4

Task Manager

4.1 Introduction

The **task manager** provides a comprehensive set of directives to create, delete, and administer tasks. The directives provided by the task manager are:

Name	Directive Description
task_create	Create a task
task_ident	Get ID of a task
task_start	Start a task
task_restart	Restart a task
task_delete	Delete a task
task_suspend	Suspend a task
task_resume	Resume a task
task_set_priority	Set task priority
task_mode	Change current task's mode
task_get_note	Get task notepad entry
task_set_note	Set task notepad entry
task_wake_after	Wake up after interval
task_wake_when	Wake up when specified

4.2 Background

4.2.1 Task Definition

Many definitions of a task have been proposed in computer literature. Unfortunately, none of these definitions encompasses all facets of the concept in a manner which is operating system independent. Several of the more common definitions are provided to enable each user to select a definition which best matches their own experience and understanding of the task concept:

- *a "dispatchable" unit.*
- *an entity to which the processor is allocated.*
- *an atomic unit of a real-time, multiprocessor system.*
- *single threads of execution which concurrently compete for resources.*
- *a sequence of closely related computations which can execute concurrently with other computational sequences.*

From **RTEMS**' perspective, a task is the smallest thread of execution which can compete on its own for system resources. A task is manifested by the existence of a task control block (TCB).

4.2.2 Task Control Block

The **Task Control Block** (TCB) is an **RTEMS** defined data structure which contains all the information that is pertinent to the execution of a task. During system initialization, **RTEMS** reserves a TCB for each task configured. A TCB is allocated upon creation of the task and is returned to the TCB free list upon deletion of the task.

The TCB's elements are modified as a result of system calls made by the application in response to external and internal stimuli. TCBs are the only **RTEMS** internal data structure that can be accessed by an application via user extension routines. The TCB contains a task's name, ID, current priority, current and starting states, execution mode, set of notepad locations, TCB user extension pointer, scheduling control structures, as well as data required by a blocked task.

A task's context is stored in the TCB when a task switch occurs. When the task regains control of the processor, its context is restored from the TCB. When a task is restarted, the initial state of the task is restored from the starting context area in the task's TCB.

4.2.3 Task States

A task may exist in one of the following five states:

Task State	Description
executing	Currently scheduled to the CPU
ready	May be scheduled to the CPU
blocked	Unable to be scheduled to the CPU
dormant	Created task that is not started
non-existent	Uncreated or deleted task

Figure 4-1 Task States

An active task may occupy the executing, ready, blocked or dormant state, otherwise the task is considered non-existent. One or more tasks may be active in the system simultaneously. Multiple tasks communicate, synchronize, and compete for system resources with each other via system calls. The multiple tasks appear to execute in parallel, but actually each is dispatched to the CPU for periods of time determined by the **RTEMS** scheduling algorithm. The scheduling of a task is based on its current state and priority.

4.2.4 Task Priority

A task's priority determines its importance in relation to the other tasks executing on the same processor. **RTEMS** supports 255 levels of priority ranging from 1 to 255. Tasks of numerically smaller priority values are more important tasks than tasks of numerically larger priority values. For example, a task at priority level 5 is of higher privilege than a task at priority level 10. There is no limit to the number of tasks assigned to the same priority.

Each task has a priority associated with it at all times. The initial value of this priority is assigned at task creation time. The priority of a task may be changed at any subsequent time.

Priorities are used by the scheduler to determine which ready task will be allowed to execute. In general, the higher the logical priority of a task, the more likely it is to receive processor execution time.

4.2.5 Task Mode

A task's mode is a combination of the following four components:

- *preemption*
- *timeslicing*
- *ASR processing*
- *interrupt level*

It is used to modify **RTEMS**' scheduling process and to alter the execution environment of the task.

The preemption component allows a task to determine when control of the processor is relinquished. If preemption is disabled (**NO_PREEMPT**), the task will retain control of the processor as long as it is in the executing state -- even if a higher priority task is made ready. If preemption is enabled (**PREEMPT**) and a higher priority task is made ready, then the processor will be taken away from the current task immediately and given to the higher priority task.

