8

Semaphore Manager

8.1 Introduction

The **semaphore manager** utilizes standard Dijkstra counting semaphores to provide synchronization and mutual exclusion capabilities. The directives provided by the semaphore manager are:

Name	Directive Description
semaphore_create	Create a semaphore
semaphore_ident	Get ID of a semaphore
semaphore_delete	Delete a semaphore
semaphore_obtain	Acquire a semaphore
semaphore_release	Release a semaphore

8.2 Background

A semaphore can be viewed as a protected variable whose value can be modified only with the **semaphore_create**, **semaphore_obtain**, and **semaphore_release** directives. **RTEMS** supports both binary and counting semaphores. A **binary semaphore** is restricted to values of zero or one, while a **counting semaphore** can assume any non-negative integer value.

A binary semaphore can be used to control access to a single resource. In particular, it can be used to enforce mutual exclusion for a critical section in user code. In this instance, the semaphore would be created with an initial count of one

to indicate that no task is executing the critical section of code. Upon entry to the critical section, a task must issue the **semaphore_obtain** directive to prevent other tasks from entering the critical section. Upon exit from the critical section, the task must issue the **semaphore_release** directive to allow another task to execute the critical section.

A counting semaphore can be used to control access to a pool of two or more resources. For example, access to three printers could be administered by a semaphore created with an initial count of three. When a task requires access to one of the printers, it issues the **semaphore_obtain** directive to obtain access to a printer. If a printer is not currently available, the task can wait for a printer to become available or return immediately. When the task has completed printing, it should issue the **semaphore_release** directive to allow other tasks access to the printer.

Task synchronization may be achieved by creating a semaphore with an initial count of zero. One task waits for the arrival of another task by issuing a **semaphore_obtain** directive when it reaches a synchronization point. The other task performs a corresponding **semaphore_release** operation when it reaches its synchronization point, thus unblocking the pending task.

8.2.1 Nested Resource Access

Deadlock occurs when a task owning a binary semaphore attempts to acquire that same semaphore and blocks as result. Since the semaphore is allocated to a task, it cannot be deleted. Therefore, the task that currently holds the semaphore and is also blocked waiting for that semaphore will never execute again.

RTEMS addresses this problem by allowing the task holding the binary semaphore to obtain the same binary semaphore multiple times in a nested manner. Each **semaphore_obtain** must be accompanied with a **semaphore_release**. The semaphore will only be made available for acquisition by other tasks when the outermost **semaphore_obtain** is matched with a **semaphore_release**.

8.2.2 Priority Inversion

Priority inversion is a form of indefinite postponement which is common in multitasking, preemptive executives with shared resources. Priority inversion occurs when a high priority tasks requests access to shared resource which is currently allocated to low priority task. The high priority task must block until the low priority task releases the resource. This problem is exacerbated when the low priority task is prevented from executing by one or more medium priority tasks. Because the low priority task is not executing, it cannot complete its interaction

with the resource and release that resource. The high priority task is effectively prevented from executing by lower priority tasks.

8.2.3 Priority Inheritance

Priority inheritance is an algorithm that calls for the lower priority task holding a resource to have its priority increased to that of the highest priority task blocked waiting for that resource. Each time a task blocks attempting to obtain the resource, the task holding the resource may have its priority increased.

RTEMS supports priority inheritance for local, binary semaphores that use the priority task wait queue blocking discipline. When a task of higher priority than the task holding the semaphore blocks, the priority of the task holding the semaphore is increased to that of the blocking task. When the task holding the task completely releases the binary semaphore (i.e. not for a nested release), the holder's priority is restored to the value it had before any higher priority was inherited.

The **RTEMS** implementation of the priority inheritance algorithm takes into account the scenario in which a task holds more than one binary semaphore. The holding task will execute at the priority of the highest priority task blocked waiting for any of the semaphores the task holds. Only when the task releases **ALL** of the binary semaphores it holds will its priority be restored to the normal value.

