### 3.2.4 Data Structures for Message Management

Definitions for events and asynchronous signals are as follows:

| | |
|---|---|
| S_EXEC0 | System Software defined |
| S_EXEC1 | System Software defined |
| S_EXEC2 | System Software defined |
| S_EXEC3 | System Software defined |
| S_EXEC4 | System Software defined |
| S_EXEC5 | System Software defined |
| S_EXEC6 | System Software defined |
| S_EXEC7 | System Software defined |
| S_EXEC8 | System Software defined |
| S_EXEC9 | System Software defined |
| S_EXEC10 | System Software defined |
| S_EXEC11 | System Software defined |
| S_EXEC12 | System Software defined |
| S_EXEC13 | System Software defined |
| S_EXEC14 | System Software defined |
| S_EXEC15 | System Software defined |
| S_USER0 | User defined |
| S_USER1 | User defined |
| S_USER2 | User defined |
| S_USER3 | User defined |
| S_USER4 | User defined |
| S_USER5 | User defined |
| S_USER6 | User defined |
| S_USER7 | User defined |
| S_USER8 | User defined |
| S_USER9 | User defined |
| S_USER10 | User defined |
| S_USER11 | User defined |
| S_USER12 | User defined |
| S_USER13 | User defined |
| S_USER14 | User defined |
| S_USER15 | User defined |

### 3.2.5 Q_CREATE

**NAME**

q_create – "Create a Message Queue"

**SYNOPSIS**

```
#include <message.h>
uint q_create (name, count, flags, &qid )

        uint name;     /* user defined 4-byte name */
        uint count;    /* maximum message and reserved buffer count */
        uint flags;    /* process method */
        uint qid;      /* message queue id - returned by this call */
```

The flags values are:

| | | |
|---|---|---|
| PRIOR | set | to process by priority |
| | clear | to process by FIFO |
| GLOBAL | set | to indicate the queue is a multiprocessor global resource. |
| | clear | to indicate the queue is local |
| TYPE | set | to process typed messages |
| | clear | to process messages without regard to type |
| LIMIT | set | to limit queue entries to number in count field |
| | clear | NO limit on queue entries and no reserved buffers |
| RESVD | set | to reserve system buffers equal to count when LIMIT is set |
| | clear | NO reserved system buffers when LIMIT is set |

**DESCRIPTION**

The *q_create* directive creates a message queue by allocating and initializing a message queue data structure. A message queue is created by name. A message *qid* is returned. Subsequent sending and receiving calls must reference the message queue with its message *qid*.

By setting the **PRIOR** value in the flags field, tasks waiting for messages in the queue will be processed by task priority order. Otherwise the tasks waiting for messages will be processed by first in, first out (FIFO) order.

By setting the **TYPE** value in the flags field, messages sent to this queue may be processed by type.

The user may put a limit on the number of messages at the message queue by setting the **LIMIT** value in the flags field, and placing the count in the *count* field. The user may additionally reserve a number of system message buffers equal to the count in the *count* field by setting the **RESVD** value in the flags field.

By setting the **GLOBAL** value in the flags field, the message *qid* will be sent to all processors in the system, to be entered into a global resource table. The system is defined as the collection of interconnected processors. The message queue is always created on the local node.

The maximum number of message queues that can be in existence at one time is a configuration parameter.

The maximum number of system message buffers is a configuration parameter.

## RETURN VALUE

If the *q_create* directive succeeds, the *qid* is filled in, and 0 is returned.

If the call was not successful, an error code is returned.

## ERROR CONDITIONS

Too many message queues.

No more system message buffers.

## NOTES

Not callable from ISR.

Will not cause a preempt.

### 3.2.6 Q_IDENT

**NAME**

q_ident -- "Obtain id of a Message Queue"

**SYNOPSIS**

```
#include <message.h>
uint q_ident ( name, node, &qid )

                uint name;    /* user defined 4-byte name */
                uint node;    /* node identifier */
                              /* 0 indicates any node */
                uint qid;     /* message queue id - returned by this call */
```

**DESCRIPTION**

The *q_ident* directive allows a task to identify a previously created message queue by name and receive the message *qid* to use for send and receive directives for the queue.

