3.4. TASK_START

Start a task.

Synopsis

```
task_start( tid, start_addr, arguments )
```

Input Parameters

tid	: task_id	kernel defined task identifier
start_addr	: *	task start address
arguments	: *	arguments passed to task

Output Parameters

<none>

Completion Status

OK	task_start operation successful
ILLEGAL_USE	operation not callable from XSR or ISR
INVALID_PARAMETER	a parameter refers to an illegal address
INVALID_ID	task does not exist
OBJECT_DELETED	task specified has been deleted
INVALID_ADDRESS	invalid start address
INVALID_ARGUMENTS	invalid number or type or size of arguments
TASK_ALREADY_STARTED	task has been started already
OBJECT_PROTECTED	task has NOTERMINATION parameter set
NODE_NOT_REACHABLE	node on which task resides is not reachable

Description

The task_start operation starts a task at the given address. The task must have been previously created with the task_create operation. The task is started with the priority and mode specified when the task was created.

* The specification of start address and the number and type of arguments are language binding dependent. For a high level language, the start address will likely be the name of a procedure and the arguments would be passed to the procedure as parameters.

3.5. TASK_RESTART

Restart a task.

Synopsis

```
task_restart( tid, arguments )
```

Input Parameters

tid	: task_id	kernel defined identifier
arguments	: *	arguments passed to task

Output Parameters

<none>

Literal Values

tid	= SELF	The calling task restarts itself

Completion Status

OK	task_restart operation successful
ILLEGAL_USE	operation not callable from ISR
INVALID_PARAMETER	a parameter refers to an illegal address
INVALID_ID	task does not exist
OBJECT_DELETED	task specified has been deleted
INVALID_ARGUMENTS	invalid number or type or size of arguments
TASK_NOT_STARTED	task has not yet been started
OBJECT_PROTECTED	task has NOTERMINATION parameter set
NODE_NOT_REACHABLE	node on which task resides is not reachable

Description

The task_restart operation interrupts the current thread of execution of the specified task and forces the task to restart at the address given in the task_start call which originally started the task. The stack pointer is reset to its original value. No assumption can be made about the original content of the stack at this time.

Any resources allocated to the task are not affected during the task_restart operation. The tasks themselves are responsible for the proper management of such resources through task_restart.

If the task's active mode has the parameter NOTERMINATION set, then the task will not be restarted and the completion status OBJECT_PROTECTED will be returned.

* The specification of the number and type of the arguments is language binding dependent. For a high level language, it is likely that these arguments will be passed as parameters to the procedure whose name was given as start address in the original task_start call.

3.6. TASK_SUSPEND

Suspend a task.

Synopsis

```
task_suspend( tid )
```

Input Parameters

tid	: task_id	kernel defined task identifier

Output Parameters

<none>

Literal Values

tid	= SELF	The calling task suspends itself

Completion Status

OK	task_suspend operation successful
INVALID_PARAMETER	a parameter refers to an illegal address
INVALID_ID	task does not exist
OBJECT_DELETED	task specified has been deleted
OBJECT_PROTECTED	task has NOPREEMPT parameter set
TASK_ALREADY_SUSPENDED	task already suspended
NODE_NOT_REACHABLE	node on which task resides is not reachable

Description

This operation temporarily suspends the specified task until the suspension is lifted by a call to task_resume. While it is suspended, a task cannot be scheduled to run.

If the task's active mode has the parameter NOPREEMPT set the operation will fail and return the completions status OBJECT_PROTECTED, unless the task suspends itself. In which case the operation will always be successful.

3.7. TASK_RESUME

Resume a suspended task.

Synopsis

```
task_resume( tid )
```

Input Parameters

tid	: task_id	kernel defined task identifier

Output Parameters

<none>

Completion Status

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

Description

The task_resume operation lifts the task's suspension immediately after the point at which it was suspended. The task must have been suspended with a call to the task_suspend operation.

3.8. TASK_SET_PRIORITY

Set priority of a task.