8.2.4 Building an Attribute Set

In general, an attribute set is built by a bitwise OR of the desired attribute components. The following table lists the set of valid semaphore attributes:

CONSTANT	DESCRIPTION	DEFAULT
FIFO	tasks wait by FIFO	*
PRIORITY	tasks wait by priority	
BINARY_SEMAPHORE	restrict values to 0 and 1	*
COUNTING_SEMAPHORE	no restriction on values	
NO_INHERIT_PRIORITY	do not use priority inheritance	*
INHERIT_PRIORITY	use priority inheritance	
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. An attribute listed as a default is not required

to appear in the attribute list, although it is a good programming practice to specify default attributes. If all defaults are desired, the attribute **DEFAULT_ATTRIBUTES** should be specified on this call.

This example demonstrates the **attribute_set** parameter needed to create a local semaphore with the task priority waiting queue discipline. The **attribute_set** parameter passed to the **semaphore_create** directive could be either **PRIORITY** or **LOCAL | PRIORITY**. The **attribute_set** parameter can be set to **PRIORITY** because **LOCAL** is the default for all created tasks. If a similar semaphore were to be known globally, then the **attribute_set** parameter would be **GLOBAL | PRIORITY**.

8.2.5 Building an Option Set

In general, an option is built by a bitwise OR of the desired option components. The set of valid options for the **semaphore_obtain** directive are listed in the following table:

CONSTANT	DESCRIPTION	DEFAULT
WAIT	task will wait for semaphore	*
NO_WAIT	task should not wait	

Option 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. An option listed as a default is not required to appear in the list, although it is a good programming practice to specify default options. If all defaults are desired, the option **DEFAULT_OPTIONS** should be specified on this call.

This example demonstrates the **option** parameter needed to poll for a semaphore. The **option** parameter passed to the **semaphore_obtain** directive should be **NO_WAIT**.

8.3 Operations

8.3.1 Creating a Semaphore

The **semaphore_create** directive creates a binary or counting semaphore with a user-specified name as well as an initial count. If a binary semaphore is created with a count of zero (0) to indicate that it has been allocated, then the task creating the semaphore is considered the current holder of the semaphore. At create time the method for ordering waiting tasks in the semaphore's task wait queue (by FIFO or task priority) is specified. Additionally, the priority inheritance algorithm may

be selected for local, binary semaphores that use the priority task wait queue blocking discipline. **RTEMS** allocates a **Semaphore Control Block** (SMCB) from the SMCB free list. This data structure is used by **RTEMS** to manage the newly created semaphore. Also, a unique semaphore ID is generated and returned to the calling task.

8.3.2 Obtaining Semaphore IDs

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

8.3.3 Acquiring a Semaphore

The **semaphore_obtain** directive is used to acquire the specified semaphore. A simplified version of the **semaphore_obtain** directive can be described as follows:

```
if semaphore's count is greater than zero
   then decrement semaphore's count
   else wait for release of semaphore
return SUCCESSFUL
```

When the semaphore cannot be immediately acquired, one of the following situations applies:

- *By default, the calling task will wait forever to acquire the semaphore.*
- *Specifying **NO_WAIT** forces an immediate return with an error status code.*
- *Specifying a timeout limits the interval the task will wait before returning with an error status code.*

If the task waits to acquire the semaphore, then it is placed in the semaphore's task wait queue in either FIFO or task priority order. If the task blocked waiting for a binary semaphore using priority inheritance and the task's priority is greater than that of the task currently holding the semaphore, then the holding task will inherit the priority of the blocking task. All tasks waiting on a semaphore are returned an error code when the semaphore is deleted.

8.3.4 Releasing a Semaphore

The **semaphore_release** directive is used to release the specified semaphore. A simplified version of the **semaphore_release** directive can be described as follows:

```
if no tasks are waiting on this semaphore
   then increment semaphore's count
   else assign semaphore to a waiting task
return SUCCESSFUL
```

If this is the outermost release of a binary semaphore that uses the priority task wait queue blocking discipline and the task does not currently hold any other binary semaphores, then the task performing the **semaphore_release** will have its priority restored to its normal value.