If the message queue name is not unique, the message *qid* returned may not correspond to the message queue named in this call.

The message queue may exist on the local processor or any remote processor in a multiprocessor configuration, as long as the queue was created with the GLOBAL flags value set (see *q_create*). If the message queue name is not unique within the multiprocessor configuration, a non-zero node identifier must be specified in the *node* field.

**RETURN VALUE**

If the *q_ident* directive succeeds, the *qid* is filled in, and 0 is returned.

If the call was not successful, an error code is returned.

**ERROR CONDITIONS**

Named message queue does not exist.

Invalid node identifier.

**NOTES**

Can be called from within an ISR.

Will not cause a preempt.

### 3.2.7 Q_DELETE

**NAME**

q_delete -- "Delete a Message Queue"

**SYNOPSIS**

```
#include <message.h>
uint q_delete ( qid )

          uint qid;    /* message queue id returned from q_create or q_ident */
```

**DESCRIPTION**

The *q_delete* directive deletes the message queue identified by the *qid*, freeing the data structure.

When a message queue is deleted, it could be in one of three states: empty, tasks waiting for messages, messages waiting for tasks. If empty, the data structure of the message queue is returned to the system. If tasks are waiting, each is made ready and given a return code indicating a deleted message queue. If messages are waiting, then each system message buffer is returned to the system message buffer pool, and the message it is carrying is therefore lost.

The message queue must exist on the local processor. If the message queue was created with the **GLOBAL** flags value set in a multiprocessor configuration, a notification will be sent to all processors in the system, so the *qid* can be deleted from the global resource table.

The requester does not have to be the creator of the message queue. Any task knowing the *qid* can delete it.

**RETURN VALUE**

If the *q_delete* directive successfully deleted the message queue, then 0 is returned.

If the call was not successful, an error code is returned.

**ERROR CONDITIONS**

Message *qid* is invalid.

Message queue not created from local node.

**NOTES**

Cannot be called from within an ISR.

May cause a preempt if a task waiting at the message queue has a higher priority than the running task, and the preempt mode is in effect. A preempt will not occur if all tasks waiting at the message queue exist on a remote processor in a multiprocessor configuration.

## 3.2.8 Q_SEND

### NAME

q_send -- "Send a Message to a Message Queue"

### SYNOPSIS

```
#include <message.h>
uint q_send ( qid, buffer )

        uint qid;             /* message queue id returned from q_create or q_ident */
        long (*buffer)[4];    /* pointer to message buffer */
```

### DESCRIPTION

The *q_send* directive sends a message to the queue identified by the *qid*.

If a task is already waiting at the queue, the message is copied to that task's indicated receiving buffer. The waiting task is then made ready. If there is no task waiting, the message is copied to a system message buffer which is then placed at the end of the message queue.

Once sent, the task's message buffer may be reused immediately. A message is fixed length, 16-bytes.

The message queue may exist on the local processor or any remote processor in a multiprocessor configuration, as long as the queue was created with the GLOBAL flags value set (see *q_create* ).

### RETURN VALUE

If the *q_send* directive successfully sent a message, then 0 is returned.

If the call was not successful, an error code is returned.

### ERROR CONDITIONS

Message *qid* is invalid.

Out of system message buffers.

Message queue at maximum count.

ISR cannot reference remote node.

### NOTES

Can be called from within an ISR, except when the queue was not created from the local node.

May cause a preempt if a task waiting at the message queue has a higher priority than the running task, and the preempt mode is in effect. A preempt will not occur if a task waiting exists on a remote processor in a multiprocessor configuration.

### 3.2.9 Q_URGENT

**NAME**

q_urgent -- "Place an Urgent Message at the Head of a Message Queue"

**SYNOPSIS**

```
#include <message.h>
uint q_urgent ( qid, buffer )

        uint qid;             /* message queue id returned from q_create or q_ident */
        long (*buffer)[4];    /* pointer to message buffer */
```

**DESCRIPTION**

The *q_urgent* directive sends a message to the queue identified by the *qid*. This call is the same as the *q_send* call, except, if there are other messages at the queue, this message is put at the head of the queue.

