Debug Extension

to the

Real Time Executive Interface Definition

DRAFT 2.0

Prepared by:

MOTOROLA Microcomputer Division

Abstract:

This specification defines a basic set of functions that constitute the Debug Extension to the Real Time Executive Interface Definition. Draft 2.0 is for public review. MOTOROLA retains the right to modify this definition as appropriate during implementation. Draft 2.0 will be submitted to the VITA technical committee no later than 01/25/88.

PRELIMINARY

September 9, 1988

DISCLAIMER

This document is prepared by Motorola Inc., Microcomputer Division. Interest in the Debug Extension to the RTEID is welcome 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)

Debug Ex

1.

TABLE OF CONTENTS

LIST OF FIGURES

REVISION RECORD		
Issue	**Revision Description**	**Date**
1	Initial version. Internal Only.	06/01/87
2	Draft 2.0, limited distribution.	01/25/88
3		

1. DEBUG EXTENSIONS

The debug extensions to the RTEID support several features targeted for use in debugging tasks and interrupt service routines (ISR's). Since debugging is inherently non-real time, systems running under debug control may not exhibit true real time performance.

1.1 Debugging Tasks

Most debugging can be performed by debugging a task or a collection of tasks. In this type of debugging, the actual debug task can reside on the local cpu, or it can be remote if the appropriate **GLOBAL** flags are set.

1.1.1 Controlling Tasks

The relationship between the debug task and the task being debugged is established using the *db_control* directive in the "set" mode. The task issuing the *db_control* directive in the set mode must provide a message queue. This message queue is used to communicate between the executive and the task that issued the *db_control* directive. After completion of the *db_control* directive, the task being debugged becomes controlled, and cannot compete for processor time unless directed to execute by the debug task using the *db_unblock* directive. The *db_block* directive is used to block execution of the controlled task. The *db_control* directive in the "clear" mode is used to terminate the relationship between the debug task and the controlled task.

1.1.2 Read/Write Memory

To read and write memory belonging to the controlled task the pair of directives *db_getmem* and *db_setmem* are provided. *Db_getmem* reads memory from an address of the controlled task and copies it to a buffer provided by the debug task for a length specified by the debug task. *Db_setmem* writes memory to an address of the controlled task copying it from a buffer provided by the debug task for a length specified by the debug task.

1.1.3 Read/Write Registers

To read and write the processor registers belonging to the controlled task the pair of directives *db_getreg* and *db_setreg* are provided. *Db_getreg* reads a register belonging to the controlled task and copies it to a buffer provided by the debug task. *Db_setreg* writes to a register belonging to the controlled task by copying it from a buffer provided by the debug task.

1.1.4 Exceptions in Tasks

When a controlled task issues an exception, such as a bus error, the executive will prevent further execution by placing the controlled task in a *blocked* state. The executive will also format a message containing information about the exception and place it on the message queue identified by the debug task in the *db_control* directive.

1.1.5 The debug_msg message queue

The executive requires the ability to inform the debug task about abnormal activity that occurs when a controlled task executes. This is done by using a message queue specified by the debug task when the *db_control* directive is issued. This message queue is used to pass information from the executive to the debug task. When a controlled task is running and suffers an exception, the executive will block further execution of the task, and inform the debug task of the exception by posting a message on the *debug_msg* queue. The format of the message is:

Bytes	Meaning
0..3	Task id of task causing exception.
4..7	Exceptions vector offset.
8..11	Address of the Exception Stack Frame
12..15	Program counter at the point of the exception

1.1.6 Trace and Breakpoint

A fundamental feature in debugging a task or ISR is the ability to control its execution. This is typically done either by causing the controlled task to single step one instruction, or by having the controlled task execute up to a particular breakpoint. With the debug extensions to the RTEID, a debugger can provide these features.

1.1.6.1 Trace

In order to single step, or trace, a controlled task, the debugger must manipulate the status register of the controlled task, cause it to resume execution, and then process the resulting exception.

Tracing can be accomplished by the following steps:

1. The debug task prevents further execution of the controlled task by issuing a *db_block* directive.
2. The controlled task's status register is read using the ~~t~~db*_getreg* directive.
3. The debug task sets the trace bit in the status register, and writes it back using the ~~t~~db*_setreg* directive.
4. The debug task then permits execution of the controlled task by issuing the *db_unblock* directive.
5. Since the trace bit is set, when the controlled task executes it will take a trace exception.
6. When the trace exception occurs, the executive will block further execution of the controlled task and send a message to the debug task using the *debug_msg* message queue specified in the *db_control* directive.
7. The debug task can then receive the message, process it, and continue debugging the task.

1.1.6.2 Breakpoints

Breakpoints are accomplished in a similar fashion.

1. Execution of the controlled task is stopped using the *db_block* directive.
2. The instruction at the breakpoint locations is read and saved using the *db_getmem* directive.
3. The instruction is replaced with the breakpoint code using the *db_setmem* directive.

