8.9. 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 allowed in queue
length	: integer	length of message buffers in bytes
options	: bit_field	queue 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 successful
ILLEGAL_USE	queue_info not callable from ISR
INVALID_PARAMETER	a parameter refers to an invalid address
INVALID_ID	queue does not exist
OBJECT_DELETED	originally existing queue has been deleted before operation
NODE_NOT_REACHABLE	node on which queue resides is not reachable

Description

This operation provides information on the specified message queue. It returns its maximum number of buffers, their length 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 snap-shot of the queue's state at the time of executing the operation.

9. EVENTS

Events provide a simple method of task synchronization. Each task has the same number of events which is equal to the number of bits in the basic word length of the corresponding processor. Events have no identifiers, but are referenced using a task identifier and a bit-field. The 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 latches corresponding to the asked for combination. This means that if an event is sent more than once before being received, the second and subsequent sends are lost.

9.1. 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 successful
INVALID_PARAMETER	a parameter refers to an invalid address
INVALID_ID	task does not exist
OBJECT_DELETED	originally existing task has been deleted before operation
NODE_NOT_REACHABLE	node on which task 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.

9.2. EVENT_RECEIVE

Receive event(s).

Synopsis

```
event_receive( event, options, time_out, event_received )
```

Input Parameters

event	: bit_field	event(s) to receive
options	: bit_field	receive options
time_out	: integer	ticks to wait before timing out

Output Parameters

event_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 event(s) set
time_out	= FOREVER	wait forever - do not time out

Completion Status

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

Description

This operation blocks a task until a given combination of events occurs. By default, the task waits until all of the events have been sent. If the ANY option is set, then the task waits only until at least 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 event_received, and the corresponding event latches are cleared. If the ANY option was set, and one or more of the specified events was sent, all the events sent, satisfying the event paramater, 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 time-out expires, the TIME_OUT completion status is returned.

10. EXCEPTIONS

ORKID exceptions provide tasks with a method of handling exceptional conditions asynchronously. Each task has the same number of exceptions which is equal to the number of bits in the basic word length of the corresponding processor. Exceptions have no identifiers, but are referenced using a task identifier and a bit-field. The bit-field can indicate any number of a task's exceptions at once.

Using this bit field, any number of exceptions can be raised simultaneously to a task. Each exception, defined by one bit of the bit-field, is associated with one specific Exception Service Routine (XSR). If a task has no XSR defined for any one of the raised exceptions, then the corresponding exception bits are lost and the XSR_NOT_SET completion status is returned for the exception_raise operation. Otherwise, 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 from the receiving of events, and involves the automatic activation by the scheduler of the task's XSRs corresponding to every set bit. XSRs have to be declared via the exception_catch operation by tasks after their creation. A task may change its XSRs at any time.

An XSR is activated whenever the corresponding exception is raised to a task, and the task has not set its NOXSR mode parameter in the active mode. If the NOXSR parameter was 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, the corresponding latch is cleared and the XSR entered.

XSR code is executed in exactly the same way as other parts of the task. While it is executing, an XSR has no special privileges or restrictions compared to normal task code. The kernel automatically activates an XSR as detailed above, but the XSR will actually run only when the task would normally be scheduled to run. The XSR must normally deactivate and return to the code it interrupted with a special **ORKID** operation: exception_return; alternatively it may alter the flow of execution through the task_restart operation.

Observation:

Raising an exception to a task will not unblock a waiting task.

An 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.

If several exception bits are set at the same time, the Exception Service Routine corresponding to the highest bit-number set will be