17

Scheduling Concepts

17.1 Introduction

The concept of scheduling in real-time systems dictates the ability to provide immediate response to specific external events, particularly the necessity of scheduling tasks to run within a specified time limit after the occurrence of an event. For example, software embedded in life-support systems used to monitor hospital patients must take instant action if a change in the patient's status is detected.

The component of **RTEMS** responsible for providing this capability is appropriately called the scheduler. The scheduler's sole purpose is to allocate the all important resource of processor time to the various tasks competing for attention. The **RTEMS** scheduler allocates the processor using a priority-based, preemptive algorithm augmented to provide round-robin characteristics within individual priority groups. The goal of this algorithm is to guarantee that the task which is executing on the processor at any point in time is the one with the **highest** priority among all tasks in the **ready** state.

There are two common methods of accomplishing the mechanics of this algorithm. Both ways involve a list or chain of tasks in the ready state. One method is to randomly place tasks in the ready chain forcing the scheduler to scan the entire chain to determine which task receives the processor. The other method is to schedule the task by placing it in the proper place on the ready chain based on the designated scheduling criteria at the time it enters the ready state. Thus, when the processor is free, the first task on the ready chain is allocated the processor.

RTEMS schedules tasks using the second method to guarantee faster response times to external events.

17.2 Scheduling Mechanisms

RTEMS provides four mechanisms which allow the user to impact the task scheduling process:

- *user-selectable task priority level*
- *task preemption control*
- *task timeslicing control*
- *manual round-robin selection*

Each of these methods provides a powerful capability to customize sets of tasks to satisfy the unique and particular requirements encountered in custom real-time applications. Although each mechanism operates independently, there is a precedence relationship which governs the effects of scheduling modifications. The evaluation order for scheduling characteristics is always priority, preemption mode, and timeslicing. When reading the descriptions of timeslicing and manual round-robin it is important to keep in mind that preemption (if enabled) of a task by higher priority tasks will occur as required, overriding the other factors presented in the description.

17.2.1 Task Priority

The most significant of these mechanisms is the ability for the user to assign a priority level to each individual task when it is created and to alter a task's priority at run-time. **RTEMS** provides 255 priority levels. Level 255 is the lowest priority and level 1 is the highest. When a task is added to the **ready chain**, it is placed behind all other tasks of the same priority. This rule provides a round-robin within priority group scheduling characteristic. This means that in a group of equal priority tasks, tasks will execute in the order they become ready or FIFO order. Even though there are ways to manipulate and adjust task priorities, the most important rule to remember is:

> *The **RTEMS** scheduler will always select the highest priority task that is ready to run when allocating the processor to a task.*

17.2.2 Preemption

Another way the user can alter the basic scheduling algorithm is by manipulating the preemption mode flag (**PREEMPT_MASK**) of individual tasks. If

preemption is disabled for a task (**NO_PREEMPT**), then the task will not relinquish control of the processor until it terminates, blocks, or re-enables preemption. Even tasks which become ready to run and possess higher priority levels will not be allowed to execute. Note that the preemption setting has no effect on the manner in which a task is scheduled. It only applies once a task has control of the processor.

17.2.3 Timeslicing

Timeslicing or round-robin scheduling is an additional method which can be used to alter the basic scheduling algorithm. Like preemption, timeslicing is specified on a task by task basis using the timeslicing mode flag (**TIMESLICE_MASK**). If timeslicing is enabled for a task (**TIMESLICE**), then **RTEMS** will limit the amount of time the task can execute before the processor is allocated to another task. Each tick of the real-time clock reduces the currently running task's timeslice. When the execution time equals the timeslice, **RTEMS** will dispatch another task of the same priority to execute. If there are no other tasks of the same priority ready to execute, then the current task is allocated an additional timeslice and continues to run. Remember that a higher priority task will preempt the task (unless preemption is disabled) as soon as it is ready to run, even if the task has not used up its entire timeslice.