The timeslicing component is used by the **RTEMS** scheduler to determine how the processor is allocated to tasks of equal priority. If timeslicing is enabled (**TIMESLICE**), then **RTEMS** will limit the amount of time the task can execute before the processor is allocated to another ready task of equal priority. The length of the timeslice is application dependent and specified in the **Configuration Table**. If timeslicing is disabled (**NO_TIMESLICE**), then the task will be allowed to execute until a task of higher priority is made ready. If **NO_PREEMPT** is selected, then the timeslicing component is ignored by the scheduler.

The asynchronous signal processing component is used to determine when received signals are to be processed by the task. If signal processing is enabled (**ASR**), then signals sent to the task will be processed the next time the task executes. If signal processing is disabled (**NO_ASR**), then all signals received by the task will remain posted until signal processing is enabled. This component affects only tasks which have established a routine to process asynchronous signals.

The interrupt level component is used to determine which interrupts will be enabled when the task is executing. **INTERRUPT_LEVEL**(**n**) specifies that the task will execute at interrupt level **n**.

CONSTANT	DESCRIPTION	DEFAULT
PREEMPT	enable preemption	*
NO_PREEMPT	disable preemption	
NO_TIMESLICE	disable timeslicing	*
TIMESLICE	enable timeslicing	
ASR	enable ASR processing	*
NO_ASR	disable ASR processing	
INTERRUPT_LEVEL(0)	enable all interrupts	*
INTERRUPT_LEVEL(n)	execute at interrupt level n	

Figure 4-2 Task Mode Constants

4.2.6 Accessing Task Arguments

All **RTEMS** tasks are invoked with a single argument which is specified when they are started or restarted. The argument is commonly used to communicate startup information to the task. The simplest manner in which to define a task which accesses it argument is:

```
rtems_task user_task(
  rtems_task_argument argument
);
```

Application tasks requiring more information may view this single argument as an index into an array of parameter blocks.

4.2.7 Floating Point Considerations

Creating a task with the **FLOATING_POINT** flag results in additional memory being allocated for the TCB to store the state of the numeric coprocessor during task switches. This additional memory is **NOT** allocated for **NO_FLOATING_POINT** tasks. Saving and restoring the context of a **FLOATING_POINT** task takes longer than that of a **NO_FLOATING_POINT** task because of the relatively large amount of time required for the numeric coprocessor to save or restore its computational state.

Since **RTEMS** was designed specifically for embedded military applications which are floating point intensive, the executive is optimized to avoid unnecessarily saving and restoring the state of the numeric coprocessor. The state of the numeric coprocessor is only saved when a **FLOATING_POINT** task is dispatched and that task was not the last task to utilize the coprocessor. In a

system with only one **FLOATING_POINT** task, the state of the numeric coprocessor will never be saved or restored.

Although the overhead imposed by **FLOATING_POINT** tasks is minimal, some applications may wish to completely avoid the overhead associated with **FLOATING_POINT** tasks and still utilize a numeric coprocessor. By preventing a task from being preempted while performing a sequence of floating point operations, a **NO_FLOATING_POINT** task can utilize the numeric coprocessor without incurring the overhead of a **FLOATING_POINT** context switch. This approach also avoids the allocation of a floating point context area. However, if this approach is taken by the application designer, **NO** tasks should be created as **FLOATING_POINT** tasks. Otherwise, the floating point context will not be correctly maintained because **RTEMS** assumes that the state of the numeric coprocessor will not be altered by **NO_FLOATING_POINT** tasks.

If the supported processor type does not have hardware floating capabilities or a standard numeric coprocessor, **RTEMS** will not provide built-in support for hardware floating point on that processor. In this case, all tasks are considered **NO_FLOATING_POINT** whether created as **FLOATING_POINT** or **NO_FLOATING_POINT** tasks. A floating point emulation software library must be utilized for floating point operations.

