio_ceiling)
- tid_t mutex_owner (mutex_t *m)
- int8_t mutex_lock (mutex_t *m, const trtime_t *timeout, uint8_t options)
- int8_t mutex_unlock (mutex_t *m)

3.10.2 Function Documentation

3.10.2.1 void mutex_init (mutex_t * *m*, tid_t *prio_ceiling*)

Initialize mutex.

Calling Context:

1. Before os_start().
2. From within a task.
3. From an ISR.

Parameters:

m Pointer to mutex structure

prio_ceiling The priority ceiling, if the priority celing protocol is used. If priority inheritance is used, then the prio_ceiling should be set to ILLEGAL_ELEM. The prio_ceiling MUST be unique (no task should have this as its base priority).

3.10.2.2 tid_t mutex_owner (mutex_t * *m*)

Get mutex owner This returns the task id of the owner of a specified mutex.

This is an instantaneous snapshot of the state of the mutex. Due to preemption, the owner may have changed by the time the callee acts on this information. Example:

```
tid_t tmp;
tmp = mutex_owner( m);  // M has been initialized previously
do_other_stuff();
if (tmp == other_task)
    do_more_stuff();    // the owner of m may have changed.
```

Use critical sections to prevent a block of code from being executed uninterrupted. NOTE: The mutex must be intialized before this call, else the return value is meaningless.

Calling Context:

1. Before os_start().
2. From within a task.
3. From an ISR.

Returns:

The id of the mutex owner or ILLEGAL_ELEM if it is unlocked.

3.10.2.3 int8_t mutex_lock (mutex_t * *m*, const trtime_t * *timeout*, uint8_t *options*)

Lock mutex.

This is used to exclusively lock a mutex by a task. The locking behavior depends on the mutex protocol that has been configured. If the specified mutex is not locked, it is locked by the calling task at the end of the call, and a SUCCESS is returned. Attempts by a task to lock a mutex that it already holds also return SUCCESS. Attempts by an ISR to lock a mutex will result in an error code of ERR_LOCK_ISR, since ISRs are not allowed to lock mutexes. If a task already has locked TIROS_MAX_MUTEXES mutexes, then ERR_FULL will be returned, as it is not allowed to lock any more mutexes. Attempts to lock a mutex owned by a different task will result in blocking. If the O_NONBLOCKING option is specified, then the ERR_WOULDBLOCK

error code is returned, else the task is made to wait until the timeout, when ERR_TIMEOUT is returned. If the task is explicitly resumed during its wait, it receives the ERR_RESUMED code.

Calling Context:

1. From within a task.
2. Call from ISR will error.

Parameters:

m Pointer to mutex

timeout Timeout. This can be 0 for infinite timeout.

options Combination of [O_RELATIVE_TIME | O_NONBLOCKING].

Returns:

{SUCCESS , ERR_LOCK_ISR, ERR_LOCK_PRIO_CEIL, ERR_WOULDBLOCK, ERR_-TIMEOUT, ERR_RESUMED, ERR_FULL}.

3.10.2.4 int8_t mutex_unlock (mutex_t * *m*)

Unlock Mutex.

Calling Context:

1. From within a task. Must be owner to succeed.
2. Call from ISR will error.

Parameters:

m Pointer to mutex.

Returns:

{SUCCESS, ERR_NOTOWNER}

3.11 Counting semaphores

3.11.1 Detailed Description

Counting semaphores can be used to keep a shared count by different tasks.

They have a value between 0 to a specified MAX value.

The P() operation (Prolaag) decreases the count of the counting semaphore. If the count is zero, the task blocks since it cannot be decreased further. The V() operation (Verhoog) increments the count of the counting semaphore. This operation is non-blocking.

Counting semaphores are useful for event notifications. For example, a message server task can perform a P() operation on a message counting semaphore. When a message has to be sent to the message server, a message client task performs a V() operation on the same semaphore. This unblocks the message server. If multiple client tasks send messages, the message server has a count of the number of times the message counter has been V()ed.

If multiple servers are blocked on a counting semaphore, the highest priority blocked server is alerted when the semaphore is incremented.

NOTE: A custom type csemval_t is used for the semaphore count. This is of type signed integer. By default, it is an 8-bit integer (max val of 127). This can be overridden (see tr_types.h).

NOTE: Always initialize before use.

```
csem_t cs; // Previously initialized (possibly in main)
void task1(void *dummy)
{
    while(1) {
      csem_P(&cs, 0, 0);  // Wait for semaphore
      do_stuff();
    }
}
void task2(void *dummy)
{
    while(1) {
      do_stuff();
      do_other_stuff();
      // Release task1
      csem_V(&cs);       // Release semaphore
      do_stuff();
    }
}
```

Functions

- void csem_init (csem_t *cs, csemval_t init_val, csemval_t max_val)
- csemval_t csem_count (csem_t *cs)
- csemval_t csem_P (csem_t *cs, const trtime_t *timeout, uint8_t options)
- csemval_t csem_V (csem_t *cs)

3.11.2 Function Documentation

3.11.2.1 void csem_init (csem_t * *cs*, csemval_t *init_val*, csemval_t *max_val*)

Initialize counting semaphore.

Calling Context:

1. Before os_start().
2. From within a task.
3. From an ISR.

Parameters:

cs Pointer to counting semaphore

init_val Initial value

max_val Maximum value for the counting semaphore. The semaphore cannot be incremented past this number.

3.11.2.2 csemval_t csem_count (csem_t * *cs*)

Obtain the value of a counting semaphore.

Calling Context:

1. Before os_start().
2. From within a task.
3. From an ISR.

NOTE: the counting semaphore must previously have been initialized for this to have any meaningful return value.

Parameters:

cs Pointer to counting semaphore

Returns:

The value of the semaaphore

3.11.2.3 csemval_t csem_P (csem_t * *cs*, const trtime_t * *timeout*, uint8_t *options*)

Decrease (prolaag) the value of a counting semaphore.

This decreases the value of a counting semaphore. If the value is zero, the caller blocks until another task increments the semaphore OR until the timeout duration OR until another task explicitly resumes (

See also:

os_task_resume) this task. The caller can prevent blocking by using the O_NONBLOCKING option. On success, the value returned is the value of the semaphore immediately after the decrease.

Calling Context:

1. From within a task.
2. From an ISR (error if blocking needed).

Parameters:

cs Pointer to counting semaphore

timeout Timeout. This can be 0, for infinite timeout.

options [O_RELATIVE_TIME | O_NONBLOCKING]

Returns:

{Semaphore count if successful, ERR_WOULDBLOCK_ISR, ERR_WOULDBLOCK_MUTEX, ERR_TIMEOUT, ERR_RESUMED }

3.11.2.4 csemval_t csem_V (csem_t * *cs*)

Increase (verhoog) the value of a counting semaphore.

This is a non-blocking function. If the maximum value of the semaphore is reached, an ERR_FULL message is returned.

NOTE: On success, the value returned is the value of the semaphore immediately after the increase. If there are other tasks waiting to csem_P on the counting semaphore, the value returned is that before these other tasks have completed the csem_P operation. Example:

```
csemval_t tmp;
// Current state of counting semaphore cs is 0.  There are two
// tasks blocked doing a csem_P on it.
tmp = csem_V(&cs);  // Value of tmp is 1, because of the csem_V

tmp = csem_count(&cs); // Value of tmp is 0, because one of the tasks that
                       // that was blocked on csem_P(&cs) has
                       // completed the prolaag and now the value
                       // of cs is back to zero.
```

Calling Context:

1. From within a task.
2. From an ISR.

Parameters:

cs Pointer to counting semaphore

Returns:

{Semaphore count or ERR_FULL}

3.12 Message Queues

3.12.1 Detailed Description

Message Queues can be used to send arbitrary messages between tasks.

Messages are sent in the form of generic pointers to memory. TiROS handles these message pointers in an opaque manner and does not care what they point to. It is important to note that the maximum size of the message queue has to be known when the message queue is intialized. It cannot be resized on the fly. A message server can wait on a message queue or poll it (with the O_NONBLOCKING option). A message client can send a message on the queue which will unblock any waiting server. If multiple tasks are waiting on the same message queue, the one with the highest priority will be unblocked first.

NOTE: A custom type mqind_t is used for the queue length. This is of type unsigned integer. By default, it is an 8-bit integer (max val of 255). This can be overridden (see tr_types.h).

NOTE: Always initialize before use.

```
#define TST_QSZ 4
osword_t tst_Q[msgQ_MEMSZ(TST_QSZ)];   // Previously initialized
void task1(void *dummy)
{
  int8_t rc;
  osptr_t rx_val;
  while(1) {
      rc = msgQ_recv( (msgQ_t*)tst_Q, 0, 0, &rx_val);
      if (rc == SUCCESS) {
         do_something(rx_val);
      }
  }
}
void task2(void *dummy)
{
  int8_t rc;
  while(1) {
      rc = msgQ_send( (msgQ_t*)tst_Q, 7); // Send 7 to task1
  }
}
```

Defines

- #define msgQ_MEMSZ(qlen)

Functions

- void msgQ_init (msgQ_t *m, mqind_t qlen)
- mqind_t msgQ_count (msgQ_t *m)
- int8_t msgQ_send (msgQ_t *m, osptr_t tx_value)
- int8_t msgQ_recv (msgQ_t *m, const trtime_t *timeout, uint8_t options, osptr_t *rx_value)

3.12.2 Define Documentation

3.12.2.1 #define msgQ_MEMSZ(qlen)

Value:

```
((sizeof(struct msgQ) +  \
                (mqind_t)qlen * sizeof(osptr_t) + \
                sizeof(osword_t)-1) /   sizeof(osword_t))
```

Get the memory size for a message queue This macro makes sure that the memory is word aligned.

Parameters:

qlen Length of the message queue

Returns:

Memory occupied (in words)

Definition at line 1002 of file tiros.h.

3.12.3 Function Documentation

3.12.3.1 void msgQ_init (msgQ_t * *m*, mqind_t *qlen*)

Initialize a message queue.

Calling Context:

1. Before os_start().
2. From within a task.
3. From an ISR.

Parameters:

m Pointer to message queue

qlen Length of the queue. NOTE: The queue memory is contained in the msgQ structure. It is EXTREMELY IMPORTANT that the qlen argument to this call be less than or equal to the length of the message queue that was used as an argument for msgQ_MEMSZ( ). Failing to follow this rule will result in buffer overflows.

Eg.:

```
msgQ_t *m;
qlen = 10;

osword_t dummy[ msgQ_MEMSZ(qlen)];
m = (msgQ_t *) dummy;
msgQ_init(m, qlen);
```

The memory should not be transient. It should be global, or allocated on the heap.

3.12.3.2 mqind_t msgQ_count (msgQ_t * *m*)

The number of messages in the queue.

Calling Context:

1. Before os_start().
2. From within a task.
3. From an ISR.

Note: The message queue must be initialized before this call, else the return value is meaningless

Parameters:

m Pointer to the message queue

Returns:

Number of messages in the queue

3.12.3.3 int8_t msgQ_send (msgQ_t * *m*, osptr_t *tx_value*)

Post a message to the queue.

Calling Context:

1. From within a task.
2. From an ISR.

In the current implementation, if the queue is full, an error is returned. The task is not allowed to block here. This may be changed in the future.

Parameters:

m Pointer to message queue

tx_value The value to be added to the queue.

Returns:

{SUCCESS, ERR_FULL}

3.12.3.4 int8_t msgQ_recv (msgQ_t * *m*, const trtime_t * *timeout*, uint8_t *options*, osptr_t * *rx_value*)

Wait for message.

Calling Context:

1. From within a task.
2. From an ISR (error if blocking needed).

Parameters:

m Pointer to message queue

timeout Timeout. This can be 0 for infinite timeout.

options [O_NONBLOCKING | O_RELATIVE_TIME].

→ ***rx_value*** Pointer to memory where the message should be stored.

Returns:

{SUCCESS, ERR_WOULDBLOCK_ISR, ERR_WOULDBLOCK_MUTEX, ERR_TIMEOUT, ERR_RESUMED}.

3.13 Event Flags

3.13.1 Detailed Description

Event flags allow tasks to wait for a specific combination of events.

An event flag is a bitmask. Waiting tasks can specify a subset of the bitmask as events of interest. They can either wait for ALL of the events to occur or ANY of the events to occur.

NOTE: Always initialize before use.

```
eflag_t notification;   // Previously initialized
#define ALERT0  0x01
#define ALERT1  0x02
void task1(void *dummy)
{
  int8_t rc;
  while(1) {
      rc = eflag_wait( &notification, ALERT0, 0, 0, );
      if (rc == SUCCESS) {
         do_something();
      }
  }
}
void task2(void *dummy)
{
  int8_t rc;
  while(1) {
      // Trigger event
      rc = eflag_set(&notification, ALERT0|ALERT1, O_EFLAG_TRIGGER);
      do_other_stuff();
  }
}
*
```

Functions

- void eflag_init (eflag_t *ef, flag_t initval)
- int8_t eflag_set (eflag_t *ef, flag_t setbits, uint8_t options)
- flag_t eflag_get (eflag_t *ef)
- int8_t eflag_wait (eflag_t *ef, flag_t checkbits, const trtime_t *timeout, uint8_t options)

3.13.2 Function Documentation

3.13.2.1 void eflag_init (eflag_t * *ef*, flag_t *initval*)

Initialize an event flag.

*

Calling Context:

1. Before os_start().
2. From within a task.
3. From an ISR.

Parameters:

ef Pointer to the event flag.

initval Initial value for the event flag

3.13.2.2 int8_t eflag_set (eflag_t * *ef*, flag_t *setbits*, uint8_t *options*)

Set an event flag.

