3.3.2 SM_IDENT

NAME

sm_ident -- "Obtain the id of a Semaphore"

SYNOPSIS

```
#include <semaphore.h>
uint sm_ident ( name, node, &smid )

        uint name;    /* semaphore name  */
        uint node;    /* node identifier */
                      /* 0 indicates any node */
        uint smid;    /* semaphore id - returned by this call */
```

DESCRIPTION

The *sm_ident* directive allows a task to identify a previously created semaphore by name and receive the *smid* to use in *sm_p*, *sm_v* and *sm_delete* directives for this semaphore.

If the semaphore name is not unique, the *smid* returned may not correspond to the semaphore named in this call.

The semaphore may exist on the local processor or any remote processor in a multiprocessor configuration, as long as the semaphore was created with the GLOBAL flags value set (see *sm_create*). If the semaphore name is not unique within the multiprocessor configuration, a non-sero node identifier must be specified in the *node* field.

RETURN VALUE

If *sm_ident* succeeds, the *smid* will be filled in, and 0 is returned.

If *sm_ident* does not succeed, an error code is returned.

ERROR CONDITIONS

Named semaphore does not exist.

Invalid node identifier.

NOTES

Can be called from within an ISR.

3.3.3 SM_DELETE

NAME

sm_delete – "Delete Semaphore"

SYNOPSIS

```
#include <semaphore.h>
uint sm_delete ( smid )

        uint smid;    /* semaphore id as returned by sm_create or sm_ident */
```

DESCRIPTION

The semaphore identified by the *smid* is deleted from the system.

If tasks are waiting for the semaphore when the semaphore is deleted, each is made ready and given a return code indicating a deleted semaphore.

The semaphore must exist on the local processor. If the semaphore was created with the GLOBAL flags value set in a multiprocessor configuration, a notification will be sent to all processors in the system, so the *smid* can be deleted from the global resource table.

The requester does not have to be the creator of the semaphore. Any task knowing the *smid* can delete it.

RETURN VALUE

If *sm_delete* successfully deleted the semaphore, 0 is returned.

If the semaphore was not successfully deleted, an error code is returned.

ERROR CONDITIONS

Invalid *smid*.

Semaphore not created from local node.

NOTES

Not callable from ISR.

May cause a preempt if a task waiting for the semaphore has a higher priority than the running task, and the preempt mode is in effect. A preempt will not occur if all tasks waiting for the semaphore exist on a remote processor in a multiprocessor configuration.

3.3.4 SM_P

NAME

sm_p -- "Access Semaphore"

SYNOPSIS

```
#include <semaphore.h>
uint sm_p ( smid, flags, timeout )

        uint smid;      /* semaphore id as returned by sm_create or sm_ident  */
        uint flags;     /* wait option */
        uint timeout;   /* number of ticks to wait */
                        /* 0 indicates wait forever */
```

The flags field values are:

NOWAIT	set	return immediately with error if semaphore count is negative
	clear	wait for resource

DESCRIPTION

If the **NOWAIT** flags value is clear, the current semaphore count of the semaphore identified by the *smid* is decremented by one. If the count is zero or positive, the requesting task continues execution, returning without error. If the count is negative, the requesting task must wait for access to the resource, and is put on a waiting list.

If the **NOWAIT** flags value is set, and the count is negative, an error is returned. If the count is zero or positive, zero is returned.

The semaphore identified by the *smid* may exist on the local processor or any remote processor in a multiprocessor configuration, as long as the semaphore was created with the **GLOBAL** flags value set (see *sm_create*).

When *sm_p* is called from an ISR, the no-wait option is forced by the executive.

RETURN VALUE

If *sm_p* succeeded, then 0 is returned.

If the call was not successful, an error code is returned.

ERROR CONDITIONS

Invalid *smid*.

Timeout (if wait and timeout is selected).

The semaphore count is negative (if no wait is selected).

Semaphore deleted.

ISR cannot reference remote node.

NOTES

Can be called from within an ISR, except when the semaphore was not created on the local node. The no-wait option is forced by the executive.

The running task will be blocked if the count is negative.

3.3.5 SM_V

NAME

sm_v -- "Release Semaphore"

SYNOPSIS

```
#include <semaphore.h>
uint sm_v ( smid )

        uint smid;    /* semaphore id as returned by sm_create or sm_ident */
```

DESCRIPTION

The current semaphore count of the semaphore identified in the *smid* field is incremented by one.

