buffer and a successful completion status returned.

If the message queue is empty, and NOWAIT was not specified in the options, then the task is blocked and put on the queue's wait queue. At that moment the time-out period is started. If the time-out expires then the TIME_OUT completion status is returned.

If NOWAIT was specified and the queue is empty, then the QUEUE_EMPTY completion status is returned.

If the queue is deleted while the task is waiting on a message from it, then the QUEUE_DELETED completion status is returned.

Otherwise, when the task reaches the head of the queue and a message is sent, or if a message is broadcast while the task is anywhere in the queue, then the task receives the message and is returned a successful completion status.

8.8. 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 successful
ILLEGAL_USE	queue_flush 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

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.

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 activated. After executing the exception_return operation in this XSR the routine corresponding to the bit with the second highest bit-number will be activated etc. An XSR running without the NOXSR bit in its mode will be interrupted by an exception of higher priority, i.e. with a higher bit-number. Exceptions of equal and lower priority will be latched.

The exception_return operation will return either to the interrupted task, reinstating its original mode, or to the interrupted XSR with its original mode. This is also true in case of explicit change of an XSR's mode via task_set_mode.

10.1. EXCEPTION_CATCH

Specify a task's Exception Service Routine for a given exception bit.

Synopsis

```
exception_catch( bit_number, new_xsr, new_mode, old_xsr, old_mode )
```

Input Parameters

```
bit_number : integer        exception bit-number
new_xsr    : address        address of XSR
new_mode   : bit_field      execution mode to be ored in
```

Output Parameters

```
old_xsr    : address        address of old XSR
old_mode   : bit_field      mode of old XSR
```

Literal Values

```
new_xsr    = NULL_XSR       task henceforth will have no XSR
                            for the given exception bit
new_mode   + NOXSR          XSRs cannot be activated
           + NOTERMINATION  task cannot be restarted or deleted
           + NOPREEMPT      task cannot be preempted
           + NOINTERRUPT    task cannot be interrupted
           = ZERO           no mode set

old_mode                    same as new_mode

old_xsr    = NULL_XSR       task previously had no XSR for the given
                            exception bit
```

Completion Status

```
OK                          exception_catch successful
ILLEGAL_USE                 exception_catch not callable from ISR
INVALID_PARAMETER           a parameter refers to an invalid address
INVALID_MODE                invalid mode value
INVALID_BIT                 invalid exception bit-number
```

Description

This operation designates a new Exception Service Routine (XSR) for the exception given by bit_number for the calling task. The task supplies the start address of the XSR, and the mode which will be ored to the active mode of the interrupted task or XSR to produce the active mode of this XSR. If this operation returns a successful completion status, the exception given by bit_number will henceforth cause the XSR at the given address to be activated, if the running task does not have the NOXSR mode set.

The kernel returns the address of the previous XSR and the mode of that

XSR for the specified exception.
Note that if a task has no XSR defined for the given exception a 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 for this exception without designating a new one.

Observation:

This operation can be used for defining the corresponding XSR for the first time and when a task wishes to use a different XSR temporarily. Once finished with the temporary XSR, the original one can be simply reinstated using the old_xsr and old_mode values.

10.2. EXCEPTION_RAISE

Raise exception(s) to a task.

Synopsis

```
exception_raise( tid, exception )
```

Input Parameters

tid	: task_id	kernel defined task id
exception	: bit_field	exception(s) to be raised

Output Parameters

<none>

Completion Status

OK	exception_raise 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
XSR_NOT_SET	no handler routine for given exception(s)
NODE_NOT_REACHABLE	node on which task resides is not reachable

Description

This operation raises one or more exceptions to a task. If the task in question has XSR(s) defined for the given exception(s), then unless it has the NOXSR mode value set, the highest priority XSR will be activated immediately and will run when the task would be normally scheduled. If NOXSR is set, this XSR will be activated as soon as the task clears this parameter.

If the task has no XSR(s) for the given exception(s), then this operation returns the XSR_NOT_SET completion status.

10.3. EXCEPTION_RETURN

Return from Exception Service 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.

