Real Time Executive Interface Definition

DRAFT 2.1

Prepared by:

MOTOROLA Microcomputer Division

and

Software Components Group

Abstract:

This specification defines a basic set of functions that constitute the Real Time Executive Interface Definition. Draft 2.1 is for public review, MOTOROLA/SCG retain the right to modify this definition as appropriate during implementation. Draft 2.1 will be submitted to the VITA technical committee no later than 01/25/88.

PRELIMINARY

REAL TIME EXECUTIVE INTERFACE DEFINITION

January 22, 1988

DISCLAIMER

This RTEID specification is being proposed to be used as the basis for formal standardization by the VME International Trade Association (VITA). However, since the standardization process has just begun, any standard resulting from this document might be different from this document . Any Product designed to this document might not be compatible with the final standard. No responsibility is assumed for such incompatibilities and no liability is assumed for any product built to conform to this document.

While considerable effort has been expended to make this document comprehensive, reliable, and unambiguous, it is still being published in preliminary form for public study and comment.

This document is prepared by Motorola Inc., Microcomputer Division. The design and development of RTEID is a joint effort of Motorola Inc., Microcomputer Division and Software Components Group, Inc. Interest in the RTEID is welcomed and encouraged any technical questions, suggestions or comments may be directed to:

Motorola Inc.
Microcomputer Division
Dept: RTEID
2900 South Diablo Way
Tempe, Arizona 85282
Tel: (602)438-3500
Fax: (602)438-3581
Tlx: 4998071 (MOTPHE)

Software Components Group, Inc.
4655 Old Ironsides Drive
Santa Clara, California 95054
Tel: ~~(408)727-0707~~ 408 - 437 - 0700
Fax: (408)727-0904
Tlx: 757697 (softcom)

John Gilbert - tech staff
Linda Mungy - Sales

REVISION RECORD		
Issue	Revision Description	Date
1	Initial version. Internal Only.	05/06/87
2	Added semaphores and debug management.	06/01/87
3	Preliminary Draft, limited distribution.	06/17/87
4	Design review of SCG's comments.	07/24/87
5	SCG/MOT Technical review.	08/20/87
6	SCG/MOT Technical review.	08/28/87
7	SCG/MOT Technical review.	09/14/87
8	SCG/MOT Technical review for Draft 2.1	12/14/87
9	Added Debug Extensions for Draft 2.1	12/22/87
10	Added I/O Interface for Draft 2.1	01/15/88
11	Removed Debug Extensions from Draft 2.1	01/22/88
12	Final Draft 2.1 submitted to VITA	01/25/88
13		

TABLE OF CONTENTS

Page

1. INTRODUCTION

1.1 Overview

This document is intended to serve the following major purposes:

- To serve as a reference source for the definition of the external interfaces to services that are provided by all Real Time Executive environments. This includes source-code interfaces and run-time behavior as seen by an application-program. It does not include the details of how the kernel implements these functions.

- To serve as a complete definition of Real Time Executive external interfaces, so that application source-code that conforms to these interfaces, will execute as defined in all Real Time Executive environments. It is assumed that source-code is recompiled for the proper target hardware. The basic objective is to facilitate the writing of applications-program source-code that is directly portable across all Real Time Executive implementations.

This document describes the basic set of functionality that makes up the Base System. This functionality has been structured to provide a minimal, stand alone run-time environment for application-programs originally written in a high-level language, such as C.

Other extensions to this Base System will be defined as a continuing effort to produce this standard Real Time Executive Run Time Environment.

It is anticipated that all conforming systems must support the source code interfaces and runtime behavior of the Base System. A system may conform to some, none, or all of the extensions.

1.2 Definitions

executive	That portion of software that constitutes the kernel or performs specific services on behalf of programs tasks.
Real Time Executive	Same as executive.
node	A processor within a multiprocessor system configuration.
local node	The processor within a multiprocessor system configuration on which the current operation is being executed.
remote node	A processor within a multiprocessor system configuration on which the current operation is *not* being executed.
target	The destination remote node in a multiprocessor system configuration.

1.3 Typedefs and Structures

For ease of documentation, the following typedefs are used in this document.

```
typedef    unsigned int    uint;       /* 32-bit unsigned integer */

typedef    void            (*ptf)();   /* pointer to a function that returns nothing */
```

LIST OF TABLES

2. Basic System Services

The Basic System Services is intended to support a minimal run-time environment for executable applications. The Basic System Services defines a set of Real Time Executive components needed by applications-programs. This basic set would be supported by any conforming system. It defines each component's source-code interface and run-time behavior, but does not specify its implementation. Source-code interfaces described are for the C language.

While only the run-time behavior of these components is supported by the Basic System Services, the source-code interfaces to these components are defined because an objective of the Real Time Executive Interface Definition is to facilitate application-program source-code portability across all Real Time Executive implementations. It is assumed that an application-program targeted to run on a system that provides only the Basic System Services (a run-time environment) would be compiled on a system supporting software development.

3. EXECUTIVE FACILITIES

The facilities of the executive have been grouped by function, and are discussed in the following paragraphs.

TABLE 1. Directives

Name	Input Parameters					Output Parameters
t_create	name	superstk	userstk	priority	flags	&tid
t_ident	name	node				&tid
t_start	tid	saddr	mode	argp		
t_restart	tid	argp				
t_delete	tid					
t_suspend	tid					
t_resume	tid					
t_setpri	tid	priority				&ppriority
t_mode	mode	mask				&pmode
t_getreg	tid	regnum				&regval
t_setreg	tid	regnum	regval			
q_create	name	count	flags			&qid
q_ident	name	node				&qid
q_delete	qid					
q_send	qid	buffer				
q_urgent	qid	buffer				
q_broadcast	qid	buffer				&count
q_receive	qid	buffer	flags	timeout		
ev_send	tid	event				
ev_receive	eventin	flags	timeout			&eventout
as_catch	asraddr	mode				
as_send	tid	signal				
as_return						
sm_create	name	count	flags			&smid
sm_ident	name	node				&smid
sm_delete	smid					
sm_p	smid	flags	timeout			
sm_v	smid					
tm_set	timebuf					
tm_get	timebuf					
tm_wkafter	ticks					
tm_wkwhen	timebuf					
tm_evafter	ticks	event				&tmid
tm_evwhen	timebuf	event				&tmid
tm_cancel	tmid					
tm_tick						
i_return						
k_fatal	errcode					

Name	Input Parameters						Output Parameters	
rn_create	name	paddr	length	pagesize	flags		&rnid	&bytes
rn_ident	name						&rnid	
rn_delete	rnid							
rn_getseg	rnid	size	flags	timeout			&segaddr	
rn_retseg	rnid	segaddr						
pt_create	name	paddr	length	bsize	flags		&ptid	&bnum
pt_ident	name	node					&ptid	
pt_delete	ptid							
pt_getbuf	ptid						&bufaddr	
pt_retbuf	ptid	bufaddr						
mm_l2p	tid	laddr					&paddr	&length
mm_p2l	tid	paddr					&laddr	&length
mm_pmap	tid	laddr	paddr	length	flags			
mm_unmap	tid	laddr						
mm_pread	paddr	laddr	length					
mm_pwrite	paddr	laddr	length					
mm_ptcreate	name	paddr	length	bsize	laddr	flags	&ptid	&bnum
m_ext2int	external						&internal	
m_int2ext	internal						&external	

TABLE 2. Directive Usage

Name	Remote	ISR	ISR to Remote
t_create	no	no	-
t_ident	yes	yes	yes
t_start	no	no	-
t_restart	no	no	-
t_delete	no	no	-
t_suspend	yes	no	-
t_resume	yes	yes	no
t_setpri	yes	no	-
t_mode	no	no	-
t_getreg	yes	yes	no
t_setreg	yes	yes	no
q_create	no	no	-
q_ident	yes	yes	yes
q_delete	no	no	-
q_send	yes	yes	no
q_urgent	yes	yes	no
q_broadcast	yes	yes	no
q_receive	yes	yes	no
ev_send	yes	yes	no
ev_receive	yes	no	-
as_catch	no	no	-
as_send	yes	yes	no
as_return	no	no	-
sm_create	no	no	-
sm_ident	yes	yes	yes
sm_delete	no	no	-
sm_p	yes	yes	no
sm_v	yes	yes	no
tm_set	yes	yes	no
tm_get	no	yes	no
tm_wkafter	no	no	-
tm_wkwhen	no	no	-
tm_evafter	no	no	-
tm_evwhen	no	no	-
tm_cancel	no	no	-
tm_tick	no	yes	no
i_return	no	yes	-
k_fatal	no	yes	-

Name	Remote	ISR	ISR to Remote
rn_create	no	no	-
rn_ident	yes	yes	yes
rn_delete	no	no	-
rn_getseg	no	no	-
rn_retseg	no	no	-
pt_create	no	no	-
pt_ident	yes	yes	yes
pt_delete	no	no	-
pt_getbuf	yes	yes	yes
pt_retbuf	yes	yes	yes
mm_l2p	no	yes	no
mm_p2l	no	no	-
mm_pmap	no	yes	no
mm_unmap	no	yes	no
mm_pread	no	no	-
mm_pwrite	no	no	-
mm_ptcreate	no	no	-
m_ext2int	no	yes	no
m_int2ext	no	yes	no

NO

3.1 Task Management

A task is a function that can execute concurrently with other functions within a multitasking environment. A task typically accepts one or more inputs, performs some processing function based on the input, and responds with one or more outputs.

A task is created using the *t_create* directive. Once a task is created, other tasks can refer to it and act on its behalf in allocating resources to it. A task is started with the *t_start* directive. Once a task has been started, it can execute its function and vie with other tasks for processor time according to its relative priority.

A task may be deleted with the *t_delete* directive. All knowledge of the task is removed from the system, and other tasks referring to it will be returned an error.

All tasks have a task identifier *(tid)*. The *tid* is assigned to the task at creation time, and must be used in all subsequent calls to the executive to identify that task. The *t_ident* directive may be used to obtain the *tid* of another task when the task name is known.