Synopsis

```
task_set_priority( tid, new_prio, old_prio )
```

Input Parameters

tid	: task_id	kernel defined task id
new_prio	: prio	task's new priority

Output Parameters

old_prio	: prio	task's previous priority

Literal Values

tid	= SELF	The calling task sets its own priority
new_prio	= CURRENT	There will be no change in priority

Completion Status

OK	task_set_priority operation successful
ILLEGAL_USE	operation not callable from ISR
INVALID_PARAMETER	a parameter refers to an illegal address
INVALID_ID	task does not exist
OBJECT_DELETED	task specified has been deleted
INVALID_PRIORITY	invalid priority value
NODE_NOT_REACHABLE	node on which task resides is not reachable

Description

This operation sets the priority of the specified task to new_prio. The new_prio parameter is specified as CURRENT if the calling task merely wishes to find out the current value of the specified task's priority. (see also 3. Task Priority)

3.9. TASK_SET_MODE

Set mode of own task.

Synopsis

```
task_set_mode( new_mode, mask, old_mode )
```

Input Parameters

new_mode	: bit_field	new task mode settings
mask	: bit_field	significant bits in mode

Output Parameters

old_mode	: bit_field	task's previous mode

Literal Values

new_mode	+ NOXSR	XSRs cannot be activated
	+ NOTERMINATION	task cannot be restarted or deleted
	+ NOPREEMPT	task cannot be preempted
	+ NOINTERRUPT	interrupt handling routine cannot be activated
old_mode	+ NOXSR	XSRs cannot be activated
	+ NOTERMINATION	task cannot be restarted or deleted
	+ NOPREEMPT	task cannot be preempted
	+ NOINTERRUPT	interrupt handling routine cannot be activated
mask		(same as mode)

Completion Status

OK	task_set_mode operation successful
ILLEGAL_USE	operation not callable from ISR
INVALID_PARAMETER	a parameter refers to an illegal address
INVALID_MODE	invalid mode or mask value

Description

This operation sets a new active mode for the task or its XSR. If called from a task's XSR then the XSR mode is changed, otherwise the main task's mode is changed.

The mode parameters which are to be changed are given in mask. If a parameter is to be set then it is also given in mode, otherwise it is left out. For both mask and mode, the logical OR (!) of the symbolic values for the mode parameters are passed to the operation.

For example, to clear NOINTERRUPT and set NOPREEMPT, mask = NOINTERRUPT ! NOPREEMPT, and mode = NOPREEMPT. To return the current mode without altering it, the mask should simply be set to zero. (see also 3. Task Modes)

3.10. TASK_READ_NOTE_PAD

Read one of a task's note-pad locations.

Synopsis

```
task_read_note_pad( tid, loc_number, loc_value )
```

Input Parameters

tid	: task_id	kernel defined task id
loc_number	: lnum	note-pad location number

Output Parameters

loc_value	: integer	note-pad location value

Literal Values

tid	= SELF	The calling task reads its own notepad

Completion Status

OK	task_read_note_pad operation successful
INVALID_PARAMETER	a parameter refers to an illegal address
INVALID_ID	task does not exist
INVALID_LOCATION	note-pad number does not exist
NODE_NOT_REACHABLE	node on which task resides is not reachable

Description

This operation returns the value contained in the specified notepad location of the task identified by tid. (see also 3. Task Notepads)

3.11. TASK_WRITE_NOTE_PAD

Write one of a task's note-pad locations.

Synopsis

```
task_write_note_pad( tid, loc_number, loc_value )
```

Input Parameters

tid	: task_id	kernel defined task id
loc_number	: lnum	note-pad location number
loc_value	: integer	note-pad location value

Output Parameters

<none>

Literal Values

tid	= SELF	The calling task writes into its own notepad

Completion Status

OK	task_write_note_pad operation successful
INVALID_PARAMETER	a parameter refers to an illegal address
INVALID_ID	task does not exist
OBJECT_DELETED	task specified has been deleted
INVALID_LOCATION	note-pad number does not exist
NODE_NOT_REACHABLE	node on which task resides is not reachable

Description

This operation writes the specified value into the specified notepad location of the task identified by tid. (see also 3. Task Notepads)

4. REGIONS

A region is an area of memory within a node which is organized by an **ORKID** compliant kernel into a pool of segments of varying size. The area of memory to become a region is declared to the kernel by a task when the region is created, and is thereafter managed by the kernel until it is explicitly deleted by a task.

