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.

6.1. POOL_CREATE

Create a pool.

Synopsis

```
   pool_create( name, addr, length, buff_size, options, pid )
```

Input Parameters

name	:	string	user defined pool name
addr	:	address	start address of pool
length	:	integer	length of pool in bytes
buff_size	:	integer	pool buffer size in bytes
options	:	bit_field	pool create options

Output Parameters

pid	:	pool_id	kernel defined pool identifier

Literal Values

options	+ GLOBAL	pool is global within the shared memory subsystem
	+ FORCED_DELETE	deletion will go ahead even if there are unrealeased buffers

Completion Status

OK	pool_create successful
ILLEGAL_USE	pool_create not callable from ISR
INVALID_PARAMETER	a parameter refers to an invalid address
INVALID_BUFF_SIZE	buff_size not supported
INVALID_OPTIONS	invalid options value
TOO_MANY_OBJECTS	too many pools on the node or in the system
POOL_OVERLAP	area given overlaps an existing pool

Description

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

The FORCED_DELETE option governs the deletion possibility of the pool (see 6.2 pool_delete).

6.2. POOL_DELETE

Delete a pool.

Synopsis

```
pool_delete( pid )
```

Input Parameters

pid	: pool_id	kernel defined pool identifier

Output Parameters

<none>

Completion Status

OK	pool_delete successful
ILLEGAL_USE	pool_delete not callable from ISR
INVALID_PARAMETER	a parameter refers to an invalid address
INVALID_ID	pool does not exist
OBJECT_DELETED	originally existing pool has been deleted before operation
POOL_IN_USE	buffers from this pool are still allocated
OBJECT_NOT_LOCAL	pool_delete not allowed on non-local pools

Description

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

6.3. POOL_IDENT

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

Synopsis

```
   pool_ident( name, nid, pid)
```

Input Parameters

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

Output Parameters

pid	: pool_id	kernel defined pool identifier

Literal Values

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

Completion Status

OK	pool_ident successful
ILLEGAL_USE	pool_ident not callable from ISR
INVALID_PARAMETER	a parameter refers to an invalid address
INVALID_ID	node does not exist
NAME_NOT_FOUND	pool does not exist on node
NODE_NOT_REACHABLE	node is not reachable

Description

This operation searches the kernel data structure in the node(s) specified for a pool with the given name, and returns its identifier if found. If OTHER_NODES or ALL_NODES is specified, the node search order is implementation dependent. If there is more than one pool with the same name, then the pid of the first one found is passed back.

Observation:

This operation may return the pid of a GLOBAL pool that is not in the same shared memory subsystem as the node containing the calling task.

6.4. POOL_GET_BUFF

Get a buffer from a pool.

Synopsis

```
   pool_get_buff( pid, buff_addr )
```

Input Parameters

pid	: pool_id	kernel defined pool identifier

Output Parameters

buff_addr	: address	address of obtained buffer

Completion Status

OK	pool_get_buff successful
ILLEGAL_USE	pool_get_buff not callable from ISR
INVALID_PARAMETER	a parameter refers to an invalid address
INVALID_ID	pool does not exist
OBJECT_DELETED	originally existing task has been deleted before operation
NO_MORE_MEMORY	no more buffers available in pool
POOL_NOT_SHARED	pool not in shared memory subsystem
NODE_NOT_REACHABLE	node on which pool resides is not reachable

Description

The pool_get_buff requests for a single buffer from the pool's free memory. If the kernel cannot immediately fulfil the request, it returns the completion status NO_MORE_MEMORY, otherwise the address of the allocated buffer is returned. The exact allocation algorithm is implementation dependent.

6.5. POOL_RET_BUFF

Return a buffer to its pool.

Synopsis

```
pool_ret_buff( pid, buff_addr )
```

Input Parameters

pid	: pool_id	kernel defined pool identifier
buff_addr	: address	address of buffer to be returned

Output Parameters

<none>

Completion Status

OK	pool_ret_buff successful
ILLEGAL_USE	pool_ret_buff not callable from ISR
INVALID_PARAMETER	a parameter refers to an invalid address
INVALID_ID	pool does not exist
OBJECT_DELETED	originally existing pool has been deleted before operation
POOL_NOT_SHARED	pool not in shared memory sybsystem
INVALID_BUFF	no buffer allocated from pool at buff_addr
NODE_NOT_REACHABLE	node on which pool resides is not reachable

Description

This operation returns the given buffer to the given pool's free space. The kernel checks that the buffer was previously allocated from the pool and returns INVALID_BUFF if it wasn't.

6.6. POOL_INFO

Obtain information on a pool.