If a task is already waiting at the queue, the message is copied to that task's indicated receiving buffer. The task is then made ready. If there is no task waiting, the message is copied to a system buffer which is then placed at the head c the message queue.

Once sent, the task's message area may be reused immediately. A message is fixed length, 16-bytes.

The message queue may exist on the local processor or any remote processor in a multiprocessor configuration, as long as the queue was created with the GLOBAL flags value set (see *q_create* ).

**RETURN VALUE**

If the *q_urgent* directive successfully sent a message, then 0 is returned.

If the call was not successful, an error code is returned.

**ERROR CONDITIONS**

Message *qid* is invalid.

Out of system message buffers.

Message queue at maximum count.

ISR cannot reference remote node.

**NOTES**

Can be called from within an ISR, except when the queue was not created from the local node.

May cause a preempt if a task waiting at the message queue has a higher priority than the running task, and the preempt mode is in effect. A preempt will not occur if a task waiting exists on a remote processor in a multiprocessor configuration.

### 3.2.10 Q_BROADCAST

#### NAME

q_broadcast -- "Broadcast N Identical Messages to a Message Queue"

#### SYNOPSIS

```
#include <message.h>
uint q_broadcast ( qid, buffer, &count )

        uint qid;            /* message queue id returned from q_create or q_ident */
        long (*buffer)[4];   /* pointer to message buffer */
        uint count;          /* number of tasks made ready - returned by this call */
```

#### DESCRIPTION

The *q_broadcast* directive sends as many messages as necessary to make ready all tasks waiting on the queue identified by the *qid*. The number of tasks readied is returned to the caller in *count*.

Once sent, the task's message buffer may be reused immediately.

The message queue may exist on the local processor or any remote processor in a multiprocessor configuration, as long as the queue was created with the GLOBAL flags value set (see *q_create* ).

#### RETURN VALUE

If the *q_broadcast* directive succeeds, the *count* is filled in with the number of tasks readied, and 0 is returned.

If the call was not successful, an error code is returned.

#### ERROR CONDITIONS

Message *qid* is invalid.

ISR cannot reference remote node.

#### NOTES

Can be called from within an ISR, except when the queue was not created from the local node.

May cause a preempt if a task waiting at the message queue has a higher priority than the running task, and the preempt mode is in effect. A preempt will not occur if a task waiting exists on a remote processor in a multiprocessor configuration.

### 3.2.11 Q_RECEIVE

**NAME**

q_receive -- "Receive a Message from a Message Queue"

**SYNOPSIS**

```
#include <message.h>
uint q_receive ( qid, buffer, flags, timeout )

        uint qid;            /* message queue id returned from q_create or q_ident */
        long (*buffer)[4];   /* pointer to message buffer */
        uint flags;          /* options */
        uint timeout;        /* number of ticks to wait */
                             /* 0 indicates wait forever */
```

The flags values are:

| | | |
|---|---|---|
| NOWAIT | set | if the task is to return immediately |
| | clear | if the task is to wait for a message |

**DESCRIPTION**

The *q_receive* directive allows a task to request a message from the message queue identified by *qid*.

If there is a message at the message queue, it is copied into the requester's buffer.

If there is no message at the message queue, then the NOWAIT flag determines what to do. If the NOWAIT flags value is set, the task returns immediately with -1 and the no message at queue error number. If the NOWAIT flags value is clear, the task is put on a wait list for the message queue, according the queue's attributes (FIFO or priority).

The *timeout* field is used to determine how long to wait. A zero in the *timeout* field indicates no timeout -- wait forever. A non-zero entry in the *timeout* field indicates that the task will run after that many ticks, if a message has not been received, or before if a message is received.

When *q_receive* is called from an ISR, the no wait option is forced by the executive. Thus there will be no waiting for a message. An error will be returned if there is no message.

The message queue may exist on the local processor or any remote processor in a multiprocessor configuration, as long as the queue was created with the GLOBAL flags value set (see *q_create* ).