Calling Context:

1. From within a task.
2. From an ISR.

Parameters:

ef Pointer to the event flag.

setbits The new bitvalues to be set or cleared.

options [O_EFLAG_CLEAR, O_EFLAG_TRIGGER]

```
O_EFLAG_CLEAR   : Clear the specified bits.  Default is to set them.
O_EFLAG_TRIGGER : Just pulse the values in the prescribed
                  fashion. Don't set them permanently.  If any of the
                  flag values were set before the trigger, they stay
                  set after the call.
```

Returns:

{SUCCESS, ERR_FAILED}

3.13.2.3 flag_t eflag_get (eflag_t * *ef*)

Peek at the value of an event flag.

This can be used for polling.

Calling Context:

1. Before os_start().
2. From within a task.

3. From an ISR.

Parameters:

ef Pointer to the flag.

Returns:

Current setting for the event flag

3.13.2.4 int8_t eflag_wait (eflag_t * *ef*, flag_t *checkbits*, const trtime_t * *timeout*, uint8_t *options*)

Wait for an event flag.

A blocking call from an ISR will result in an error. So, there is usually no need to call this from an ISR. eflag_get may be preferable.

Calling Context:

1. From within a task.
2. From an ISR (error if blocking needed).

Parameters:

ef Pointer to the event flag.

checkbits Bits to be checked.

← ***timeout*** Timeout. This can be 0 for infinite timeout.

options [O_NONBLOCKING | O_RELATIVE_TIME | O_EFLAG_AND | O_EFLAG_OR].

Returns:

{SUCCESS, ERR_WOULDBLOCK_ISR, ERR_WOULDBLOCK_MUTEX, ERR_TIMEOUT, ERR_RESUMED}

3.14 Advanced Features and Debugging

3.14.1 Detailed Description

TiROS has API calls for debugging and per-task customization.

Notification of context switching Using the TIROS_USER_CTXT_UPCALL feature, a user supplied function, user_ctxt_upcall(), can be kept informed of context switching. This can be used in creative ways to implement per-task hardware customization. For example, the upcall can be used to save the low-power state of the current task and set the low-power state of the next task. This will result in per-task power control.

Debugging with TiROS Two types of debugging information can be retrieved from TiROS.

1. One is a simple text based error log that is activated by setting TIROS_DEBUG_LEVEL higher than 0. Output messages depend on the debug level.

```
0 - No debugging messages.
1 - Critical error messages.
2 - Adds warning mesages.
3 - Adds informational messages.
4 - Adds trace messages.
```

   The debug output from TiROS gets channeled to the putchar(int char) function. To get debugging information, ensure that putchar() is properly implemented. By implementing putchar() appropriately, the output can be directed to a serial port or to a file.

2. Another method of debugging (and the most powerful) is to periodically capture the internal state of TiROS. This is done by a user task by calling the osint_snapshot function. To enable this function, define TIROSINT_DATA_DEBUG in the proj_config.h file, and include tr_debug.h in your source file. The osint_snapshot() function can be used to capture the TiROS state into a memory buffer (no need for putchar()) which can then be output as desired. An example of the use of this form of debugging is provided as an example in os_examples/os_debug_example

Data Structures

- struct os_data_hdr

  A representation of the debug header data.

- struct os_data_global

  This structure is a representation of the global debug data.

- struct os_data_per_task

  This structure is a representation of the variable internal data.

- struct os_internal_debug_data

  This structure describes the data returned by osint_snapshot.

Defines

- #define TIROS_DEBUG_DATA_SZ (sizeof (struct os_internal_debug_data))

Functions

- int osint_snapshot (unsigned char *buffer, int memsz)

3.14.2 Define Documentation

3.14.2.1 #define TIROS_DEBUG_DATA_SZ (sizeof (struct os_internal_debug_data))

Recommended size of the buffer to hold the TiROS internal debug data.

Definition at line 188 of file tr_debug.h.

3.14.3 Function Documentation

3.14.3.1 int osint_snapshot (unsigned char * *buffer*, int *memsz*)

A snapshot of all the internal os data.

This can be used by a dedicated task to send out debugging info The data is packed into a tight architecture independent little-endian byte stream. The type of data that is packed is illustrated in os_internal_debug_data. To use this function, tr_debug.h should be included by the source file and TIROSINT_DATA_DEBUG should be defined in proj_config.h

The TiROS state is written in a compact endian-neutral format. This can be parsed using the tiros_parse program that is included with the distribution. This form of debugging provides a wealth of information. Here is example output, parsed by tiros_parse, showing the debug data output by the example in os_examples/os_debug_example. This output was obtained from an msp430_gcc port.

NOTE: The information about the debugging task itself may not be fully accurate. It will reflect the state of the task at the last context switch.

```
--------------------------------------------------
os_options = 0x07,  data_sizes1 = 0x15, data_sizes2 = 0x01, max_procs = 8
Record size = 240
 TIROS Settings
PrioSort: 1      reg_passing: 1         stk_chk_info: 1          oswordt_sz: 2
tidtsz: 1        subtimet_sz: 2          flagt_sz: 2    osptrwordt_sz: 2
```

```
max_procs: 8
gdata_len: 12    data_per_task_len: 28  task_tcb_len: 20         record_sz: 240
********************************************************************************
         5:41515        RUN:  1 RDY:  1 WT:  0  NESTING:  0     CTXT_CNT:   12
--------------------------------------------------------------------------------
TSK     CP      PRIO    EFF     FLAGS   MUTXS   TIMEOUT                 LK_PTR  EVTS    ...
0       0x1310    0       0     0x10      0              6:  640        0xffff  0x0000  ...
1       0x13aa    1       1     0x00      0              5:41483        0xffff  0x0000  ...
2       0x1442    2       2     0x10      0     4294967295:65535        0x1530  0x0001  ...
3       0x150e    3       3     0x00      0              0:    0        0xffff  0x0000  ...
4       0xffff    0       0     0x00      0              0:    0        0x0000  0x0000  ...
5       0xffff    0       0     0x00      0              0:    0        0x0000  0x0000  ...
6       0xffff    0       0     0x00      0              0:    0        0x0000  0x0000  ...
7       0xffff    0       0     0x00      0              0:    0        0x0000  0x0000  ...
------------------------------------------------------------------------
  ...    STK     STKSZ   PC      CUR_STK MAX_STK R/W Q   LK Q
  ...    0x12b8     79   0x472a     35      36   0x02    0xff
  ...    0x1356     79   0xffff      0      38   0x03    0xff
  ...    0x13f4     79   0x472a     40      41   0xff    0xff
  ...    0x1492     79   0x59ec     17      30   0xff    0xff
  ...    0x0000      0   0xffff      0       0   0xff    0xff
  ...    0x0000      0   0xffff      0       0   0xff    0xff
  ...    0x0000      0   0xffff      0       0   0xff    0xff
  ...    0x0000      0   0xffff      0       0   0xff    0xff
------------------------------------------------------------------------
```

As can be seen, there is a prolific amount of information:

- the sizes of all the data types including task control block sizes.
- the TIROS_MAX_PROCS value that was configured into the program during compilation.
- Time.
- The running task ID, task IDs of the heads of the ready queue and wait queue.
- the ISR nesting level.
- number of context switches since reset.

On a per-task basis, the following information is available:

- TSK: The task id.
- CP: The current context pointer.
- PRIO: The priority of the task.
- EFF: The effective priorityof the task.
- FLAGS: Internal flags used to mark the task's state.
- MUTXS: The number of mutexes held by the task.
- TIMEOUT: The wait or sleep timeout.

- LK_PTR: Pointer to a lock that the task is waiting on if any.
- EVTS: Event bitmask that the task is waiting on.
- STK: The base stack pointer provided during task initialization. (only if TIROS_STK_CHECK is defined).
- STKSZ: The stack size provided during task initialization. (only if TIROS_STK_CHECK is defined).
- PC: The program counter (or instruction pointer) is available if the hardware port supports this.
- CUR_STK: The current stack usage is provided if the hardware port supports this.
- MAX_STK: The maximum stack usage of the task (throughout its run so far) is provided if the hardware port supports this.

The R/W Q and LK Q columns can be used to decipher the order of the tasks in the read/wait list and lock list respectively.

Parameters:

← ***memaddr*** Memory address to copy the data structures. If this is set to ILLEGAL_ADDR, the return value contains the size that the data structures would occupy in bytes. It is recommended that buffer be TIROS_DEBUG_DATA_SZ long.

← ***memsz*** Size of the memory location.

Returns:

If memaddr is set to ILLEGAL_ADDR< this returns the size of the data structures, else ERR_FAILED, if not enough memory. the actual length of the byte stream, if success.

3.15 Return Codes.

3.15.1 Detailed Description

The common return codes and their descriptions are given below.

Error return codes have a major and minor number. The major number provides the error family. The minor number gives more detail.

Defines

- #define ERR_MAJOR(x) (x & (~7))
- #define ERR_MINOR(x) (x & 7)
- #define ILLEGAL_ELEM ((tid_t) ~0)
- #define SUCCESS 0
- #define EMINOR_ISR 0x01
- #define EMINOR_PRIO_RULE 0x02
- #define ERR_LOCK -8
- #define ERR_LOCK_ISR (ERR_LOCK | EMINOR_ISR)
- #define ERR_LOCK_PRIO_CEIL (ERR_LOCK | EMINOR_PRIO_RULE)
- #define ERR_WOULDBLOCK -16
- #define ERR_WOULDBLOCK_ISR (ERR_WOULDBLOCK | EMINOR_ISR)
- #define ERR_WOULDBLOCK_MUTEX (ERR_WOULDBLOCK | EMINOR_PRIO_RULE)
- #define ERR_FULL -24
- #define ERR_NOTOWNER -32
- #define ERR_TIMEOUT -40
- #define ERR_RESUMED -48
- #define ERR_NOSUCHTASK -56
- #define ERR_TASKBLOCKED -64
- #define ERR_PRIO_IN_USE -72
- #define ERR_FAILED -80

3.15.2 Define Documentation

3.15.2.1 #define ERR_MAJOR(x) (x & (~7))

Extract the major error code out of the return value.

Definition at line 1273 of file tiros.h.

3.15.2.2 #define ERR_MINOR(x) (x & 7)

Extract the minor error code out of the return value.

Definition at line 1275 of file tiros.h.

3.15.2.3 #define ILLEGAL_ELEM ((tid_t) ∼0)

Notation for an invalid task.

Definition at line 1279 of file tiros.h.

3.15.2.4 #define SUCCESS 0

Return code success.

Definition at line 1283 of file tiros.h.

3.15.2.5 #define EMINOR_ISR 0x01

Minor error code indicating that error occurence was due to invocation from an ISR.

Definition at line 1287 of file tiros.h.

3.15.2.6 #define EMINOR_PRIO_RULE 0x02

Minor error code indicating that action would violate rules to prevent priority inversion.

Definition at line 1291 of file tiros.h.

3.15.2.7 #define ERR_LOCK -8

Mutex locking error.

Definition at line 1297 of file tiros.h.

3.15.2.8 #define ERR_LOCK_ISR (ERR_LOCK | EMINOR_ISR)

Mutex locking error: locking from ISR.

Definition at line 1300 of file tiros.h.

3.15.2.9 #define ERR_LOCK_PRIO_CEIL (ERR_LOCK | EMINOR_PRIO_RULE)

Mutex locking error: denied by priority ceiling.

Definition at line 1303 of file tiros.h.

3.15.2.10 #define ERR_WOULDBLOCK -16

Operation would cause blocking, option O_NONBLOCKING specified.

Definition at line 1309 of file tiros.h.

3.15.2.11 #define ERR_WOULDBLOCK_ISR (ERR_WOULDBLOCK | EMINOR_ISR)

This has been called from an ISR but would result in blocking.

Definition at line 1312 of file tiros.h.

3.15.2.12 #define ERR_WOULDBLOCK_MUTEX (ERR_WOULDBLOCK | EMINOR_PRIO_RULE)

This call would block and is denied because a Mutex is being held.

The current combination of synchronization primitives would result in the unwanted blocking of a mutex.

Definition at line 1317 of file tiros.h.

3.15.2.13 #define ERR_FULL -24

A message queue is full or a counting semaphore is maxed out.

Definition at line 1324 of file tiros.h.

3.15.2.14 #define ERR_NOTOWNER -32

Failure: Not the owner of the specified mutex, or the mutex is not locked.

Definition at line 1330 of file tiros.h.

3.15.2.15 #define ERR_TIMEOUT -40

A timeout has occured.

Definition at line 1335 of file tiros.h.

3.15.2.16 #define ERR_RESUMED -48

Task was resumed while sleeping or waiting for a lock.

Definition at line 1338 of file tiros.h.

3.15.2.17 #define ERR_NOSUCHTASK -56

Specified task does not exist.

Definition at line 1342 of file tiros.h.

3.15.2.18 #define ERR_TASKBLOCKED -64

The operation cannot be completed because a specified task is blocked.

Definition at line 1347 of file tiros.h.

3.15.2.19 #define ERR_PRIO_IN_USE -72

The operation failed because the desired priority is in use.

Definition at line 1350 of file tiros.h.

3.15.2.20 #define ERR_FAILED -80

Failure: reason unspecified.

Definition at line 1353 of file tiros.h.

Chapter 4

Mutual Exclusion (Mutex)

TiROS supports mutual exclusion (Mutex) mechanisms to provide tasks with synchronized access to shared resources.

In an embedded system, mutexes have to be used with care. Mutexes are subject to priority inversion and deadlock. Real-time operating systems implement algorithms to deal with these problems. All implementations are not alike. The system designer must be cognizant of the limitations of the alorithms and the limitations of a particular implementations. TiROS supports two algorithms to deal with mutexes: Cascaded priority inheritance with delayed fallback and Immediate Priority Ceiling Protocol. The following sections are provided so that the system designer can have a good understanding of mutexes, their implementations in TiROS, and the associated limitations:

1. What is a Mutex?
2. Mutexes vs. Critical Sections.
3. Pitfalls of mutexes.
4. Priority Inheritance Protocol.
5. Priority Ceiling Protocol.
6. Rules for using mutexes in TiROS.