8.3.5 Deleting a Semaphore

The **semaphore_delete** directive removes a semaphore from the system and frees its control block. A semaphore can be deleted by any local task that knows the semaphore's ID. As a result of this directive, all tasks blocked waiting to acquire the semaphore will be readied and returned a status code which indicates that the semaphore was deleted. Any subsequent references to the semaphore's name and ID are invalid.

8.4 Directives

This section details the semaphore 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.

8.4.1 SEMAPHORE_CREATE - Create a semaphore

CALLING SEQUENCE:

```
rtems_status_code rtems_semaphore_create(
  rtems_name        name,
  rtems_unsigned32  count,
  rtems_attribute   attribute_set,
  rtems_id         *id
);
```

DIRECTIVE STATUS CODES:

SUCCESSFUL	semaphore created successfully
INVALID_NAME	invalid task name
TOO_MANY	too many semaphores created
NOT_DEFINED	invalid attribute set
INVALID_NUMBER	invalid starting count for binary semaphore
MP_NOT_CONFIGURED	multiprocessing not configured
TOO_MANY	too many global objects

DESCRIPTION:

This directive creates a semaphore which resides on the local node. The created semaphore has the user-defined name specified in **name** and the initial count specified in **count**. For control and maintenance of the semaphore, **RTEMS** allocates and initializes a SMCB. The **RTEMS**-assigned semaphore id is returned in **id**. This semaphore id is used with other semaphore related directives to access the semaphore.

Specifying **PRIORITY** in **attribute_set** causes tasks waiting for a semaphore to be serviced according to task priority. When **FIFO** is selected, tasks are serviced in First In-First Out order.

NOTES:

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

Priority inheritance is only supported for local, binary semaphores that use the priority task wait queue blocking discipline.

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

CONSTANT	DESCRIPTION	DEFAULT
FIFO	tasks wait by FIFO	*
PRIORITY	tasks wait by priority	
BINARY_SEMAPHORE	restrict values to 0 and 1	*
COUNTING_SEMAPHORE	no restriction on values	
NO_INHERIT_PRIORITY	do not use priority inheritance	*
INHERIT_PRIORITY	use priority inheritance	
LOCAL	local task	*
GLOBAL	global task	

Semaphores should not be made global unless remote tasks must interact with the created semaphore. This is to avoid the system overhead incurred by the creation of a global semaphore. When a global semaphore is created, the semaphore'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 semaphores, is limited by the **maximum_global_objects** field in the **Configuration Table**.

8.4.2 SEMAPHORE_IDENT - Get ID of a semaphore

CALLING SEQUENCE:

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

DIRECTIVE STATUS CODES:

SUCCESSFUL	semaphore identified successfully
INVALID_NAME	semaphore name not found
INVALID_NODE	invalid node id

DESCRIPTION:

This directive obtains the semaphore **id** associated with the semaphore **name**. If the semaphore name is not unique, then the semaphore id will match one of the semaphores with that name. However, this semaphore id is not guaranteed to correspond to the desired semaphore. The semaphore id is used by other semaphore related directives to access the semaphore.

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 semaphores 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.

8.4.3 SEMAPHORE_DELETE - Delete a semaphore

CALLING SEQUENCE:

```
rtems_status_code rtems_semaphore_delete(
  rtems_id id
);
```

DIRECTIVE STATUS CODES:

SUCCESSFUL	semaphore deleted successfully
INVALID_ID	invalid semaphore id
ILLEGAL_ON_REMOTE_OBJECT	cannot delete remote semaphore
RESOURCE_IN_USE	binary semaphore is in use

DESCRIPTION:

This directive deletes the semaphore specified by **id**. All tasks blocked waiting to acquire the semaphore will be readied and returned a status code which indicates that the semaphore was deleted. The SMCB for this semaphore is reclaimed by **RTEMS**.