**RETURN VALUE**

If the *q_receive* directive succeeds, then 0 is returned.

If the call was not successful, an error code is returned.

## ERROR CONDITIONS

Message *qid* is invalid.

No message at queue ( if no wait is selected ).

Message queue deleted.

Timed out with no message ( if wait and timeout is selected ).

ISR cannot reference remote node.

## NOTES

Can be called from within an ISR, except when the queue was not created from the local node. The executive will force the options to no wait.

The requesting task may be blocked if there is no message available, and the wait option is selected.

### 3.2.12 EV_SEND

**NAME**

ev_send -- "Send Event to a Task"

**SYNOPSIS**

```
uint ev_send ( tid, event )

                    uint tid;      /* task id as returned by t_create or t_ident */
                    uint event;    /* event set */
```

**DESCRIPTION**

The *ev_send* directive sends an event to a task. The *event* field describes the set of events the task wishes to send. Thirty-two events are available. Sixteen are available as *system* events and sixteen are available as *user* events.

The task identified by the *tid* may exist on the local processor or any remote processor in a multiprocessor configuration, as long as the task was created with the **GLOBAL** flags value set (see *t_create* ).

Events sent to tasks not waiting for an event are left pending.

**RETURN VALUE**

If the *ev_send* directive succeeds, then 0 is returned.

If the call was not successful, an error code is returned.

**ERROR CONDITIONS**

Invalid *tid*.

ISR cannot reference remote node.

**NOTES**

Can be called from within an ISR, except when the task was not created from the local node.

May cause a preempt if the task waiting for the event has a higher priority than the running task, and the preempt mode is in effect. A preempt will not occur if the task waiting exists on a remote processor in a multiprocessor configuration.

### 3.2.13 EV_RECEIVE

#### NAME

ev_receive -- "Receive Event"

#### SYNOPSIS

uint ev_receive ( eventin, flags, timeout, &eventout )

```
uint eventin;    /* input event condition */
uint flags;      /* options */
uint timeout;    /* number of ticks to wait */
                 /* 0 indicates wait forever */
uint eventout;   /* output events - returned by this call */
```

The flags values are:

| | | |
|---|---|---|
| NOWAIT | set | if the task is to return immediately |
| | clear | if the task is to wait for event condition |
| ANY | set | return when any one of the indicated events has occurred |
| | clear | return when all of the indicated events have occurred |

#### DESCRIPTION

The *ev_receive* directive allows a task to receive an event condition. The event condition to receive is a set of events specified in the *eventin* field.

The task may elect to wait for the event condition, or return immediately by setting the **NOWAIT** value in the flags field. The task may elect to receive all of the events, or receive any one of them by setting the **ANY** value in the flags field.

When pending events satisfy the event condition, the events are cleared and the task will remain running. Otherwise, if the task elects to wait, the task will become blocked. The task will be made ready to run when the event condition is satisfied by new events, or the timeout condition is met.

When pending events do not satisfy the event condition, and the task elects not to wait, the task returns immediately with -1 and the no event available error number.

If the *eventin* field is 0, *ev_receive* will return the pending events, but the events will remain pending.

The *timeout* field is used to determine how long to wait. A zero in the *timeout* field indicates no timeout -- wait forever. A non-zero entry in the *timeout* field indicates that the task will run after that many ticks, if the event condition is not satisfied, or before if the event condition is satisfied.

## RETURN VALUE

If the *ev_receive* directive succeeds, *eventout* is filled in with the output events, and 0 is returned.

If the call was not successful, an error code is returned.

## ERROR CONDITIONS

Event not satisfied ( if no wait is selected ).

Timed out with no event ( if wait and timeout is selected ).

## NOTES

Cannot be called from within an ISR.

The requesting task may be blocked if the event condition is not satisfied, and the wait option is selected.