The TiROS API for management of mutexes is provided in the Mutexes section of the API description.

4.1 What is a Mutex?

Mutexes are used by a task to provide exclusive access to a shared resource. Mutexes can be used in systems where multiple tasks/threads/processes compete to use the same resources. It is illustrated below:

```
void task1(void) {
        char *ch;
        ch = malloc(100);  // Allocate 100 bytes.
                           // malloc on most embedded systems
                           // is not thread-safe (re-entrant).
                           // It may be possible that task1 is
                           // preempted in the middle of the
                           // malloc and task2 runs and invokes
                           // malloc,
                           // resulting in memory inconsistencies.
        do_stuff(ch);
        free(ch);          // free releases memory. free and
                           // malloc work together. So they both
                           // have to be protected from
                           // concurrent access.
}

void task2(void) {
        char *ch2;
        ch2 = malloc(50);
        do_other_stuff(ch2);
        free(ch2);
}
```

The example uses malloc as a shared resource, but it is applicable for any shared resource, such as serial ports, shared data, etc. In a system with concurrent tasks, these shared resources should be protected from concurrent use. To protect a shared resource, a mutex is associated with it. To use the shared resource, a task locks the mutex. If another task find the mutex locked, it has to wait for the mutex to be unlocked. Then it can lock the mutex and use the shared resource.

Using mutexes, the code would be replaced with:

```
mutex mem_mutex;           // Also do appropriate initialization
                           // at startup
void task1(void) {
        char *ch;
        lock(mem_mutex);  // locking memory allocation routine
        ch = malloc(100);
        unlock(mem_mutex);  // unlocking

        do_stuff(ch);

        lock(mem_mutex);   //locking
        free(ch);          // Free should also be protected.
        unlock(mem_mutex); //unlocking
}

void task2(void) {
        char *ch2;
        lock(mem_mutex);   // If mem_mutex is already locked,
                           // then this task will wait here
                           // until it is unlocked.
        ch2 = malloc(50);
        unlock(mem_mutex);

        do_other_stuff(ch2);
```

```
            lock(mem_mutex);
            free(ch2);
            unlock(mem_mutex);
}
```

Mutexes can be implemented in many ways (In the simplest case, it could be a global interrupt disable and enable). All implementations are not alike. To design reliable systems, the user of a real-time operating system should have a knowledge of the limitations of the operating system mutex implementation.

The following sections explain the different types of mutex implementations and their limitations. Figures showing task scheduling with mutexes are used to illustrate mutex properties. The legend for the illustrations is given below.

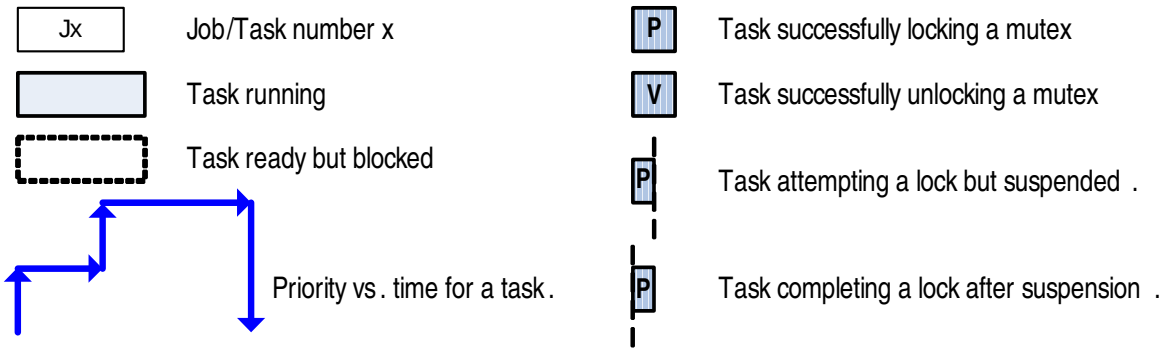

Figure 4.1: Legend for mutex diagrams

4.2 Mutexes vs. Critical Sections

The simplest implementation of a mutex would be to disable global interrupts on mutex lock and restore it on unlock. Simple and fast as it is, it has the undesirable effect of blocking high priority tasks that do not need to be blocked and increasing interrupt response latency. The figure below illustrates this.

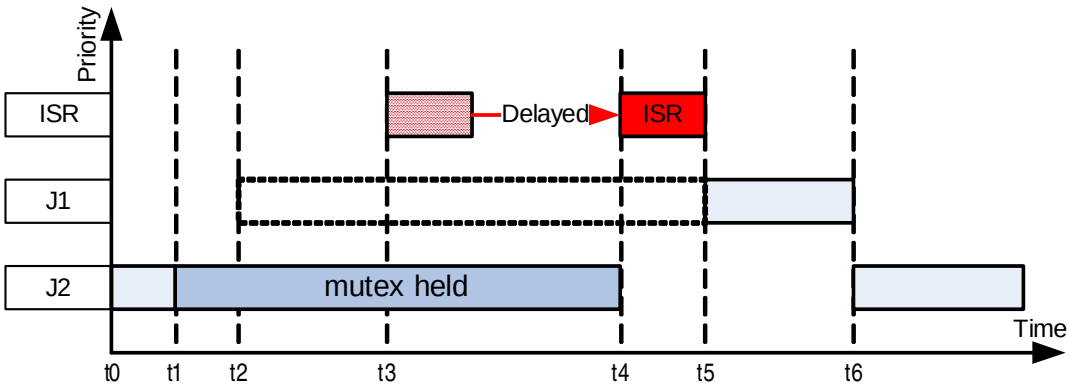

Figure 4.2: Scenario 1: Mutex implementation with interrupt enable/disable

1. Low priority task, J2 is running at time, t0.

2. At time t1, J2 acquires a lock on mutex, M.

3. At time t2, higher priority task, J1, ought to be waking up for execution. However, since the implementation disables interrupts, it stays blocked.

4. At time t3, an interrupt is delivered. However, since interrupts are disabled, the interrupt is not serviced at this time.

5. At time t4, J2 gives up its lock on the mutex and the interrupt state is reenabled. At this time, the interrupt service routine can run.

6. Finally, at time t5, when the ISR is done, higher priority task, J1 can run.

This implementation has the undesirable effect of blocking higher priority tasks and also interrupt routines. In some situations, where the mutex is held for very short periods of time, this may be suitable.In TiROS, this can be achieved using Critical Sections.

For the rest of the discussion, we will restrict ourselves to mutex implementations that do not block interrupts or other high priority tasks. A better mutex implementation of the previous scenario will look as shown in the next figure.

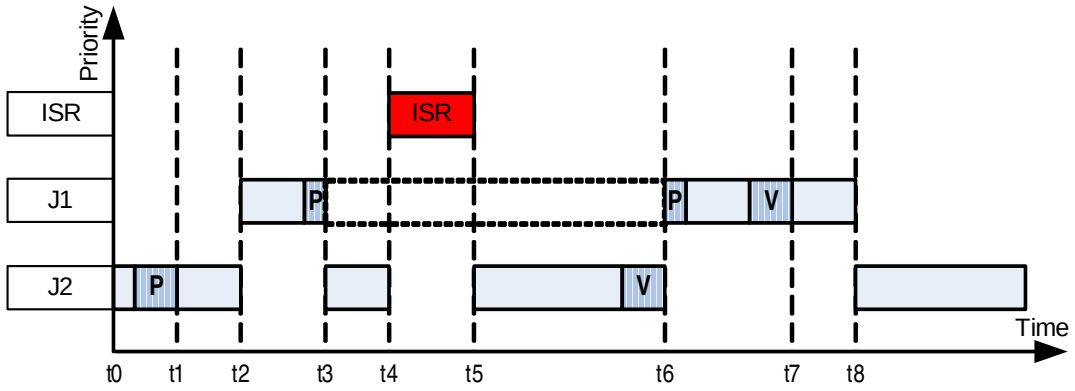

Figure 4.3: The non-trivial Mutex implementation of Scenario 1.

1. Low priority task, J2 is running at time, t0.
2. At time t1, J2 acquires a lock on mutex, M.
3. At time t2, higher priority task, J1, wakes up and runs.
4. At time t3, J1, tries to lock M, but is blocked because M has already been blocked by J2.
5. At time t4, an interrupt is delivered. The ISR runs.
6. At time t5, the ISR is complete and J2 resumes.
7. At time t6, J2 releases the lock on M. J1 can now lock M and continue.
8. At time t7, J1 releases the lock on M.
9. At time t8, J1 yields control (or goes to sleep) and J2 is free to run.

Note that interrupts are not delayed by the task holding the lock.

4.3 Pitfalls of mutexes

4.3.1 Priority Inversion

Consider a system with three tasks: J0, J1, J2. The priorities for these tasks are P0, P1, P2, where P0 $>$ P1 $>$ P2. J2 is the lowest priority task. Consider the following situation:

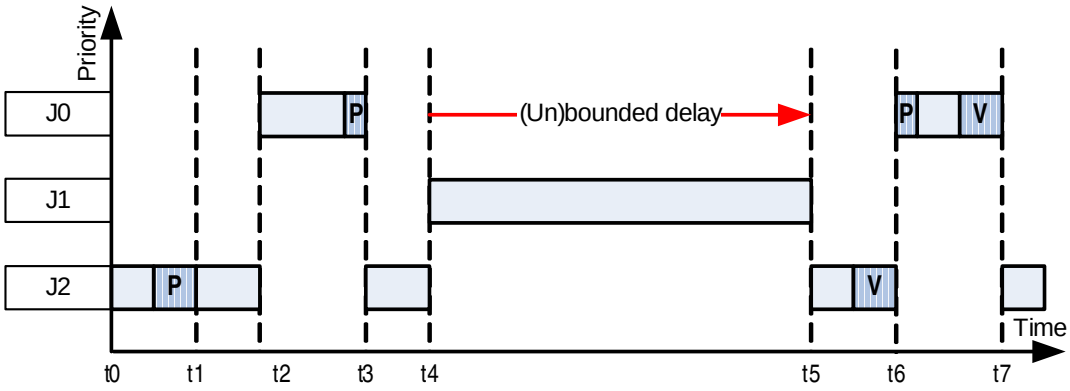

Figure 4.4: Scenario 2: Priority Inversion.

1. At time t1, low priority task, J2 acquires mutex, M.

2. At time t2, high priority task, J0, wakes up and runs.

3. At time t3, J0 tries to lock M but has to wait because it has already been locked by J2..

4. At time t4, medium priority task, J1,starts running a long computation. The low-priority task, J2, cannot complete because J1 is running. Since J2, cannot finish and unlock the mutex, high-priority task J0, is indirectly blocked.

5. At time t5, J1, goes to sleep and low priority task J2 is able to run.

6. At time t6, J2, gives up the lock, this enables J0 to acquire the lock and J0 runs.

Effectively, the high-priority task, J0, is blocked by medium-priority task J1. This condition is called priority-inversion. This is not desirable in a real-time system because the priority-inversion may be unbounded. There are two primary techniques to solve priority-inversion.

- The Priority Inheritance Protocol.
- The Priority Ceiling Protocol.

4.3.2 Deadlock

A deadlock is a condition where two or more tasks are waiting on each other to finish. None of the tasks can advance. Consider this simple scenario illustrated in the figure below:

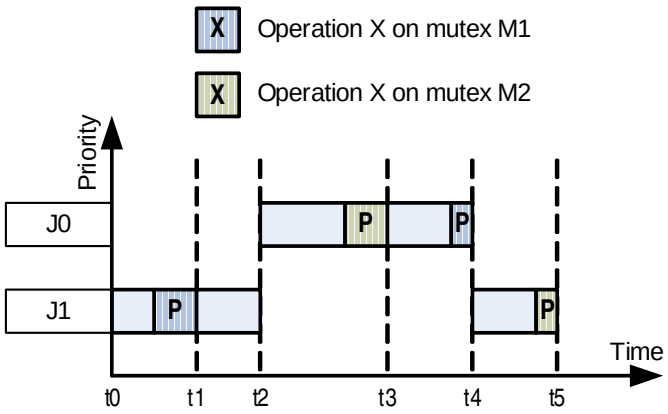

Figure 4.5: Scenario 3: Deadlock.

1. At time t1, low priority task, J1 acquires mutex, M1.

2. At time t2, high priority task, J0, wakes up and runs.

3. At time t3, J0 locks mutex M2.

4. At time t4, J0 tries to lock mutex M1 but has to wait because it has already been locked by J1.

5. At time t5, J0 tries to lock mutex M2 that is already locked by J1. Now, task J0 is waiting on J1 and J1 is waiting on J0. Neither task can progress. They are deadlocked.

In a simple system, deadlocks can be avoided by careful design rules (Example: a given set of mutexes should always be acquired in the same time sequence by any task). In a complex system where many library functions internally use mutexes, this can be more difficult. The Priority Ceiling Protocol can prevent deadlocks.

4.4 Priority Inheritance Protocol

The priority inheritance protocol prevents priority inversion by temporarily raising the priority of the task holding the mutex. Its priority is raised to the priority of the task that is waiting on the mutex, if that is greater. The priority inheritance prevents Priority Inversion as follows.