4.2.8 Building a Task's Attribute Set

In general, an attribute set is built by a bitwise OR of the desired components. The set of valid task attribute components is provided in the following table:

CONSTANT	DESCRIPTION	DEFAULT
NO_FLOATING_POINT	does not use coprocessor	*
FLOATING_POINT	uses numeric coprocessor	
LOCAL	local task	*
GLOBAL	global task	

Attribute values are specifically designed to be mutually exclusive, therefore bitwise OR and addition operations are equivalent as long as each attribute appears exactly once in the component list. A component listed as a default is not required to appear in the component list, although it is a good programming practice to specify default components. If all defaults are desired, then **DEFAULT_ATTRIBUTES** should be used.

This example demonstrates the **attribute_set** parameter needed to create a local task which utilizes the numeric coprocessor. The **attribute_set** parameter could be **FLOATING_POINT** or **LOCAL | FLOATING_POINT**. The **attribute_set**

parameter can be set to **FLOATING_POINT** because **LOCAL** is the default for all created tasks. If the task were global and used the numeric coprocessor, then the **attribute_set** parameter would be **GLOBAL | FLOATING_POINT**.

4.2.9 Building a Mode and Mask

In general, a mode and its corresponding mask is built by a bitwise OR of the desired components. The set of valid mode constants and each mode's corresponding mask constant is provided in the following table:

MODE CONSTANT	MASK CONSTANT	DESCRIPTION
PREEMPT	PREEMPT_MASK	enable preemption
NO_PREEMPT	PREEMPT_MASK	disable preemption
NO_TIMESLICE	TIMESLICE_MASK	disable timeslicing
TIMESLICE	TIMESLICE_MASK	enable timeslicing
ASR	ASR_MASK	enable ASR processing
NO_ASR	ASR_MASK	disable ASR processing
INTERRUPT_LEVEL(0)	INTERRUPT_MASK	enable all interrupts
INTERRUPT_LEVEL(n)	INTERRUPT_MASK	execute at interrupt level n

Mode values are specifically designed to be mutually exclusive, therefore bitwise OR and addition operations are equivalent as long as each mode appears exactly once in the component list. A mode component listed as a default is not required to appear in the mode component list, although it is a good programming practice to specify default components. If all defaults are desired, the mode **DEFAULT_MODES** and the mask **ALL_MODE_MASKS** should be used.

The following example demonstrates the **mode** and **mask** parameters used with the **task_mode** directive to place a task at interrupt level 3 and make it non-preemptible. The **mode** should be set to **INTERRUPT_LEVEL(3) | NO_PREEMPT** to indicate the desired preemption mode and interrupt level, while the **mask** parameter should be set to **INTERRUPT_MASK | NO_PREEMPT_MASK** to indicate that the calling task's interrupt level and preemption mode are being altered.

4.3 Operations

4.3.1 Creating Tasks

The **task_create** directive creates a task by allocating a task control block, assigning the task a user-specified name, allocating it a stack and floating point

context area, setting a user-specified initial priority, setting a user-specified initial mode, and assigning it a task ID. Newly created tasks are initially placed in the dormant state. All **RTEMS** tasks execute in the most privileged mode of the processor.

4.3.2 Obtaining Task IDs

When a task is created, **RTEMS** generates a unique task ID and assigns it to the created task until it is deleted. The task ID may be obtained by either of two methods. First, as the result of an invocation of the **task_create** directive, the task ID is stored in a user provided location. Second, the task ID may be obtained later using the **task_ident** directive. The task ID is used by other directives to manipulate this task.

4.3.3 Starting and Restarting Tasks

The **task_start** directive is used to place a dormant task in the **ready** state. This enables the task to compete, based on its current priority, for the processor and other system resources. Any actions, such as suspension or change of priority, performed on a task prior to starting it are nullified when the task is started.

With the **task_start** directive the user specifies the task's starting address and argument. The argument is used to communicate some startup information to the task. As part of this directive, **RTEMS** initializes the task's stack based upon the task's initial execution mode and start address. The starting argument is passed to the task in accordance with the target processor's calling convention.