### 3.2.14 AS_CATCH

**NAME**

as_catch -- "Catch Signals"

**SYNOPSIS**

```
uint as_catch ( asraddr, mode )

            ptf asraddr;    /* address of Asynchronous Signal Routine (asr) */
                            /* 0 indicates asr is invalid */
            uint mode;      /* mode value for asr */
```

The *mode* value is defined as follows:

| | | |
|---|---|---|
| NOPREEMPT | set | to disable preempting |
| | clear | to enable preempting |
| TSLICE | set | to enable timeslicing |
| | clear | to disable timeslicing |
| DISASR | set | to disable asr processing |
| | clear | to enable asr processing |
| SUPV | set | to execute in supervisor mode |
| | clear | to execute in user mode |
| LEVEL | | interrupt level when SUPV is set |

**DESCRIPTION**

The *as_catch* directive allows a task to specify what action to take when catching signals.

The asr address is established when *as_catch* is called with a non-zero address in the *asraddr* field. Zero is not a valid asr address. The asr is invalidated when *as_catch* is called with the *asraddr* field equal zero. Asynchronous signal processing will be discontinued until re-enabled with a valid asr address in another *as_catch* call.

When a signal is caught, the task is not unblocked. Signals are latched until the task becomes the running task, at which time the task is dispatched to its asr. The task will execute the asr according to the values specified in the *mode* field. The signal condition will be passed to the task, along with the the task's current PC and mode, on the task's stack in a signal stack frame. The signal condition contains all of the signals which have been received since the last time the task was executing.

The asr is responsible for saving and restoring all registers it uses.

The *as_return* directive must be executed to return the task to its previous dispatch address.

Only one asr per task is allowed.

**RETURN VALUE**

The *as_catch* directive always succeeds, and returns 0.

## ERROR CONDITIONS

None.

## NOTES

Cannot be called from within an ISR.

Will not cause a preempt.

### 3.2.15 AS_SEND

**NAME**

as_send -- "Send Signal to a Task"

**SYNOPSIS**

```
uint as_send ( tid, signal )

                uint tid;       /* task id as returned by t_create or t_ident */
                uint signal;    /* signal set */
```

**DESCRIPTION**

The *as_send* directive sends signals to a task. The *signal* field describes the set of signals it wishes to send. Thirty-two signals are available. Sixteen are available as *system* signals and sixteen are available as *user* signals.

The signal set must be sent to tasks which have specified an asr using the *as_catch* directive. If the task identified by the *tid* does not have a valid asr, the caller returns with the invalid asr error.

When a signal is sent to a task with a valid and enabled asr, the task will be dispatched to the asr address when it becomes the running task. Signals sent to a blocked task are latched until the task becomes the running task. Duplicate signals are not queued.

The task identified by the *tid* may exist on the local processor or any remote processor in a multiprocessor configuration, as long as the task was created with the GLOBAL flags value set (see *t_create* ).

**RETURN VALUE**

If the *as_send* directive successfully sent the signal, then 0 is returned.

If the call was not successful, an error code is returned.

**ERROR CONDITIONS**

Invalid *tid*.

Invalid asr.

ISR cannot reference remote node.

**NOTES**

Can be called from within an ISR, except when the task was not created from the local node.

### 3.2.15 AS_SEND

**NAME**

as_send -- "Send Signal to a Task"

**SYNOPSIS**

```
uint as_send ( tid, signal )

                uint tid;       /* task id as returned by t_create or t_ident */
                uint signal;    /* signal set */
```

**DESCRIPTION**

The *as_send* directive sends signals to a task. The *signal* field describes the set of signals it wishes to send. Thirty-two signals are available. Sixteen are available as *system* signals and sixteen are available as *user* signals.

The signal set must be sent to tasks which have specified an asr using the *as_catch* directive. If the task identified by the *tid* does not have a valid asr, the caller returns with the invalid asr error.

When a signal is sent to a task with a valid and enabled asr, the task will be dispatched to the asr address when it becomes the running task. Signals sent to a blocked task are latched until the task becomes the running task. Duplicate signals are not queued.

The task identified by the *tid* may exist on the local processor or any remote processor in a multiprocessor configuration, as long as the task was created with the GLOBAL flags value set (see *t_create* ).

