4.11. 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	: integer	note-pad location number
loc_value	: word	note-pad location value

Output Parameters

<none>

Literal Values

tid	= SELF	the calling task writes into its own note-pad.

Completion Status

OK	task_write_note_pad 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
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 note-pad location of the task identified by tid (see also 4. Task Note-Pads). ORKID compliant kernels have a minimum of 16 note-pad locations, indexed via loc_number starting at one.

4.12 TASK_INFO

Obtain information on a task.

Synopsis

```
task_info( tid, priority, mode, options, event, exception, state )
```

Input Parameters

tid	:	task_id	kernel defined task id

Output Parameters

priority	:	integer	task priority
mode	:	bit_field	task mode
options	:	bit_field	task options
event	:	bit_field	event(s) latched for task
exception	:	bit_field	exception(s) latched for task
state	:	integer	task's execution state

Literal Values

tid	=	SELF	the calling task requests information on itself
state	=	RUNNING	task is executing
		READY	task is ready for execution
		BLOCKED	task is blocked
		SUSPENDED	task is suspended

Completion Status

OK	task_info successful
ILLEGAL_USE	task_info not callable from ISR
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 provides information on the specified task. It returns the task's priority, mode, options, event and exception latches and the execution state. The latched bits in the task's event and exception bit_fields are returned without interfering with the state of these latches. The task execution state indicates the state from the scheduler's point of view. If the task is blocked and subsequently suspended the SUSPENDED state will be passed back. All return values except options reflect the dynamic state of a task and should be used with care as they are just snapshots of this state at the time of executing the operation.
The operation, when called from an Exception Service Routine (XSR), returns this XSR's mode.

5. REGIONS

A region is an area of memory within a node which is organized by the kernel into a collection 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 pools, 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. This address has to be explicitly translated by the sender via int_to_ext and by the receiver via ext_to_int.

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 battery backed-up SRAM, 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.

5.1. 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

Literal Values

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

Completion Status

OK	region_create successful
ILLEGAL_USE	region_create not callable from ISR
INVALID_PARAMETER	a parameter refers to an invalid address
INVALID_GRANULARITY	granularity not supported
INVALID_OPTIONS	invalid options value
TOO_MANY_OBJECTS	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.

The FORCED_DELETE option governs the deletion possibility of the region. (see 5.2. region_delete)

5.2. REGION_DELETE

Delete a region.

Synopsis

```
region_delete( rid )
```

Input Parameters

rid	: region_id	kernel defined region identifier

Output Parameters

<none>

Literal Values

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

Completion Status

OK	region_delete successful
ILLEGAL_USE	region_delete not callable from ISR
INVALID_PARAMETER	a parameter refers to an invalid address
INVALID_ID	region does not exist
OBJECT_DELETED	originally existing region has been deleted before operation
REGION_IN_USE	segments from this region are still allocated

Description

Unless the FORCED_DELETE option was specified at creation, 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.