If the count is zero or negative, the first task in the waiting list is removed from the list and is made ready to await execution. If the task is of higher priority than the running task, it will cause a preempt.

RETURN VALUE

If *sm_v* succeeded, then 0 is returned.

If the call was not successful, an error code is returned.

ERROR CONDITIONS

Invalid *smid*.

ISR cannot reference remote node.

NOTES

Can be called from within an ISR, except when the semaphore was not created on the local node.

May cause a preempt if a task waiting on the semaphore has a higher priority than the running task, and the preempt mode is in effect. A preempt will not occur if the task waiting exists on a remote processor in a multiprocessor configuration.

3.4 Time Management

The executive time manager supports two concepts of time: calendar time and elapsed time. These functions depend on periodic timer interrupts, and will not work without timer hardware.

The *tm_set* directive allows a task to inform the time manager of the current date and time (e.g., March 21, 1985; 12:04). The *tm_get* directive allows a task to request the current date and time from the time manager (e.g., March 27, 1986; 09:24).

The *tm_wkafter* directive allows a task to remove itself from the running state and enter into a wait state for a specified number of ticks. *After* the elapsed time expires, the task is made ready.

The *tm_wkwhen* directive allows a task to remove itself from the running state and enter into a wait state until a specific date and time is reached. *When* the date and time is reached, the task is made ready.

The *tm_evafter* directive allows a task to receive a timer event *after* the specified number of system clock ticks have occurred. The requesting task is not blocked by this call. To receive the event, the *ev_receive* directive must be used.

The *tm_evwhen* directive allows a task to receive a timer event *when* the specified date and time is reached. The requesting task is not blocked by this call. To receive the event, the *ev_receive* directive must be used.

The *tm_cancel* directive allows a task to cancel a timer event scheduled by the *tm_evafter* or *tm_evwhen* directives.

The *tm_tick* directive allows a task or an interrupt service routine to inform the system of the occurrence of a system clock tick. This information is used to maintain correct calendar time, execute timeslicing, and decrement ticks from tasks which are currently being delayed or timing out.

Tick and timeslice are configuration parameters. A tick is defined to be some integral number of milliseconds. A timeslice is defined to be some integral number of ticks.

The directives provided by the time manager are:

Directive	Function
tm_set	Set date and time
tm_get	Get date and time
tm_wkafter	Wake after interval
tm_wkwhen	Wake when date and time
tm_evafter	Send event after interval
tm_evwhen	Send event when date and time
tm_cancel	Cancel timer event
tm_tick	Announce tick

3.4.1 Timebuf Structure

The time and date buffer structure is defined as follows:

```
struct   time_ds          {
         struct t_date    date;    /* date */
         struct t_time    time;    /* time */
         uint             ticks;   /* current elapsed ticks between seconds */
};
```

Date is defined as follows:

```
struct   t_date    {
         short     year;     /* year, A.D. */
         char      month;    /* month, 1->12 */
         char      day;      /* day, 1->31 */
};
```

Time is defined as follows:

```
struct   t_time    {
         short     hour;     /* hour, 0->23 */
         char      minute;   /* minute, 0-> 59 */
         char      second;   /* second, 0-> 59 */
};
```

3.4.2 TM_SET

NAME

tm_set – "Set System Time and Date"

SYNOPSIS

```
#include <time.h>
uint tm_set ( timebuf )

        struct time_ds *timebuf;    /* pointer to time and date structure */
```

DESCRIPTION

The *tm_set* directive sets or resets the date and time of *all* nodes within the system. The parameters within the time and date structure are validated, and an error will be returned if they are out of range.

After this call is successfully completed, the system maintains the date and time based upon the frequency of system clock ticks. The current date and time may be obtained by using the *tm_get* directive.

RETURN VALUE

If *tm_set* successfully set the date and time, then 0 is returned.

If the date and time were not successfully set, an error code is returned.

ERROR CONDITIONS

Date input parameter error.

Time input parameter error.

Ticks input parameter error.

NOTES

Callable from ISR.

May cause a preempt if setting the time causes a task on the timeout list to become ready, and that task has a higher priority than the running task, and the preempt mode is in effect.