**RETURN VALUE**

If the *as_send* directive successfully sent the signal, then 0 is returned.

If the call was not successful, an error code is returned.

**ERROR CONDITIONS**

Invalid *tid*.

Invalid asr.

ISR cannot reference remote node.

**NOTES**

Can be called from within an ISR, except when the task was not created from the local node.

### 3.2.16 AS_RETURN

**NAME**

as_return -- "Return from Signal Routine"

**SYNOPSIS**

void as_return ( )

**DESCRIPTION**

The *as_return* must be used by tasks to return from an asynchronous signal routine (asr).

**RETURN VALUE**

None.

**ERROR CONDITIONS**

Not in asr.

**NOTES**

This call is only used to return from an asr. Refer to the *as_catch* and *as_send* directives.

## 3.3 Semaphore Management

The semaphore manager provides a set of directives to use in arbitrating access to a shared resource (many-to-one). The semaphores primitives provided can be used to fulfill different sets of requirements:

1. To control access to a single resource that is either available or not, the user can create a semaphore with an initial value of 1.

2. To control access to a pool of "n" resources where at any moment "m" of those resources are available ( $0 <= m <= n$ ) and "n-m" are not, the user can create a semaphore with an initial value of "n".

Arbitrating access to shared resources requires signaling that a predefined event has occurred. Sophisticated synchronisation also requires a counter to record the number of events sent but not yet received, and a list of tasks awaiting receipt of the event.

The semaphore data structure fulfills all the previous requirements. A semaphore possesses a name to distinguish it from the other semaphores within the system, a semaphore id to enable quick access to the semaphore, the requisite semaphore count variable to count the events, and a list of waiting tasks. In addition to the semaphore count variable, the semaphore contains an initial count, used as an initial assignment value for the semaphore count.

The synchronisation rules for semaphores ar :

1. The semaphore count is decremented by 1, when a task does a *sm_p* operation. The task continues execution if the count is then greater than or equal to zero. If the count is less than zero, the task is put on a waiting list for the semaphore.

2. The semaphore count is incremented by one when a task does a *sm_v* operation. If the count is less than or equal to zero, the first task in the semaphore waiting list is placed in the ready state.

The directives provided by the semaphore manager are:

| Directive | Function |
|---|---|
| sm_create | Get a semaphore |
| sm_ident | Obtain the id of a Semaphore |
| sm_delete | Delete a semaphore |
| sm_p | Access semaphore |
| sm_v | Release semaphore |

### 3.3.1 SM_CREATE

#### NAME

sm_create -- "Create a Semaphore"

#### SYNOPSIS

```
#include <semaphore.h>
uint sm_create ( name, count, flags, &smid )

        uint name;     /* semaphore name */
        uint count;    /* initial count */
        uint flags;    /* semaphore flags */
        uint smid;     /* semaphore id - returned by this call */
```

The flags field values are:

| | | |
|---|---|---|
| PRIOR | set | to process wait list by priority |
| | clear | to process wait list by FIFO |
| GLOBAL | set | to indicate the semaphore is a multiprocessor global resource. |
| | clear | to indicate the semaphore is local. |

#### DESCRIPTION

The *sm_create* directive creates a semaphore and assigns it an initial count equal to the value in the *count* field. The semaphore id is returned in *smid*. The *smid* must be used in subsequent *sm_p, sm_v,* and *sm_delete* calls.

By setting the **PRIOR** value in the flags field, tasks waiting on a semaphore will be processed in task priority order. Otherwise the tasks will be processed in first in, first out (FIFO) order.

By setting the **GLOBAL** value in the flags field, the *smid* will be sent to all processors in the system, to be entered into a global resource table. The system is defined as the collection of interconnected processors. The semaphore is always created on the local node.

The maximum number of semaphores that can be in existence at one time is a configuration parameter.

#### RETURN VALUE

If *sm_create* successfully created the semaphore, the *smid* is filled in, and 0 is returned.

If the semaphore was not successfully created, an error code is returned.

#### ERROR CONDITIONS

Too many semaphores.

## NOTES

Not callable from ISR.

Will not cause a preempt.