All tasks have a priority. A task's priority is a measure of the task's importance relative to all other tasks within the system and indicate its "need to run" in a multitasking environment where many tasks may be ready to run at any moment. A task is given a priority at creation time. A task's priority may be changed with the *t_setpri* directive.

A task's mode of execution is set up initially with the *t_start* directive, and may be changed using the *t_mode* directive. The mode of a task specifies its ability to be preempted, timesliced, to execute in user mode, to execute in supervisor mode at an optional interrupt level, and to disable/enable its asynchronous signal routine.

The task manager provides the pair of directives, *t_suspend* and *t_resume*, to control execution of another task.

A task is provided with a set of eight user and eight system defined software registers which may be set with the *t_setreg* directive, and read with the *t_getreg* directive.

The directives provided by the task manager are:

Directive	Function
t_create	Create a task
t_ident	Obtain id of a task
t_delete	Delete a task
t_start	Start a task
t_restart	Restart a task
t_suspend	Suspend a task
t_resume	Resume a task
t_setpri	Set task priority
t_mode	Change task mode
t_getreg	Get task register
t_setreg	Set task register

3.1.1 T_CREATE

NAME

t_create -- "Create a Task"

SYNOPSIS

uint t_create (name, superstk, userstk, priority, flags, &tid)

```
uint name;      /* user defined 4-byte task name */
uint superstk;  /* supervisor stack size in bytes */
uint userstk;   /* user stack size in bytes */
uint priority;  /* task priority */
uint flags;     /* task attributes */
uint tid;       /* task id - returned by this call */
```

Flags is defined as follows:

CMASK		Coprocessor mask 0 = no coprocessor
GLOBAL	set	to indicate the task is a multiprocessor global resource.
	clear	to indicate the task is local

DESCRIPTION

The *t_create* directive creates a task by allocating and initialising a task data structure. A task is created by name. A task id is returned to the caller in the *tid* field. The *tid* must be used in all calls to the executive requiring a *tid*.

The task is allocated a user stack and supervisor stack as determined by the values in the *userstk* and *superstk* fields. A minimum supervisor stack is required, and an error will be returned if the *superstk* value is too small. There is no minimum user stack required.

By setting the GLOBAL value in the flags field, the *tid* will be sent to all processors in the system, to be entered into a global resource table. The system is defined as the collection of interconnected processors. The task is always created on the local node.

The newly created task will be placed in the dormant state. The *t_start* directive will make the task ready, in priority order. The executive will support a minimum of 32 priorities.

The maximum number of tasks is a configuration parameter.

RETURN VALUE

If *t_create* successfully created a task, the *tid* is filled in, and 0 is returned.

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

ERROR CONDITIONS

Too many tasks.

No more memory for stack(s) segment.

Superstk too small.

Invalid priority.

NOTES

Not callable from ISR.

Will not cause a preempt.

3.1.2 T_IDENT

NAME

t_ident -- "Obtain id of a task"

SYNOPSIS

uint t_ident (name, node, &tid)

```
uint name;    /* user defined 4-byte task name */
              /* 0 indicates requesting task */
uint node;    /* node identifier */
              /* 0 indicates any node */
uint tid;     /* task id - returned by this call */
```

DESCRIPTION

This directive allows a task to obtain the *tid* of itself or another task in the system. The *tid* must then be used in all calls to the executive requiring a *tid*.

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

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

RETURN VALUE

If *t_ident* succeeded, the *tid* is filled in, and 0 is returned.

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

ERROR CONDITIONS

Task with this name does not exist.

Invalid node identifier.

NOTES

Can be called from within an ISR.

Will not cause a preempt.

3.1.3 T_START

NAME

t_start – "Start a Task"

SYNOPSIS

uint t_start (tid, saddr, mode, argp)

```
uint tid;          /* task id as returned from t_create or t_ident */
ptf saddr;         /* start execution address of task */
uint mode;         /* initial mode value of task */
long (*argp)[4];   /* pointer to argument list */
```

The *mode* value is defined as follows:

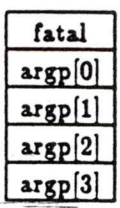
NOPREEMPT	set	to disable preempting
	clear	to enable preempting
TSLICE	set	to enable timeslicing
	clear	to disable timeslicing
NOASR	set	to disable asynchronous signal processing
	clear	to enable asynchronous signal processing
SUPV	set	to execute in supervisor mode
	clear	to execute in user mode
LEVEL		interrupt level when SUPV is set

DESCRIPTION

The task identified by the *tid* is made ready, based on its current priority, to await execution. A task can be started only from the dormant state.

Saddr is the logical address where the task wants to start execution. *Mode* contains the flag values to enable/disable preempting, timeslicing, asynchronous processing, supervisor mode and an optional interrupt level when the task starts execution.

Argp is a pointer to a list of four arguments. These arguments are pushed onto the stack of the task being started. A fifth argument, the executive's fatal error handler, is also pushed onto the task's stack. Should the task attempt to exit the procedure (which normally causes unpredictable behavior), the executive's fatal error handler will be executed. The user must take this frame into consideration when calculating the size of a task's stack(s).



The task identified by the *tid* must exist on the local processor, even if the task was created with the GLOBAL flags value set (see *t_create*).

RETURN VALUE

If *t_start* successfully started the task, then 0 is returned.

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

ERROR CONDITIONS

Invalid *tid*.

Task not in dormant state.

Task not created from local node.

NOTES

Not callable from ISR.

May cause a preempt if the task being started has a higher priority than the running task, and the preempt mode is in effect.

3.1.4 T_RESTART

NAME

t_restart – "Restart a Task"

SYNOPSIS

uint t_restart (tid, argp)

```
uint tid;       /* task id as returned from t_create or t_ident */
long argp[4];   /* pointer to argument list */
```

DESCRIPTION

The task identified by the *tid* is made ready. If the task was blocked, the executive unblocks it. The task's *superstk, userstk,* and *priority* are set to their original values established when the task was created using *t_create*. The task's start address *saddr* and *mode* are set to their original values established when the task was started using *t_start*. A task can be restarted from any state.

Argp is a pointer to a list of four arguments. These arguments are pushed onto the stack of the task being restarted. This argument list may be different from the original argument list. A fifth argument, the executive's fatal error handler, is also pushed onto the task's stack. Should the task attempt to exit the procedure (which normally causes unpredictable behavior), the executive's fatal error handler will be executed.

Tasks which anticipate being restarted can use the arguments to distinguish between initial startup and a restart.

Due to the capability of this call to unblock a task, this call is useful to delete a task in the system. Tasks which anticipate being deleted can use the arguments to distinguish between initial startup and deletion.

fatal
argp[0]
argp[1]
argp[2]
argp[3]

The task identified by the *tid* must exist on the local processor, even if the task was created with the GLOBAL flags value set (see *t_create*).

RETURN VALUE

If *t_restart* successfully restarted the task, then 0 is returned.

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

ERROR CONDITIONS

Invalid *tid*.

Task has never been started.

Task not created from local node.

NOTES

Not callable from ISR.

May cause a preempt if the task being restarted has a higher priority than the running task, and the preempt mode is in effect.

3.1.5 T_DELETE

NAME

t_delete – "Delete a Task"

SYNOPSIS

uint t_delete (tid)

```
uint tid;   /* task id as returned from t_create or t_ident  */
            /* 0 indicates requesting task */
```

DESCRIPTION

This directive allows a task to delete itself, or the task identified in the *tid* field. The executive halts execution of the task and frees the task data structure.

The task identified by the *tid* must exist on the local processor, even if the task was created with the GLOBAL flags value set (see *t_create*).

RETURN VALUE

If the task identified in the *tid* field is the requesting task, then *t_delete* always succeeds, and there is no return.

If the task identified in the *tid* field is not the requesting task, and *t_delete* successfully deleted the task, then 0 is returned to the requesting task.

If the task identified in the *tid* field is not the requesting task, and the call was not successful, an error code is returned to the requesting task.

ERROR CONDITIONS

Invalid *tid*.

Task not created on local node.

NOTES

Not callable from ISR.

A new task is scheduled when the requesting task deletes itself, and there is no return.

Tasks are responsible for returning resources to the executive before deleting itself. It is suggested that a task needing to delete another task use *as_send* or *t_restart* to inform the task to return its resources and then delete itself.

3.1.6 T_SUSPEND

NAME

t_suspend -- "Suspend Task"

SYNOPSIS

uint t_suspend (tid)

```
uint tid;   /* task id as returned from t_create or t_ident */
            /* 0 indicates requesting task */
```

DESCRIPTION

The executive will prevent future execution of the task identified in the *tid* field. The task identified by the *tid* is placed in a suspended state. The suspended state is in addition to the other wait states; waiting for memory, for a message, for an event, for a semaphore, or for a timeout.

The *t_resume* directive issued by another task removes the suspended state. The task is made ready unless blocked by any other wait state.

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

RETURN VALUE

If the task identified in the *tid* field is the requesting task, then *t_suspend* always succeeds and returns 0 when the task runs.

If the task identified in the *tid* field is not the requesting task, and *t_suspend* successfully put the task in the suspend state, then 0 is returned to the requesting task.

If the task identified in the *tid* field is not the requesting task, and the call was not successful, an error code is returned to the requesting task.

ERROR CONDITIONS

Invalid *tid.*

Task already suspended.

NOTES

Not callable from ISR.

The running task will be blocked if suspending itself.

3.1.7 T_RESUME

NAME

t_resume – "Resume a Task"

SYNOPSIS

```
uint t_resume ( tid )

        uint tid;    /* task id as returned from t_create or t_ident */
```

DESCRIPTION

The *t_resume* directive removes the task identified in the *tid* field from the suspended state.

If the task was waiting for memory, for a message, for an event, for a semaphore, or for a timeout, then the task will not be scheduled. Otherwise, the task is scheduled to await execution. If the task is the highest priority ready to run task, it will cause a preempt.

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

RETURN VALUE

If *t_resume* successfully resumed the task, then 0 is returned.

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

ERROR CONDITIONS

Invalid *tid*.