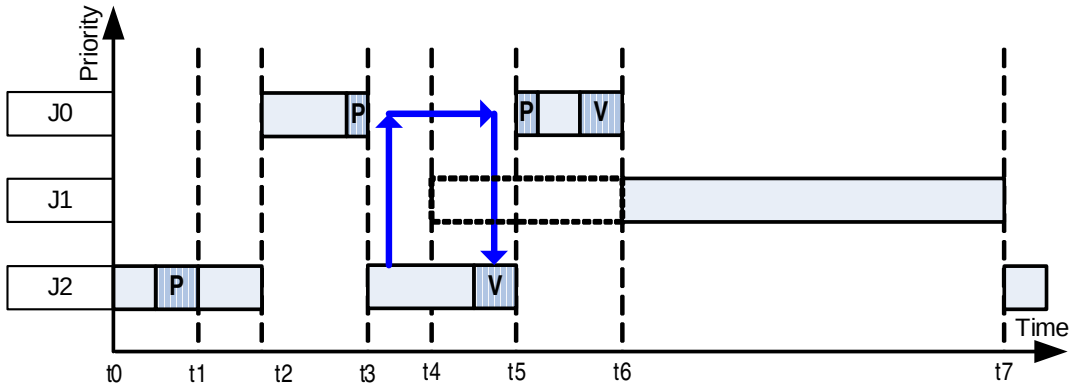

Figure 4.6: Priority Inheritance prevents the priority inversion in Scenario 2.

1. At time t1, low priority task, J2 acquires mutex, M.

2. At time t2, high priority task, J0, wakes up and runs.

3. At time t3, J0 tries to lock M but has to wait because it has already been locked by J2. Since high priority task J0 is now waiting on J2, the priority of J2 is temporarily raised to that of J0.

4. At time t4, medium priority task, J1, wakes up. However, it cannot run because J2 is running with the priority of J0 which is higher than that of J1. J1 is kept waiting.

5. At time t5, J2 releases its lock on the mutex. Its priority reverts back to its original priority. Now that J2 has released its lock, J0 which has been waiting for this can acquire the lock. J0 runs because it is the highest priority task that is now "runnable".

6. At time t6, J0, gives up the lock. Now, medium priority task, J1, which has been waiting can run its long computation.

In a situation with multiple mutexes, this can become more complicated. **Most real-time operating systems do not implement the full priority inheritance protocol but instead implement an approximation to it.** We describe some of the practical implementations of priority inheritance used in embedded real-time operating systems.

4.4.1 Simple priority inheritance (no support for multiple mutexes)

The previous example showed a mutex implementation that works well in a simple case. But consider the following case, where multiple mutexes are involved.

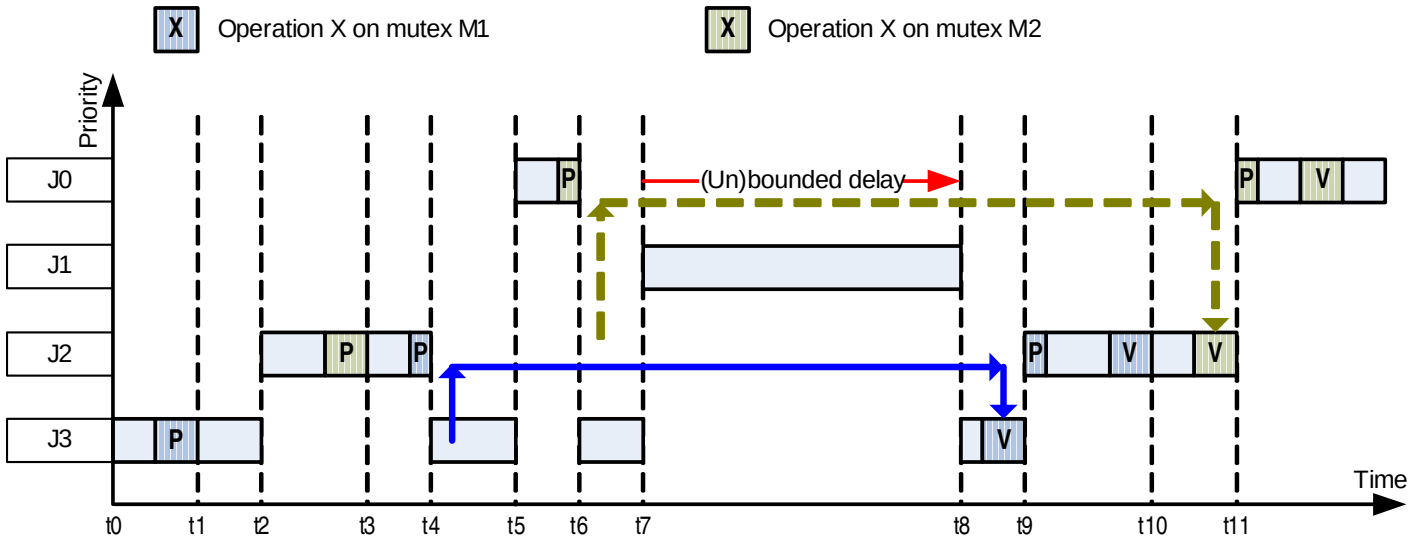

Figure 4.7: Scenario 4: Simple PIP is insufficient for multiple mutexes.

1. At time t1, low priority task, J3 acquires mutex, M1.
2. At time t2, J2 wakes and runs.
3. At time t3, J2 locks mutex M2.
4. At time t4, J2 attempts to lock mutex, M1. J2 is blocked. J3 starts running with the priority of J2.
5. At time t5, J0 wakes and runs.
6. At time t6, J0 attempts to lock mutex M2. M2 is already locked by J2. J0 is blocked and the priority of J2 is raised to that of J0. **However since J2 is already blocked on another mutex M1, this has no effect.** J3 continues to run at the priority of J2.
7. At time t7, J1 wakes up and runs, preempting J2. Now J0 is effectively blocked by J1 which is running a long computation.

This is problematic because J1 can run for an arbitrarily long period of time, leading to unbounded priority inversion. In this case, it is not sufficient to raise the priority of J2. As J2 is blocked on yet another mutex, M1, we have to follow the chain of mutexes and raise the priority of J3, which is the owner of mutex M1. This is what is done by the full priority inheritance protocol.

4.4.2 Full cascaded priority inheritance protocol

As opposed to the previous scenarion, the full priority inheritance protocol has to operate correctly on a chain of blocked tasks, each waiting on a mutex held by a previous task in the chain. The previous scenario

(Scenario 4) is repeated, this time using the full cascased priority inheritance protocol. It is illustrated in the following figure.

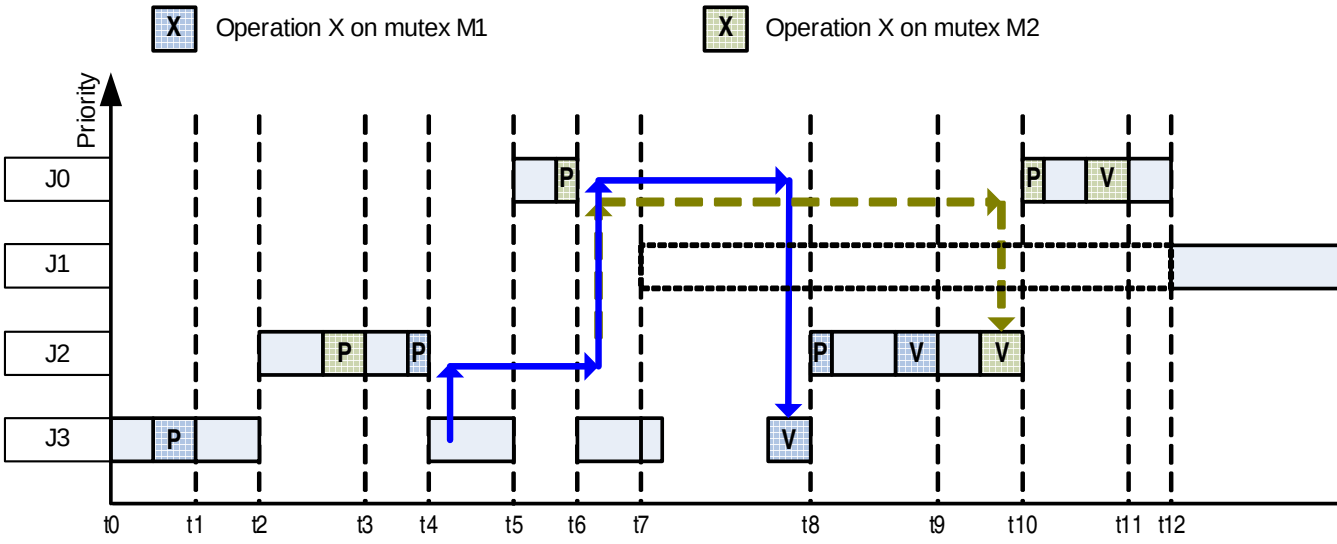

Figure 4.8: Scenario 4: Full PIP: Priority is transitive across locks.

The priority inheritance protocol is simple to describe but complicated to implement in full. Conditions such as nested mutexes, and arbitrary arrival of high-priority tasks have to be considered. In practice, very few operating systems implement the full priority inheritance protocol for the following reasons:

- The full implementation requires keeping track of all locking resources owned by a task etc.
- When a lock is released, the effective priority has to be re-computed. To do this all the locks held by a task have to be examined.
- This algorithm has poor worst-case performance.

For the full PIP, the effective priority of a task is always recomputed in the following case:

- When the task releases a lock.
- When another task blocks(or cancels a block) on a mutex held by the task.
- When another task blocks(or cancels a block) on a different task that in turn is blocking on a mutex held by the task. This applies to larger chains of tasks "related by mutexes".

This is described in Scenario 5 below.

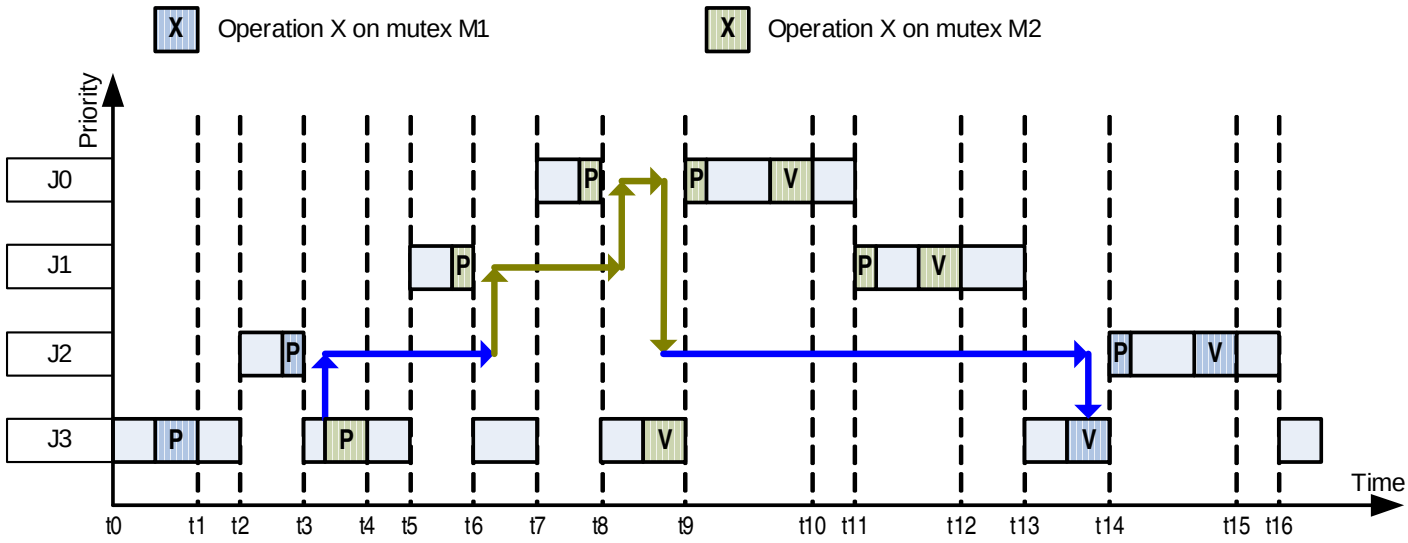

Figure 4.9: Scenario 5: Relinquishing priority with full PIP.

1. At time t1, low priority task, J3 acquires mutex, M1.
2. At time t2, J2 wakes and runs.
3. At time t3, J2 attempts to lock mutex M1, but is blocked. The priority of J3 is raised to that of J2.
4. At time t4, J3 has locked mutex M2.
5. At time t5, J1 wakes and runs, thus preempting J3.
6. At time t6, J1 attempts to lock mutex M2 which is locked by J3. It is blocked. This raises the priority of J3 to that of J1.
7. At time t7, J0 wakes, preempts J3.
8. At time t8, J0 blocks on M2. The priority of J3 is now raised to that of J0.
9. At time t9, J3 releases the lock on M2. Its priority is decreased to that of J2, because J2 is still waiting for mutex M1 which is still locked by J3. Now J0 can finish its lock on M2.
10. At time t10, J0 releases its lock on M2.
11. At time t11, J0 goes to sleep. The next highest priority task is J1 which is now also allowed to complete its lock on M2.
12. At time t12, J1 releases M2.
13. At time t13, J1 goes to sleep. Now J3 continues running at the priority of J2.

14. At time t14, J3 releases its lock. Its priority is lowered to its original priority. Now J2 is the highest priority runnable task. J2 is now allowed to complete its lock on M1.

15. At time t15, J2 releases M1.

4.4.3 Cascaded priority inheritance with delayed fallback

This is an approximation to the full priority inheritance protocol. The difference between this approximate algorithm and the full algorithm is the effect upon the effective priority of a task when a lock is released.

The effective priority is only lowered when there are no more mutexes held by the task. This modified algorithm is much simpler to implement than the full protocol. It requires much less state, memory, and overhead. In practice, most systems that implement priority inheritance implement the modified algorithm. The behavior of the modified algorithm with Scenario 5 is described below.