4. The debug task then executes the controlled task with the *db_unblock* directive.
5. The controlled task will execute until it reaches the breakpoint code. At this point it will take an exception.
6. The executive will block further execution of the debug task and post a message to the *debug_msg* message queue specified in the *db_control* directive.
7. The debugger will receive the message and perform the appropriate action.

1.1.7 Directives

The directives provided by the debug manager are:

Directive	Function
db_control	Control a task
db_remote	Perform directive on remote cpu
db_block	Prevent a task from running
db_unblock	Run a task under control
db_getmem	Get a task's memory
db_setmem	Set a task's memory
db_getreg	Get a task's register
db_setreg	Set a task's register

1.1.8 DB_CONTROL

NAME

db_control -- "Control a Task During Debug"

SYNOPSIS

```
uint db_control ( tid, mode, qid )

          uint tid;       /* task id as returned from t_create or t_ident */
          uint mode;      /* new mode */
          uint qid;       /* debug_msg qid */
```

DESCRIPTION

Db_control is used to establish or remove debug control over a task.

The *tid* parameter specifies the task to be controlled. This task may exist on the local processor, or any remote processor in the multiprocessing configuration if the task was created with the **GLOBAL** flag set (see *t_create).*

The *mode* specifies what type of action is to be performed when an exception occurs.

DB_TASK_CONTROL	set	to establish control over task
	clear	to remove control over task

These values are mutually exclusive.

The message queue identified by the *qid* parameter is used by the executive to report exceptions to the debug task. This queue must exist and if debugging is to be done on multiple cpu's, then this queue must have been created with the **GLOBAL** flag set.

RETURN VALUE

If *db_control* successfully completes, 0 is returned.

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

ERROR CONDITIONS

Invalid *tid.*

Task already under debug control.

NOTES

Not callable from ISR.

Asserting control over a task will place it in the *blocked* state.

Removing debug control from a task will *unblock* the task if it was blocked.

Will not cause a preempt when *mode* is *set*.

May cause a preempt when *mode* is *clear* by unblocking a higher priority task.

1.1.9 DB_REMOTE

NAME

db_remote -- "Remote Request"

SYNOPSIS

uint db_remote (cpuid, request, &rval, arg1, ..., argN)

```
        uint cpuid;     /* Identifies remote cpu */
        uint request;   /* Identifies request to be performed */
        uint rval;      /* Return value of remote call - returned by this call */
        uint arg1;      /* First argument of request */

        uint argN;      /* Last argument of request */
```

DESCRIPTION

The *db_remote* directive will cause a directive to be executed on a remote cpu.

The *cpuid* identifies the remote cpu, the *request* specifies which RTEID request (including debug extensions) is to be performed, and *arg1-argN* specify the arguments.

Arg1-argN are the arguments for the request and their meaning is specific to the directive identified by *request*. Any addresses specific to the calling task are treated as external physical addresses.

RETURN VALUE

If *db_remote* successfully completes, then *rval* contains the return value of the remote directive, and 0 is returned.

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

ERROR CONDITIONS

Invalid *cpuid*.

Invalid request.

Other error returns are based on the specific directive identified by *request*.

NOTES

This request operates as if a task on the remote system issues the request on behalf of the caller. The actual execution of the remote request may be performed by the ISR which processes remote requests, or may be performed by a system task on the target system.

Since not all RTEID directives may be executed on a non-local cpu, the *db_remote* directive will provide this function. It is especially important for debuggers which need to create tasks and manage resources on the target cpu.

This directive is also needed to access resources that are local to a remote cpu. For example, this directive could be used to suspend a task which does not have the **GLOBAL** flag set (assuming the task is local to a remote cpu).

Several directives have the address of return buffers as input parameters. The caller of *db_remote* must specify addresses which are external to the target processor (designated by *cpuid).*

1.1.10 DB_BLOCK

NAME

db_block -- "Prevent a Task Under Debug Control from Running"

SYNOPSIS

```
uint db_block ( tid )

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

DESCRIPTION

The *db_block* directive prevents the task identified in the *tid* field from executing. The controlling relationship must have been previously established using the *db_control* directive.

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** flag set (see *t_create).*

RETURN VALUE

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

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

ERROR CONDITIONS

Invalid *tid.*

Task not in controlled state.

Task already blocked.

NOTES

Not callable from ISR.

1.1.11 DB_UNBLOCK

NAME

db_unblock -- "Release a Task"

SYNOPSIS

```
uint db_unblock ( tid )

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

DESCRIPTION

Db_unblock allows the task identified by the *tid* field to resume execution under control of the requesting task. The controlling relationship must have been previously established using the *db_control* directive.

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** flag set (see *t_create).*

RETURN VALUE

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

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

ERROR CONDITIONS

Invalid *tid.*

Task not in controlled state.

Task not blocked.

NOTES

Not callable from ISR.

May cause a preempt.