17.2.4 Manual Round-Robin

The final mechanism for altering the **RTEMS** scheduling algorithm is called manual round-robin. Manual round-robin is invoked by using the **task_wake_after** directive with a time interval of **YIELD_PROCESSOR**. This allows a task to give up the processor and be immediately returned to the ready chain at the end of its priority group. If no other tasks of the same priority are ready to run, then the task does not lose control of the processor.

17.2.5 Dispatching Tasks

The dispatcher is the **RTEMS** component responsible for allocating the processor to a ready task. In order to allocate the processor to one task, it must be deallocated or retrieved from the task currently using it. This involves a concept called a context switch. To perform a context switch, the dispatcher saves the context of the current task and restores the context of the task which has been allocated to the processor. Saving and restoring a task's context is the storing/loading of all the essential information about a task to enable it to continue execution without any effects of the interruption. For example, the contents of a task's register set must be the same when it is given the processor as they were

when it was taken away. All of the information that must be saved or restored for a context switch is located either in the TCB or on the task's stacks.

Tasks that utilize a numeric coprocessor and are created with the **FLOATING_POINT** attribute require additional operations during a context switch. These additional operations are necessary to save and restore the floating point context of **FLOATING_POINT** tasks. To avoid unnecessary save and restore operations, 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.

17.3 Task State Transitions

Tasks in an **RTEMS** system must always be in one of the five allowable task states. These states are: **executing**, **ready**, **blocked**, **dormant**, and **non-existent**.

A task occupies the **non-existent** state before a **task_create** has been issued on its behalf. A task enters the **non-existent** state from any other state in the system when it is deleted with the **task_delete** directive. While a task occupies this state

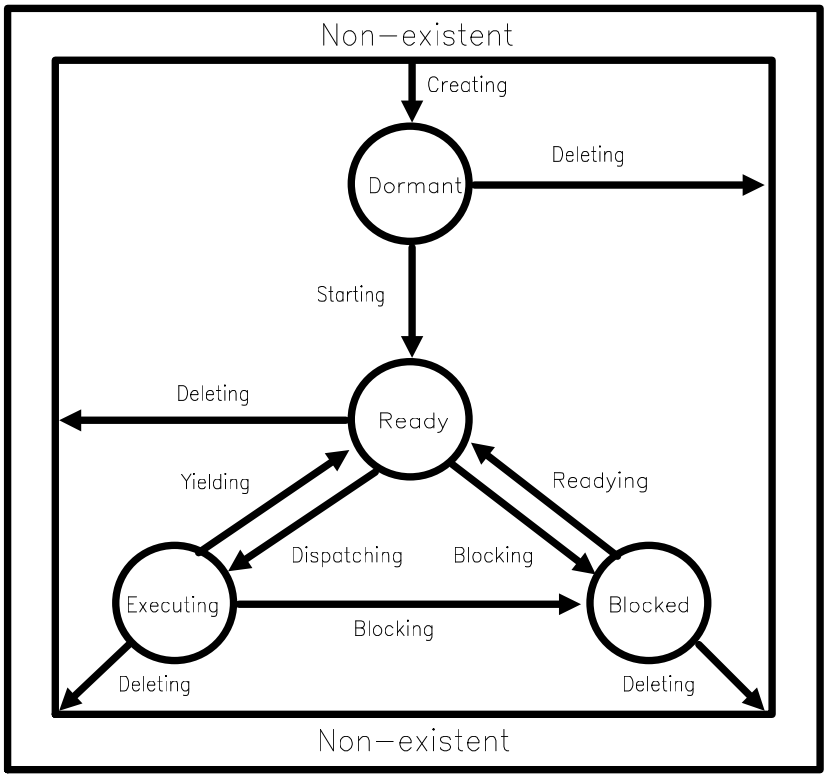

Figure 17-1 RTEMS Task State Transitions

it does not have a TCB or a task ID assigned to it; therefore, no other tasks in the system may reference this task.