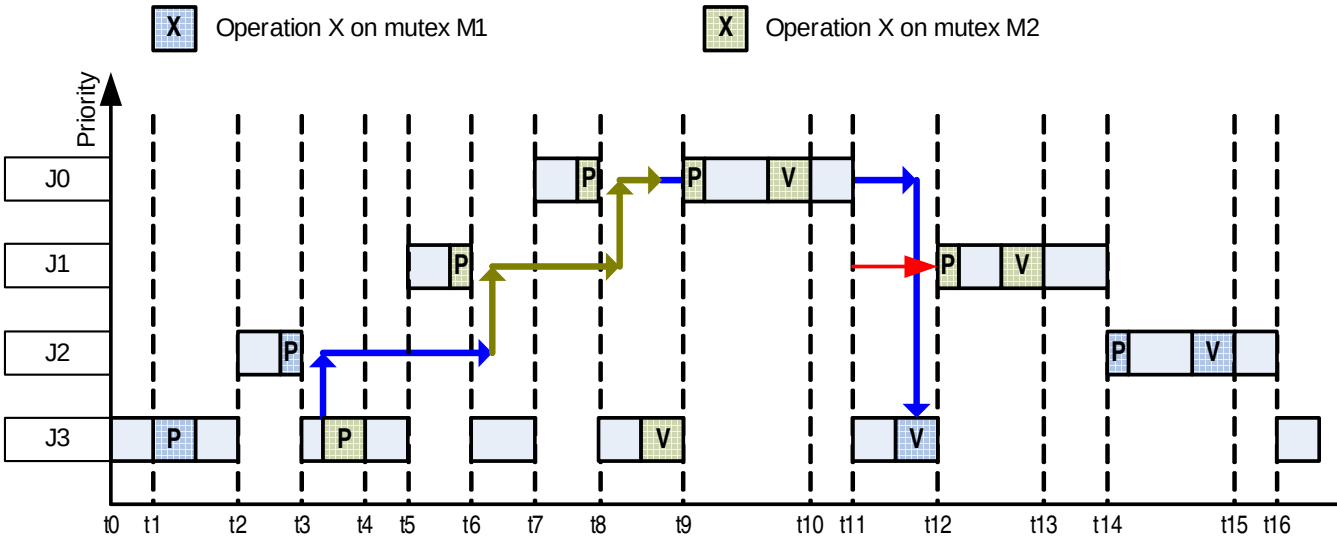

Figure 4.10: Scenario 5: Cascaded priority inheritance with lazy fallback.

1. At times t0-t8, the behavior is the same as that of the full PIP.

2. At time t9, J3 releases the lock on M2. In this approximate algorithm, its priority is not immediately decreased. Note, that now there are two tasks with the same priority (J3 and J0). Depending on the specifics of the task scheduler for the operating sytem, either one of these tasks may continue running. It may be preferable to have task J0 continue because it is at a higher priority task than J3. On the other hand, swithching a task involves a context-switch overhead. Either implementation is correct because mutexes are assumed to hold a resource for only short periods of time. And in any case, there is no unbounded priority inversion. For this case, we choose J0 as the task to continue.

3. At time t10, J0 releases the lock on M2.

4. At time t11, J0 goes to sleep. The highest priority task is J3, which has the effective priority of J0. J3 runs.

5. At time t12, J3 releases its lock on M1. It reverts back to its original priority. The highest priority task is now J1 which allowed to complete its lock on M2.

6. At time t13, J1 releases its lock on M2.

7. At time t14, J1 sleeps. This enables J2 to run and complete its lock on M1.

8. At time t15, J2 releses its lock on M1.

9. At time t16, J2 goes to sleep. This allows J3 to run.

4.5 Priority Ceiling Protocol

In the priority ceiling protocol, a mutex is associated with a ceiling priority. Only tasks whose priority is lower than the ceiling protocol, are allowed to use the mutex. It works like PIP, in that a task blocked by the mutex raises the owner of the mutex to its priority. This priority cannot go above the ceiling set for the mutex because by definition, tasks with a higher priority are not permitted to use the mutex and thus cannot block on it. The **Priority Ceiling Protocol can prevent deadlock.** The disadvantage of using the PCP protocol is that task priorities must be known ahead of time. Tasks are not allowed to dynamically change their priorities (except as part of the protocol).

4.5.1 Original Priority Ceiling Protocol

The original priority ceiling protocol is specified as follows:

1. Every task has a static priority.

2. Every mutex has a priority ceiling. A task is not allowed to lock a task whose ceiling priority is lower than its effective priority. Correspondingly, the priority ceiling of a mutex should be greater than or equal to that of any task that might use it.

3. A task is only allowed to lock a mutex if its priority is higher than all priority ceilings of all active mutexes in the system. This sounds complex but in a system where every task has a unique priority, the scheduler automatically takes care of this (if a task holding a mutex cannot block on other primitives as in TiROS).

4. When a high priority task blocks on a mutex, the task owning the mutex is raised to the priority of the high-priority task. This is exactly as is in the Priority Inheritance Protocol. However, there is a maximum priority level which is the priority ceiling of the mutex.

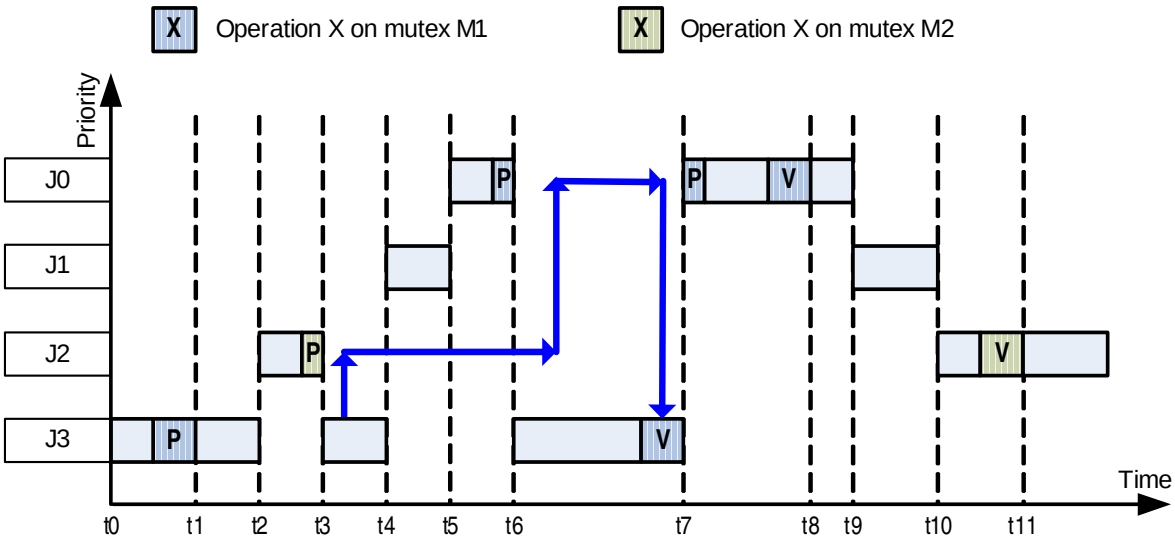

Figure 4.11: Scenario 6: Original Priority Ceiling Protocol

1. There are two mutexes M1, and M2. The highest priority task that will ever access M1 is J0. Thus the ceiling priority of M1 is greater than or equal to the priority of J0. The highest priority task that will access M2 is J2. The ceiling priority of M2 is greater than or equal to the priority of J2.
2. At time t1, low priority task, J3 acquires mutex, M1.
3. At time t2, J2 wakes and runs.
4. At time t3, J2 attempts to lock mutex M2, but is blocked. Note that J2 is blocked even though mutex M2 is available. This blockage happens by the application of Rule 3) above. The priority of J2 is not higher than the priority ceilings of other active mutexes (i.e., M1). Since J2 is blocked by J3, J3 resumes with the priority of J2.
5. At time t4, J1 wakes and runs.
6. At time t5, J0 wakes and runs.
7. At time t6, J0 attempts to lock mutex M1, but is blocked. M1 is already locked by J3. The priority of J3 is raised to that of J0.
8. At time t7, J3 releases the lock on M1. It is restored to its original priority. J0 can now run and complete the lock on M1.
9. At time t8, J0 releases its lock on M1.
10. At time t9, J0 goes to sleep. This allows J1 to run.
11. At time t10, J1 goes to sleep allowing J2 to run. J2 can now complete its lock on M2.
12. At time t11, J2 releases the lock on M2.

4.5.2 Immediate Priority Ceiling Protocol

The Immediate Priority Ceiling Protocol (also called Priority Protect Protocol in POSIX, Priority Ceiling Emulation Protocol in Real-Time Java) is a modification to the original PCP protocol. This is the algorithm that is typically implemented in operating systems that support the priority ceiling protocol. It is simpler to implement and provides higher performance than the original protocol. It is specified as follows:

1. Every task has a static priority (Similar to OPCP).
2. Every mutex has a priority ceiling. A task is not allowed to lock a mutex whose ceiling priority is lower than its effective priority. (Similar to OPCP).
3. When a task locks a mutex, its priority is immediately raised to the priority ceiling (Differs from OPCP).

To illustrate the differences from OPCP, we will re-enact Scenario 6 using IPCP.

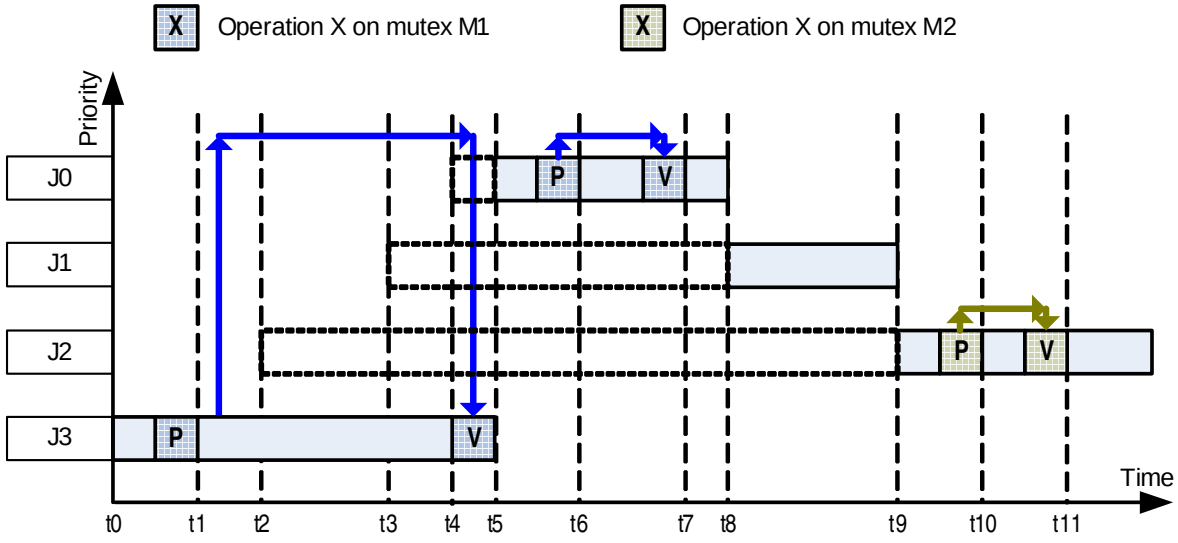

Figure 4.12: Scenario 6: Immediate Priority Ceiling Protocol

1. The ceiling priority of M1 is greater than or equal to the priority of J0. The ceiling priority of M2 is greater than or equal to the priority of J2.
2. At time t1, low priority task, J3 acquires mutex, M1. The priority of J3 is raised to the priority ceiling (greather than that of J0).
3. At time t2, J2 wakes but cannot run.
4. At time t3, J1 wakes but cannot run.

5. At time t4, J0 wakes but cannot run..
6. At time t5, J3 unlocks M1. Its original priority is restored and J0 now runs.
7. At time t6, J0 locks M1 and runs at the priority ceiling of M1.
8. At time t7, J0 releases its lock on M1 and its original priority is restored.
9. At time t8, J0 goes to sleep. This allows J1 to run.
10. At time t9, J1 goes to sleep allowing J2 to run.
11. At time t10, J2 locks M2 and runs at the priority ceiling of M2.
12. At time t11, J2 unlocks M2 and its original priority is restored.

Note also that IPCP uses fewer task-switches. This translates to reduced overhead.

4.5.3 Deadlock Prevention using PCP

The Priority Ceiling Protocol has an important property that is extremely valuable in building reliable embedded systems. It can eliminate deadlock!

Consider the scenario that was presented earlier to illustrate Deadlock. Now, lets apply IPCP (OPCP would work just as well) to this scenario.

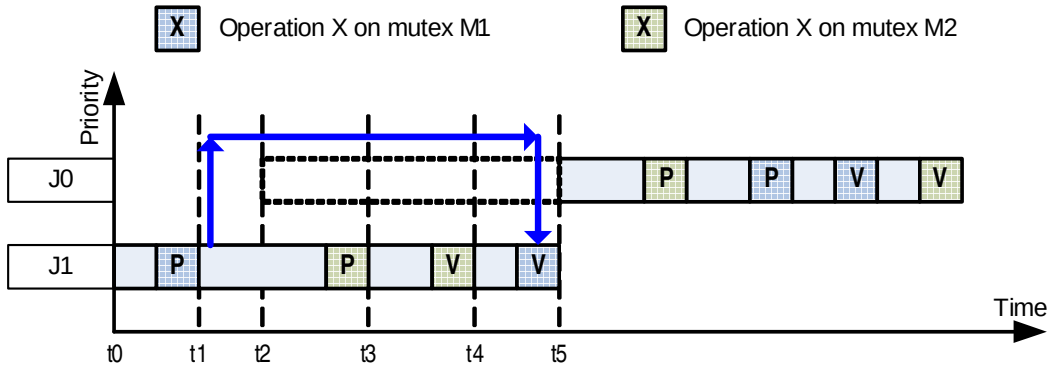

Figure 4.13: Deadlock prevention using IPCP

1. In this example, M1 and M2 have the same ceiling priority. They both have a priority greater than or equal to the priority of J0.
2. At time t1, low-priority task J1 has locked mutex M1. The priority of J1 is thus raised to the priority ceiling.

