7.8. QUEUE_FLUSH

Flush all messages on a queue.

Synopsis

```
queue_flush( qid, count )
```

Input Parameters

qid	: queue_id	kernel defined queue identifier

Output Parameters

count	: integer	number of flushed messages

Completion Status

OK	queue_flush operation successful
ILLEGAL_USE	operation not callable from ISR
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 semaphore resides is not reachable

Description

If there were one or more messages in the specified queue, then they are removed from the queue, their buffers deallocated and their number returned in count. If there were no messages in the queue, then a count of zero is returned.

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.

9.1. EXCEPTION_CATCH

Specify a task's asynchronous exception handling routine.

Synopsis

```
exception_catch( new_XSR, mode, old_XSR, old_mode )
```

Input Parameters

new_XSR	: address	address of exception handling routine
mode	: bit_field	startup execution mode of XSR

Output Parameters

old_XSR	: address	address of previous XSR
old_mode	: bit-filed	mode associated with old XSR

Literal Values

new_XSR	= NULL_XSR	task henceforth will have no XSR
mode	+ NOXHR	XSR cannot be activated
	+ NOTERMINATION	task cannot be restarted or deleted
	+ NOPREEMPT	task cannot be preempted
	+ NOINTERRUPT	interrupt handling routine cannot be activated
old_XSR	= NULL_XSR	task previously had no XSR

Completion Status

OK	exceptions_catch operation successful
ILLEGAL_USE	operation not callable from ISR
INVALID_PARAMETER	a parameter refers to an illegal address
INVALID_ADDRESS	new_XSR refers to an illegal address
INVALID_MODE	invalid mode value

Description

This operation designates a new exception handling routine (XSR) for the current task. The task supplies the start address of the XSR, and the mode in which it will be started. If this operation returns a successful completion status, an exception sent to the task will henceforth cause the XSR at the given address to be activated.

The kernel returns the address of the previous XSR and the mode associated with that XSR.

Observation:

This can be used when a task wishes to use a different XSR temporarily. Once finished with the temporary XSR, the original one can be simply reinstated.

Note that if tasks are created without an XSR in a particular

implementation, the first call to exception_catch will return the symbolic value NULL_XSR in old_XSR. This same value can be passed as the new_XSR input parameter, which removes the current XSR from the task without designating a new one.

9.2. EXCEPTION_RAISE

Raise exceptions to a task.

Synopsis

```
exception_raise( tid, exceptions )
```

Input Parameters

tid	: task_id	kernel defined task id
exceptions	: bit_field	exceptions to be raised

Output Parameters

<none>

Completion Status

OK	exceptions_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
XSR_NOT_SET	task has no exception handler routine
NODE_NOT_REACHABLE	node on which semaphore resides is not reachable

Description

This operation raises one or more exceptions to a task. If the task in question has an XSR, then unless it has the NOXHR modal parameter set, the XSR will be activated immediately and run not later than the task would normally be scheduled. If NOXHR is set, the XSR will be activated as soon as the task clears this parameter.

If the task has no current XSR, then this operation returns the XSR_NOT_SET completion status.

9.3. EXCEPTION_RETURN

Return from Asynchronous Exception Handling Routine.

Synopsis

```
exception_return( )
```

Input Parameters

<none>

Output Parameters

<none>

Completion Status

<not applicable>

Description

This operation transfers control from an XSR back to the code which it interrupted. It has no parameters and does not produce a completion status. This operation must be used to deactivate an XSR.

The behavior of exception_return when not called from an XSR is undefined.

10. CLOCK

Each ORKID kernel maintains a node clock. This is a single data object in the kernel data structure which contains the current date and time. The clock is updated at every tick, the frequency of which is node dependent. The range of dates the clock is allowed to take is implementation dependent.

In a multi-node system, the different node clocks will very likely be synchronized, although this is not necessarily done automatically by the kernel. Since nodes could be in different time zones in widely distributed systems, the node clock specifies the local time zone, so that all nodes can synchronize their clocks to the same absolute time.

The data structure containing the clock value passed in clock operations is language binding dependent. It identifies the date and time down to the nearest tick, along with the local time zone. The time zone value is defined as the number of hours ahead (positive value) or behind (negative value) Greenwich Mean Time (GMT).

When the system starts up, the clock may be uninitialised. If this is the case, attempts at reading it before it has been set result in an error completion status, rather than returning a random value.

10.1. CLOCK_SET

Set node time and date.

Synopsis

```
clock_set( clock )
```

Input Parameters

clock	: clock_buf	current time and date

Output Parameters

<none>

Completion Status

OK	clock_set operation successful
ILLEGAL_USE	operation not callable from XSR or ISR
INVALID_PARAMETER	a parameter refers to an illegal address
INVALID_CLOCK	invalid clock value

Description

This operation sets the node clock to the specified value. The kernel checks the supplied date and time in clock_buf to ensure that they are legal. This is purely a syntactic check - the operation will accept any legal value. The exact structure of the data supplied is language binding dependent.

10.2. CLOCK_GET

Get node time and date.