When a task is created via the **task_create** directive it enters the **dormant** state. This state is not entered through any other means. Although the task exists in the system, it cannot actively compete for system resources. It will remain in the **dormant** state until it is started via the **task_start** directive, at which time it enters the **ready** state. The task is now permitted to be scheduled for the processor and to compete for other system resources.

A task occupies the **blocked** state whenever it is unable to be scheduled to run. A running task may block itself or be blocked by other tasks in the system. The running task blocks itself through voluntary operations that cause the task to wait. The only way a task can block a task other than itself is with the **task_suspend** directive. A task enters the blocked state due to any of the following conditions:

- *A task issues a **task_suspend** directive which blocks either itself or another task in the system.*
- *The running task issues a **message_queue_receive** directive with the wait option and the message queue is empty.*
- *The running task issues an **event_receive** directive with the wait option and the currently pending events do not satisfy the request.*
- *The running task issues a **semaphore_obtain** directive with the wait option and the requested semaphore is unavailable.*
- *The running task issues a **task_wake_after** directive which blocks the task for the given time interval. If the time interval specified is zero, the task yields the processor and remains in the ready state.*
- *The running task issues a **task_wake_when** directive which blocks the task until the requested date and time arrives.*
- *The running task issues a **region_get_segment** directive with the wait option and there is not an available segment large enough to satisfy the task's request.*
- *The running task issues a **rate_monotonic_period** directive and must wait for the specified rate monotonic period to conclude.*

A blocked task may also be suspended. Therefore, both the suspension and the blocking condition must be removed before the task becomes ready to run again.

A task occupies the **ready** state when it is able to be scheduled to run, but currently does not have control of the processor. Tasks of the same or higher priority will yield the processor by either becoming blocked, completing their

timeslice, or being deleted. All tasks with the same priority will execute in **FIFO** order. A task enters the **ready** state due to any of the following conditions:

- *A running task issues a **task_resume** directive for a task that is suspended and the task is not blocked waiting on any resource.*
- *A running task issues a **message_queue_send**, **message_queue_broadcast**, or a **message_queue_urgent** directive which posts a message to the queue on which the blocked task is waiting.*
- *A running task issues an **event_send** directive which sends an event condition to a task which is blocked waiting on that event condition.*
- *A running task issues a **semaphore_release** directive which releases the semaphore on which the blocked task is waiting.*
- *A timeout interval expires for a task which was blocked by a call to the **task_wake_after** directive.*
- *A timeout period expires for a task which blocked by a call to the **task_wake_when** directive.*
- *A running task issues a **region_return_segment** directive which releases a segment to the region on which the blocked task is waiting and a resulting segment is large enough to satisfy the task's request.*
- *A rate monotonic period expires for a task which blocked by a call to the **rate_monotonic_period** directive.*
- *A timeout interval expires for a task which was blocked waiting on a message, event, semaphore, or segment with a timeout specified.*
- *A running task issues a directive which deletes a message queue, a semaphore, or a region on which the blocked task is waiting.*
- *A running task issues a **task_restart** directive for the blocked task.*
- *The running task, with its preemption mode enabled, may be made **ready** by issuing any of the directives that may unblock a task with a higher priority. This directive may be issued from the running task itself or from an ISR.*

A ready task occupies the **executing** state when it has control of the CPU. A task enters the **executing** state due to any of the following conditions:

- *The task is the highest priority ready task in the system.*

- *The running task blocks and the task is next in the scheduling queue. The task may be of equal priority as in round-robin scheduling or the task may possess the highest priority of the remaining ready tasks.*
- *The running task may reenable its preemption mode and a task exists in the ready queue that has a higher priority than the running task.*
- *The running task lowers its own priority and another task is of higher priority as a result.*
- *The running task raises the priority of a task above its own and the running task is in preemption mode.*