3. At time t2, task J0 wakes. However, it cannot run because J1 is running, which has higher or equal priority as J1.
4. At time t3, task J1 locks mutex M2.
5. At time t4, task J1 releases mutex M2.
6. At time t5, task J1 releases mutex M1. It then reverts to its original priority. Since the priority of J0 is now greater than that of J1, task J0 is now allowed to run. It can then proceed to lock and unlock M1, M2 without obstruction.

This scenario is similar to that shown earlier (Deadlock), but with PCP applied. It shows how deadlock can be prevented.

4.6 Rules for using mutexes in TiROS

1. To deal with priority inversion, TiROS provides an option of using Cascaded priority inheritance with delayed fallback or Immediate Priority Ceiling Protocol. One and only one of these algorithms can be used at a time.
2. A mutex cannot be used from an interrupt service routine. Mutex locks and unlocks have to occur in pairs. To enforce priority protocols, a mutex has to be owned by a task. If a global resource also has to be used from an interrupt handler, consider using Critical Sections. Also see Usage with Interrupt Service Routines.
3. A task acquiring multiple mutexes must release them in the reverse order.
4. A task that is holding a mutex is not allowed to perform a blocking-wait on other types of synchronization primitives (locks) such as Counting semaphores, Message Queues, and Event Flags. Non-blocking operations are permitted. Blocking on additional mutexes is also permitted. This is because the priority protocols are not transitive across different types of locks. There is no concept of an owner with counting semaphores or message queues. Priority protocols only operate on owned locks, i.e., mutexes. In the default configuration, explicit sleeping is also disallowed while holding a mutex. However, using the TIROS_ALLOW_SLEEP_W_MUTEX configuration directive, this can be bypassed. This can cause undesirable blocking but it is assumed that the user has an explicit reason for doing this. So use this with care.
5. All tasks should have unique priorities.
6. Mutexes should have unique ceiling priorities that do not correspond to any task priorities.

Chapter 5

Porting to other hardware

5.1 Porting Guide

This section describes the porting of TiROS to a new platform.

The hardware specific port includes several files located in ROOT/src/tiros/port/port_xxx A tr_port.h is required. There may be an associated tr_port.c and other assembly files if needed. There will also be a file called porttime.h which contains the time addition, subtraction specific to your port. The tr_port.h file has several sections.

1. In a comment section at the top of the file, describe the port and the configuration options specific to the port.
2. Include the user's configuration file proj_config.h.
3. Define basic types and OS configuration for your port.
4. Critical Sections and Interrupts.
5. Context Switching Functions.
6. Port specific time functions.
7. Setup for the kernel trap.
8. Debug Functions.

5.2 Define basic types and OS configuration for your port

Defines

- #define subtime_t uint16_t
- #define LT_INLINE static inline
- #define TRPORT_INF_TIME {∼0 , ∼0 }
- #define TIROS_REGISTER_PASSING
- #define ILLEGAL_STACK ((osstkptr_t) ∼0)
- #define ILLEGAL_ADDR ((osptr_t) ∼0)

Typedefs

- typedef unsigned char uint8_t
- typedef char int8_t
- typedef unsigned short uint16_t
- typedef short int16_t
- typedef unsigned long uint32_t
- typedef long int32_t
- typedef unsigned int osword_t
- typedef void ∗ osptr_t
- typedef osword_t ∗ osstkptr_t
- typedef void(∗) osfnptr_t (void ∗)
- typedef unsigned int osptrword_t

5.2.1 Define Documentation

5.2.1.1 #define subtime_t uint16_t

Define the type field used for subunits in the time structure.

Definition at line 103 of file template/tr_port.h.

5.2.1.2 #define LT_INLINE static inline

Do you want the time functions inlined ?

Definition at line 105 of file template/tr_port.h.

5.2.1.3 #define TRPORT_INF_TIME {∼0 , ∼0 }

Define the value of infinite time for this port.

Definition at line 115 of file template/tr_port.h.

5.2.1.4 #define TIROS_REGISTER_PASSING

Will you be using register passing? This is an optimization and may be difficult on some platforms.

In that case, comment this out.

Definition at line 121 of file template/tr_port.h.

5.2.1.5 #define ILLEGAL_STACK ((osstkptr_t) ∼0)

Create a definition for an invalid stack pointer.

Definition at line 157 of file template/tr_port.h.

5.2.1.6 #define ILLEGAL_ADDR ((osptr_t) ∼0)

Create a definition for an illegal memory address.

Definition at line 159 of file template/tr_port.h.

5.2.2 Typedef Documentation

5.2.2.1 typedef unsigned char uint8_t

Define the different integer types.

or include a header file such as stdint.h that already has these defined.

Definition at line 91 of file template/tr_port.h.

5.2.2.2 typedef unsigned int osword_t

Basic type for the hardware word.

An 8 bit system would have an 8-bit integer, a 16-bit system would have a 16 bit integer, etc. On a system with stdint.h, this could be uintptr_t

Definition at line 127 of file template/tr_port.h.

5.2.2.3 typedef void∗ osptr_t

Type to use for generic pointer.

Pointer types: Separate pointer types are defined for a generic pointer, stack pointer, and function pointer. On many architectures, this may be the same. However, there may be a need to keep this different on some hardware. For example, on an 8051, the generic pointer may be set to refer to the IDATA memory, while the stack pointer may be set to XDATA (which takes more bytes for address storage). Other hardware with

interesting addressing such as a 20-bit address or a 23-bit address may need to keep these different types of pointers separate

Definition at line 141 of file template/tr_port.h.

5.2.2.4 typedef osword_t* osstkptr_t

Type to use for stack pointer.

Definition at line 144 of file template/tr_port.h.

5.2.2.5 typedef void(*) osfnptr_t(void *)

Type to use for a function pointer.

Definition at line 147 of file template/tr_port.h.

5.2.2.6 typedef unsigned int osptrword_t

This is an integer type that can fully represent osptr_t.

If a pointer takes two bytes of storage, then a 16 bit integer should be used, if it takes 3 bytes of storage, then a 32 bit integer should be used, etc. If your system has a <stdint.h> file, then this could be of type uintptr_t (defined in stdint.h)

Definition at line 154 of file template/tr_port.h.

5.3 Critical Sections and Interrupts

5.3.1 Detailed Description

These functions are defined for using critical sections and dealing with interrupts.

Defines

- #define OS_PORT_CRITICAL_ENABLE()
- #define OS_PORT_CRITICAL_BEGIN()
- #define OS_PORT_CRITICAL_END()
- #define OS_PORT_ISR_BEGIN()
- #define OS_PORT_INT_ENABLE()
- #define OS_PORT_INT_DISABLE()

5.3.2 Define Documentation

5.3.2.1 #define OS_PORT_CRITICAL_ENABLE()

Enable critical section calls in the the called function.

OS_PORT_CRITICAL_ENABLE() will be called at the beginning of any function that uses OS_-CRITICAL_BEGIN or OS_CRITICAL_END. This will be the first call after variable declaration. This is a good place to define variables that may be used by OS_BEGIN_CRITICAL and OS_CRITICAL_END

Definition at line 176 of file template/tr_port.h.

5.3.2.2 #define OS_PORT_CRITICAL_BEGIN()

Begin a critical section A simple way to do this would be to disable interrupts.

However, it would be better if the current interrupt state were stored and then interrupts were disabled. Upon ending the critical section, the stored state should be restored. The stored state can be saved in a local variable, which can be defined within OS_PORT_CRITICAL_ENABLE()

Definition at line 185 of file template/tr_port.h.

5.3.2.3 #define OS_PORT_CRITICAL_END()

End a critical section This may be as simple as reenabling interrupts, but it is better to renable the interrupt state stored in OS_PORT_BEGIN_CRITICAL().

Definition at line 191 of file template/tr_port.h.

5.3.2.4 #define OS_PORT_ISR_BEGIN()

Value:

```
extern uint8_t os_isr_nesting;                          \
        os_isr_nesting++;
```

This macro will be called at the beginning of an ISR.

Leave this as such

Definition at line 196 of file template/tr_port.h.

5.3.2.5 #define OS_PORT_INT_ENABLE(void)

Enable interrupts This allows the user to explicitly enable interrupts.

Definition at line 202 of file template/tr_port.h.

5.3.2.6 #define OS_PORT_INT_DISABLE(void)

Disable interrupts This allows the user to explicitly disable interrupts.

Definition at line 206 of file template/tr_port.h.

5.4 Context Switching Functions

5.4.1 Detailed Description

These functions should be implemented to manage a per task stack, and saving and restoring context.

The context is stored so that it looks like the way the stack would if all the registers were stored by an interrupt function.

Defines

- #define TRPORT_MIN_CTXT_SZ 64

Functions

- osstkptr_t hal_stk_init (osword_t *stk, osword_t stacksize, osfnptr_t pc, osptr_t init_arg)
- static void hal_retval_set (osstkptr_t ctxt_ptr, osptr_t retval)
- osstkptr_t hal_ctxt_switch (void)
- void hal_ctxt_load (osstkptr_t ctxt_ptr)
- void hal_init (void)

5.4.2 Define Documentation

5.4.2.1 #define TRPORT_MIN_CTXT_SZ 64

Define the minimum size of the context in oswords.

Definition at line 221 of file template/tr_port.h.

5.4.3 Function Documentation

5.4.3.1 osstkptr_t hal_stk_init (osword_t * *stk*, osword_t *stacksize*, osfnptr_t *pc*, osptr_t *init_arg*)

Initialize a stack and create a new context for a task.

Parameters:

stk Pointer to the stack.

stacksize Size of the stack.

PC Program counter.

init_arg Initial argument passed to the task.

Returns:

The pointer to the task context to be used in load/save context.

5.4.3.2 static void hal_retval_set (osstkptr_t *ctxt_ptr*, osptr_t *retval*) [inline, static]

Set a return value for a task whose context is stored.

This is only needed if TIROS_REGISTER_PASSING is defined for your port. This injects a return value into the context of a task that is currently frozen.

Parameters:

ctxt_ptr Pointer to the context of the task.

retval The return value to be injected into the task context

Definition at line 239 of file template/tr_port.h.

5.4.3.3 osstkptr_t hal_ctxt_switch (void)

Context switch from user level (non-ISR) This function is called from within an OS call, to switch the running process.

It also provides a return value. This is a pseudo return-value, This function does not actually determine the value returned. The return value is set by hal_retval_set which is often invoked when waking the process up from a blocked state. This function may have to be implemented in assembly based on the port. The function takes the following form

```
osstkptr_t hal_ctxt_switch(void) {
     osstkptr_t new_ctxtptr, current_ctxtptr;
     current_ctxtptr = save_current_context();
     new_ctxtptr = osint_taskswitcher(current_ctxtptr);
     load_new_context(new_ctxtptr);

     // The code should not reach this far.  When
     // new context is loaded, it will jump out of the
     // hal_ctxt_switch function.
       return 0;
}
```

The function may also take the following form when there is no easy access to the current stack value.

```
osstkptr_t hal_ctxt_switch(void) {
     osstkptr_t ctxt_ptr, new_ctxt_ptr;
     ctxt_ptr = osint_running_task_ctxt();
     if (stkptr != ILLEGAL_STACK) {
           save_current_context(ctxt_ptr);
           new_ctxt_ptr = osint_taskswitcher(ctxt_ptr);
           load_new_context(new_ctxt_ptr);
```

```
      } else {
            new_ctxt_ptr = osint_taskswitcher(ctxt_ptr);
            load_new_context(new_ctxt_ptr);
      }
      // The code should not reach this far.  When
      // new context is loaded, it will jump out of the
      // hal_ctxt_switch function.
        return 0;
}
```

Returns:

Return value.

5.4.3.4 void hal_ctxt_load (osstkptr_t *ctxt_ptr*)

Load the given context.

Parameters:

ctxt_ptr Pointer to task context

5.4.3.5 void hal_init (void)

Initialize the hardware.

Any hardware specific initialization can be performed her. Timer and kernel trap initialization can also be performed within this function.

5.5 Port specific time functions

Defines

- #define TRPORT_RESPONSE_TIME { 0, 80 }

Functions

- void hal_time_get (trtime_t *lt)
- int hal_time_set (const trtime_t *lt)
- void hal_alarm_set (const trtime_t *lt)

5.5.1 Define Documentation

5.5.1.1 #define TRPORT_RESPONSE_TIME { 0, 80 }

OS response time.

The OS wakes a task at the absolute time specified. There might however be some little overhead involved in waking it up that might delay the wakeup. So a task that wants to wake up for a very short duration of time might actually get delayed because it takes some time for the OS to put it to sleep and wake it up. To avoid this, TiROS takes the response overhead into account when scheduling

Definition at line 344 of file template/tr_port.h.

5.5.2 Function Documentation

5.5.2.1 void hal_time_get (trtime_t * *lt*)

Get the current time.

Parameters:

→ ***lt*** Pointer to time structure

5.5.2.2 int hal_time_set (const trtime_t * *lt*)

Set the current time.

This may not always be possible, depending on the port.

Parameters:

← ***lt*** Pointer to time structure.

Returns:

1 if successful, 0 if not possible.