Task not suspended.

ISR cannot reference remote node.

NOTES

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

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

3.1.8 T_SETPRI

NAME

t_setpri – "Set Task Priority"

SYNOPSIS

uint t_setpri (tid, priority, &ppriority)

```
uint tid;       /* task id as returned from t_create or t_ident */
                /* 0 indicates requesting task */
uint priority;  /* task priority */
                /* 0 indicates current priority */
uint ppriority; /* previous priority - returned by this call */
```

DESCRIPTION

This directive changes the current priority of the task identified in the *tid* field to the new value specified by *taskattr*. A task may change its own priority or the priority of another task. The task will be scheduled according to the new priority.

Priority level zero is reserved by the system, and may not be used as a priority. If zero is specified in the *priority* field, the task's current priority will be returned. The executive will support a minimum of 32 priorities.

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

RETURN VALUE

If *t_setpri* successfully changed the task priority, the *ppriority* is filled in, and 0 is returned.

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

ERROR CONDITIONS

Invalid *tid*.

Invalid priority.

NOTES

Not callable from ISR.

May cause a preempt if the running task lowers its own priority, or raises the priority of another task, and the preempt mode is in effect. A preempt will not occur if the task having its priority raised exists on a remote processor in a multiprocessor configuration.

3.1.9 T_MODE

NAME

t_mode – "Change Task Mode"

SYNOPSIS

uint t_mode (mode, mask, &pmode)

```
uint mode;    /* new mode */
uint mask;    /* mask */
uint pmode;   /* previous mode - returned by this call */
```

The *mode* and *mask* values are defined as follows:

NOPREEMPT	set	to disable preempting
	clear	to enable preempting
TSLICE	set	to enable timeslicing
	clear	to disable timeslicing
NOASR	set	to disable asynchronous signal processing
	clear	to enable asynchronous signal processing
SUPV	set	to execute in supervisor mode
	clear	to execute in user mode
LEVEL		interrupt level when SUPV is set

DESCRIPTION

T_mode enables and disables several modes of execution for the calling task. A task may enable/disable timeslicing, enable/disable preempting, enable/disable asynchronous signal processing, or execute in supervisor mode at an optional interrupt level.

Tasks have the ability to process signals asynchronously. Any task with a valid asynchronous signal routine (asr) which needs to temporarily disable asynchronous processing should use this directive.

To change a particular mode, the user must indicate which mode is being changed by setting the appropriate value in the *mask* parameter, and then set the appropriate value in the *mode* parameter to the new mode. For example, if the user only wants to change the preempt mode characteristic, he would set the mask value to NOPREEMPT and the mode value to NOPREEMPT to disable preempting, or the mode field to 0 to enable preempting.

If the preempt mode is not in effect, timeslicing will not take place.

RETURN VALUE

The *t_mode* call always succeeds, *pmode* is filled in, and 0 is returned.

NOTES

Not callable from ISR.

May cause a preempt if the running task enables preempting.

Refer to *as_catch* for discussion on receiving asynchronous signals.

3.1.10 T_GETREG

NAME

t_getreg – "Get a task's register"

SYNOPSIS

uint t_getreg (tid, regnum, &regval)

```
uint tid;      /* task id as returned from t_create or t_ident */
uint regnum;   /* register number */
uint regval;   /* register value - returned by this call */
```

The *regnum* field values are:

S_REG0	System defined register 0
S_REG1	System defined register 1
S_REG2	System defined register 2
S_REG3	System defined register 3
S_REG4	System defined register 4
S_REG5	System defined register 5
S_REG6	System defined register 6
S_REG7	System defined register 7
U_REG0	User defined register 0
U_REG1	User defined register 1
U_REG2	User defined register 2
U_REG3	User defined register 3
U_REG4	User defined register 4
U_REG5	User defined register 5
U_REG6	User defined register 6
U_REG7	User defined register 7

DESCRIPTION

The executive returns the register value in the *regval* field for the register identified in the *regnum* field and the task identified by the *tid*.

The task identified in the *tid* field may exist on the local processor, or any remote processor in the multiprocessing configuration if the task was created with the GLOBAL flags value set (see *t_create*).

RETURN VALUE

If *t_getreg* is successful, *regval* is filled in, and 0 is returned.

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

ERROR CONDITIONS

Invalid *tid*.

Invalid register number.

ISR cannot reference remote node.

NOTES

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

Will not cause a preempt.

3.1.11 T_SETREG

NAME

t_setreg -- "Set a task's register"

SYNOPSIS

uint t_setreg (tid, regnum, regval)

```
uint tid;       /* task id as returned from t_create or t_ident */
uint regnum;    /* register number */
uint regval;    /* register value  */
```

The *regnum* field values are:

S_REG0	System defined register 0
S_REG1	System defined register 1
S_REG2	System defined register 2
S_REG3	System defined register 3
S_REG4	System defined register 4
S_REG5	System defined register 5
S_REG6	System defined register 6
S_REG7	System defined register 7
U_REG0	User defined register 0
U_REG1	User defined register 1
U_REG2	User defined register 2
U_REG3	User defined register 3
U_REG4	User defined register 4
U_REG5	User defined register 5
U_REG6	User defined register 6
U_REG7	User defined register 7

DESCRIPTION

The executive sets the register identified in the *regnum* field for the task identified by the *tid* with the value in the *regval* field.

The task identified in the *tid* field may exist on the local processor, or any remote processor in the multiprocessing configuration if the task was created with the GLOBAL flags value set (see *t_create*).

RETURN VALUE

If *t_setreg* successfully set the register value, 0 is returned.

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

ERROR CONDITIONS

Invalid *tid*.

Invalid register number.

ISR cannot reference remote node.

NOTES

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

Will not cause a preempt.

3.2 Message, Event, and Signal Management

The executive supports communication and synchronisation between tasks using messages and events. Asynchronous communication is supported using signals.

3.2.1 Message Manager

The message queue is the data structure supporting inter-task communication and synchronization. One or more tasks may send messages to the message queue, and one or more tasks may request messages from the queue.

Message queues are created at run time using the *q_create* directive. The creator assigns a 4-byte name and attributes to the queue. The attributes define whether tasks waiting on messages from the queue will wait first-in, first-out (FIFO), or by task priority, and whether the queue will limit the number of messages queued to a specified maximum, or allow an unlimited number of messages.

A message queue is identified by both a name, assigned by the creator, and a message queue id (*qid*), assigned by the executive at *q_create* time. The *qid* is returned to the caller by the *q_create* directive, and must be used by tasks to send and receive messages from the message queue. Tasks other than the task which created the message queue can obtain the *qid* by using the *q_ident* directive.

Messages are sent to the message queue from any task which knows the *qid*, using the *q_send*, *q_urgent*, and *q_broadcast* directives.

When a message arrives at the queue, it will be copied into one of two places. If there is one or more tasks waiting at the queue, then the message is copied into the message buffer belonging to the waiting task. The task is removed from the wait list and is made ready. If there are no tasks waiting at the queue, then the message is copied into a system message buffer (the executive maintains a pool of system message buffers for this purpose). This system message buffer is entered into the message queue. If the message was sent using *q_send*, the message is entered at the tail of the queue. If the message was sent using *q_urgent*, the message is entered at the head of the queue. The *q_broadcast* directive sends a message to all tasks waiting at the queue, so they become ready to run. The count of readied tasks is returned to the caller.

Messages are received from the message queue using the *q_receive* directive. When this directive is called, and a message is in the queue, the message is copied to the task's message buffer, and the directive is complete. When no message is in the queue, there are several ways to proceed. If the calling task asked to wait, the task will be entered into the queue's wait list according the queue's attributes (FIFO or priority). If the calling task asked to wait with timeout, the task will be entered into a timeout list. If the calling task asked not to wait, the task will be returned to with an error code for no message available.

Message queues can be deleted by tasks knowing the *qid* using the *q_delete* directive. If any messages are queued, the executive will claim and return the system message buffers to the system message buffer pool. If any tasks are waiting on the queue, then the executive will remove them from the wait list and make them ready. Waiting tasks will return from the *q_receive* directive with the message queue deleted error.

The message manager defines a message as being fixed length, 16-bytes. The content of the message is user defined. It may be used to carry data, pointers to data, or nothing at all.

The directives provided by the message manager are:

Directive	Function
q_create	Create queue
q_ident	Obtain id of a queue
q_delete	Delete queue
q_send	Send message
q_urgent	Urgent message
q_broadcast	Broadcast message
q_receive	Receive message

3.2.2 Event Manager

Although inter-task synchronisation can be accomplished using the message queue, the executive also provides a second, higher performance method of inter-task synchronization, using events.

Events are different from messages in that they are directed at other tasks. They are also different from messages in that they carry no information, and they cannot be queued. The final difference is tasks can wait for several events at one time, but cannot wait on multiple message queues at one time.

Every task in the system has the ability to send and receive events. Events are simply bits encoded into a event mask. Thirty-two events are available; sixteen will be available as *system* events and sixteen will be available as *user* events. A task can send one or more events to another task using the *ev_send* directive. The *tid* of the destination task is required as input, along with the event set.

A task can receive events using the *ev_receive* directive. The events to receive are input to the directive, along with an option to wait on all of the events, or just one of them. If the events are already pending, then the event mask is cleared before returning to the calling task. If the event condition cannot be satisfied, and the calling task asked to wait, the task will be blocked. If the calling task asked to wait with timeout, the task will be entered into a timeout list. Tasks that do not want to wait for the event condition must specify this as an option. If the event condition was not pending, then an error code for event condition not met is returned.

The directives provided by the event manager are:

Directive	Function
ev_send	Send event
ev_receive	Receive event

3.2.3 Signal Manager

Asynchronous communication is supported through the use of signals.

Signals, like events, are simply bits encoded into a signal mask. Thirty-two signals are available; sixteen will be available as *system* signals and sixteen will be available as *user* signals.

A task can send one or more signals to another task using the *as_send* directive. If the receiving task has set up an asynchronous signal routine (asr) using the *as_catch* directive, the task will be dispatched to the signal routine.