The **task_restart** directive restarts a task at its initial starting address with its original priority and execution mode, but with a possibly different argument. The new argument may be used to distinguish between the original invocation of the task and subsequent invocations. The task's stack and control block are modified to reflect their original creation values. Although references to resources that have been requested are cleared, resources allocated by the task are **NOT** automatically returned to **RTEMS**. A task cannot be restarted unless it has previously been started (i.e. dormant tasks cannot be restarted). All restarted tasks are placed in the **ready** state.

4.3.4 Suspending and Resuming Tasks

The **task_suspend** directive is used to place either the caller or another task into a suspended state. The task remains suspended until a **task_resume** directive is issued. This implies that a task may be suspended as well as blocked waiting either to acquire a resource or for the expiration of a timer.

The **task_resume** directive is used to remove another task from the suspended state. If the task is not also blocked, resuming it will place it in the ready state, allowing it to once again compete for the processor and resources. If the task was blocked as well as suspended, this directive clears the suspension and leaves the task in the blocked state.

4.3.5 Delaying the Currently Executing Task

The **task_wake_after** directive creates a sleep timer which allows a task to go to sleep for a specified interval. The task is blocked until the delay interval has elapsed, at which time the task is unblocked. A task calling the **task_wake_after** directive with a delay interval of **YIELD_PROCESSOR** ticks will yield the processor to any other ready task of equal or greater priority and remain ready to execute.

The **task_wake_when** directive creates a sleep timer which allows a task to go to sleep until a specified date and time. The calling task is blocked until the specified date and time has occurred, at which time the task is unblocked.

4.3.6 Changing Task Priority

The **task_set_priority** directive is used to obtain or change the current priority of either the calling task or another task. If the new priority requested is **CURRENT_PRIORITY** or the task's actual priority, then the current priority will be returned and the task's priority will remain unchanged. If the task's priority is altered, then the task will be scheduled according to its new priority.

The **task_restart** directive resets the priority of a task to its original value.

4.3.7 Changing Task Mode

The **task_mode** directive is used to obtain or change the current execution mode of the calling task. A task's execution mode is used to enable preemption, timeslicing, ASR processing, and to set the task's interrupt level.

The **task_restart** directive resets the mode of a task to its original value.

4.3.8 Notepad Locations

RTEMS provides sixteen notepad locations for each task. Each notepad location may contain a **note** consisting of four bytes of information. **RTEMS** provides two directives, **task_set_note** and **task_get_note**, that enable a user to access and change the notepad locations. The **task_set_note** directive enables the user to set a task's notepad entry to a specified note. The **task_get_note** directive allows the

user to obtain the note contained in any one of the sixteen notepads of a specified task.

4.3.9 Task Deletion

RTEMS provides the **task_delete** directive to allow a task to delete itself or any other task. This directive removes all **RTEMS** references to the task, frees the task's control block, removes it from resource wait queues, and deallocates its stack as well as the optional floating point context. The task's name and ID become inactive at this time, and any subsequent references to either of them is invalid. In fact, **RTEMS** may reuse the task ID for another task which is created later in the application.

Unexpired delay timers (i.e. those used by **task_wake_after** and **task_wake_when**) and timeout timers associated with the task are automatically deleted, however, other resources dynamically allocated by the task are **NOT** automatically returned to **RTEMS**. Therefore, before a task is deleted, all of its dynamically allocated resources should be deallocated by the user. This may be accomplished by instructing the task to delete itself rather than directly deleting the task. Other tasks may instruct a task to delete itself by sending a "delete self" message, event, or signal, or by restarting the task with special arguments which instruct the task to delete itself.

4.4 Directives

This section details the task manager's directives. A subsection is dedicated to each of this manager's directives and describes the calling sequence, related constants, usage, and status codes.