5.5.2.3 void hal_alarm_set (const trtime_t * *lt*)

Set an alarm for the specified time.

When the alarm is reached, osint_alarm_reached is called

Parameters:

← ***lt*** Pointer to time structure. If lt is set to zero, then disable alarms.

5.6 Setup for the kernel trap

5.6.1 Detailed Description

The kernel trap is used to reduce interrupt latency.

When an interrupt occurs, the task context is not saved. At the end of the ISR, there is a check made to see if a new task has been enabled by the processing within the interrupt.

In most cases, there is no change and processing returns to the task that was interrupted. If this is the case, there is no overhead for saving and restoring all the task context.

In cases where a different task must run, a software interrupt is raised at the end of the ISR. When the ISR returns, the kernel trap ISR is invoked due to the software interrupt. This can save the context for the current task and restore the context for the new task.

Defines

- #define TIROS_KERNEL_TRAP_ENABLED 1

Functions

- void OS_KERNEL_TRAP (void)

5.6.2 Define Documentation

5.6.2.1 #define TIROS_KERNEL_TRAP_ENABLED 1

Is a kernel trap possible on this architecture? If not, set TIROS_KERNEL_TRAP_ENABLED to 0.

Definition at line 374 of file template/tr_port.h.

5.6.3 Function Documentation

5.6.3.1 void OS_KERNEL_TRAP (void)

Implement the code to Invoke a kernel trap (i.e., create a software interrupt).

OS_KERNEL_TRAP() is called from within an ISR. On hardware where kernel trap is not possible, this function may set a flag which determines if the scheduler has to be invoked at the end of the ISR.

Chapter 6

Port Documentation

6.1 HAL for MSP430 family with mspgcc compiler

This is the TiROS hardware abstraction layer for the Texas Instruments MSP430x family of embedded microcontrollers.

This microcontroller family can support software interrupts. These microcontrollers can be driven from a 32kHz crystal oscillator. The time functions have been written assuming such an oscillator drives the timer. If this is not the case, modify the functions in porttime.h and the timer initialization functions in tr_port.c.

Note that in the MSP430 architecture, the low-power state of the processor is stored in the status register. Since this is part of the context, per-task power control is automatic. To keep the processor in low-power state during idle time, simple define the idle task as

```
void idle_task(void *dummy)
{
  while(1) {
      _BIS_SR(LPM3_bits);
  }
}
```

Tested with TI MSP430x1611

Configuration Options:

TRPORT_MSP_OSTIMER [A | B] : Options are A or B. The MSP430 family supports two timers, TimerA and TimerB. TimerA is available on the entire family line. TimerB is only available on some. By default, if this configuration option is not set, TimerA is used as source for the timer interrupt.

TIROS_KERNEL_TRAP_ENABLED [0|1]: If this define is set to 1, OS kernel context switches from ISRs are made using a software trap (i.e.) by causing an interrupt under software control. This makes ISRs very efficient. The MSP430/GCC port supports kernel traps. So this is enabled as a default. To disable it, set TIROS_KERNEL_TRAP_ENABLED to 0 in proj_config.h.

USER_DEFINED_KERNEL_TRAP : If TIROS_KERNEL_TRAP_ENABLED is set to 1, kernel traps are enabled. The default implementation uses Port 2, pins. By setting a specific pin, an interrupt is forced. The user can override this default and specify a different interrupt handler for this purpose. In this case, create user definitions for isr_kernel_trap and OS_KERNEL_TRAP, and hal_setup_kernel_trap() . Do not define or set these functions if TIROS_KERNEL_TRAP_ENABLED is set to 0.

Writing ISRs.

If no TiROS API functions will be called, interrupts will stay disabled, then the ISR can be written very simply. See the requirements in Usage with Interrupt Service Routines.

```
 interrupt (IVECTOR) ISR_function(void)
{
 // do stuff
}
```

If TiROS API calls have to be used, the form of the ISR depends on whether TIROS_KERNEL_TRAP_ENABLED is set (It is set as a default). If TIROS_KERNEL_TRAP_ENABLED is set to 1, the ISR is simple

```
interrupt (IVECTOR) ISR_function(void)
{
  int x;
  OS_CRITICAL_ENABLE();    // Only needed if any critical sections
                           // will be used and interrupt nesting is enabled.

  OS_ISR_BEGIN();          // Mark the beginning of an ISR. Note
                           // that if OS_CRITICAL_ENABLE() is also
                           // present, then this comes after
                           // that.

 x = 3;
   Do more stuff ....

 OS_ISR_END();
}
```

If TIROS_KERNEL_TRAP_ENABLED is set to 0, the ISR would look as below. Use the ISR implementation in tr_port.c as a reference.

```
interrupt (IVECTOR) ISR_function(void) __attribute__ ( (naked))
interrupt (IVECTOR) ISR_function(void)
{
   HALINT_ISR_ENTRY();
   OS_ISR_BEGIN();
   perform_actions();
   OS_ISR_END();
   HALINT_ISR_EXIT();
}
```

6.2 HAL for MSP430 family with IAR compiler

This is the TiROS hardware abstraction layer for the Texas Instruments MSP430x family of embedded microcontrollers with the IAR compiler.

This microcontroller family can support software interrupts. These microcontrollers can be driven from a 32kHz crystal oscillator. The time functions have been written assuming such an oscillator drives the timer. If this is not the case, modify the functions in porttime.h and the timer initialization functions in tr_port.c.

Note that in the MSP430 architecture, the low-power state of the processor is stored in the status register. Since this is part of the context, per-task power control is automatic. To keep the processor in low-power state during idle time, simple define the idle task as

```
void idle_task(void *dummy)
{
  while(1) {
    // Set low power state here
  }
}
```

Tested with TI MSP430x1611 To use the IAR tools for development: Set the following directories in the include path for project options (C compiler->Preprocessor)

1. $PROJ_DIR$ (containing your proj_config.h file)
2. path to the inc directory
3. path to src/tiros
4. path to src/tiros/port/msp430_iar

Configuration Options:

TRPORT_MSP_OSTIMER [A | B] : Options are A or B. The MSP430 family supports two timers, TimerA and TimerB. TimerA is available on the entire family line. TimerB is only available on some. By default, if this configuration option is not set, TimerA is used as source for the timer interrupt.

TIROS_KERNEL_TRAP_ENABLED [0|1]: If this define is set to 1, OS kernel context switches from ISRs are made using a software trap (i.e.) by causing an interrupt under software control. This makes ISRs very efficient. The MSP430_IAR port currently requires kernel traps. So this is enabled as a default.

USER_DEFINED_KERNEL_TRAP : If TIROS_KERNEL_TRAP_ENABLED is set to 1, kernel traps are enabled. The default implementation uses Port 2, pins. By setting a specific pin, an interrupt is forced. The user can override this default and specify a different interrupt handler for this purpose. In this case, create user definitions for isr_kernel_trap and OS_KERNEL_TRAP, and hal_setup_kernel_-trap() . Do not define or set these functions if TIROS_KERNEL_TRAP_ENABLED is set to 0.

Writing ISRs.

If no TiROS API functions will be called, interrupts will stay disabled, then the ISR can be written very simply. See the requirements in Usage with Interrupt Service Routines.

```
#pragma vector=VECTOR_NUM
 __interrupt void ISR_function(void)
{
 // do stuff
}
```

If TiROS APIs will be used, the form of the ISR depends on whether TIROS_KERNEL_TRAP_ENABLED is set (It is set as a default). If TIROS_KERNEL_TRAP_ENABLED is set to 1, the ISR is simple

```
#pragma vector=VECTOR_NUM
__interrupt void ISR_function(void)
{
  int x;
  OS_CRITICAL_ENABLE();   // Only needed if any critical sections
                          // will be used and interrupt nesting is enabled.

  OS_ISR_BEGIN();         // Mark the beginning of an ISR. Note
                          // that if OS_CRITICAL_ENABLE() is also
                          // present, then this comes after
                          // that.

 x = 3;
   Do more stuff ....

 OS_ISR_END();
}
```

This port currently requires TIROS_KERNEL_TRAP_ENABLED.

6.3 Posix HAL

This port makes a posix system look like real hardware to TiROS.

TiROS runs within a posix process. This port provides the context switching and time functions to TiROS. It simulates interrupts using posix signals. The posix port does not use kernel traps (They work on Linux but do not work reliably on Cygwin). The signal implementation under Cygwin is such that storing and restoring a context within a signal handler does not work properly. So we bypass this for the Cygwin environment and use direct Win32 api calls. This port has been tested on Linux 2.6.x kernel series/Glibc and on Cygwin 5.1. This port does not work on Mac OS X (as of 10.4), because the needed posix functions are not implemented.

This port does not allow the time to be set.

Currently, this port does not use the user provided stack memory for the task stack. This port stores the context on the user provided memory. For the task stack, it allocates new memory (memory size defined by TIROS_POSIX_STKSZ)

Configuration Options:

PORT_TIMER_TYPE : Options are 1 or 2. Setting PORT_TIMER_TYPE to 1 uses the system real time. Setting PORT_TIMER_TYPE to 2, uses the process time. This option is irrelevant to Cygwin as only real time is available.

TIROS_POSIX_STKSZ : The stack size allocated to each task. When os_task_create is called, the stack location and stack size for the task are passed along to the port. The Posix port stores the context at this provided stack location. The context size is fixed based on the registers etc. The rest of this provided stack is unused. The stack for the task is allocated separately using a mmap. By default, this stack is 16384 bytes. This was done because sometimes simple calls like printf on a posix system will easily overrun a small stack (tens of bytes) that might be specified for an embedded system. Allocating a larger stack on a posix system might help in running the code on both platforms without modification. Override the default by specifying TIROS_POSIX_STKSZ.

Writing ISRs for the Posix port

The posix port uses unix/posix signals to fake interrupts. Interrupt handlers are thus signal handlers. The posix port does not use kernel traps (for compatibility). So the form of the ISR must be as follows.

```
extern int halint_kernel_trap_flagged;  // Flag in hal
extern uint8_t os_isr_nesting;
void signal_handler(int sig)
{
  OS_ISR_BEGIN();
  halint_kernel_trap_flagged = 0;
  // Perform actions
  xxx();

    // Re-enable interrupts if desired
  if (os_isr_nesting  < TIROS_MAX_ISR_NESTING) {
     OS_INT_ENABLE();
     yyy();
  }
  zzz();
  // End actions
  OS_ISR_END();
  if (halint_kernel_trap_flagged) {
     hal_ctxt_switch();
  }
}
```

Chapter 7

License and Usage

7.1 License

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Lesser General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU GENERAL PUBLIC LICENSE
TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

```
  0. This License applies to any program or other work which contains
a notice placed by the copyright holder saying it may be distributed
under the terms of this General Public License.  The "Program", below,
refers to any such program or work, and a "work based on the Program"
means either the Program or any derivative work under copyright law:
that is to say, a work containing the Program or a portion of it,
either verbatim or with modifications and/or translated into another
language.  (Hereinafter, translation is included without limitation in
the term "modification".)  Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not
covered by this License; they are outside its scope.  The act of
running the Program is not restricted, and the output from the Program
is covered only if its contents constitute a work based on the
Program (independent of having been made by running the Program).
Whether that is true depends on what the Program does.

  1. You may copy and distribute verbatim copies of the Program's
source code as you receive it, in any medium, provided that you
conspicuously and appropriately publish on each copy an appropriate
copyright notice and disclaimer of warranty; keep intact all the
notices that refer to this License and to the absence of any warranty;
and give any other recipients of the Program a copy of this License
along with the Program.

You may charge a fee for the physical act of transferring a copy, and
you may at your option offer warranty protection in exchange for a fee.

  2. You may modify your copy or copies of the Program or any portion
of it, thus forming a work based on the Program, and copy and
distribute such modifications or work under the terms of Section 1
above, provided that you also meet all of these conditions:

    a) You must cause the modified files to carry prominent notices
    stating that you changed the files and the date of any change.

    b) You must cause any work that you distribute or publish, that in
    whole or in part contains or is derived from the Program or any
    part thereof, to be licensed as a whole at no charge to all third
    parties under the terms of this License.

    c) If the modified program normally reads commands interactively
    when run, you must cause it, when started running for such
    interactive use in the most ordinary way, to print or display an
    announcement including an appropriate copyright notice and a
    notice that there is no warranty (or else, saying that you provide
    a warranty) and that users may redistribute the program under
    these conditions, and telling the user how to view a copy of this
    License.  (Exception: if the Program itself is interactive but
    does not normally print such an announcement, your work based on
    the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole.  If
identifiable sections of that work are not derived from the Program,
and can be reasonably considered independent and separate works in
themselves, then this License, and its terms, do not apply to those
sections when you distribute them as separate works.  But when you
```

```
distribute the same sections as part of a whole which is a work based
on the Program, the distribution of the whole must be on the terms of
this License, whose permissions for other licensees extend to the
entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest
your rights to work written entirely by you; rather, the intent is to
exercise the right to control the distribution of derivative or
collective works based on the Program.

In addition, mere aggregation of another work not based on the Program
with the Program (or with a work based on the Program) on a volume of
a storage or distribution medium does not bring the other work under
the scope of this License.

  3. You may copy and distribute the Program (or a work based on it,
under Section 2) in object code or executable form under the terms of
Sections 1 and 2 above provided that you also do one of the following:

    a) Accompany it with the complete corresponding machine-readable
    source code, which must be distributed under the terms of Sections
    1 and 2 above on a medium customarily used for software interchange; or,

    b) Accompany it with a written offer, valid for at least three
    years, to give any third party, for a charge no more than your
    cost of physically performing source distribution, a complete
    machine-readable copy of the corresponding source code, to be
    distributed under the terms of Sections 1 and 2 above on a medium
    customarily used for software interchange; or,

    c) Accompany it with the information you received as to the offer
    to distribute corresponding source code.  (This alternative is
    allowed only for noncommercial distribution and only if you
    received the program in object code or executable form with such
    an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for
making modifications to it.  For an executable work, complete source
code means all the source code for all modules it contains, plus any
associated interface definition files, plus the scripts used to
control compilation and installation of the executable.  However, as a
special exception, the source code distributed need not include
anything that is normally distributed (in either source or binary
form) with the major components (compiler, kernel, and so on) of the
operating system on which the executable runs, unless that component
itself accompanies the executable.

If distribution of executable or object code is made by offering
access to copy from a designated place, then offering equivalent
access to copy the source code from the same place counts as
distribution of the source code, even though third parties are not
compelled to copy the source along with the object code.

  4. You may not copy, modify, sublicense, or distribute the Program
except as expressly provided under this License.  Any attempt
otherwise to copy, modify, sublicense or distribute the Program is
void, and will automatically terminate your rights under this License.
However, parties who have received copies, or rights, from you under
```

this License will not have their licenses terminated so long as such
parties remain in full compliance.