Synopsis

```
clock_get( clock )
```

Input Parameters

<none>

Output Parameters

clock	: clock_buf	current time and date

Completion Status

OK	clock_get operation successful
INVALID_PARAMETER	a parameter refers to an illegal address
CLOCK_NOT_SET	clock has not been initialized

Description

This operation returns the current date and time in the node clock. If the node clock has not yet been set, then the CLOCK_NOT_SET completion status is returned. The exact structure of the clock_buf data returned is language binding dependent.

10.3. CLOCK_TICK

Announce a tick to the clock.

Synopsis

```
clock_tick(  )
```

Input Parameters

<none>

Output Parameters

<none>

Completion Status

OK	clock_tick operation successful

Description

This operation increments the current node time by one tick. There are no parameters and the operation always succeeds. Every node must contain a mechanism which keeps the node clock up to date by calling upon CLOCK_TICK.

11. TIMERS

ORKID defines two types of timers. The first type is the sleep timer. This type allows a task to sleep either for a given period, or up until a given time, and then wake and continue. Obviously a task can set only one such timer in operation at a time, and once set, it cannot be cancelled. These timers have no identifier.

The second type of timer is the event timer. This type allows a task to send events to itself either after a given period or at a given time. A task can have more than one event timer running at a time. Each event timer is assigned an identifier by the kernel when the event is set. This identifier can be used to cancel the timer.

Timers are purely local objects. They affect only the calling task, either by putting it to sleep or sending it events. Timers exist only while they are running. When they expire or are cancelled, they are deleted from the kernel data structure.

11.1. TIMER_WAKE_AFTER

Wake after a specified time interval.

Synopsis

```
timer_wake_after( ticks )
```

Input Parameters

ticks	: integer	number of ticks to wait

Output Parameters

<none>

Completion Status

OK	timer_wake_after operation successful
ILLEGAL_USE	operation not callable from XSR or ISR
INVALID_PARAMETER	a parameter refers to an illegal address

Description

This operation causes the calling task to be blocked for the given number of ticks. The task is woken after this interval has expired, and is returned a successful completion status. If the node clock is set using the clock_set operation during this interval, the number of ticks left does not change.

11.2. TIMER_WAKE_WHEN

Wake at a specified wall time.

Synopsis

```
timer_wake_when( clock )
```

Input Parameters

```
clock       : clock_buf    time and date to wake
```

Output Parameters

```
<none>
```

Completion Status

OK	timer_wake_when operation successful
ILLEGAL_USE	operation not callable from XSR or ISR
INVALID_PARAMETER	a parameter refers to an illegal address
INVALID_CLOCK	invalid clock value

Description

This operation causes the calling task to be blocked up until a given date and time. The task is woken at this time, and is returned a successful completion status. The kernel checks the supplied clock_buf data for validity. The exact structure of that data is language binding dependent.

If the node clock is set while the timer is running, the wall time at which the task is woken remains valid. If the node time is set to after the timer wake time, then the timer is deemed expired and the task is woken immediately and returned a successful completion status.

11.3. TIMER_EVENT_AFTER

Send event after a specified time interval.

Synopsis

```
timer_event_after( ticks, event, tmid )
```

Input Parameters

ticks	: integer	number of ticks to wait
event	: bit_field	event to send

Output Parameters

tmid	: timer_id	kernel defined timer identifier

Completion Status

OK	timer_event_after operation successful
INVALID_PARAMETER	a parameter refers to an illegal address
TOO_MANY_TIMERS	too many timers on the node

Description

This operation starts an event timer which will send the given events to the calling task after the specified number of ticks. The kernel returns an identifier which can be used to cancel the timer. If the node clock is set using the clock_set operation during this interval, the number of ticks left does not change.

11.4. TIMER_EVENT_WHEN

Send event at the specified wall time and date.

Synopsis

```
timer_event_when( clock, event, tmid )
```

Input Parameters

clock	: clock_buf	time and date to send event
event	: bit_field	event(s) to send

Output Parameters

tmid	: timer_id	kernel defined timer identifier

Completion Status

OK	timer_event_when operation successful
INVALID_PARAMETER	a parameter refers to an illegal address
INVALID_CLOCK	invalid clock value
TOO_MANY_TIMERS	too many timers on node

Description

This operation starts an event timer which will send the given events to the calling task at the given date and time. The kernel returns an identifier which can be used to cancel the timer.

If the node clock is set while the timer is running, the wall time at which the task is woken remains valid. If the node time is set to after the timer wake time, then the timer is deemed expired and the events are sent to the calling task immediately .

11.5. TIMER_CANCEL

Cancel a running event timer.

Synopsis

```
timer_cancel( tmid )
```

Input Parameters

tmid	: timer_id	kernel defined timer identifier

Output Parameters

<none>

Completion Status

OK	timer_cancel operation successful
INVALID_PARAMETER	a parameter refers to an illegal address
INVALID_ID	timer does not exist

Description

This operation cancels an event timer previously started using the timer_event_after or timer_event_when operations. The user specifies the timer using the identifier returned by these operations. If the given timer has expired or has been cancelled, the INVALID_ID completion status is returned.