3.4.3 TM_GET

NAME

tm_get – "Get System Time and Date"

SYNOPSIS

```
#include <time.h>
uint tm_get ( timebuf )

        struct time_ds *timebuf;    /* pointer to time and date structure */
```

DESCRIPTION

The requester is allowed to get the current date and time as maintained by the system. If the date and time have not been set via the *tm_set* directive, then an error is returned, and the buffer contents will be meaningless.

RETURN VALUE

If *tm_get* successfully got the date and tim ., *timebuf* will be filled in, and 0 is returned.

If the date and time have not been set, an error code is returned.

ERROR CONDITIONS

Date and time have not been set.

NOTES

Callable from ISR.

Will not cause a preempt.

3.4.4 TM_WKAFTER

NAME

tm_wkafter -- "Wake After Interval"

SYNOPSIS

```
#include <time.h>
uint tm_wkafter ( ticks )

        uint ticks;     /* number of ticks to wait */
```

DESCRIPTION

The executive stops the execution of the requesting task until the specified number of system clock ticks have occurred. Execution resumes at the location following the *tm_wkafter* directive.

If the system clock frequency is 100 ticks per second, and the requester wants to wait for 2 seconds, then the input parameter will be 100*2, or 200 ticks.

The relative scheduling priority of the task will influence when the task actually gets to run again. A manual round-robin may be performed by executing *tm_wkafter(0)*. This causes the requesting task to yield the processor to other tasks at the same priority, if any exist.

The number of ticks remaining until the task is awakened will not be modified by the executive if the system date and time are reset via the *tm_set* directive.

The maximum duration is 2**32 - 1 ticks.

RETURN VALUE

Tm_wkafter always succeeds and returns 0.

ERROR CONDITIONS

None.

NOTES

Not callable from ISR.

The requesting task will be blocked until the interval is expired.

3.4.5 TM_WKWHEN

NAME

#include <time.h>
tm_wkwhen -- "Wake When Date and Time"

SYNOPSIS

```
#include <time.h>
uint tm_wkwhen ( timebuf )

        struct time_ds *timebuf;    /* pointer to time and date structure */
```

DESCRIPTION

The executive stops execution of the requesting task until the specified date and time is reached. Execution resumes at the location following the *tm_wkwhen* directive.

If the system date and time are reset via the *tm_set* directive, the requested date and time when the task will be awakened will be modified by the executive. Therefore, if the date and time are reset *ahead* of the requested time, the task may be awakened *late*.

The relative scheduling priority of the task will influence when the task actually gets to run again.

The current elapsed ticks in the *ticks* field within the timebuf structure are ignored.

RETURN VALUE

If *tm_wkwhen* is successful, then 0 is returned.

If the date and time are invalid, an error code is returned.

ERROR CONDITIONS

Date and time have not been set.

Date input parameter error.

Time input parameter error.

NOTES

Not callable from ISR.

The requesting task will be blocked until the date and time is reached.

3.4.8 TM_EVAFTER

NAME

tm_evafter -- "Send Event After Interval"

SYNOPSIS

```
#include <time.h>
uint tm_evafter ( ticks, event, &tmid )

        uint ticks;     /* number of ticks until event */
        uint event;     /* event condition */
        uint tmid;      /* timer id - returned by this call */
```

DESCRIPTION

The *tm_evafter* directive allows a task to receive a timer event after the specified number of system clock ticks have occurred. The requesting task is not blocked by this call. To receive the event, the *ev_receive* directive must be used.

If the system clock frequency is 100 ticks per second, and the requester wants to receive an event after 2 seconds, then the input parameter will be 100*2, or 200 ticks.

The number of ticks remaining until the timer event is sent will not be modified by the executive if the system date and time are reset via the *tm_set* directive.

The maximum duration is 2**32 - 1 ticks.

RETURN VALUE

Tm_evafter always succeeds, the *tmid* is filled in, and 0 is returned.

ERROR CONDITIONS

Too many timers.

NOTES

Not callable from ISR.

Will not cause a preempt.

The requesting task will not be blocked.

3.4.7 TM_EVWHEN

NAME

tm_evwhen -- "Send Event When Date and Time"

SYNOPSIS

```
#include <time.h>
uint tm_evwhen ( timebuf, event, &tmid )

        struct time_ds *timebuf;    /* pointer to time and date structure */
        uint event;                 /* event condition */
        uint tmid;                  /* timer id - returned by this call */
```

DESCRIPTION

The *tm_evwhen* directive allows a task to receive a timer event when the specified date and time is reached. The requesting task is not blocked by this call. To receive the event, the *ev_receive* directive must be used.