  5. You are not required to accept this License, since you have not
signed it. However, nothing else grants you permission to modify or
distribute the Program or its derivative works. These actions are
prohibited by law if you do not accept this License. Therefore, by
modifying or distributing the Program (or any work based on the
Program), you indicate your acceptance of this License to do so, and
all its terms and conditions for copying, distributing or modifying
the Program or works based on it.

  6. Each time you redistribute the Program (or any work based on the
Program), the recipient automatically receives a license from the
original licensor to copy, distribute or modify the Program subject to
these terms and conditions. You may not impose any further
restrictions on the recipients' exercise of the rights granted herein.
You are not responsible for enforcing compliance by third parties to
this License.

  7. If, as a consequence of a court judgment or allegation of patent
infringement or for any other reason (not limited to patent issues),
conditions are imposed on you (whether by court order, agreement or
otherwise) that contradict the conditions of this License, they do not
excuse you from the conditions of this License. If you cannot
distribute so as to satisfy simultaneously your obligations under this
License and any other pertinent obligations, then as a consequence you
may not distribute the Program at all. For example, if a patent
license would not permit royalty-free redistribution of the Program by
all those who receive copies directly or indirectly through you, then
the only way you could satisfy both it and this License would be to
refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under
any particular circumstance, the balance of the section is intended to
apply and the section as a whole is intended to apply in other
circumstances.

It is not the purpose of this section to induce you to infringe any
patents or other property right claims or to contest validity of any
such claims; this section has the sole purpose of protecting the
integrity of the free software distribution system, which is
implemented by public license practices. Many people have made
generous contributions to the wide range of software distributed
through that system in reliance on consistent application of that
system; it is up to the author/donor to decide if he or she is willing
to distribute software through any other system and a licensee cannot
impose that choice.

This section is intended to make thoroughly clear what is believed to
be a consequence of the rest of this License.

  8. If the distribution and/or use of the Program is restricted in
certain countries either by patents or by copyrighted interfaces, the
original copyright holder who places the Program under this License
may add an explicit geographical distribution limitation excluding
those countries, so that distribution is permitted only in or among
countries not thus excluded. In such case, this License incorporates

the limitation as if written in the body of this License.

  9. The Free Software Foundation may publish revised and/or new versions
of the General Public License from time to time. Such new versions will
be similar in spirit to the present version, but may differ in detail to
address new problems or concerns.

Each version is given a distinguishing version number. If the Program
specifies a version number of this License which applies to it and "any
later version", you have the option of following the terms and conditions
either of that version or of any later version published by the Free
Software Foundation. If the Program does not specify a version number of
this License, you may choose any version ever published by the Free Software
Foundation.

  10. If you wish to incorporate parts of the Program into other free
programs whose distribution conditions are different, write to the author
to ask for permission. For software which is copyrighted by the Free
Software Foundation, write to the Free Software Foundation; we sometimes
make exceptions for this. Our decision will be guided by the two goals
of preserving the free status of all derivatives of our free software and
of promoting the sharing and reuse of software generally.

    NO WARRANTY

  11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY
FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN
OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES
PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED
OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS
TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE
PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING,
REPAIR OR CORRECTION.

  12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING
WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR
REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES,
INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING
OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED
TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY
YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER
PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE
POSSIBILITY OF SUCH DAMAGES.

     END OF TERMS AND CONDITIONS

Index

Advanced Features and Debugging, 47

Context Switching Functions, 79
Counting semaphores, 36
crit_secs
    OS_CRITICAL_BEGIN, 17
    OS_CRITICAL_ENABLE, 17
    OS_CRITICAL_END, 18
    OS_INT_DISABLE, 18
    OS_INT_ENABLE, 18
Critical Sections, 17
Critical Sections and Interrupts, 77
csem_count
    os_csem, 37
csem_init
    os_csem, 37
csem_P
    os_csem, 37
csem_V
    os_csem, 38

Define basic types and OS configuration for your port, 74

eflag_get
    os_eflag, 45
eflag_init
    os_eflag, 44
eflag_set
    os_eflag, 45
eflag_wait
    os_eflag, 46
EMINOR_ISR
    ret_codes, 52
EMINOR_PRIO_RULE
    ret_codes, 52
ERR_FAILED
    ret_codes, 54
ERR_FULL
    ret_codes, 53
ERR_LOCK
    ret_codes, 52
ERR_LOCK_ISR
    ret_codes, 52
ERR_LOCK_PRIO_CEIL
    ret_codes, 52
ERR_MAJOR
    ret_codes, 51
ERR_MINOR
    ret_codes, 51
ERR_NOSUCHTASK
    ret_codes, 53
ERR_NOTOWNER
    ret_codes, 53
ERR_PRIO_IN_USE
    ret_codes, 54
ERR_RESUMED
    ret_codes, 53
ERR_TASKBLOCKED
    ret_codes, 53
ERR_TIMEOUT
    ret_codes, 53
ERR_WOULDBLOCK
    ret_codes, 52
ERR_WOULDBLOCK_ISR
    ret_codes, 53
ERR_WOULDBLOCK_MUTEX
    ret_codes, 53
Event Flags, 44

func_options
    O_EFFECTIVE_PRIO, 15
    O_EFLAG_AND, 16
    O_EFLAG_CLEAR, 15
    O_EFLAG_OR, 16

O_EFLAG_TRIGGER, 15
O_NONBLOCKING, 15
O_RELATIVE_TIME, 15
Function Options, 15

hal_alarm_set
port_time_funcs, 83
hal_ctxt_load
port_ctxt_funcs, 81
hal_ctxt_switch
port_ctxt_funcs, 80
hal_init
port_ctxt_funcs, 81
hal_retval_set
port_ctxt_funcs, 80
hal_stk_init
port_ctxt_funcs, 79
hal_time_get
port_time_funcs, 82
hal_time_set
port_time_funcs, 82

ILLEGAL_ADDR
port_config, 75
ILLEGAL_ELEM
ret_codes, 51
ILLEGAL_STACK
port_config, 75
isr
OS_ISR_BEGIN, 21
OS_ISR_END, 21

LT_INLINE
port_config, 74

Message Queues, 40
msgQ_count
os_msgq, 42
msgQ_init
os_msgq, 41
msgQ_MEMSZ
os_msgq, 41
msgQ_recv
os_msgq, 43
msgQ_send
os_msgq, 42
mutex_init
os_mutex, 33
mutex_lock
os_mutex, 34
mutex_owner
os_mutex, 34
mutex_unlock
os_mutex, 35
Mutexes, 33

O_EFFECTIVE_PRIO
func_options, 15
O_EFLAG_AND
func_options, 16
O_EFLAG_CLEAR
func_options, 15
O_EFLAG_OR
func_options, 16
O_EFLAG_TRIGGER
func_options, 15
O_NONBLOCKING
func_options, 15
O_RELATIVE_TIME
func_options, 15
OS initialization and running, 22
os_advanced
osint_snapshot, 48
TIROS_DEBUG_DATA_SZ, 48
OS_CRITICAL_BEGIN
crit_secs, 17
OS_CRITICAL_ENABLE
crit_secs, 17
OS_CRITICAL_END
crit_secs, 18
os_csem
csem_count, 37
csem_init, 37
csem_P, 37
csem_V, 38
os_eflag
eflag_get, 45
eflag_init, 44
eflag_set, 45
eflag_wait, 46
os_init
os_initialization, 22
os_initialization

os_init, 22
os_start, 22
OS_INT_DISABLE
crit_secs, 18
OS_INT_ENABLE
crit_secs, 18
OS_ISR_BEGIN
isr, 21
OS_ISR_END
isr, 21
OS_KERNEL_TRAP
port_kernel_trap, 84
os_msgq
msgQ_count, 42
msgQ_init, 41
msgQ_MEMSZ, 41
msgQ_recv, 43
msgQ_send, 42
os_mutex
mutex_init, 33
mutex_lock, 34
mutex_owner, 34
mutex_unlock, 35
OS_PORT_CRITICAL_BEGIN
port_crit_secs, 77
OS_PORT_CRITICAL_ENABLE
port_crit_secs, 77
OS_PORT_CRITICAL_END
port_crit_secs, 77
OS_PORT_INT_DISABLE
port_crit_secs, 78
OS_PORT_INT_ENABLE
port_crit_secs, 78
OS_PORT_ISR_BEGIN
port_crit_secs, 77
os_prio_get
os_task, 25
os_prio_set
os_task, 25
os_self_tid
os_task, 25
os_start
os_initialization, 22
os_task
os_prio_get, 25
os_prio_set, 25
os_self_tid, 25
os_task_create, 24
os_task_del, 26
os_task_resume, 27
os_task_suspend, 26
taskfunc_t, 23
TIROS_MIN_CTXT_SZ, 23
os_task_create
os_task, 24
os_task_del
os_task, 26
os_task_resume
os_task, 27
os_task_suspend
os_task, 26
os_time
os_time_get, 28
os_time_set, 28
os_wake_at, 29
os_time_get
os_time, 28
os_time_set
os_time, 28
os_time_utils
secs_to_trtime, 31
time_add, 31
time_compare, 31
time_lessthan, 32
time_sub, 31
trtime_t, 30
trtime_to_secs, 30
os_wake_at
os_time, 29
osfnptr_t
port_config, 76
osint_snapshot
os_advanced, 48
osptr_t
port_config, 75
osptrword_t
port_config, 76
osstkptr_t
port_config, 76
osword_t
port_config, 75

Port specific time functions, 82
port_config
    ILLEGAL_ADDR, 75
    ILLEGAL_STACK, 75
    LT_INLINE, 74
    osfnptr_t, 76
    osptr_t, 75
    osptrword_t, 76
    osstkptr_t, 76
    osword_t, 75
    subtime_t, 74
    TIROS_REGISTER_PASSING, 74
    TRPORT_INF_TIME, 74
    uint8_t, 75
port_crit_secs
    OS_PORT_CRITICAL_BEGIN, 77
    OS_PORT_CRITICAL_ENABLE, 77
    OS_PORT_CRITICAL_END, 77
    OS_PORT_INT_DISABLE, 78
    OS_PORT_INT_ENABLE, 78
    OS_PORT_ISR_BEGIN, 77
port_ctxt_funcs
    hal_ctxt_load, 81
    hal_ctxt_switch, 80
    hal_init, 81
    hal_retval_set, 80
    hal_stk_init, 79
    TRPORT_MIN_CTXT_SZ, 79
port_kernel_trap
    OS_KERNEL_TRAP, 84
    TIROS_KERNEL_TRAP_ENABLED, 84
port_time_funcs
    hal_alarm_set, 83
    hal_time_get, 82
    hal_time_set, 82
    TRPORT_RESPONSE_TIME, 82

ret_codes
    EMINOR_ISR, 52
    EMINOR_PRIO_RULE, 52
    ERR_FAILED, 54
    ERR_FULL, 53
    ERR_LOCK, 52
    ERR_LOCK_ISR, 52
    ERR_LOCK_PRIO_CEIL, 52
    ERR_MAJOR, 51
    ERR_MINOR, 51
    ERR_NOSUCHTASK, 53
    ERR_NOTOWNER, 53
    ERR_PRIO_IN_USE, 54
    ERR_RESUMED, 53
    ERR_TASKBLOCKED, 53
    ERR_TIMEOUT, 53
    ERR_WOULDBLOCK, 52
    ERR_WOULDBLOCK_ISR, 53
    ERR_WOULDBLOCK_MUTEX, 53
    ILLEGAL_ELEM, 51
    SUCCESS, 52
Return Codes., 51

secs_to_trtime
    os_time_utils, 31
Setup for the kernel trap, 84
subtime_t
    port_config, 74
SUCCESS
    ret_codes, 52

Task creation and management, 23
taskfunc_t
    os_task, 23
Time services provided by the OS, 28
time_add
    os_time_utils, 31
time_compare
    os_time_utils, 31
time_lessthan
    os_time_utils, 32
time_sub
    os_time_utils, 31
TIROS_DEBUG_DATA_SZ
    os_advanced, 48
TIROS_KERNEL_TRAP_ENABLED
    port_kernel_trap, 84
TIROS_MIN_CTXT_SZ
    os_task, 23
TIROS_REGISTER_PASSING
    port_config, 74
TRPORT_INF_TIME
    port_config, 74
TRPORT_MIN_CTXT_SZ
    port_ctxt_funcs, 79
TRPORT_RESPONSE_TIME

port_time_funcs, 82
trtime_t
os_time_utils, 30
trtime_to_secs
os_time_utils, 30

uint8_t
port_config, 75
Usage with Interrupt Service Routines, 19
Utility functions for time operations, 30