4.4.1 TASK_CREATE - Create a task

CALLING SEQUENCE:

```
rtems_status_code rtems_task_create(
  rtems_name           name,
  rtems_task_priority  initial_priority,
  rtems_unsigned32     stack_size,
  rtems_mode           initial_modes,
  rtems_attribute      attribute_set,
  rtems_id            *id
);
```

DIRECTIVE STATUS CODES:

SUCCESSFUL	task created successfully
INVALID_NAME	invalid task name
INVALID_SIZE	stack too small
INVALID_PRIORITY	invalid task priority
MP_NOT_CONFIGURED	multiprocessing not configured
TOO_MANY	too many tasks created
UNSATISFIED	not enough memory for stack/FP context
TOO_MANY	too many global objects

DESCRIPTION:

This directive creates a task which resides on the local node. It allocates and initializes a TCB, a stack, and an optional floating point context area. The **mode** parameter contains values which sets the task's initial execution mode. The **FLOATING_POINT** attribute should be specified if the created task is to use a numeric coprocessor. For performance reasons, it is recommended that tasks not using the numeric coprocessor should specify the **NO_FLOATING_POINT** attribute. If the **GLOBAL** attribute is specified, the task can be accessed from remote nodes. The task id, returned in **id**, is used in other task related directives to access the task. When created, a task is placed in the dormant state and can only be made ready to execute using the directive **task_start**.

NOTES:

This directive will not cause the calling task to be preempted.

Valid task priorities range from a high of 1 to a low of 255.

RTEMS supports a maximum of 256 interrupt levels which are mapped onto the interrupt levels actually supported by the target processor.

The requested stack size should be at least **MINIMUM_STACK_SIZE** bytes. The value of **MINIMUM_STACK_SIZE** is processor dependent. Application developers should consider the stack usage of the device drivers when calculating the stack size required for tasks which utilize the driver.

The following task attribute constants are defined by **RTEMS**:

CONSTANT	DESCRIPTION	DEFAULT
NO_FLOATING_POINT	does not use coprocessor	*
FLOATING_POINT	uses numeric coprocessor	
LOCAL	local task	*
GLOBAL	global task	

The following task mode constants are defined by **RTEMS**:

MODE CONSTANT	MASK CONSTANT	DESCRIPTION
PREEMPT	PREEMPT_MASK	enable preemption
NO_PREEMPT	PREEMPT_MASK	disable preemption
NO_TIMESLICE	TIMESLICE_MASK	disable timeslicing
TIMESLICE	TIMESLICE_MASK	enable timeslicing
ASR	ASR_MASK	enable ASR processing
NO_ASR	ASR_MASK	disable ASR processing
INTERRUPT_LEVEL(0)	INTERRUPT_MASK	enable all interrupts
INTERRUPT_LEVEL(n)	INTERRUPT_MASK	execute at interrupt level n

Tasks should not be made global unless remote tasks must interact with them. This avoids the system overhead incurred by the creation of a global task. When a global task is created, the task's name and id must be transmitted to every node in the system for insertion in the local copy of the global object table.

The total number of global objects, including tasks, is limited by the **maximum_global_objects** field in the **Configuration Table**.

4.4.2 TASK_IDENT - Get ID of a task

CALLING SEQUENCE:

```
rtems_status_code rtems_task_ident(
  rtems_name        name,
  rtems_unsigned32  node,
  rtems_id         *id
);
```

DIRECTIVE STATUS CODES:

SUCCESSFUL	task identified successfully
INVALID_NAME	invalid task name
INVALID_NODE	invalid node id

DESCRIPTION:

This directive obtains the task id associated with the task name specified in **name**. A task may obtain its own id by specifying **SELF** or its own task name in **name**. If the task name is not unique, then the task id returned will match one of the tasks with that name. However, this task id is not guaranteed to correspond to the desired task. The task id, returned in **id**, is used in other task related directives to access the task.

NOTES:

This directive will not cause the running task to be preempted.

If **node** is **SEARCH_ALL_NODES**, all nodes are searched with the local node being searched first. All other nodes are searched with the lowest numbered node searched first.