If the system date and time are reset via the *tm_set* directive, the requested date and time of the timer event will be modified by the executive. Therefore, if the date and time are reset *ahead* of the requested time, the task may receive the timer event *late*.

The current elapsed ticks in the *ticks* field within the timebuf structure are ignored.

RETURN VALUE

If *tm_evwhen* is successful, the *tmid* is filled in, and 0 is returned.

If the date and time are invalid, an error code is returned.

ERROR CONDITIONS

Too many timers.

Date and time have not been set.

Date input parameter error.

Time input parameter error.

NOTES

Not callable from ISR.

Will not cause a preempt.

The requesting task will not be blocked.

3.4.8 TM_CANCEL

NAME

tm_cancel -- "Cancel Timer Event"

SYNOPSIS

```
#include <time.h>
uint tm_cancel ( tmid )

        uint tmid;    /* timer id - as returned from tm_evafter or tm_evwhen */
```

DESCRIPTION

The *tm_cancel* directive allows a task to cancel the timer event identified by the *tmid*. The timer event may have been scheduled by the *tm_evafter* or *tm_evwhen* directives.

RETURN VALUE

If *tm_cancel* successfully canceled the timer event, then 0 is returned.

If the call was not successful, an error code is returned.

ERROR CONDITIONS

Invalid *tmid*.

Timer event not set.

NOTES

Not callable from ISR.

Will not cause a preempt.

The *timer event not set* error may occur if the specified *tmid* has expired. The caller may need to clear the event condition associated with the *tmid*.

3.4.9 TM_TICK

NAME

tm_tick -- "Announce Tick"

SYNOPSIS

uint tm_tick ( )

DESCRIPTION

This call is used to inform the executive that a system clock tick has occurred. This information is used by the time manager to maintain correct calendar time, execute timeslicing, and decrement ticks from tasks which are currently being delayed or timing out. When a timeslice or timeout expires, the task is made ready.

RETURN VALUE

***Tm_tick* always succeeds and returns 0.**

ERROR CONDITIONS

None.

NOTES

Can be called from within an ISR.

3.5 Interrupt Handling

Fast interrupt response and the ability to preempt from an Interrupt Service Routine (ISR) are important features of a real time executive.

In order to provide the fastest possible interrupt service mechanism, the executive will allow tasks and ISRs to directly claim interrupt vectors by writing directly to the vector table.

An ISR usually communicates with tasks within the system using RTEID directives. The directives which are callable from ISRs are identified in the **NOTES** section of each directive. Directives called from an ISR will always return immediately to the ISR, without going though the normal dispatch cycle. The postponed dispatch is required to complete the ISR before any tasks are dispatched.

The *i_return* directive provides the real-time exit mechanism for ISRs. Since an ISR can make a task other than the running task ready to run, i.e. by sending a message from the ISR, it becomes extremely important NOT to exit the ISR with the RTE instruction. This would return control to the running task at the time of the interrupt, which may not be the highest priority task ready to run. To ensure the highest priority task runs, all ISRs must exit using the *i_return* directive, which may cause the running task to be preempted.

The directives provided by the interrupt manager are:

Directive	Function
i_return	Return from Interrupt

3.5.1 I_RETURN

NAME

i_return -- "Return from Interrupt"

SYNOPSIS

void i_return ()

DESCRIPTION

The *i_return* directive will allow the executive to return control to the highest priority task in the system following the interrupt processing. The interrupt routine may have caused a task of higher priority than the task running at the time of interrupt, to become ready.

RETURN VALUE

None.

ERROR CONDITIONS

None.

NOTES

Can only be called from an ISR.

3.6 Fatal Errors

Occasionally, the executive, application or system software will detect an unrecoverable error condition. Such a condition is called a *fatal error* and normally halts execution on the local node. Such errors include checksum errors, not enough memory, etc.

The executive will provide a fatal error handler which is responsible for processing fatal errors. The exact manner in which fatal errors are processed is implementation dependent. For example, the executive may simply **STOP**, or it may pass control to a debugger or other user provided fatal error handling routine.

