5.3. 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 successful
ILLEGAL_USE	region_ident not callable from ISR
INVALID_PARAMETER	a parameter refers to an invalid address
NAME_NOT_FOUND	region 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 the first one found.

5.4. 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 successful
ILLEGAL_USE	region_get_seg 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
NO_MORE_MEMORY	not enough contiguous memory in the region to allocate segment of requested size

Description

The region_get_seg operation requests a given sized segment from a given region's free memory. If the kernel cannot fulfil the request immediately, it returns the completion status NO_MORE_MEMORY, otherwise the address of the allocated segment is passed back in seg_addr. 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.

5.5. 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 successful
ILLEGAL_USE	region_ret_seg 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
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. The kernel checks that this segment was previously allocated from this region, and returns INVALID_SEGMENT if it wasn't.

5.6. REGION_INFO

Obtain information on a region.

Synopsis

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

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
options	: bit_field	region create options

Completion Status

OK	region_info successful
ILLEGAL_USE	region_info 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

Description

This operation provides information on the specified region. It returns the size in bytes 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 in bytes 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 allocation granularity and options.

6. POOLS

A pool is an area of memory within a shared memory subsystem which is organized by the kernel into a collection of fixed size buffers. The area of memory to become a pool is declared to the kernel by a task when the pool is created, and is thereafter managed by the kernel until it is explicitly deleted by a task. The task also specifies the size of the buffers to be allocated from the pool. Any restrictions imposed on the buffer size are implementation dependent.

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

Once the pool has been created, tasks may request buffers one at a time from it, and can return them in any order. Because the buffers are all the same size, there is no fragmentation problem in pools. The exact allocation algorithms are implementation dependent. Addresses of buffers obtained via pool_get_buff are translated to the callers address map for immediate use.

Observation:

Buffer addresses passed from one node to another in e.g. a message have to be explicitly translated by the sender via int_to_ext and by the receiver via ext_to_int.