If **node** is a valid node number which does not represent the local node, then only the tasks exported by the designated node are searched.

This directive does not generate activity on remote nodes. It accesses only the local copy of the global object table.

4.4.3 TASK_START - Start a task

CALLING SEQUENCE:

```
rtems_status_code rtems_task_start(
  rtems_id            id,
  rtems_task_entry    entry_point,
  rtems_task_argument argument
);
```

DIRECTIVE STATUS CODES:

SUCCESSFUL	task started successfully
INVALID_ADDRESS	invalid task entry point
INVALID_ID	invalid task id
INCORRECT_STATE	task not in the dormant state
ILLEGAL_ON_REMOTE_OBJECT	cannot start remote task

DESCRIPTION:

This directive readies the task, specified by **tid**, for execution based on the priority and execution mode specified when the task was created. The starting address of the task is given in **entry_point**. The task's starting argument is contained in **argument**. This argument can be a single value or used as an index into an array of parameter blocks.

NOTES:

The calling task will be preempted if its preemption mode is enabled and the task being started has a higher priority.

Any actions performed on a **dormant** task such as suspension or change of priority are nullified when the task is initiated via the **task_start** directive.

4.4.4 TASK_RESTART - Restart a task

CALLING SEQUENCE:

```
rtems_status_code rtems_task_restart(
  rtems_id            id,
  rtems_task_argument argument
);
```

DIRECTIVE STATUS CODES:

SUCCESSFUL	task restarted successfully
INVALID_ID	task id invalid
INCORRECT_STATE	task never started
ILLEGAL_ON_REMOTE_OBJECT	cannot restart remote task

DESCRIPTION:

This directive resets the task specified by **id** to begin execution at its original starting address. The task's priority and execution mode are set to the original creation values. If the task is currently blocked, **RTEMS** automatically makes the task ready. A task can be restarted from any state, except the dormant state.

The task's starting argument is contained in **argument**. This argument can be a single value or an index into an array of parameter blocks. This new argument may be used to distinguish between the initial **task_start** of the task and any ensuing calls to **task_restart** of the task. This can be beneficial in deleting a task. Instead of deleting a task using the **task_delete** directive, a task can delete another task by restarting that task, and allowing that task to release resources back to **RTEMS** and then delete itself.

NOTES:

If **id** is **SELF**, the calling task will be restarted and will not return from this directive.

The calling task will be preempted if its preemption mode is enabled and the task being restarted has a higher priority.

The task must reside on the local node, even if the task was created with the **GLOBAL** option.

4.4.5 TASK_DELETE - Delete a task

CALLING SEQUENCE:

```
rtems_status_code rtems_task_delete(
  rtems_id id
);
```

DIRECTIVE STATUS CODES:

SUCCESSFUL	task restarted successfully
INVALID_ID	task id invalid
ILLEGAL_ON_REMOTE_OBJECT	cannot restart remote task

DESCRIPTION:

This directive deletes a task, either the calling task or another task, as specified by **id**. **RTEMS** stops the execution of the task and reclaims the stack memory, any allocated delay or timeout timers, the TCB, and, if the task is **FLOATING_POINT**, its floating point context area. **RTEMS** does not reclaim the following resources: region segments, partition buffers, semaphores, timers, or rate monotonic periods.

NOTES:

A task is responsible for releasing its resources back to **RTEMS** before deletion. To insure proper deallocation of resources, a task should not be deleted unless it is unable to execute or does not hold any **RTEMS** resources. If a task holds **RTEMS** resources, the task should be allowed to deallocate its resources before deletion. A task can be directed to release its resources and delete itself by restarting it with a special argument or by sending it a message, an event, or a signal.

Deletion of the current task (**SELF**) will force **RTEMS** to select another task to execute.

When a global task is deleted, the task id must be transmitted to every node in the system for deletion from the local copy of the global object table.

The task must reside on the local node, even if the task was created with the **GLOBAL** option.