Synopsis

```
   pool_info( pid, buffers, free_buffers, buff_size, options )
```

Input Parameters

pid	: pool-id	kernel defined pool identifier

Output Parameters

buffers	: integer	number of buffers in the pool
free_buffers	: integer	number of free buffers in the pool
buff_size	: integer	pool buffer size in bytes
options	: bit_field	pool create options

Completion Status

OK	pool_info successful
ILLEGAL_USE	pool_info not callable from ISR
INVALID_PARAMETER	a parameter refers to an invalid address
INVALID_ID	pool does not exist
OBJECT_DELETED	originally existing pool has been deleted before operation
NODE_NOT_REACHABLE	node on which the pool resides is not reachable

Description

This operation provides information on the specified pool. It returns its overall number of buffers, the number of free buffers in the pool, its buffer size in bytes and options. The number of free buffers in the pool should be used with care as it is just a snap-shot of the pools's usage at the time of executing the operation.

7. SEMAPHORES

The semaphores defined in **ORKID** are standard Dijkstra counting semaphores. Semaphores provide for the fundamental need of synchronization in multi-tasking systems, i.e. mutual exclusion, resource management and sequencing.

Semaphore Behavior

The following should not be understood as a recipe for implementations.

During a sem_claim operation, the semaphore count is decremented by one. If the resulting semaphore count is greater than or equal to zero, then the calling task continues to execute. If the count is less than zero, the task blocks from processor usage and is put on a waiting queue for the semaphore. During a sem_release operation, the semaphore count is incremented by one. If the resulting semaphore count is less than or equal to zero, then the first task in the waiting queue for this semaphore is unblocked and is made eligible for processor usage.

Semaphore Usage

Mutual exclusion is achieved by creating a counting semaphore with an initial count of one. A resource is guarded with this semaphore by requiring all operations on the resource to be proceeded by a sem_claim operation. Thus, if one task has claimed a resource, all other tasks requiring the resource will be blocked until the task releases the resource with a sem_release operation.

In situations where multiple copies of a resource exist, the semaphore may be created with an initial count equal to a number of copies. A resource is claimed with the sem_claim operation. When all available copies of the resource have been claimed, a task requiring the resource will be blocked until return of one of the claimed copies is announced by a sem_release operation.

Sequencing is achieved by creating a semaphore with an initial count of zero. A task may pend the arrival of another task by performing a sem_claim operation when it reaches a synchronization point. The other task performs a sem_release operation when it reaches its synchronization point, unblocking the pending task.

Semaphore Options

ORKID defines the following option symbols, which may be combined.

+ GLOBAL — Semaphores created with the GLOBAL option set are visible and accessible from any node in the system.

+ FIFO — Semaphores with the FIFO option set enter additional tasks at the end of their waiting queue. Without this option, the tasks are enqueued in order of task priority. **ORKID** does not require reordering of semaphore waiting queues when a waiting task has his priority changed.

7.1. SEM_CREATE

Create a semaphore.

Synopsis

```
sem_create( name, init_count, options, sid )
```

Input Parameters

name	:	string	user defined semaphore name
init_count	:	integer	initial semaphore count
options	:	bit_field	semaphore create options

Output Parameters

sid	:	sem_id	kernel defined semaphore identifier

Literal Values

options	+ GLOBAL	the new semaphore will be visible throughout the system
	+ FIFO	tasks will be queued in first in first out order

Completion Status

OK	sem_create successful
ILLEGAL_USE	sem_create not callable from ISR
INVALID_PARAMETER	a parameter refers to an invalid address
INVALID_COUNT	initial count is negative
INVALID_OPTIONS	invalid options value
TOO_MANY_OBJECTS	too many semaphores on the node or in the system

Description

This operation creates a new semaphore in the kernel data structure, and returns its identifier. The semaphore is created with its count at the value given by the init_count parameter. The task queue, initially empty, will be ordered by task priority, unless the FIFO option is set, in which case it will be first in first out.

7.2. SEM_DELETE

Delete a semaphore.

Synopsis

```
   sem_delete( sid )
```

Input Parameters

sid	: sem_id	kernel defined semaphore identifier

Output Parameters

<none>

Completion Status

OK	sem_delete successful
ILLEGAL_USE	sem_delete not callable from ISR
INVALID_PARAMETER	a parameter refers to an invalid address
INVALID_ID	semaphore does not exist
OBJECT_DELETED	originally existing semaphore has been deleted before operation
OBJECT_NOT_LOCAL	sem_delete not allowed on non-local semaphore

Description

The sem_delete operation deletes a semaphore from the kernel data structure. The semaphore is deleted immediately, even though there are tasks waiting in its queue. These latter are all unblocked and are returned the SEMAPHORE_DELETED completion status.