A task may asynchronously receive signals by establishing an asynchronous signal routine (asr) to catch them using the *as_catch* directive. When a signal is caught, the task will be dispatched to the asr address when it becomes the running task. The signal condition will be passed to the task to enable it to determine what signals occurred.

The *as_return* directive must be executed to return the task to its previous dispatch address.

The directives provided by the signal manager are:

Directive	Function
as_catch	Catch signal
as_send	Send signal
as_return	Return from signal

3.2.4 Data Structures for Message Management

Definitions for events and asynchronous signals are as follows:

S_EXEC0	System Software defined
S_EXEC1	System Software defined
S_EXEC2	System Software defined
S_EXEC3	System Software defined
S_EXEC4	System Software defined
S_EXEC5	System Software defined
S_EXEC6	System Software defined
S_EXEC7	System Software defined
S_EXEC8	System Software defined
S_EXEC9	System Software defined
S_EXEC10	System Software defined
S_EXEC11	System Software defined
S_EXEC12	System Software defined
S_EXEC13	System Software defined
S_EXEC14	System Software defined
S_EXEC15	System Software defined

S_USER0	User defined
S_USER1	User defined
S_USER2	User defined
S_USER3	User defined
S_USER4	User defined
S_USER5	User defined
S_USER6	User defined
S_USER7	User defined
S_USER8	User defined
S_USER9	User defined
S_USER10	User defined
S_USER11	User defined
S_USER12	User defined
S_USER13	User defined
S_USER14	User defined
S_USER15	User defined

3.2.5 Q_CREATE

NAME

q_create – "Create a Message Queue"

SYNOPSIS

```
#include <message.h>
uint q_create (name, count, flags, &qid )

        uint name;    /* user defined 4-byte name */
        uint count;   /* maximum message and reserved buffer count */
        uint flags;   /* process method */
        uint qid;     /* message queue id - returned by this call */
```

The flags values are:

PRIOR	set	to process by priority
	clear	to process by FIFO
GLOBAL	set	to indicate the queue is a multiprocessor global resource.
	clear	to indicate the queue is local
TYPE	set	to process typed messages
	clear	to process messages without regard to type
LIMIT	set	to limit queue entries to number in count field
	clear	NO limit on queue entries and no reserved buffers
RESVD	set	to reserve system buffers equal to count when LIMIT is set
	clear	NO reserved system buffers when LIMIT is set

DESCRIPTION

The *q_create* directive creates a message queue by allocating and initialising a message queue data structure. A message queue is created by name. A message *qid* is returned. Subsequent sending and receiving calls must reference the message queue with its message *qid*.

By setting the **PRIOR** value in the flags field, tasks waiting for messages in the queue will be processed by task priority order. Otherwise the tasks waiting for messages will be processed by first in, first out (FIFO) order.

By setting the **TYPE** value in the flags field, messages sent to this queue may be processed by type.

The user may put a limit on the number of messages at the message queue by setting the **LIMIT** value in the flags field, and placing the count in the *count* field. The user may additionally reserve a number of system message buffers equal to the count in the *count* field by setting the **RESVD** value in the flags field.

By setting the **GLOBAL** value in the flags field, the message *qid* will be sent to all processors in

the system, to be entered into a global resource table. The system is defined as the collection of interconnected processors. The message queue is always created on the local node.

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

The maximum number of system message buffers is a configuration parameter.

RETURN VALUE

If the *q_create* directive succeeds, the *qid* is filled in, and 0 is returned.

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

ERROR CONDITIONS

Too many message queues.

No more system message buffers.

NOTES

Not callable from ISR.

Will not cause a preempt.

3.2.6 Q_IDENT

NAME

q_ident -- "Obtain id of a Message Queue"

SYNOPSIS

```
#include <message.h>
uint q_ident ( name, node, &qid )

        uint name;   /* user defined 4-byte name */
        uint node;   /* node identifier */
                     /* 0 indicates any node */
        uint qid;    /* message queue id - returned by this call */
```

DESCRIPTION

The *q_ident* directive allows a task to identify a previously created message queue by name and receive the message *qid* to use for send and receive directives for the queue.

If the message queue name is not unique, the message *qid* returned may not correspond to the message queue named in this call.

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

RETURN VALUE

If the *q_ident* directive succeeds, the *qid* is filled in, and 0 is returned.

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

ERROR CONDITIONS

Named message queue does not exist.

Invalid node identifier.

NOTES

Can be called from within an ISR.

Will not cause a preempt.

3.2.7 Q_DELETE

NAME

q_delete – "Delete a Message Queue"

SYNOPSIS

```
#include <message.h>
uint q_delete ( qid )

        uint qid;    /* message queue id returned from q_create or q_ident */
```

DESCRIPTION

The *q_delete* directive deletes the message queue identified by the *qid*, freeing the data structure.

When a message queue is deleted, it could be in one of three states: empty, tasks waiting for messages, messages waiting for tasks. If empty, the data structure of the message queue is returned to the system. If tasks are waiting, each is made ready and given a return code indicating a deleted message queue. If messages are waiting, then each system message buffer is returned to the system message buffer pool, and the message it is carrying is therefore lost.

The message queue must exist on the local processor. If the message queue 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 *qid* can be deleted from the global resource table.

The requester does not have to be the creator of the message queue. Any task knowing the *qid* can delete it.

RETURN VALUE

If the *q_delete* directive successfully deleted the message queue, then 0 is returned.

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

ERROR CONDITIONS

Message *qid* is invalid.

Message queue not created from local node.

NOTES

Cannot be called from within an ISR.

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

3.2.8 Q_SEND

NAME

q_send -- "Send a Message to a Message Queue"

SYNOPSIS

```
#include <message.h>
uint q_send ( qid, buffer )

        uint qid;           /* message queue id returned from q_create or q_ident */
        long (*buffer)[4];  /* pointer to message buffer */
```

DESCRIPTION

The *q_send* directive sends a message to the queue identified by the *qid*.

If a task is already waiting at the queue, the message is copied to that task's indicated receiving buffer. The waiting task is then made ready. If there is no task waiting, the message is copied to a system message buffer which is then placed at the end of the message queue.

Once sent, the task's message buffer may be reused immediately. A message is fixed length, 16-bytes.

The message queue may exist on the local processor or any remote processor in a multiprocessor configuration, as long as the queue was created with the GLOBAL flags value set (see *q_create*).

RETURN VALUE

If the *q_send* directive successfully sent a message, then 0 is returned.

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

ERROR CONDITIONS

Message *qid* is invalid.

Out of system message buffers.

Message queue at maximum count.

ISR cannot reference remote node.

NOTES

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

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

3.2.9 Q_URGENT

NAME

q_urgent -- "Place an Urgent Message at the Head of a Message Queue"

SYNOPSIS

```
#include <message.h>
uint q_urgent ( qid, buffer )

        uint qid;           /* message queue id returned from q_create or q_ident */
        long (*buffer)[4];  /* pointer to message buffer */
```

DESCRIPTION

The *q_urgent* directive sends a message to the queue identified by the *qid*. This call is the same as the *q_send* call, except, if there are other messages at the queue, this message is put at the head of the queue.

If a task is already waiting at the queue, the message is copied to that task's indicated receiving buffer. The task is then made ready. If there is no task waiting, the message is copied to a system buffer which is then placed at the head of the message queue.

Once sent, the task's message area may be reused immediately. A message is fixed length, 16-bytes.

The message queue may exist on the local processor or any remote processor in a multiprocessor configuration, as long as the queue was created with the GLOBAL flags value set (see *q_create*).

RETURN VALUE

If the *q_urgent* directive successfully sent a message, then 0 is returned.

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

ERROR CONDITIONS

Message *qid* is invalid.

Out of system message buffers.

Message queue at maximum count.

ISR cannot reference remote node.

NOTES

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

May cause a preempt if a task waiting at the message queue has a higher priority than the running task, and the preempt mode is in effect. A preempt will not occur if a task waiting exists on

a remote processor in a multiprocessor configuration.

3.2.10 Q_BROADCAST

NAME

q_broadcast -- "Broadcast N Identical Messages to a Message Queue"

SYNOPSIS

```
#include <message.h>
uint q_broadcast ( qid, buffer, &count )

        uint qid;           /* message queue id returned from q_create or q_ident */
        long (*buffer)[4];  /* pointer to message buffer */
        uint count;         /* number of tasks made ready - returned by this call */
```

DESCRIPTION

The *q_broadcast* directive sends as many messages as necessary to make ready all tasks waiting on the queue identified by the *qid*. The number of tasks readied is returned to the caller in *count*.

Once sent, the task's message buffer may be reused immediately.

The message queue may exist on the local processor or any remote processor in a multiprocessor configuration, as long as the queue was created with the GLOBAL flags value set (see *q_create*).

RETURN VALUE

If the *q_broadcast* directive succeeds, the *count* is filled in with the number of tasks readied, and 0 is returned.

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

ERROR CONDITIONS

Message *qid* is invalid.

ISR cannot reference remote node.

NOTES

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

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

3.2.11 Q_RECEIVE

NAME

q_receive -- "Receive a Message from a Message Queue"

SYNOPSIS

```
#include <message.h>
uint q_receive ( qid, buffer, flags, timeout )

        uint qid;           /* message queue id returned from q_create or q_ident */
        long (*buffer)[4];  /* pointer to message buffer */
        uint flags;         /* options */
        uint timeout;       /* number of ticks to wait */
                            /* 0 indicates wait forever */
```

The flags values are:

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

DESCRIPTION

The *q_receive* directive allows a task to request a message from the message queue identified by *qid*.

If there is a message at the message queue, it is copied into the requester's buffer.

If there is no message at the message queue, then the NOWAIT flag determines what to do. If the NOWAIT flags value is set, the task returns immediately with -1 and the no message at queue error number. If the NOWAIT flags value is clear, the task is put on a wait list for the message queue, according the queue's attributes (FIFO or priority).

