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 Munz - 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).

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_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