NOTES:

The calling task will be preempted if it is enabled by the task's execution mode and a higher priority local task is waiting on the deleted semaphore. The calling task will **NOT** be preempted if all of the tasks that are waiting on the semaphore are remote tasks.

The calling task does not have to be the task that created the semaphore. Any local task that knows the semaphore id can delete the semaphore.

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

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

Proxies, used to represent remote tasks, are reclaimed when the semaphore is deleted.

8.4.4 SEMAPHORE_OBTAIN - Acquire a semaphore

CALLING SEQUENCE:

```
rtems_status_code rtems_semaphore_obtain(
  rtems_id         id,
  rtems_unsigned32 option_set,
  rtems_interval   timeout
);
```

DIRECTIVE STATUS CODES:

SUCCESSFUL	semaphore obtained successfully
UNSATISFIED	semaphore not available
TIMEOUT	timed out waiting for semaphore
OBJECT_WAS_DELETED	semaphore deleted while waiting
INVALID_ID	invalid semaphore id

DESCRIPTION:

This directive acquires the semaphore specified by **id**. The **WAIT** and **NO_WAIT** components of the **options** parameter indicate whether the calling task wants to wait for the semaphore to become available or return immediately if the semaphore is not currently available. With either **WAIT** or **NO_WAIT**, if the current semaphore count is positive, then it is decremented by one and the semaphore is successfully acquired by returning immediately with a successful return code.

If the calling task chooses to return immediately and the current semaphore count is zero or negative, then a status code is returned indicating that the semaphore is not available. If the calling task chooses to wait for a semaphore and the current semaphore count is zero or negative, then it is decremented by one and the calling task is placed on the semaphore's wait queue and blocked. If the semaphore was created with the **PRIORITY** attribute, then the calling task is inserted into the queue according to its priority. However, if the semaphore was created with the **FIFO** attribute, then the calling task is placed at the rear of the wait queue. If the semaphore was created with the **INHERIT_PRIORITY** attribute, then the priority of the task currently holding the binary semaphore is guaranteed to be greater than or equal to that of the blocking task.

The **timeout** parameter specifies the maximum interval the calling task is willing to be blocked waiting for the semaphore. If it is set to **NO_TIMEOUT**, then the calling task will wait forever. If the semaphore is available or the **NO_WAIT** option component is set, then **timeout** is ignored.

NOTES:

The following semaphore acquisition option constants are defined by **RTEMS**:

CONSTANT	DESCRIPTION	DEFAULT
WAIT	**task will wait for semaphore**	*
NO_WAIT	**task should not wait**	

Attempting to obtain a global semaphore which does not reside on the local node will generate a request to the remote node to access the semaphore. If the semaphore is not available and **NO_WAIT** was not specified, then the task must be blocked until the semaphore is released. A proxy is allocated on the remote node to represent the task until the semaphore is released.

8.4.5 SEMAPHORE_RELEASE - Release a semaphore

CALLING SEQUENCE:

```
rtems_status_code rtems_semaphore_release(
  rtems_id id
);
```

DIRECTIVE STATUS CODES:

SUCCESSFUL	semaphore released successfully
INVALID_ID	invalid semaphore id
NOT_OWNER_OF_RESOURCE	calling task does not own semaphore

DESCRIPTION:

This directive releases the semaphore specified by **id**. The semaphore count is incremented by one. If the count is zero or negative, then the first task on this semaphore's wait queue is removed and unblocked. The unblocked task may preempt the running task if the running task's preemption mode is enabled and the unblocked task has a higher priority than the running task.

NOTES:

The calling task may be preempted if it causes a higher priority task to be made ready for execution.

Releasing a global semaphore which does not reside on the local node will generate a request telling the remote node to release the semaphore.

If the task to be unblocked resides on a different node from the semaphore, then the semaphore allocation is forwarded to the appropriate node, the waiting task is unblocked, and the proxy used to represent the task is reclaimed.

The outermost release of a local, binary, priority inheritance semaphore may result in the calling task having its priority lowered. This will occur if the calling task holds no other binary semaphores and it has inherited a higher priority.