The *timeout* field is used to determine how long to wait. A zero in the *timeout* field indicates no timeout -- wait forever. A non-zero entry in the *timeout* field indicates that the task will run after that many ticks, if a message has not been received, or before if a message is received.

When *q_receive* is called from an ISR, the no wait option is forced by the executive. Thus there will be no waiting for a message. An error will be returned if there is no message.

The message queue may exist on the local processor or any remote processor in a multiprocessor configuration, as long as the queue was created with the GLOBAL flags value set (see *q_create*).

RETURN VALUE

If the *q_receive* directive succeeds, then 0 is returned.

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

ERROR CONDITIONS

Message *qid* is invalid.

No message at queue (if no wait is selected).

Message queue deleted.

Timed out with no message (if wait and timeout is selected).

ISR cannot reference remote node.

NOTES

Can be called from within an ISR, except when the queue was not created from the local node. The executive will force the options to no wait.

The requesting task may be blocked if there is no message available, and the wait option is selected.

3.2.12 EV_SEND

NAME

ev_send – "Send Event to a Task"

SYNOPSIS

uint ev_send (tid, event)

```
uint tid;      /* task id as returned by t_create or t_ident */
uint event;    /* event set */
```

DESCRIPTION

The *ev_send* directive sends an event to a task. The *event* field describes the set of events the task wishes to send. Thirty-two events are available. Sixteen are available as *system* events and sixteen are available as *user* events.

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

Events sent to tasks not waiting for an event are left pending.

RETURN VALUE

If the *ev_send* directive succeeds, then 0 is returned.

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

ERROR CONDITIONS

Invalid *tid*.

ISR cannot reference remote node.

NOTES

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

May cause a preempt if the task waiting for the event 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.

3.2.13 EV_RECEIVE

NAME

ev_receive -- "Receive Event"

SYNOPSIS

uint ev_receive (eventin, flags, timeout, &eventout)

```
uint eventin;    /* input event condition */
uint flags;      /* options */
uint timeout;    /* number of ticks to wait */
                 /* 0 indicates wait forever */
uint eventout;   /* output events - returned by this call */
```

The flags values are:

NOWAIT	set	if the task is to return immediately
	clear	if the task is to wait for event condition
ANY	set	return when any one of the indicated events has occurred
	clear	return when all of the indicated events have occurred

DESCRIPTION

The *ev_receive* directive allows a task to receive an event condition. The event condition to receive is a set of events specified in the *eventin* field.

The task may elect to wait for the event condition, or return immediately by setting the NOWAIT value in the flags field. The task may elect to receive all of the events, or receive any one of them by setting the ANY value in the flags field.

When pending events satisfy the event condition, the events are cleared and the task will remain running. Otherwise, if the task elects to wait, the task will become blocked. The task will be made ready to run when the event condition is satisfied by new events, or the timeout condition is met.

When pending events do not satisfy the event condition, and the task elects not to wait, the task returns immediately with -1 and the no event available error number.

If the *eventin* field is 0, *ev_receive* will return the pending events, but the events will remain pending.

The *timeout* field is used to determine how long to wait. A zero in the *timeout* field indicates no timeout -- wait forever. A non-zero entry in the *timeout* field indicates that the task will run after that many ticks, if the event condition is not satisfied, or before if the event condition is satisfied.

RETURN VALUE

If the *ev_receive* directive succeeds, *eventout* is filled in with the output events, and 0 is returned.

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

ERROR CONDITIONS

Event not satisfied (if no wait is selected).

Timed out with no event (if wait and timeout is selected).

NOTES

Cannot be called from within an ISR.

The requesting task may be blocked if the event condition is not satisfied, and the wait option is selected.

3.2.14 AS_CATCH

NAME

as_catch -- "Catch Signals"

SYNOPSIS

uint as_catch (asraddr, mode)

```
ptf asraddr;    /* address of Asynchronous Signal Routine (asr) */
                /* 0 indicates asr is invalid */
uint mode;      /* mode value for asr */
```

The *mode* value is defined as follows:

NOPREEMPT	set	to disable preempting
	clear	to enable preempting
TSLICE	set	to enable timeslicing
	clear	to disable timeslicing
DISASR	set	to disable asr processing
	clear	to enable asr processing
SUPV	set	to execute in supervisor mode
	clear	to execute in user mode
LEVEL		interrupt level when SUPV is set

DESCRIPTION

The *as_catch* directive allows a task to specify what action to take when catching signals.

The asr address is established when *as_catch* is called with a non-zero address in the *asraddr* field. Zero is not a valid asr address. The asr is invalidated when *as_catch* is called with the *asraddr* field equal zero. Asynchronous signal processing will be discontinued until re-enabled with a valid asr address in another *as_catch* call.

When a signal is caught, the task is not unblocked. Signals are latched until the task becomes the running task, at which time the task is dispatched to its asr. The task will execute the asr according to the values specified in the *mode* field. The signal condition will be passed to the task, along with the the task's current PC and mode, on the task's stack in a signal stack frame. The signal condition contains all of the signals which have been received since the last time the task was executing.

The asr is responsible for saving and restoring all registers it uses.

The *as_return* directive must be executed to return the task to its previous dispatch address.

Only one asr per task is allowed.

RETURN VALUE

The *as_catch* directive always succeeds, and returns 0.

ERROR CONDITIONS

None.

NOTES

Cannot be called from within an ISR.

Will not cause a preempt.

3.2.15 AS_SEND

NAME

as_send -- "Send Signal to a Task"

SYNOPSIS

uint as_send (tid, signal)

```
uint tid;      /* task id as returned by t_create or t_ident */
uint signal;   /* signal set */
```

DESCRIPTION

The *as_send* directive sends signals to a task. The *signal* field describes the set of signals it wishes to send. Thirty-two signals are available. Sixteen are available as *system* signals and sixteen are available as *user* signals.

The signal set must be sent to tasks which have specified an asr using the *as_catch* directive. If the task identified by the *tid* does not have a valid asr, the caller returns with the invalid asr error.

When a signal is sent to a task with a valid and enabled asr, the task will be dispatched to the asr address when it becomes the running task. Signals sent to a blocked task are latched until the task becomes the running task. Duplicate signals are not queued.

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

RETURN VALUE

If the *as_send* directive successfully sent the signal, then 0 is returned.

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

ERROR CONDITIONS

Invalid *tid*.

Invalid asr.

ISR cannot reference remote node.

NOTES

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

3.2.15 AS_SEND

NAME

as_send -- "Send Signal to a Task"

SYNOPSIS

uint as_send (tid, signal)

```
uint tid;      /* task id as returned by t_create or t_ident */
uint signal;   /* signal set */
```

DESCRIPTION

The *as_send* directive sends signals to a task. The *signal* field describes the set of signals it wishes to send. Thirty-two signals are available. Sixteen are available as *system* signals and sixteen are available as *user* signals.

The signal set must be sent to tasks which have specified an asr using the *as_catch* directive. If the task identified by the *tid* does not have a valid asr, the caller returns with the invalid asr error.

When a signal is sent to a task with a valid and enabled asr, the task will be dispatched to the asr address when it becomes the running task. Signals sent to a blocked task are latched until the task becomes the running task. Duplicate signals are not queued.

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

RETURN VALUE

If the *as_send* directive successfully sent the signal, then 0 is returned.

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

ERROR CONDITIONS

Invalid *tid*.

Invalid asr.

ISR cannot reference remote node.

NOTES

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

3.2.16 AS_RETURN

NAME

as_return -- "Return from Signal Routine"

SYNOPSIS

void as_return ( )

DESCRIPTION

The *as_return* must be used by tasks to return from an asynchronous signal routine (asr).

RETURN VALUE

None.

ERROR CONDITIONS

Not in asr.

NOTES

This call is only used to return from an asr. Refer to the *as_catch* and *as_send* directives.

3.3 Semaphore Management

The semaphore manager provides a set of directives to use in arbitrating access to a shared resource (many-to-one). The semaphores primitives provided can be used to fulfill different sets of requirements:

1. To control access to a single resource that is either available or not, the user can create a semaphore with an initial value of 1.

2. To control access to a pool of "n" resources where at any moment "m" of those resources are available (0 <= m <= n) and "n-m" are not, the user can create a semaphore with an initial value of "n".

Arbitrating access to shared resources requires signaling that a predefined event has occurred. Sophisticated synchronisation also requires a counter to record the number of events sent but not yet received, and a list of tasks awaiting receipt of the event.

The semaphore data structure fulfills all the previous requirements. A semaphore possesses a name to distinguish it from the other semaphores within the system, a semaphore id to enable quick access to the semaphore, the requisite semaphore count variable to count the events, and a list of waiting tasks. In addition to the semaphore count variable, the semaphore contains an initial count, used as an initial assignment value for the semaphore count.

The synchronisation rules for semaphores ar :

1. The semaphore count is decremented by 1, when a task does a *sm_p* operation. The task continues execution if the count is then greater than or equal to zero. If the count is less than zero, the task is put on a waiting list for the semaphore.

2. The semaphore count is incremented by one when a task does a *sm_v* operation. If the count is less than or equal to zero, the first task in the semaphore waiting list is placed in the ready state.

The directives provided by the semaphore manager are:

Directive	Function
sm_create	Get a semaphore
sm_ident	Obtain the id of a Semaphore
sm_delete	Delete a semaphore
sm_p	Access semaphore
sm_v	Release semaphore

3.3.1 SM_CREATE

NAME

sm_create – "Create a Semaphore"

SYNOPSIS

```
#include <semaphore.h>
uint sm_create ( name, count, flags, &smid )

        uint name;    /* semaphore name */
        uint count;   /* initial count */
        uint flags;   /* semaphore flags */
        uint smid;    /* semaphore id - returned by this call */
```

The flags field values are:

PRIOR	set	to process wait list by priority
	clear	to process wait list by FIFO
GLOBAL	set	to indicate the semaphore is a multiprocessor global resource.
	clear	to indicate the semaphore is local.

DESCRIPTION

The *sm_create* directive creates a semaphore and assigns it an initial count equal to the value in the *count* field. The semaphore id is returned in *smid*. The *smid* must be used in subsequent *sm_p*, *sm_v*, and *sm_delete* calls.