Each region has a granularity, defined when the region is created. The actual size of segments allocated is always a multiple of the granularity, although the required segment size is given in bytes.

Once a region has been created, a task is free to claim variable sized segments from it and return them in any order. The kernel will do its best to satisfy all requests for segments, although fragmentation may cause a segment request to be unsuccessful, despite there being more than enough total memory remaining in the region. The memory management algorithms used are implementation dependent.

Regions, as opposed to partitions, tasks, etc., are only locally accessible. In other words, regions cannot be declared global and a task cannot access a region on another node. This does not stop a task from using the memory in a region on another node, for example in an area of memory shared between the nodes, but all claiming of segments must be done by a co-operating task in the appropriate node and the address passed back.

Observation:

Regions are intended to provide the first subdivisions of the physical memory available to a node. These subdivisions may reflect differing physical nature of the memory, giving for example a region of RAM, a region of ROM, a region of shared memory, etc.. Regions may also subdivide memory into areas for different uses, for example a region for kernel use and a region for user task use.

4.1. REGION_CREATE

Create a region.

Synopsis

```
region_create( name, addr, length, granularity, options, rid )
```

Input Parameters

name	: string	user defined region name
addr	: address	start address of the region
length	: integer	length of region in bytes
granularity	: integer	allocation granularity in bytes
options	: bit_field	region create options

Output Parameters

rid	: region_id	kernel defined region identifier

Completion Status

OK	region_create operation successful
ILLEGAL_USE	operation not callable from XSR or ISR
INVALID_PARAMETER	a parameter refers to an illegal address
INVALID_ADDRESS	area given not within actual memory present
INVALID_GRANULARITY	granularity not supported
INVALID_OPTIONS	invalid options value
TOO_MANY_REGIONS	too many regions on the node
REGION_OVERLAP	area given overlaps an existing region

Description

This operation declares an area of memory to be organized as a region by the kernel. The process of formatting the memory to operate as a region may require a memory overhead which may be taken from the new region itself. It can never be assumed that all of the memory in the region will be available for allocation. The overhead percentage will be implementation dependent.

Observation:

Currently ORKID defines no options, the parameter is there as a place holder for future extensions and implementations desiring to provide additional options.

4.2. REGION_DELETE

Delete a region.

Synopsis

```
region_delete( rid, options )
```

Input Parameters

rid	: region_id	kernel defined region identifier
options	: bit_field	region deletion options

Output Parameters

<none>

Literal Values

options	+ FORCED_DELETE	deletion will go ahead even though there are unreleased segments

Completion Status

OK	region_delete operation successful
ILLEGAL_USE	operation not callable from ISR
INVALID_PARAMETER	a parameter refers to an illegal address
INVALID_ID	region does not exist
OBJECT_DELETED	region specified has been deleted
INVALID_OPTIONS	invalid options value
REGION_IN_USE	segments from this region are still allocated

Description

Unless the FORCED_DELETE option was specified, this operation first checks whether the region has any segments which have not been returned. If this is the case, then the REGION_IN_USE completion status is returned. If not, and in any case if FORCED_DELETE was specified, then the region is deleted from the kernel data structure.

4.3. REGION_IDENT

Obtain the identifier of a region with a given name.

Synopsis

```
region_ident( name, rid )
```

Input Parameters

name	: string	user defined region name

Output Parameters

rid	: region_id	kernel defined region identifier

Completion Status

OK	region_ident operation successful
ILLEGAL_USE	operation not callable from XSR or ISR
INVALID_PARAMETER	a parameter refers to an illegal address
NAME_NOT_FOUND	name does not exist on node

Description

This operation searches the kernel data structure in the local node for a region with the given name, and returns its identifier if found. If there is more than one region with the same name, the kernel will return the identifier of one of them, the choice being implementation dependent.