7.3. SEM_IDENT

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

Synopsis

```
   sem_ident( name, nid, sid )
```

Input Parameters

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

Output Parameters

sid	: sem_id	kernel defined semaphore identifier

Literal Values

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

Completion Status

OK	sem_ident successful
ILLEGAL_USE	sem_ident not callable from ISR
INVALID_PARAMETER	a parameter refers to an invalid address
INVALID_ID	node does not exist
NAME_NOT_FOUND	semaphore does not exist on node
NODE_NOT_REACHABLE	node is not reachable

Description

This operation searches the kernel data structure in the node(s) specified for a semaphore with the given name, and returns its identifier if found. If OTHER_NODES or ALL_NODES is specified, the node search order is implementation dependent. If there is more than one semaphore with the same name in the node(s) specified, then the sid of the first one found is returned.

7.4. SEM_CLAIM

Claim a semaphore (P operation).

Synopsis

```
   sem_claim( sid, options, time_out )
```

Input Parameters

sid	: sem_id	kernel defined semaphore identifier
options	: bit_field	semaphore wait options
time_out	: integer	ticks to wait before timing out

Output Parameters

<none>

Literal Values

options	+ NOWAIT	do not wait - return immediately if semaphore not available
time_out	= FOREVER	wait forever - do not time out

Completion Status

OK	sem_claim successful
ILLEGAL_USE	sem_claim not callable from ISR
INVALID_PARAMETER	a parameter refers to an invalid address
INVALID_ID	semaphore does not exist
OBJECT_DELETED	originally existing semaphore has been deleted before operation
TIME_OUT	sem_claim timed out
SEMAPHORE_DELETED	semaphore deleted while blocked in sem_claim
SEMAPHORE_NOT_AVAILABLE	semaphore unavailable with NOWAIT option
SEMAPHORE_UNDERFLOW	semaphore counter underflowed
NODE_NOT_REACHABLE	node on which semaphore resides is not reachable

Description

This operation performs a claim from the given semaphore. It first checks if the NOWAIT option has been specified and the counter is zero or less, in which case the SEMAPHORE_NOT_AVAILABLE completion status is returned. Otherwise, the counter is decreased. If the counter is now zero or more, then the claim is successful, otherwise the calling task is put on the semaphore queue. If the counter underflowed the SEMAPHORE_UNDERFLOW completion status is returned. If the semaphore is deleted while a task is waiting on its queue, then the task is unblocked and the sem_claim operation returns the SEMAPHORE_DELETED completion status to the task. Otherwise the task is blocked either until the timeout expires, in which case the TIME_OUT completion status is returned, or until the task reaches the head of the queue and another task performs a sem_release operation on this semaphore, leading to the return of the successful completion status.

7.5. SEM_RELEASE

Release a semaphore (V operation).

Synopsis

```
sem_release( sid )
```

Input Parameters

sid	: sem_id	kernel defined semaphore identifier

Output Parameters

<none>

Completion Status

OK	sem_release successful
INVALID_PARAMETER	a parameter refers to an invalid address
INVALID_ID	semaphore does not exist
OBJECT_DELETED	originally existing semaphore has been deleted before operation
SEMAPHORE_OVERFLOW	semaphore counter overflowed
NODE_NOT_REACHABLE	node on which semaphore resides is not reachable

Description

This operation increments the semaphore counter by one. If the resulting semaphore count is less than or equal to zero then the first task in the semaphore queue is unblocked, and returned the successful completion status. If the counter overflowed the SEMAPHORE_OVERFLOW completion status is returned.

7.6. SEM_INFO

Obtain information on a semaphore.

Synopsis

```
sem_info( sid, options, count, tasks_waiting )
```

Input Parameters

sid	: sem-id	kernel defined semaphore identifier

Output Parameters

options	: bit_field	semaphore create options
count	: integer	semaphore count at time of call
tasks_waiting	: integer	number of tasks waiting in the semaphore queue

Completion Status

OK	sem_info successful
ILLEGAL_USE	sem_info not callable from ISR
INVALID_PARAMETER	a parameter refers to an invalid address
INVALID_ID	semaphore does not exist
OBJECT_DELETED	originally existing semaphore has been deleted before operation
NODE_NOT_REACHABLE	node on which semaphore resides is not reachable

Description

This operation provides information on the specified semaphore. It returns its create options, the value of it's counter, and the number of tasks waiting on the semaphore queue. The latter two values should be used with care as they are just a snap-shot of the semaphore's state at the time of executing the operation.