By setting the PRIOR value in the flags field, tasks waiting on a semaphore will be processed in task priority order. Otherwise the tasks will be processed in first in, first out (FIFO) order.

By setting the GLOBAL value in the flags field, the *smid* will be sent to all processors in the system, to be entered into a global resource table. The system is defined as the collection of interconnected processors. The semaphore is always created on the local node.

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

RETURN VALUE

If *sm_create* successfully created the semaphore, the *smid* is filled in, and 0 is returned.

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

ERROR CONDITIONS

Too many semaphores.

NOTES

Not callable from ISR.

Will not cause a preempt.

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

Not callable from ISR.

Requester will be blocked when the wait option is selected and the memory is not available.

3.7.7 RN_RETSEG

NAME

rn_retseg -- "Return a Segment"

SYNOPSIS

```
#include <memory.h>
uint rn_retseg ( rnid, segaddr )

        uint rnid;      /* region id as returned by rn_create or rn_ident */
        char *segaddr;  /* segment address as returned by rn_getseg */
```

DESCRIPTION

This directive returns a segment to its region. If possible, the segment is merged with neighboring segments. The resulting segment then becomes available for subsequent allocation, or allocation to tasks already waiting.

RETURN VALUE

If *rn_retseg* successfully returned the segment, then 0 is returned.

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

ERROR CONDITIONS

Invalid *rnid*.

Segment not from specified region.

NOTES

Not callable from ISR.

May cause a preempt if a task waiting for memory becomes ready as a result of this call and has a higher priority than the running task, and the preempt mode is in effect.

3.7.8 PT_CREATE

NAME

pt_create – "Create a Partition"

SYNOPSIS

```
#include <memory.h>
uint pt_create ( name, paddr, length, bsize, flags, &ptid, &bnum)

        uint name;      /* user defined 4-byte partition name */
        char *paddr;    /* physical start address of partition */
        uint length;    /* physical length in bytes */
        uint bsize;     /* size of buffers in bytes */
        uint flags;     /* partition attributes */
        uint ptid;      /* partition id - returned by this call */
        uint bnum;      /* number of buffers in partition - returned by this call */
```

Flags field values:

GLOBAL	set	to indicate the partition is a multiprocessor global resource.
	clear	to indicate the partition is local

DESCRIPTION

This directive allows the user to create a partition of fixed size buffers from a contiguous memory area. The partition id will be returned in *ptid* by the executive to use for *pt_getbuf* and *pt_retbuf* directives for the partition. The number of buffers created by the executive will be returned in *bnum*.

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

The executive may use memory within the partition for partition and buffer data structures. Therefore, the product of the buffer count and buffer size will be slightly less than the length of the partition.

By setting the GLOBAL value in the flags field, the *ptid* will be sent to all processors in the system, to be entered into a global resource table. The system is defined as the collection of interconnected processors.

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

RETURN VALUE

If *pt_create* successfully created the partition, the *ptid* and *bnum* are filled in and 0 is returned.

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

ERROR CONDITIONS

Too many partitions.

NOTES

Not callable from ISR.

Will not cause a preempt.

3.7.9 PT_IDENT

NAME

pt_ident -- "Obtain id of a Partition"

SYNOPSIS

```
#include <memory.h>
uint pt_ident ( name, node, &ptid )

        uint name;    /* user defined 4-byte partition name */
        uint node;    /* node identifier */
                      /* 0 indicates any node */
        uint ptid;    /* partition id - returned by this call */
```

DESCRIPTION

This directive allows a task to identify a previously created partition by name and obtain the *ptid* to use for *pt_getbuf* and *pt_retbuf* directives for the partition.

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

The partition may have been created by the local processor or any remote processor in a multiprocessor configuration, as long as the partition was created with the GLOBAL flags value set (see *pt_create*). If the partition name is not unique within the multiprocessor configuration, a non-zero node identifier must be specified in the *node* field.

RETURN VALUE

If *pt_ident* directive succeeds, the *ptid* will be filled in and 0 is returned.

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

ERROR CONDITIONS

Named partition does not exist.

Invalid node identifier.

NOTES

Can be called from within an ISR.

Will not cause a preempt.

3.7.10 PT_DELETE

NAME

pt_delete – "Delete a Partition"

SYNOPSIS

```
#include <memory.h>
uint pt_delete ( ptid )

        uint ptid;    /* partition id as returned by pt_create or pt_ident */
```

DESCRIPTION

This directive removes a partition, provided that none of its buffers is still allocated.

After this directive has successfully executed, the executive will reject any *pt_getbuf* or *pt_retbuf* directives for the partition.

The partition must exist on the local processor. If the partition 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 *ptid* can be deleted from the global resource table.

RETURN VALUE

If *pt_delete* successfully removed the partition, then 0 is returned.

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

ERROR CONDITIONS

Invalid *ptid*.

Cannot delete – some buffers in use.

Partition not created from local node.

NOTES

Not callable from ISR.

Will not cause a preempt.

3.7.11 PT_GETBUF

NAME

pt_getbuf -- "Get a Buffer"

SYNOPSIS

```
#include <memory.h>
uint pt_getbuf ( ptid, &bufaddr )

        uint ptid;          /* partition id as returned by pt_create or pt_ident */
        char *bufaddr;      /* buffer address - returned by this call */
```

DESCRIPTION

The *pt_getbuf* directive will get a buffer from a buffer partition. The buffer address will be returned in *bufaddr* as a result of this call.

The partition may have been created by the local processor or any remote processor in a multiprocessor configuration, as long as the partition was created with the **GLOBAL** flags value set (see *pt_create*).

RETURN VALUE

If *pt_getbuf* successfully got a buffer, then the address of the buffer is returned in *bufaddr* and 0 is returned.

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

ERROR CONDITIONS

Invalid *ptid*.

Partition out of free buffers.

NOTES

Can be called from within an ISR.

Will not cause a preempt.

3.7.12 PT_RETBUF

NAME

pt_retbuf – "Return a Buffer"

SYNOPSIS

```
#include <memory.h>
uint pt_retbuf ( ptid, bufaddr )

        uint ptid;          /* partition id as returned by pt_create or pt_ident */
        char *bufaddr;      /* buffer start address as returned by pt_getbuf */
```

DESCRIPTION

The *pt_retbuf* directive will return a buffer to the partition from which it was originally allocated.

Buffers are not automatically released when a task is deleted.

RETURN VALUE

If *pt_retbuf* successfully returned the buffer, then 0 is returned.

If the buffer was not returned, a value of -1 is returned, and *errno* is set to indicate the error.

ERROR CONDITIONS

Invalid *ptid*.

Buffer not from specified partition.

NOTES

Can be called from within an ISR.

Will not cause a preempt.

3.8 MMU Management

The executive can optionally support the PMMU (M68851 and M68030) to provide memory protection, dynamic task loading, and dynamic memory allocation.

To provide these services, the executive adopts an MMU model which defines the *pagesize*, the structure and depth of the memory map tree, and the degree of control each task has over its own memory map. Different implementations of the RTEID are free to choose different models. However, the model chosen should allow the standard memory management services (regions and partitions) to operate in a consistent and intuitive manner in both an MMU and non-MMU environment.

Logically, the RTEID adopts a sectioned view of the logical address space associated with each task. Memory objects are mapped into a task's logical address space in variable size MMU sections. A single section is contiguous in the logical and possibly the underlying physical address spaces. Thus, the MMU is used to define a set of mappings for each task in the form:

(logical address, length) --> physical address range

Based on this model, the RTEID defines how the memory management services should operate, and defines additional services to manage the MMU directly.

3.8.1 Segments vs. Sections

MMU sections should not be confused with region segments. A segment is a block of memory allocated from a region. It can exist on any CPU in the M68000 family. A section is only meaningful on the M68030 or M68020/M68851 combination, and refers to a contiguous block of memory which is mapped into a task's address space.

3.8.2 Regions

When a task calls *rn_getseg* to obtain a segment from a region, the segment is automatically mapped into the task's logical address space at an executive assigned address. Because *rn_getseg* performs the mapping, the corresponding region is not mapped into the address space of tasks using it. This means that allocated sections are accessible only by the allocating task, and those tasks which explicitly are given access to the segment using the MMU directives. Thus, a segment is fully protected from inadvertent access by other tasks.

3.8.3 Partitions

When a task executes a *pt_create* or *pt_ident* directive, the entire partition is mapped into the task's address space. Thus, tasks which share a partition can share and access any buffers allocated from the partition. However, protection is on the partition level, and individual buffers are not protected.

The directives provided by the MMU manager are:

Directive	Function
mm_l2p	Logical to physical
mm_p2l	Physical to logical
mm_pmap	Map physical
mm_unmap	Unmap logical
mm_pread	Physical read
mm_pwrite	Physical write
mm_ptcreate	Create logical partition

3.8.4 MM_L2P

NAME

mm_l2p -- "Logical to Physical"

SYNOPSIS

```
#include <memory.h>
uint mm_l2p ( tid, laddr, &paddr, &length )

        uint tid;       /* task id as returned by t_create or t_ident */
        char *laddr;    /* logical start address */
        char *paddr;    /* physical start address - returned by this call */ */
        uint length;    /* remaining length in bytes - returned by this call */
```

DESCRIPTION

This directive calculates the physical address within the section associated with the logical address belonging to the task identified by the *tid*.

The physical start address is returned in the *paddr* field. The number of bytes remaining in the section is returned in the *length* field.

RETURN VALUE

If *mm_l2p* was successful, then the physical start address is returned in *paddr*, the number of bytes remaining is returned in *length*, and 0 is returned.

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

ERROR CONDITIONS

Invalid *tid*.

Unmapped logical address.

Task not created on local node.

ISR cannot reference remote node.

NOTES

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

Will not cause a preempt.