4.4. REGION_GET_SEG

Get a segment from a region.

Synopsis

```
region_get_seg( rid, seg_size, seg_addr )
```

Input Parameters

rid	: region_id	kernel defined region id
seg_size	: integer	requested segment size in bytes

Output Parameters

seg_addr	: address	address of obtained segment

Completion Status

OK	region_get_seg operation successful
ILLEGAL_USE	operation not callable from ISR
INVALID_PARAMETER	a parameter refers to an illegal address
INVALID_ID	region does not exist
OBJECT_DELETED	region specified has been deleted
NO_MORE_MEMORY	not enough contiguous memory in the region to allocate segment of requested size

Description

The region_get_seg operation is a request for a given sized segment from a given region's free memory pool. If the kernel cannot fulfil the request immediately, it returns the error completion status NO_MORE_MEMORY, otherwise the address of the allocated segment is returned. The allocation algorithm is implementation dependent.

Note that the actual size of the segment returned will be more than the size requested, if the latter is not a multiple of the region's granularity.

4.5. REGION_RET_SEG

Return a segment to its region.

Synopsis

```
region_ret_seg( rid, seg_addr )
```

Input Parameters

rid	: region_id	kernel defined region id
seg_addr	: address	address of segment to be returned

Output Parameters

<none>

Completion Status

OK	region_ret_seg operation successful
ILLEGAL_USE	operation not callable from ISR
INVALID_PARAMETER	a parameter refers to an illegal address
INVALID_ID	region does not exist
OBJECT_DELETED	region specified has been deleted
INVALID_SEGMENT	no segment allocated from this region at seg_addr

Description

This operation returns the given segment to the given region's free memory pool. The kernel checks that this segment was previously allocated from this region, and returns INVALID_SEGMENT if it wasn't.

4.6. REGION_INFO

Obtain information on a region.

Synopsis

```
region_info( rid, size, max_segment, granularity )
```

Input Parameters

rid	: region_id	kernel defined region id

Output Parameters

size	: integer	length in bytes of overall area in region available for segment allocation
max_segment	: integer	length in bytes of maximum segment allocatable at time of call
granularity	: integer	allocation granularity in bytes

Completion Status

OK	region_info operation successful
ILLEGAL_USE	operation not callable from ISR
INVALID_PARAMETER	a parameter refers to an illegal address
INVALID_ID	region does not exist
OBJECT_DELETED	region specified has been deleted

Description

This operation provides information on the specified region. It returns the size of the region's area for segment allocation, which may be smaller than the region length given in region_create due to a possible formatting overhead. It returns also the size of the biggest segment allocatable from the region. This value should be used with care as it is just a snap-shot of the region's usage at the time of executing the operation. Finally it returns the region's allocatable granularity.

5. PARTITIONS

Partitions are areas of memory organized by the kernel as a pool of fixed size blocks. As for regions, the creating task supplies the area of memory to be used by the partition. The task also supplies the size of the blocks to be allocated from the partition. Any restrictions imposed on the block size are implementation dependent.

Partitions are simpler structures than regions, and are intended for use where speed of allocation is essential. Partitions may also be declared global, and be operated on from more than one node. However, this makes sense only if the nodes accessing the partition are all in the same shared memory system, and the partition is in shared memory.

Once the partition created, tasks may request blocks one at a time from it, and can return them in any order. Because the blocks are all the same size, there is no fragmentation problem in partitions. The exact allocation algorithms are implementation dependent.

5.1. PARTITION_CREATE

Create a partition.