4.4.6 TASK_SUSPEND - Suspend a task

CALLING SEQUENCE:

```
rtems_status_code rtems_task_suspend(
  rtems_id id
);
```

DIRECTIVE STATUS CODES:

SUCCESSFUL	task restarted successfully
INVALID_ID	task id invalid
ALREADY_SUSPENDED	task already suspended

DESCRIPTION:

This directive suspends the task specified by **id** from further execution by placing it in the suspended state. This state is additive to any other blocked state that the task may already be in. The task will not execute again until another task issues the **task_resume** directive for this task and any blocked state has been removed.

NOTES:

The requesting task can suspend itself by specifying **SELF** as **id**. In this case, the task will be suspended and a successful return code will be returned when the task is resumed.

Suspending a global task which does not reside on the local node will generate a request to the remote node to suspend the specified task.

4.4.7 TASK_RESUME - Resume a task

CALLING SEQUENCE:

```
rtems_status_code rtems_task_resume(
  rtems_id id
);
```

DIRECTIVE STATUS CODES:

SUCCESSFUL	task restarted successfully
INVALID_ID	task id invalid
INCORRECT_STATE	task not suspended

DESCRIPTION:

This directive removes the task specified by **id** from the suspended state. If the task is in the ready state after the suspension is removed, then it will be scheduled to run. If the task is still in a blocked state after the suspension is removed, then it will remain in that blocked state.

NOTES:

The running task may be preempted if its preemption mode is enabled and the local task being resumed has a higher priority.

Resuming a global task which does not reside on the local node will generate a request to the remote node to resume the specified task.

4.4.8 TASK_SET_PRIORITY - Set task priority

CALLING SEQUENCE:

```
rtems_status_code rtems_task_set_priority(
  rtems_id            id,
  rtems_task_priority  new_priority,
  rtems_task_priority *old_priority
);
```

DIRECTIVE STATUS CODES:

SUCCESSFUL	task priority set successfully
INVALID_ID	invalid task id
INVALID_PRIORITY	invalid task priority

DESCRIPTION:

This directive manipulates the priority of the task specified by **id**. An **id** of **SELF** is used to indicate the calling task. When **new_priority** is not equal to **CURRENT_PRIORITY**, the specified task's previous priority is returned in **old_priority**. When **new_priority** is **CURRENT_PRIORITY**, the specified task's current priority is returned in **old_priority**. Valid priorities range from a high of 1 to a low of 255.

NOTES:

The calling task may be preempted if its preemption mode is enabled and it lowers its own priority or raises another task's priority.

Setting the priority of a global task which does not reside on the local node will generate a request to the remote node to change the priority of the specified task.

If the task specified by **id** is currently holding any binary semaphores which use the priority inheritance algorithm, then the task's priority cannot be lowered immediately. If the task's priority were lowered immediately, then priority inversion results. The requested lowering of the task's priority will occur when the task has released all priority inheritance binary semaphores. The task's priority can be increased regardless of the task's use of priority inheritance binary semaphores.

4.4.9 TASK_MODE - Change current task's mode

CALLING SEQUENCE:

```
rtems_status_code rtems_task_mode(
  rtems_mode  mode_set,
  rtems_mode  mask,
  rtems_mode *previous_mode_set
);
```

DIRECTIVE STATUS CODES:

SUCCESSFUL task priority set successfully

DESCRIPTION:

This directive manipulates the execution mode of the calling task. A task's execution mode enables and disables preemption, timeslicing, asynchronous signal processing, as well as specifying the current interrupt level. To modify an execution mode, the mode class(es) to be changed must be specified in the **mask** parameter and the desired mode(s) must be specified in the **mode** parameter.

NOTES:

The calling task will be preempted if it enables preemption and a higher priority task is ready to run.

Enabling timeslicing has no effect if preemption is enabled.

A task can obtain its current execution mode, without modifying it, by calling this directive with a **mask** value of **CURRENT_MODE**.

To temporarily disable the processing of a valid ASR, a task should call this directive with the **NO_ASR** indicator specified in **mode**.

