ERROR CONDITIONS

Invalid *tid*.

Invalid register number.

ISR cannot reference remote node.

NOTES

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

Will not cause a preempt.

3.2 Message, Event, and Signal Management

The executive supports communication and synchronisation between tasks using messages and events. Asynchronous communication is supported using signals.

3.2.1 Message Manager

The message queue is the data structure supporting inter-task communication and synchronization. One or more tasks may send messages to the message queue, and one or more tasks may request messages from the queue.

Message queues are created at run time using the *q_create* directive. The creator assigns a 4-byte name and attributes to the queue. The attributes define whether tasks waiting on messages from the queue will wait first-in, first-out (FIFO), or by task priority, and whether the queue will limit the number of messages queued to a specified maximum, or allow an unlimited number of messages.

A message queue is identified by both a name, assigned by the creator, and a message queue id (*qid*), assigned by the executive at *q_create* time. The *qid* is returned to the caller by the *q_create* directive, and must be used by tasks to send and receive messages from the message queue. Tasks other than the task which created the message queue can obtain the *qid* by using the *q_ident* directive.

Messages are sent to the message queue from any task which knows the *qid*, using the *q_send*, *q_urgent*, and *q_broadcast* directives.

When a message arrives at the queue, it will be copied into one of two places. If there is one or more tasks waiting at the queue, then the message is copied into the message buffer belonging to the waiting task. The task is removed from the wait list and is made ready. If there are no tasks waiting at the queue, then the message is copied into a system message buffer (the executive maintains a pool of system message buffers for this purpose). This system message buffer is entered into the message queue. If the message was sent using *q_send*, the message is entered at the tail of the queue. If the message was sent using *q_urgent*, the message is entered at the head of the queue. The *q_broadcast* directive sends a message to all tasks waiting at the queue, so they become ready to run. The count of readied tasks is returned to the caller.

Messages are received from the message queue using the *q_receive* directive. When this directive is called, and a message is in the queue, the message is copied to the task's message buffer, and the directive is complete. When no message is in the queue, there are several ways to proceed. If the calling task asked to wait, the task will be entered into the queue's wait list according the queue's attributes (FIFO or priority). If the calling task asked to wait with timeout, the task will be entered into a timeout list. If the calling task asked not to wait, the task will be returned to with an error code for no message available.

Message queues can be deleted by tasks knowing the *qid* using the *q_delete* directive. If any messages are queued, the executive will claim and return the system message buffers to the system message buffer pool. If any tasks are waiting on the queue, then the executive will remove them from the wait list and make them ready. Waiting tasks will return from the *q_receive* directive with the message queue deleted error.

The message manager defines a message as being fixed length, 16-bytes. The content of the message is user defined. It may be used to carry data, pointers to data, or nothing at all.

The directives provided by the message manager are:

Directive	Function
q_create	Create queue
q_ident	Obtain id of a queue
q_delete	Delete queue
q_send	Send message
q_urgent	Urgent message
q_broadcast	Broadcast message
q_receive	Receive message

3.2.2 Event Manager

Although inter-task synchronisation can be accomplished using the message queue, the executive also provides a second, higher performance method of inter-task synchronization, using events.

Events are different from messages in that they are directed at other tasks. They are also different from messages in that they carry no information, and they cannot be queued. The final difference is tasks can wait for several events at one time, but cannot wait on multiple message queues at one time.

Every task in the system has the ability to send and receive events. Events are simply bits encoded into a event mask. Thirty-two events are available; sixteen will be available as *system* events and sixteen will be available as *user* events. A task can send one or more events to another task using the *ev_send* directive. The *tid* of the destination task is required as input, along with the event set.

A task can receive events using the *ev_receive* directive. The events to receive are input to the directive, along with an option to wait on all of the events, or just one of them. If the events are already pending, then the event mask is cleared before returning to the calling task. If the event condition cannot be satisfied, and the calling task asked to wait, the task will be blocked. If the calling task asked to wait with timeout, the task will be entered into a timeout list. Tasks that do not want to wait for the event condition must specify this as an option. If the event condition was not pending, then an error code for event condition not met is returned.

The directives provided by the event manager are:

Directive	Function
ev_send	Send event
ev_receive	Receive event

3.2.3 Signal Manager

Asynchronous communication is supported through the use of signals.

Signals, like events, are simply bits encoded into a signal mask. Thirty-two signals are available; sixteen will be available as *system* signals and sixteen will be available as *user* signals.

A task can send one or more signals to another task using the *as_send* directive. If the receiving task has set up an asynchronous signal routine (asr) using the *as_catch* directive, the task will be dispatched to the signal routine.

A task may asynchronously receive signals by establishing an asynchronous signal routine (asr) to catch them using the *as_catch* directive. When a signal is caught, the task will be dispatched to the asr address when it becomes the running task. The signal condition will be passed to the task to enable it to determine what signals occurred.

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

The directives provided by the signal manager are:

Directive	Function
as_catch	Catch signal
as_send	Send signal
as_return	Return from signal

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