Synopsis

```
partition_create( name, addr, length, block_size, options, pid )
```

Input Parameters

name	: string	user defined partition name
addr	: address	start address of partition
length	: integer	length of partition in bytes
block_size	: integer	partition block size in bytes
options	: bit_field	partition create options

Output Parameters

pid	: part_id	kernel defined partition identifier

Literal Values

option:	+GLOBAL	partition is global within the shared memory system

Completion Status

OK	partition_create operation successful
ILLEGAL_USE	operation not callable from XSR or ISR
INVALID_PARAMETER	a parameter refers to an illegal address
INVALID_ADDRESS	area defined is not within actual memory present
INVALID_BLOCK_SIZE	block_size not supported
INVALID_OPTIONS	invalid options value
TOO_MANY_PARTITIONS	too many partitions on the node
PARTITION_OVERLAP	area given overlaps an existing partition

Description

This operation declares an area of memory to be organized as a partition by the kernel. The process of formatting the memory to operate as a partition may require a memory overhead which may be taken from the new partition. It can never be assumed that all of the memory in the partition will be available for allocation. The overhead percentage will be implementation dependent.

5.2. PARTITION_DELETE

Delete a partition.

Synopsis

```
partition_delete( pid, options )
```

Input Parameters

pid	: part_id	kernel defined partition identifier
options	: bit_field	partition deletion options

Output Parameters

<none>

Literal Values

options	+ FORCED_DELETE	deletion will go ahead even though there are unreleased blocks

Completion Status

OK	partition_delete operation successful
ILLEGAL_USE	operation not callable from ISR
INVALID_PARAMETER	a parameter refers to an illegal address
INVALID_ID	partition does not exist
OBJECT_DELETED	partition specified has been deleted
INVALID OPTIONS	invalid options value
PARTITION_IN_USE	blocks from this partition are still allocated
NODE_NOT_REACHABLE	node on which task resides is not reachable

Description

Unless the FORCED_DELETE option was specified, this operation first checks whether the partition has any blocks which have not been returned. If this is the case, then the PARTITION_IN_USE completion status is returned. If not, and in any case if FORCED_DELETE was specified, then the partition is deleted from the kernel data structure

5.3. PARTITION_IDENT

Obtain the identifier of a partition on a given node with a given name.

Synopsis

```
partition_ident( name, nid, pid, )
```

Input Parameters

name	: string	user defined partition name
nid	: node_id	node identifier

Output Parameters

pid	: part_id	kernel defined partition identifier
block_size	: integer	the partition's block size

Literal Values

nid	= LOCAL_NODE	the node containing the calling task
	= OTHER_NODES	all nodes in the system except the local node

Completion Status

OK	partition_ident operation successful
ILLEGAL_USE	operation not callable from XSR or ISR
INVALID_PARAMETER	a parameter refers to an illegal address
INVALID_NODE	node does not exist
NAME_NOT_FOUND	name does not exist on node
NODE_NOT_REACHABLE	node on which task resides is not reachable

Description

This operation searches the kernel data structure in the node(s) specified for a partition with the given name, and returns its identifier and block size if found. If OTHER_NODES is specified, the node search order is implementation dependent, but will include only those nodes in the shared memory system or subsystem containing the partition. If there is more than one partition with the same name, then the pid of the first one found is returned.

5.4. PARTITION_GET_BLK

Get a block from a partition.

Synopsis

```
partition_get_blk( pid, blk_addr )
```

Input Parameters

pid	: part_id	kernel defined partition identifier

Output Parameters

blk_addr	: address	address of obtained block

Completion Status

OK	partition_get_blk operation successful
ILLEGAL_USE	operation not callable from ISR
INVALID_PARAMETER	a parameter refers to an illegal address
INVALID_ID	partition does not exist
OBJECT_DELETED	partition specified has been deleted
NO_MORE_MEMORY	no more blocks available in partition
NODE_NOT_REACHABLE	node on which task resides is not reachable

Description

This operation is a request for a single block from the partition's free block pool. If the kernel cannot immediately fulfil the request, it returns the error completion status NO_MORE_MEMORY, otherwise the address of the allocated block is returned. The exact allocation algorithm is implementation dependent.