The set of task mode constants and each mode's corresponding mask constant is provided in the following table:

MODE CONSTANT	MASK CONSTANT	DESCRIPTION
PREEMPT	PREEMPT_MASK	enable preemption
NO_PREEMPT	PREEMPT_MASK	disable preemption
NO_TIMESLICE	TIMESLICE_MASK	disable timeslicing
TIMESLICE	TIMESLICE_MASK	enable timeslicing
ASR	ASR_MASK	enable ASR processing
NO_ASR	ASR_MASK	disable ASR processing
INTERRUPT_LEVEL(0)	INTERRUPT_MASK	enable all interrupts
INTERRUPT_LEVEL(n)	INTERRUPT_MASK	execute at interrupt level n

4.4.10 TASK_GET_NOTE - Get task notepad entry

CALLING SEQUENCE:

```
rtems_status_code rtems_task_get_note(
  rtems_id          id,
  rtems_unsigned32  notepad,
  rtems_unsigned32 *note
);
```

DIRECTIVE STATUS CODES:

SUCCESSFUL	note obtained successfully
INVALID_ID	invalid task id
INVALID_NUMBER	invalid notepad location

DESCRIPTION:

This directive returns the **note** contained in the **notepad** location of the task specified by **id**.

NOTES:

This directive will not cause the running task to be preempted.

If **id** is set to **SELF**, the calling task accesses its own notepad.

The sixteen notepad locations can be accessed using the constants **NOTEPAD_0** through **NOTEPAD_15**.

Getting a note of a global task which does not reside on the local node will generate a request to the remote node to obtain the notepad entry of the specified task.

4.4.11 TASK_SET_NOTE - Set task notepad entry

CALLING SEQUENCE:

```
rtems_status_code rtems_task_set_note(
  rtems_id         id,
  rtems_unsigned32 notepad,
  rtems_unsigned32 note
);
```

DIRECTIVE STATUS CODES:

SUCCESSFUL	task's note set successfully
INVALID_ID	invalid task id
INVALID_NUMBER	invalid notepad location

DESCRIPTION:

This directive sets the **notepad** entry for the task specified by **id** to the value **note**.

NOTES:

If **id** is set to **SELF**, the calling task accesses its own notepad locations.

This directive will not cause the running task to be preempted.

The sixteen notepad locations can be accessed using the constants **NOTEPAD_0** through **NOTEPAD_15**.

Setting a notepad location of a global task which does not reside on the local node will generate a request to the remote node to set the specified notepad entry.

4.4.12 TASK_WAKE_AFTER - Wake up after interval

CALLING SEQUENCE:

```
rtems_status_code rtems_task_wake_after(
  rtems_interval ticks
);
```

DIRECTIVE STATUS CODES:

SUCCESSFUL always successful

DESCRIPTION:

This directive blocks the calling task for the specified number of system clock ticks. When the requested interval has elapsed, the task is made ready. The **clock_tick** directive automatically updates the delay period.

NOTES:

Setting the system date and time with the **clock_set** directive has no effect on a task_wake_after blocked task.

A task may give up the processor and remain in the ready state by specifying a value of **YIELD_PROCESSOR** in ticks.

The maximum timer interval that can be specified is the maximum **value** which can be represented by the **rtems_unsigned32** type.

4.4.13 TASK_WAKE_WHEN - Wake up when specified

CALLING SEQUENCE:

```
rtems_status_code rtems_task_wake_when(
  rtems_time_of_day *time_buffer
);
```

DIRECTIVE STATUS CODES:

SUCCESSFUL	awakened at date/time successfully
INVALID_TIME_OF_DAY	invalid time buffer
NOT_DEFINED	system date and time is not set

DESCRIPTION:

This directive blocks a task until the date and time specified in **time_buffer**. At the requested date and time, the calling task will be unblocked and made ready to execute.

NOTES:

The ticks portion of **time_buffer** structure is ignored. The timing granularity of this directive is a second.