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 Debug Extension to the Real Time Executive Interface Definition 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. 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)

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 *db_getreg* directive.
3. The debug task sets the trace bit in the status register, and writes it back using the *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.

1.1.18 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*.

Invalid register number.

Task not created from local 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.

1.1.15 DB_SETREG

NAME

db_setreg -- "Set a task's register"

SYNOPSIS

uint db_setreg (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 */

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

The *regnum* field values are:

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 sets the register identified in the *regnum* field for the task identified by the *tid* with the value in the *regptr* 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 *db_setreg* successfully set the register value, then 0 is returned.

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

ERROR CONDITIONS

Invalid *tid*.

Invalid register number.

Task not created from local 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.

1.2 Debugging systems

Debugging a system is much more complex than debugging a task or collection of tasks. In order to debug a system, it should be possible to debug the interrupt service routines (ISR's) which are part of the system. This causes several problems. The interrupt mask must not be lowered outside of an ISR. Additionally, an exception in an ISR may come at any time, and may occur when any task (with a low enough interrupt mask) is executing. Since the ISR must be blocked from further execution, the current task is also blocked.

1.2.1 Controlling Systems

The control over a system is established through the use of the *db_system* directive. This will assert debug control over the entire system of tasks and ISR's executing on that particular cpu board. In order to issue this command, the debugger must not be a task on the cpu board being debugged[1].

When control is established, the type of control is specified by the *mode* parameter. If *all* is specified, then all activity, except for processing directives, is suspended when an exception occurs in an ISR. If *level* is specified, then the executive will block further dispatching at the current level and below (see the *db_level* command) and continue dispatching tasks whose interrupt mask is greater than the current level.

1.2.2 Exceptions in ISR's

When a controlled ISR issues an exception, such as a bus error, the execution of the entire system must be examined. Further activity of the ISR is suspended and further task dispatching on the system is performed based on the *mode* specified in the *db_system* directive. The executive on the controlled system will format a message containing information about the exception and place it on a message queue associated with the debug of the cpu. Note that even if the execution of a system is blocked, the execution of the directives must still be processed. Since the execution of directives continues, the debug task may issue a *db_remote* directive which will permit further execution of the controlled system.

---

1. Alternatively, the debugger could be a "higher order" entity, such as the resident debug monitor, on a single cpu system. This "higher order" entity would perform as a system debugger and be able to issue requests to the executive as if it were a remote task.

1.2.3 Directives

The following directives are used for system debugging:

Directive	Function
db_system	Control a system
db_level	Set minimum Processor mask level

1.2.4 DB_SYSTEM

NAME

db_system -- "Control a System During Debug"

SYNOPSIS

uint db_system (cpu, mode)

```
uint cpu;     /* Designates a cpu in the system */
uint mode;    /* new mode */
```

DESCRIPTION

The *cpu* parameter uniquely identifies a cpu in the system.

The *mode* parameter indicates what processing may continue in the system after an exception occurs at some point within the system. Valid *mode* settings are:

DB_SYSTEM_CONTROL	to establish control over system
DB_SYSTEM_RELEASE	to remove control over system
DB_LEVEL	block tasking at level of ISR
DB_ALL	block all task dispatching
DB_CONTINUE	continue execution on the system

If an exception occurs while a task is executing, then that task is blocked and a message is sent to the debug task. If **DB_LEVEL** was specied as the mode, then only this task will be blocked. If **DB_ALL** was specified as the mode, then all dispatching will be suspended until a *db_system* command is specified with mode set to **DB_CONTINUE.**

If an exception occurs while an ISR is executing, further system activity is indicated by the mode parameter. If **DB_LEVEL** is specified for the *mode* parameter, then when an exception occurs in an ISR, the executive will issue a *db_level* directive with the level set to that of the current interrupt priority mask. This will keep the executive from dispatching tasks whose interrupt priority mask is less than this value, and will also block interrupts at this level or less. Interrupts and tasks whose level is greater will occur normally.