There are three sources for fatal errors:

1. the executive
2. system code
3. user application code

When the executive detects a fatal error, control is automatically passed to the fatal error handler. When system code or user application code detects a fatal error, the *k_fatal* directive should be used to pass control to the fatal error handler. The error code passed to the fatal error handler describes the type of fatal error.

Fatal errors only halt execution on the local node. Remote nodes are not directly affected.

The directive provided to report fatal errors is:

Directive	Function
k_fatal	Fatal Error

3.6.1 K_FATAL

NAME

k_fatal -- "Fatal Error"

SYNOPSIS

void k_fatal (errcode)

uint errcode; /* type of error to be reported */

DESCRIPTION

The *k_fatal* directive will allow the executive to halt execution of the system in a manner as described by the *errcode*. This directive does not return to the caller.

RETURN VALUE

None.

ERROR CONDITIONS

None.

NOTES

Can be called from within an ISR.

3.7 Memory Management

The executive will support two different memory managers. A region manager provides allocation of variable sized memory segments. A partition manager provides allocation of fixed sized buffers.

3.7.1 Region Manager

A region is an area of physical contiguous memory from which the executive can dynamically allocate segments to an application. A segment is a variable length block of memory.

A region is created with the *rn_create* directive. Like all objects managed by the executive, a region has a 4 character name, and, once created, a 32-bit region id (*rnid*). Tasks other than the creator can use the *rn_ident* directive to obtain a region's *rnid*. The directives *rn_getseg* and *rn_retseg* allocate and return segments from the region.

Each region has an associated *pagesize*, specified when the region is created. The pagesize must be a power of 2. Segment lengths are always in multiples of this pagesize. For example, if a task requests a 700 byte segment from a region having a 512 byte pagesize, a 1024 byte segment is allocated.

When requesting a segment, if the request cannot immediatedly be satisfied, the requesting task may optionally wait (with or without timeout) for a segment to become available. If it elects to wait, the task is placed in a memory wait queue associated with the region. Tasks can be queued either by priority or FIFO. When a segment is returned, if possible it is merged with its neighbor segments and then the wait queue is searched. The first task, if any, whose request can be satisfied receives the segment.

In a multiprocessor system, regions may not be shared between processors. Segments may only be allocated or returned by tasks running on the processor from which the region was created. Hence, the GLOBAL flag used with the other create services is not supported by *rn_create*.

When a region is created, the executive must build data structures to manage the region. The memory containing these structures may itself be allocated from the region, in which case, the amount of allocatable memory within the region may be slightly less than the original size of the region.

The maximum number of regions that may exist at any one time is a configuration parameter.

The directives provided by the region manager are:

Directive	Function
rn_create	Create a region
rn_ident	Obtain id of a region
rn_delete	Delete a region
rn_getseg	Get a segment
rn_retseg	Return a segment

3.7.2 Partition Manager

A partition is a pool of equal sized buffers. *Pt_create* creates a partition in a physical contiguous memory area provided by the caller. Like all objects managed by the executive, partitions have a 4 character name, and, once created, a 32-bit partition id (*ptid*). Tasks other than the creator can use the *pt_ident* directive to obtain a partition's *ptid*. *Pt_getbuf* and *pt_retbuf* allocate and return buffers from the partition.

Each partition contains a specified number of fixed size buffers. The number and size of the buffers is specified when the partition is created.

In a shared memory multiprocessor configuration, partitions may be shared between processors. To do so, the caller must declare the partition **GLOBAL** when it is created. If a partition is **GLOBAL**, then the executive will arbitrate access to the partition.

Tasks may not wait for buffers. If no buffers are available an error number is returned.

When a partition is created, the executive must build data structures to manage the partition. The memory containing these structures may be allocated within the partition area provided by the caller, in which case, the partition may occupy slightly more memory than the simple product of the buffer count and buffer size.

The maximum number of partitions that may exist at any one time is a configuration parameter.

The directives provided by the partition manager are:

Directive	Function
pt_create	Create a partition
pt_ident	Obtain id of a partition
pt_delete	Delete a partition
pt_getbuf	Get a buffer
pt_retbuf	Return a buffer

3.7.3 RN_CREATE

NAME

rn_create -- "Create a Region"

SYNOPSIS

```
#include <memory.h>
uint rn_create ( name, paddr, length, pagesize, flags, &rnid, &bytes )

        uint name;      /* user defined 4-byte region name */
        char *paddr;    /* physical start address of region */
        uint length;    /* physical length in bytes */
        uint pagesize;  /* region pagesize */
        uint flags;     /* region attributes */
        uint rnid;      /* region id - returned by this call */
        uint bytes;     /* available number of bytes - returned by this call */
```

The flags field values are:

PRIOR	set	to process wait list by priority
	clear	to process wait list by FIFO

DESCRIPTION

This directive allows the user to create a region from a physical contiguous memory area. The region id will be returned in *rnid* by the executive to use in *rn_getseg* and *rn_retseg* directives for the region.