11. 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.

11.1. CLOCK_SET

Set node time and date.

Synopsis

```
clock_set( clock )
```

Input Parameters

clock	: clock_buff	current time and date

Output Parameters

<none>

Completion Status

OK	clock_set successful
ILLEGAL_USE	clock_set not callable from ISR
INVALID_PARAMETER	a parameter refers to an invalid 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_buff 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.

11.2. CLOCK_GET

Get node time and date.

Synopsis

```
clock_get( clock )
```

Input Parameters

<none>

Output Parameters

clock	: clock_buff	current time and date

Completion Status

OK	clock_get successful
INVALID_PARAMETER	a parameter refers to an invalid 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 and the contents of clock are undetermined. The exact structure of the clock_buff data returned is language binding dependent.

11.3. CLOCK_TICK

Announce a tick to the clock.

Synopsis

clock_tick( )

Input Parameters

<none>

Output Parameters

<none>

Completion Status

OK	clock_tick successful

Description

This operation increments the current node time by one tick. There are no parameters and the operation always succeeds. Nevertheless, the operation can be meaningless if the clock was not initialized beforehand. Every node must contain a mechanism which keeps the node clock up to date by calling upon clock_tick.

12. 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.

12.1. 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 successful
ILLEGAL_USE	timer_wake_after not callable from ISR

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.

12.2. TIMER_WAKE_WHEN

Wake at a specified wall time and date.

Synopsis

```
timer_wake_when( clock )
```

Input Parameters

clock	: clock_buff	time and date to wake

Output Parameters

<none>

Completion Status

OK	timer_wake_when successful
ILLEGAL_USE	timer_wake_when not callable from ISR
INVALID_PARAMETER	a parameter refers to an invalid address
INVALID_CLOCK	invalid clock value
CLOCK_NOT_SET	clock has not been initialized

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.

12.3. 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 successful
ILLEGAL_USE	timer_event_after not callable from ISR
INVALID_PARAMETER	a parameter refers to an invalid address
TOO_MANY_OBJECTS	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.

12.4. TIMER_EVENT_WHEN

Send event at the specified wall time and date.

Synopsis

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

Input Parameters

clock	: clock_buff	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 successful
ILLEGAL_USE	timer_event_when not callable from ISR
INVALID_PARAMETER	a parameter refers to an invalid address
INVALID_CLOCK	invalid clock value
TOO_MANY_OBJECTS	too many timers on the node
CLOCK_NOT_SET	clock has not been initialized

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 envent(s) are sent remains valid. If the node time is set to after the value specified in the clock parameter, then the timer is deemed expired and the events are sent to the calling task immediately.

12.5. TIMER_EVENT_EVERY

Send periodic event.

Synopsis

```
timer_event_every( ticks, event, tmid )
```

Input Parameters

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

Output Parameters

tmid	: timer_id	kernel defined timer identifier

Completion Status

OK	timer_event_every successful
ILLEGAL_USE	timer_event_every not callable from ISR
INVALID_PARAMETER	a parameter refers to an invalid address
TOO_MANY_OBJECTS	too many timers on the node

Description

This operation starts an event timer which will periodically send the given events to the calling task with the periodicity specified by the 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 the life time of the timer, the number of ticks left until the next event does not change.

Observation:

This provides a drift-free mechanism for sending an event at periodic intervals.

12.6. 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 successful
ILLEGAL_USE	timer_cancel not callable from ISR
INVALID_PARAMETER	a parameter refers to an invalid address
INVALID_ID	timer does not exist
OBJECT_DELETED	originally existing timer expired or has been canceled before operation

Description

This operation cancels an event timer previously started using the timer_event_after, timer_event_when or timer_event_every operations.

13. INTERRUPTS

ORKID defines two operations which bracket interrupt service code. It is up to each implementor to decide what functionality to put in these operations.

Observation:

The kernel may use int_enter and int_return to distinguish if Interrupt Service Routine code or task code is being executed. Typically int_return will be useed to decide if a scheduling action must take place in kernels with preemptive scheduling.