8.3. QUEUE_IDENT

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

Synopsis

```
   queue_ident( name, nid, qid )
```

Input Parameters

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

Output Parameters

qid	: queue_id	kernel defined queue identifier

Literal Values

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

Completion Status

OK	queue_ident successful
ILLEGAL_USE	queue_ident not callable from ISR
INVALID_PARAMETER	a parameter refers to an invalid address
INVALID_ID	node does not exist
NAME_NOT_FOUND	queue name does not exist on node
NODE_NOT_REACHABLE	node is not reachable

Description

This operation searches the kernel data structure in the node(s) specified for a queue with the given name, and returns its identifier if found. If OTHER_NODES or ALL_NODES is specified, the node search order is implementation dependent. If there is more than one queue with the same name in the node(s) specified, then the qid of the first one found is returned.

8.4. QUEUE_SEND

Send a message to a given queue.

Synopsis

```
   queue_send( qid, msg_buff, msg_length )
```

Input Parameters

qid	: queue_id	kernel defined queue identifier
msg_buff	: address	message starting address
msg_length	: integer	length of message in bytes

Output Parameters

<none>

Completion Status

OK	queue_send successful
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
INVALID_LENGTH	message length greater than queue's buffer length
QUEUE_FULL	no more buffers available
NODE_NOT_REACHABLE	node on which queue resides is not reachable

Description

This operations sends a message to a queue.

If the queue's wait queue contains a number of tasks waiting on messages, then the message is delivered to the task at the head of the wait queue. This task is then removed from the wait queue, unblocked and will be returned a successful completion status along with the message. Otherwise the message is appended at the end of the queue.

If the maximum queue length has been reached, then the QUEUE_FULL completion status is returned.

8.5. QUEUE_JUMP

Send a message to the head of a given queue.

Synopsis

```
queue_jump( qid, msg_buff, msg_length )
```

Input Parameters

qid	: queue_id	kernel defined queue identifier
msg_buff	: address	message starting address
msg_length	: integer	length of message in bytes

Output Parameters

<none>

Completion Status

OK	queue_jump successful
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
INVALID_LENGTH	message length greater than queue's buffer length
QUEUE_FULL	no more buffers available
NODE_NOT_REACHABLE	node on which queue resides is not reachable

Description

This operations sends a message to the head of a queue.

If the queue's wait queue contains a number of tasks waiting on messages, then the message is delivered to the task at the head of the wait queue. This task is then removed from the wait queue, unblocked and will be returned a successful completion status along with the message. Otherwise the message is prepended at the head of the queue.

If the maximum queue length has been reached, then the QUEUE_FULL completion status is returned.

8.6. QUEUE_BROADCAST

Broadcast message to all tasks blocked on a queue.

Synopsis

```
   queue_broadcast( qid, msg_buff, msg_length, count )
```

Input Parameters

qid	: queue_id	kernel defined queue identifier
msg_buff	: address	message starting address
msg_length	: integer	message length in bytes

Output Parameters

count	: integer	number of unblocked tasks

Completion Status

OK	queue_broadcast successful
ILLEGAL_USE	queue_broadcast 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
INVALID_LENGTH	message length greater than queue's buffer length
NODE_NOT_REACHABLE	node on which queue resides is not reachable

Description

This operation sends a message to all tasks waiting on a queue.

If the wait queue is empty, then no messages are sent, no tasks are unblocked and the count passed back will be zero. If the wait queue contains a number of tasks waiting on messages, then the message is delivered to each task in the wait queue. All tasks are then removed from the wait queue, unblocked and returned a successful completion status. The number of tasks unblocked is passed back in the count parameter.

This operation is atomic with respect to other operations on the queue.

8.7. QUEUE_RECEIVE

Receive a message from a queue.

Synopsis

```
queue_receive( qid, msg_buff, buff_length, options, time_out,
               msg_length )
```

Input Parameters

qid	: queue_id	kernel defined queue identifier
msg_buff	: address	starting address of receive buffer
buff_length	: integer	length of receive buffer in bytes
options	: bit_field	queue receive options
time_out	: integer	ticks to wait before timing out

Output Parameters

msg_length	: integer	received message length in bytes

Literal Values

options	+ NOWAIT	do not wait - return immediately if no message in queue
time_out	= FOREVER	wait forever - do not time out

Completion Status

OK	queue_receive successful
ILLEGAL_USE	queue_receive 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
INVALID_LENGTH	receive buffer smaller than queue's message buffer
INVALID_OPTIONS	invalid options value
TIME_OUT	queue-receive timed out
QUEUE_DELETED	queue deleted while blocked in queue_receive
QUEUE_EMPTY	queue empty with NOWAIT option
NODE_NOT_REACHABLE	node on which queue resides is not reachable

Description

This operation receives a message from a given queue. The operation first checks if the receive buffer is smaller than the queue's message buffer. If this is the case the INVALID_LENGTH completion status is returned.

Otherwise, if there are one or more messages on the queue, then the message at the head of the queue is removed and copied into the receive