The region physical start address specified in *paddr* will be long-word aligned by the executive. In systems with an MMU, the region physical start address must be on the pagesize boundary.

The available number of bytes within the region will be returned by the executive in the *bytes* field. Since the executive may use memory within the region for a region data structure, the number of bytes in *bytes* may be less than the number of bytes in *length*.

By setting the PRIOR value in the flags field, tasks which wait for segments from the region will be processed in task priority order. Otherwise, the tasks will wait in first in, first out (FIFO) order.

Regions may not be shared between processors in a shared memory multiprocessor configuration.

The maximum number of regions that can be in existence at one time is a configuration parameter.

RETURN VALUE

If *rn_create* successfully created the region, then *rnid* and *bytes* are filled in and 0 is returned.

If the region was not successfully created, an error code is returned.

ERROR CONDITIONS

Too many regions.

Paddr is not on a pagesize boundary (MMU only).

NOTES

Not callable from ISR.

8.7.4 RN_IDENT

NAME

rn_ident -- "Obtain id of a Region"

SYNOPSIS

```
#include <memory.h>
uint rn_ident ( name, &rnid )

        uint name;    /* user defined 4-byte region name */
        uint rnid;    /* region id - returned by this call */
```

DESCRIPTION

This directive allows a task to identify a previously created region by name, and obtain the *rnid* to use for *rn_getseg* and *rn_retseg* directives for the region.

The region must have been created by a task on the local processor. It may not be shared between processors in a shared memory multiprocessor configuration.

If the region name is not unique, the region id returned in *rnid* may not correspond to the region named by this call.

RETURN VALUE

If *rn_ident* directive succeeds, then the *rnid* is filled in and 0 is returned.

If the call was not successful, an error code is returned.

ERROR CONDITIONS

Named region does not exist.

NOTES

Can be called from within an ISR.

Will not cause a preempt.

3.7.5 RN_DELETE

NAME

rn_delete -- "Delete a Region"

SYNOPSIS

```
#include <memory.h>
uint rn_delete ( rnid )

        uint rnid;    /* region id as returned by rn_create or rn_ident */
```

DESCRIPTION

This directive deletes the specified region, provided that none of its segments is still allocated. After this directive has successfully executed, the executive will reject any *rn_getseg* and *rn_retseg* directives for the region.

RETURN VALUE

If *rn_delete* successfully deleted the region, then 0 is returned.

If the region was not successfully deleted, an error code is returned.

ERROR CONDITIONS

Invalid *rnid*.

Cannot delete -- outstanding segments.

NOTES

Not callable from ISR.

Will not cause a preempt.

3.7.6 RN_GETSEG

NAME

rn_getseg -- "Get a Segment"

SYNOPSIS

```
#include <memory.h>
uint rn_getseg ( rnid, size, flags, timeout, &segaddr )

        uint rnid;      /* region id as returned by rn_create or rn_ident */
        uint size;      /* segment size in bytes */
        uint flags;     /* directive options */
        uint timeout;   /* number of ticks to wait for memory */
                        /* 0 indicates forever */
        char *segaddr;  /* segment address - returned by this call */
```

The flags field values are defined as follows:

NOWAIT	set	if the task is to return immediately
	clear	if the task is to wait for memory

DESCRIPTION

This directive allocates a variable size segment from the region specified by the *rnid*. The address of the segment is returned to the caller in *segaddr*.

The actual segment length is a multiple of the region pagesize. Thus, the segment allocated may be larger than the requested size.

RETURN VALUE

If *rn_getseg* successfully allocated the segment, the address of the segment is returned in *segaddr* and 0 is returned.

If the call was not successful, an error code is returned.

ERROR CONDITIONS

Invalid *rnid*.

No memory available (no-wait only).

Timeout occurred before memory was available (wait with timeout).

Region has been deleted.

NOTES