7.9. QUEUE_INFO

Obtain information on a queue.

Synopsis

```
queue_info( qid, max_buff, length, options, messages_waiting
            tasks_waiting )
```

Input Parameters

qid	: queue_id	kernel defined queue identifier

Output Parameters

max_buff	: integer	maximum number of buffers in queue
length	: integer	length of message buffers in bytes
options	: bit_field	semaphore create options
tasks_waiting	: integer	number of tasks waiting on the message queue
messages_waiting	: integer	number of messages waiting in the message queue

Completion Status

OK	queue_info operation successful
INVALID_PARAMETER	a parameter refers to an illegal address
INVALID_ID	queue does not exist
OBJECT_DELETED	queue specified has been deleted
NODE_NOT_REACHABLE	node on which the queue resides is not reachable

Description

This operation provides information on the specified message queue. It returns its maximum number of buffers in bytes, its create options, and the number of tasks waiting for messages on this queue, respectively the number of messages waiting in the queue to be read.
The latter two values should be used with care as they are just a snapshot of the semaphores's state at the time of executing the operation.

8. EVENTS

Events provide a simple method of task synchronization. Each task has the same number of events. The maximum number of these is implementation dependent, but the minimum number is fixed at sixteen. Events have no identifiers, but are addressed using a task identifier and a bit-field. A bit-field can indicate any number of a task's events at once.

A task can wait on any combination of its events, requiring either all specified events to arrive, or at least one of them, before being unblocked. Tasks can send any combination of events to a given task. If the receiving task is not in the same node as the sending task, then the receiving task must be global.

Sending events in effect sets a one bit latch for each event. Receiving a combination of events clears the appropriate latches. This means that if an event is sent more than once before being received, the second and subsequent sends are not seen.

8.1. EVENT_SEND

Send event(s) to a task.

Synopsis

```
event_send( tid, event )
```

Input Parameters

tid	: task_id	kernel defined task identifier
event	: bit_field	event(s) to be sent

Output Parameters

<none>

Completion Status

OK	event_send operation successful
INVALID_PARAMETER	a parameter refers to an illegal address
INVALID_ID	task does not exist
OBJECT_DELETED	task specified has been deleted
NODE_NOT_REACHABLE	node on which semaphore resides is not reachable

Description

This operation sends the given event(s) to the given task. The appropriate task event latches are set. If the task is waiting on a combination of events, a check is made to see if the currently set latches satisfy the requirements. If this is the case, the given task receives the event(s) it is waiting on and the appropriate bits are cleared in the latch.

8.2. EVENT_RECEIVE

Receive event(s).

Synopsis

```
event_receive( events, options, time_out, events_received )
```

Input Parameters

events	: bit_field	event(s) to receive
options	: bit_field	receive options
time_out	: integer	max no of ticks to wait

Output Parameters

events_received : bit_field event(s) received

Literal Values

options	+ ANY	return when any of the events is sent
	+ NOWAIT	do not wait - return immediately if no events set
time_out	= FOREVER	wait forever - do not time out

Completion Status

OK	event_receive operation successful
ILLEGAL_USE	operation not callable from ISR
INVALID_PARAMETER	a parameter refers to an illegal address
INVALID_OPTIONS	invalid options value
TIME_OUT	event_receive operation timed out
NO_EVENTS	event(s) not set and NOWAIT option given

Description

This operation waits on a given combination of events to occur. By default, the operation waits until all of the events have been sent. If the ANY option is set, then the operation waits only until any one of the events has been sent.

The operation first checks the task's event latches to see if the required event(s) have already been sent. In this case the task receives the events, which are returned in events_caught, and the appropriate event latches are cleared. If the ANY option was set, and more than one of the specified events was sent, all the events sent, satisfying the events, are received.

If the required event(s) have yet to be sent, and the NOWAIT option has been specified, the NO_EVENTS completion status is returned. If NOWAIT is not specified then the task is blocked, waiting on the appropriate events to be sent. A timeout is initiated, unless the time_out value supplied is FOREVER. If all required events are sent before the timeout expires, then the events are received and a successful completion status returned. If the timeout expires, the TIME_OUT completion status is returned.

9. EXCEPTIONS

ORKID exceptions provide tasks with a method of handling exceptional conditions asynchronously. Each task has the same number of exceptions. The maximum number of these is implementation dependent, but the minimum number is fixed at sixteen. Exceptions have no identifiers, but are addressed using a task identifier and a bit field, which can indicate any number of exceptions at once.

Exceptions are identified in the same manner as events. Using a bit field, any number of exceptions can be raised simultaneously to a task. Raising an exception sets a one bit latch for each exception. If the same exception is raised more than once to a task before the task can catch them, then the second and subsequent raisings are ignored. If the target task is not in the same node as the raising task, then the target task must be global.

The 'catching' of exceptions is quite different than that of events, and involves the activation of the task's Exception Service Routine (XSR). XSRs have to be declared via the exception_catch operation to tasks after their creation. A task may change its XSR at any time.

An XSR is activated whenever one or more exceptions are raised to a task, and the task has not set its NOXHR modal parameter in the active mode. If the NOXHR parameter is set, the XSR will be activated as soon as it is cleared. When an XSR is activated, the task's current flow of execution is interrupted and the XSR entered. The XSR is passed the bit field indicating which exceptions have been sent as a parameter. The exact way how to accomplish this is defined in the language binding. The XSR always catches all exceptions which have been raised, and all the latches are cleared.

An XSR is treated by the scheduler in exactly the same way as other parts of the task. The kernel automatically activates a task's current XSR as detailed above, but the XSR is actually required to execute only when the task would normally be scheduled to run. The XSR must deactivate and return to the code which it interrupted with a special ORKID operation: EXCEPTION_RETURN. While it is active, an XSR has no special privileges or restrictions other than those necessitated by its asynchronous execution.

A XSR has its own mode with the same four mode parameters as tasks: NOXSR, NOTERMINATION, NOPREEMPT and NOINTERRUPT. The mode parameter given in the exception_catch operation is ored with the active mode at the time of the XSR's activation. The XSR will enter execution with this mode, which now becomes the active mode.

An active XSR can itself be interrupted by an exception being raised. In this case, unless the XSR's modal parameter NOXHR was set, the XSR is immediately reentered to handle the new exception. Theoretically, XSR activation can be thus nested to any depth. The kernel only considers the active mode when making scheduling decisions.