1.1.12 DB_GETMEM

NAME

db_getmem -- "Get a Task's Memory"

SYNOPSIS

uint db_getmem (tid, laddr, bufaddr, length)

```
uint tid;         /* task id as returned from t_create or t_ident */
char *laddr;      /* logical start address */
char *bufaddr;    /* buffer address */
uint length;      /* length in bytes */
```

DESCRIPTION

The executive reads memory from the task identified in the *tid* field, starting at the task's logical address *laddr,* and copies it to the buffer identified in the *bufaddr* field for the length identified in *length.*

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 flag set (see *t_create).* This directive may be used to transfer data between a logical address belonging to the task identified by the *tid* and the requesting task's buffer.

RETURN VALUE

If *db_getmem* successfully read the memory into the buffer, then 0 is returned.

If the memory was not successfully read into the buffer, an error code is returned.

ERROR CONDITIONS

Invalid *tid.*

Invalid *laddr* for the task.

Bus Error occurred during the read.

NOTES

Not callable from ISR.

Will not cause a preempt.

There is no requirement that the task identified by the *tid* be a controlled task.

Db_getmem will attempt to only read the requested data and will not access memory beyond the *laddr+length*. If *length* is 1, a byte wide read is performed. If *length* is 2, a word wide read is performed.

1.1.13 DB_SETMEM

NAME

db_setmem -- "Set a Task's Memory"

SYNOPSIS

```
uint db_setmem ( tid, laddr, bufaddr, length )

        uint tid;        /* task id as returned from t_create or t_ident */
        char *laddr;     /* logical start address */
        char *bufaddr;   /* buffer address */
        uint length;     /* length in bytes */
```

DESCRIPTION

The executive writes memory to the task identified in the *tid* field from the buffer identified in the *bufaddr* starting at the task's logical address *laddr* field for the length identified in *length*.

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** flag set (see *t_create)*. This directive may be used to transfer data between any requesting task's buffer and a logical address belonging to the task identified by the *tid*.

RETURN VALUE

If *db_setmem* successfully writes the memory from the buffer, then 0 is returned.

If the memory was not successfully written from the buffer, an error code is returned.

ERROR CONDITIONS

Invalid *tid*.

Invalid *laddr*.

Bus Error occurred during the write.

NOTES

Not callable from ISR.

Will not cause a preempt.

There is no requirement that the task identified by *tid* be a controlled task.

Db_setmem will only read the requested data and will not access memory beyond the *laddr+length*. If *length* is 1, a byte wide read is performed. If *length* is 2, a word wide read is performed.

1.1.14 DB_GETREG

NAME

db_getreg -- "Get a task's register"

SYNOPSIS

```
uint db_getreg ( tid, regnum, &regptr )

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

        union regval {
                               uint i;
                               float f;
        }
```

The *regnum* field values are:

S_STAT	Task's status byte values:
T_WTMEM	waiting for memory
T_WTMSG	waiting on message queue
T_WTEVT	waiting for event
T_WTSEM	waiting for semaphore
T_WTTIM	waiting for timeout
T_WTCTL	waiting on control
D_REG0	Task's Processor Register D0
D_REG1	Task's Processor Register D1
D_REG2	Task's Processor Register D2
D_REG3	Task's Processor Register D3
D_REG4	Task's Processor Register D4
D_REG5	Task's Processor Register D5
D_REG6	Task's Processor Register D6
D_REG7	Task's Processor Register D7
A_REG0	Task's Processor Register A0
A_REG1	Task's Processor Register A1
A_REG2	Task's Processor Register A2
A_REG3	Task's Processor Register A3
A_REG4	Task's Processor Register A4
A_REG5	Task's Processor Register A5

A_REG6	Task's Processor Register A6
A_REG7	Task's Processor Register A7
H_SR	Status Register
H_PC	Program Counter
H_VOR	Vector Offset Register
H_USP	User Stack Pointer
H_ISP	Interrupt Stack Pointer
H_MSP	Master Stack Pointer
H_VBR	Vector Base Register
H_CACR	Cache Control Register
H_CAAR	Cache Address Register
H_VBR	Vector Base Register
H_CACR	Cache Control Register
H_CAAR	Cache Address Register
FP_REG0	Task's Processor Register FP0
FP_REG1	Task's Processor Register FP1
FP_REG2	Task's Processor Register FP2
FP_REG3	Task's Processor Register FP3
FP_REG4	Task's Processor Register FP4
FP_REG5	Task's Processor Register FP5
FP_REG6	Task's Processor Register FP6
FP_REG7	Task's Processor Register FP7
FPCR	Task's Coprocessor Control Register
FPSR	Task's Coprocessor Status Register
FPIAR	Task's Coprocessor Instruction Address Register

DESCRIPTION

The executive returns the register value in the *regptr* 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 *db_getreg* is successful, *regptr* is filled in and 0 is returned.

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

ERROR CONDITIONS

Invalid *tid*.