3.8.5 MM_P2L

NAME

mm_p2l – "Physical to Logical"

SYNOPSIS

```
#include <memory.h>
uint mm_p2l ( tid, paddr, &laddr, &length )

        uint tid;       /* task id as returned by t_create or t_ident */
        char *paddr;    /* physical start address */
        char *laddr;    /* logical start address - returned by this call */ */
        uint length;    /* remaining length in bytes - returned by this call */
```

DESCRIPTION

This directive returns the logical address within the section associated with the physical address belonging to the task identified by the *tid*. The executive will only return the first valid mapping of the physical address it finds, and the logical address returned may be ambiguous if the task has a many-to-one mapping of the physical address range.

The logical start address is returned in the *laddr* field, and the number of bytes remaining in the section is returned in the *length* field.

RETURN VALUE

If *mm_p2l* was successful, then the logical address is returned in *laddr*, the number of bytes remaining is returned in *length*, and 0 is returned.

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

ERROR CONDITIONS

Invalid *tid*.

Unmapped logical address.

Task not created on local node.

NOTES

Not callable from ISR.

Will not cause a preempt.

3.8.6 MM_PMAP

NAME

mm_pmap -- "Map Physical"

SYNOPSIS

```
#include <memory.h>
uint mm_pmap ( tid, laddr, paddr, length, flags )

        uint tid;       /* task id as returned by t_create or t_ident */
        char *laddr;    /* logical start address */
        char *paddr;    /* physical start address */
        uint length;    /* length in bytes */
        uint flags;     /* section attributes */
```

The flags field values are defined as follows:

RDONLY	set	read-only
	clear	read-write

DESCRIPTION

This directive maps physical memory starting at *paddr* for the number of bytes specified in *length,* to a section at the logical start address *laddr* in the address space of the task identified by the *tid.*

The physical start address specified in *paddr* must be on the pagesize boundary. The logical start address specified in *laddr* must be on a section boundary.

If *length* is not a multiple of the pagesize, then more bytes than requested are mapped.

RETURN VALUE

If *mm_pmap* was successful, then 0 is returned.

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

ERROR CONDITIONS

Invalid *tid.*

Paddr is not on a pagesize boundary.

Laddr is not on a section boundary.

Length specified is too large.

Duplicate logical address.

Task not created on local node.

ISR cannot reference remote node.

NOTES

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

Will not cause a preempt.

3.8.7 MM_UNMAP

NAME

mm_unmap -- "Unmap Logical"

SYNOPSIS

```
#include <memory.h>
uint mm_unmap ( tid, laddr )

        uint tid;       /* task id as returned by t_create or t_ident */
        char *laddr;    /* logical start address */
```

DESCRIPTION

This directive removes the section starting at logical address *laddr* from the address space of the task identified by the *tid*.

RETURN VALUE

If *mm_unmap* was successful, then 0 is returned.

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

ERROR CONDITIONS

Invalid *tid*.

Unmapped logical address.

Task not created on local node.

ISR cannot reference remote node.

NOTES

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

Will not cause a preempt.

To return the segment to the region, the *rn_retseg* directive must be used.

3.8.8 MM_PREAD

NAME

mm_pread – "Physical read"

SYNOPSIS

```
#include <memory.h>
uint mm_pread ( paddr, laddr, length )

        uint paddr;     /* physical start address */
        char *laddr;    /* logical start address */
        uint length;    /* length in bytes */
```

DESCRIPTION

The *mm_pread* directive reads from a physical address, and writes to the logical address in the calling task's address space. The length cannot span a section boundary.

RETURN VALUE

If *mm_pread* was successful, then 0 is returned.

If the call was not successful, no data is transferred and an error code is returned.

ERROR CONDITIONS

Unmapped logical address.

Length spans section boundary.

NOTES

Not callable from ISR.

Will not cause a preempt.

3.8.9 MM_PWRITE

NAME

mm_pwrite -- "Physical write"

SYNOPSIS

```
#include <memory.h>
uint mm_pwrite ( paddr, laddr, length )

        uint paddr;     /* physical start address */
        char *laddr;    /* logical start address */
        uint length;    /* length in bytes */
```

DESCRIPTION

The *mm_pwrite* directive reads from the logical address in the calling task's address space, and writes to a physical address. The length may not span a section boundary.

RETURN VALUE

If *mm_pwrite* was successful, then 0 is returned.

If the call was not successful, no data is transferred and an error code is returned.

ERROR CONDITIONS

Unmapped logical address.

Length spans section boundary.

NOTES

Not callable from ISR.

Will not cause a preempt.

3.8.10 MM_PTCREATE

NAME

mm_ptcreate -- "Create a Logical Partition"

SYNOPSIS

```
#include <memory.h>
uint mm_ptcreate ( name, paddr, length, bsize, laddr, flags, &ptid, &bnum)

        uint name;      /* user defined 4-byte partition name */
        char *paddr;    /* physical start address of partition */
        uint length;    /* physical length in bytes */
        uint bsize;     /* size of buffers in bytes */
        char *laddr;    /* physical start address of partition */
        uint flags;     /* partition attributes */
        uint ptid;      /* partition id - returned by this call */
        uint bnum;      /* number of buffers in partition - returned by this call */
```

Flags field values:

GLOBAL	set	to indicate the partition is a multiprocessor global resource.
	clear	to indicate the partition is local

DESCRIPTION

This directive allows the user to create a logical partition of fixed size buffers from a contiguous memory area. The partition is mapped into the caller's address space at the logical address specified in *laddr*. By creating logical partitions at the same logical addresses, partitions can be easily shared between processors.

The partition id will be returned in *ptid* by the executive to use for *pt_getbuf* and *pt_retbuf* directives for the partition.

The partition physical start address must be on the pagesize boundary.

The number of buffers created by the executive will be returned in *bnum*. The executive may use memory within the partition for partition and buffer data structures. Therefore, the product of the buffer count and buffer size will be slightly less than the length of the partition.

By setting the GLOBAL value in the flags field, the *ptid* will be sent to all processors in the system, to be entered into a global resource table. The system is defined as the collection of interconnected processors.

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

RETURN VALUE

If *mm_ptcreate* successfully created the partition, the *ptid* and *bnum* are filled in and 0 is returned.

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

ERROR CONDITIONS

Too many partitions.

NOTES

Not callable from ISR.

Will not cause a preempt.

3.9 Dual-ported Memory

Dual-ported memory is commonly found in multiprocessor systems. The executive provides a method for converting internal addresses to external, and external addresses to internal, to accommodate the use of dual-ported memory and allow tasks to exchange addresses between processors.

The *internal* address will be defined as the address of a memory resource, relative to the local node which needs to access the memory resource.

The *external* address will be defined as the address of a memory resource, relative to a remote node which needs to access the memory resource.

The directives provided for dual-ported memory are:

Directive	Function
m_ext2int	Convert external address
m_int2ext	Convert internal address

3.9.1 M_EXT2INT

NAME

m_ext2int -- "Convert external address to internal address"

SYNOPSIS

uint m_ext2int (external, &internal)

```
char *external;    /* external address */
char *internal;    /* internal address - returned by this call */
```

DESCRIPTION

The *m_ext2int* call is used to convert the physical address contained in *external* into an internal address, so it can be used by the local node. The internal address is returned to the caller in *internal.*

The external (VMEbus) address is normally an address received by the local node, and the requester may not know whether its internal (local) or not. If the address contained in *external* is internal, the returned address will be same as the address in *external.*

RETURN VALUE

The *m_ext2int* directive always succeeds, the internal address is returned in *internal,* and 0 is returned.

ERROR CONDITIONS

None.

NOTES

Can be called from within an ISR.

Will not cause a preempt.

In a MMU system, a task will need to execute *mm_p2l* following this call to obtain a logical internal address.

3.9.2 M_INT2EXT

NAME

m_int2ext -- "Convert internal address to external address"

SYNOPSIS

uint m_int2ext (internal, &external)

```
char *internal;    /* internal address */
char *external;    /* external address - returned by this call */
```

DESCRIPTION

The *m_int2ext* call is used to convert the physical address contained in *internal* into an external address, so it can pass the address to a remote node within the system. The external address is returned to the requester in *external.*

The internal address is a physical address accessible by the local node within its dual-ported memory, and the external (VMEbus) address will be different.

RETURN VALUE

The *m_int2ext* directive always succeeds, the external address is returned in *external,* and 0 is returned.

ERROR CONDITIONS

None.

NOTES

Can be called from within an ISR.

Will not cause a preempt.

In a MMU system, a task will need to execute *mm_l2p* preceding this call to obtain a physical address.

4. I/O INTERFACE

This section describes a set of I/O Interface services for the RTEID. These services provide a well defined mechanism for installing and calling device drivers. They provide a structured methodology for writing drivers which both simplifies and assists in the development of drivers and enhances their portability between RTEID based ~~system~~ systems. The RTEID does not make any assumptions about the construction or operation of a driver itself.

The directives provided by the I/O Interface are:

Directive	Description
de_init	Initialize a device driver
de_open	Open a device for I/O
de_close	Close a device
de_read	Read from a device
de_write	Write to a device
de_cntrl	Special device services

4.1 Driver Properties

Device drivers shall have the following properties:

1. A driver is always called by a task and is considered to run on behalf of the task which called it.
2. A driver can make any and all RTEID calls, including additional I/O calls. I/O calls may not be called from within the driver's ISR.
3. If the driver makes a blocking service call, (e.g. *q_receive*), the calling task blocks.
4. Drivers always execute in supervisor mode regardless of the mode of the caller. Designers should account for driver stack usage when determining supervisor stack sizes for new tasks.
5. A driver may temporarily enter user mode but must return to supervisor mode prior to exiting.
6. Other than item (4) above, drivers retain the *mode* of the calling task. Thus on entry they have the same interrupt mask level, preemption, asr and time-slicing status as the caller. The driver may change any or all of these but is responsible for restoring them prior to exiting.

4.2 Data Structures

The data structures used by drivers which are supported by the I/O Interface are:

- **Driver Address Table**
  - Used by the I/O Interface to locate the driver's INIT, OPEN, CLOSE, READ, WRITE, and CNTRL routines.

- **Device Data Area Table**
  - Used by the I/O Interface to locate the driver's data area for the driver's OPEN, CLOSE, READ, WRITE, and CNTRL routines.

4.2.1 Driver Address Table

When a task makes an I/O Interface call, the executive must locate the driver associated with the specified device (major number) and operation (i.e. READ). It does so via a **Driver Address Table** provided by the user. The physical address of the table and the number of devices are specified to the executive via configuration parameters.

The **Driver Address Table** for a system with N devices can be described by the following declarations:

```
struct drvaddr drvatab[N];

struct drvaddr
{
int (*init_driver)();
int (*open_driver)();
int (*close_driver)();
int (*read_driver)();
int (*write_driver)();
int (*cntrl_driver)();
int resvd1;
int resvd2;
}
```

As shown, the **Driver Address Table** is an array of N structures, one for each device. Each structure contains eight entries. The first six entries contain pointers to functions (routines) within the driver associated with the device. The last two entries are reserved for future use.

4.2.2 Device Data Area Table

Many, if not most, devices need a data area where the device driver can store information specific to the device. Although a statically allocated area can be used, it is usually more convenient to dynamically allocate this area when the device is initialized. The I/O Interface contains services to support such dynamic allocation.

The **Device Data Area Table** is supplied and maintained by the I/O Interface. The table contains one long word entry for each device in the system. The entry is used to maintain the address of the data area for the device.

The device driver's INIT routine is responsible for allocating the device's data area and returning its address to the I/O Interface. This memory can come from any source - static data, a region, or a partition. On exit, the INIT routine must return the address of the data area to the I/O Interface. The I/O Interface saves this address in the **Device Data Area Table**. Whenever a device driver routine (other than INIT) is called, the I/O Interface passes the data area address to the driver.

4.3 Device Initialisation

During system initialisation, the executive automatically calls the driver's INIT routine for each device. They are called sequentially, beginning with device 0 and ending with the last device in the system.

Since drivers can only be called by tasks, the executive calls the driver's INIT routine on behalf of a system initialisation task, defined by configuration parameters. The *mode* of the system initialisation task (also a configuration parameter) is used as the *mode* while the executive calls the INIT routines of the drivers. If the driver's INIT routine makes a RTEID call which blocks, control is passed to an idle task provided by the executive until an interrupt unblocks the driver.

Although the driver's INIT routine is always called at system startup, it may also be called by a task, either to re-initialise a driver or when a new device driver is dynamically loaded.

4.4 Parameter Passing

All directives except *de_init* require a user provided parameter block. The format and content of the parameter block depends on and is determined entirely by the particular driver and device it controls. Its function is to pass input parameters to the driver.

In a system with an MMU, the address of the parameter block is a logical address. The I/O Interface will convert it to a physical address before passing it to the driver. Wit' 'n the parameter block, addresses may be either logical or physical, as defined by the driver. The I/O Interface does not examine or translate any fields within the parameter block.

4.5 I/O Interface in C Language

The I/O Interface may be called in the C language as follows:

Function	Parameters
de_init	(dev)
de_open	(dev, argp, &rval)
de_close	(dev, argp, &rval)
de_read	(dev, argp, &rval)
de_write	(dev, argp, &rval)
de_cntrl	(dev, argp, &rval)

dev is a 32-bit device number formatted as follows:

bits 31-16 = major device number
bits 15-0 = minor device number

argp is a pointer to a parameter block which contains device and operation specific parameters. The format and contents of the block is determined by the driver.

rval is an output parameter in which READ, WRITE and CNTRL routines may return information about the call.

4.6 I/O Interface in Assembly Language

The I/O Interface may be called by loading parameters into specific CPU registers and executing a TRAP instruction. The following assembly language interface is used:

INPUT

D0.W = function number as follows:
1 = INIT
2 = OPEN
3 = CLOSE
4 = READ
5 = WRITE
6 = CNTRL
7 = RESVD1
8 = RESVD2

D1.L = Device number (major and minor)
A0.L = Pointer to parameter block (except INIT)

OUTPUT

D0.L = Error code - 0 indicates successful return
D1.L = Return value from OPEN, CLOSE, READ, WRITE and CNTRL
A1.L = Address of device data area (INIT only)

4.7 Driver Interface in Assembly Language

The I/O Interface calls the user provided driver using the following assembly language convention:

INPUT

D0.L = tid
D1.L = Device number (major and minor)
A0.L = Physical address of parameter block (except INIT)
A1.L = Physical address of device data area (except INIT)

OUTPUT

D0.L = Error code - 0 indicates successful return
D1.L = Return value from OPEN, CLOSE, READ, WRITE and CNTRL
A1.L = Address of device data area (INIT only)

4.8 Error Handling

There are a number of errors which can occur during a driver call. In general, there are two types:

1. Errors detected by the I/O Interface.
2. Errors detected and returned by the driver.

All I/O Interface generated errors are detected prior to calling the driver. In these cases, the I/O supervisor loads register D0 with an error code and returns to the caller without ever passing control to the driver. To distinguish between I/O Interface errors and driver errors, error codes below 10000H (16-bit values) are reserved for use by the I/O Interface. Below is a list of the errors which are detected by the I/O Interface:

Illegal Function Code
Illegal Major Device Number
Illegal to call driver from ISR
Illegal parameter block address (MMU version only)

Drivers should always return error codes which are greater than 10000H (non-zero in the upper 16-bits).

Error codes returned from the driver's INIT routine are ignored by the executive. If a driver's INIT encounters a fatal error during system startup, the *k_fatal* directive may be used.

4.9 I/O Interface Routines in C Language

The I/O Interface routines as called in the C language are described in the following pages.

4.9.1 INIT

NAME

de_init – "Initialise a Device Driver"

SYNOPSIS

```
uint de_init ( dev )

uint dev;    /* 32-bit device number */
```

DESCRIPTION

The INIT routine will be called during system initialization. The function of INIT is to setup the hardware as necessary and to initialise the driver dependent variables. If the driver needs to allocate a data area for its use, it would do so in the INIT routine. The address of this data area is saved in the Device Data Area Table by the I/O Interface.

RETURN VALUE

If the call succeeds, then 0 is returned.

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

ERROR CONDITIONS

To be defined.

NOTES

Not callable from ISR.

4.9.2 OPEN

NAME

de_open -- "Open a device for I/O"

SYNOPSIS

uint de_open (dev, argp, &rval)

```
uint dev;       /* 32-bit device number */
char *argp;     /* Address of parameter block */
uint rval;      /* rval - returned by this call */
```

DESCRIPTION

The OPEN routine is generally used in conjunction with the CLOSE routine. An example is the implementation of mutual exclusion. OPEN can define the start of a task's exclusive access to a device.

RETURN VALUE

If the call succeeds, *rval* may be filled in, and 0 is returned.

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

ERROR CONDITIONS

To be defined.

NOTES

Not callable from ISR.

4.9.3 CLOSE

NAME

de_close -- "Close a device"

SYNOPSIS

```
uint de_close ( dev, argp, &rval )

uint dev;       /* 32-bit device number */
char *argp;     /* Address of parameter block */
uint rval;      /* rval - returned by this call */
```

DESCRIPTION

The CLOSE routine is generally used in conjunction with the OPEN routine. An example is the implementation of mutual exclusion. CLOSE can define the end of a task's exclusive access to a device.

RETURN VALUE

If the call succeeds, *rval* may be filled in, and 0 is returned.

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

ERROR CONDITIONS

To be defined.

NOTES

Not callable from ISR.

4.9.4 READ

NAME

de_read -- "Read from a device"

SYNOPSIS

uint de_read (dev, argp, &rval)

```
uint dev;       /* 32-bit device number */
char *argp;     /* Address of parameter block */
uint rval;      /* rval - returned by this call */
```

DESCRIPTION

The READ routine transfers data from a device to a user's buffer.

RETURN VALUE

If the call succeeds, *rval* may be filled in, and 0 is returned.

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

ERROR CONDITIONS

To be defined.

NOTES

Not callable from ISR.

4.9.5 WRITE

NAME

de_write – "Write to a device"

SYNOPSIS

```
uint de_write ( dev, argp, &rval )

uint dev;       /* 32-bit device number */
char *argp;     /* Address of parameter block */
uint rval;      /* rval - returned by this call */
```

DESCRIPTION

The WRITE routine transfers data from a user's buffer to a device.

RETURN VALUE

If the call succeeds, *rval* may be filled in, and 0 is returned.

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

ERROR CONDITIONS

To be defined.

NOTES

Not callable from ISR.

4.9.6 CNTRL

NAME

de_cntrl -- "Special device services"

SYNOPSIS

```
uint de_cntrl ( dev, argp, &rval )

uint dev;       /* 32-bit device number */
char *argp;     /* Address of parameter block */
uint rval;      /* Return value - returned by this call */
```

DESCRIPTION

The function of the CNTRL routine is driver dependent. For serial I/O drivers, this routine can define tty parameters such as the baud rate. For a disk driver, the CNTRL routine can include disk formatting.

RETURN VALUE

If the call succeeds, *rval* may be filled in, and 0 is returned.

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

ERROR CONDITIONS

To be defined.

NOTES

Not callable from ISR.

4.10 Driver Interface in C Language

In addition to the assembly language interface, a driver may support the C language interface as defined below:

Function	Parameters
dx_init	(dev, &datap, tid)
dx_open	(dev, pargp, datap, tid, &rval)
dx_close	(dev, pargp, datap, tid, &rval)
dx_read	(dev, pargp, datap, tid, &rval)
dx_write	(dev, pargp, datap, tid, &rval)
dx_cntrl	(dev, pargp, datap, tid, &rval)

dev is the 32-bit device number.

bits 31-16 = major device number
bits 15-0 = minor device number

pargp is the physical address of the parameter block which contains device and operation specific parameters. The format and contents of the block is determined by the driver.

datap is the physical address of the device's data area for all calls except INIT. *datap* is an output parameter in which the INIT routine returns the address of the device's data area.

tid is the task id of the calling task.

rval is an output parameter in which READ, WRITE and CNTRL routines may return completion information about the call.