If the *mode* parameter is **DB_ALL** and an exception occurs within an ISR, then all further activity on this system will be blocked. The only exception to this is that remote requests for RTEID directives (including debug extensions) will be serviced by the executive. The executive will become unblocked when the debug task (remotly) issues a *db_unblock* for the *cpu_id* corresponding to the system. At this point, the ISR that caused the exception will continue execution.

Issuing a *db_system* directive with *mode* set to **DB_CONTINUE** will cause the execution of the system to continue.

RETURN VALUE

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

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

ERROR CONDITIONS

Invalid *cpu*.

Invalid *mode*.

NOTES

When first establishing control over a system, the *mode* parameter must include **DB_SYSTEM_CONTROL** and may also include either **DB_ALL** or **DB_LEVEL**.

Once control has been established, the type of control may be changed by specifying a different mode.

1.2.5 DB_LEVEL

NAME

db_level -- "Set the Minimum Mask Level"

SYNOPSIS

uint db_level (level, &plevel)

```
uint level;     /* Minimum Processor Interrupt mask level*/
uint plevel;    /* Previous level - returned by this call */
```

DESCRIPTION

The *db_level* directive specifies a minimum interrupt priority mask level for further execution of the tasks and ISR's executing on the local cpu.

The *level* value is the minimum interrupt level for all tasks in the system. The executive will never set the status register's interrupt mask to a value less than *level*. Furthermore, the executive will never dispatch a task whose status register's interrupt mask is less than *level*.

RETURN VALUE

If *db_level* is successful, then the previous minimum level is returned in *plevel* and 0 is returned.

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

ERROR CONDITIONS

Level is not in a valid range (0..7).

The interrupt mask of the current task is less than *level*.

NOTES

May cause a preempt.

1.3 System Monitoring

Debugging a system involves more than debugging a collection of tasks; the performance of the entire system needs to be monitored and tuned. The *db_get_id* directive will return a unique identifier for items of particular types, or items in particular queues. The *db_get_item* directive will get information about items specified by the identifier. The information block will contain data about the system as well as some history (such as total number of calls to a directive) about the execution of the system. It is important to note that gathering statistics about the system will add a small amount of overhead to all of the calls.

The *db_get_id* directive requires an item_id as an input parameter. If the value of item_id is zero, then the first item of the specified class would be returned. If the item is non-zero, then the next item past the specified item_id will be returned. This can be used to loop through all items in a particular class. For example, to examine all tasks in the system, the following C code could be used:

```
for( item_id=0; item_id=get_item(item_id, TASK, 0); )
{
    process(item_id);
}
```

The class parameter specifies what type of item id to return and the third parameter is used to specify additional information (such as which message queue).

1.3.1 Directives

The directives provided by the system monitoring are:

Directive	Function
db_get_id	Get identifier for an item
db_get_item	Get information about an item

1.3.2 DB_GET_ID

NAME

db_get_id -- "Get an Item Identifier"

SYNOPSIS

uint db_get_id (item_id, &ret_id, class, arg)

```
uint item_id;   /* Previous item_id */
                /* 0 requests first item */
uint ret_id;    /* Returned item_id - returned by this call */
uint class;     /* Class of item */
uint arg;       /* Argument as defined by class */
```

DESCRIPTION

The *db_get_id* directive allows the debug task to receive a unique identifier as defined by *item_id* and *class,* to be returned in *ret_id.*

Item_id must be the unique id of the appropriate type from the list or queue specified by *class,* possibly further qualified by the *arg* parameter. If *item_id* is zero, then an identifier for the first element of the list or queue specified by *class* is returned. If *item_id* is non zero, then the next item past *item_id* is returned in *ret_id.*

Class specifies the list or queue that *item_id* is to be taken from. *Arg* can further specify how the selection is done by selecting a specific list or queue.

Valid class values and the appropriate value for *arg* are given in the following table.

Class Value	Returned item id	Meaning of arg
TASK	task id	
MESSAGE_QUE	message queue id	
SEMAPHORE	semaphore id	
REGION	region id	
PARTITION	partition id	
MESSAGE	message id	message queue id
TASK_IN_MESQ	task id	message queue id
TASK_IN_SEMQ	task id	semaphore id
TASK_IN_SEGQ	task id	region id
SEGMENT	segment id	region id
BUFFER	buffer id	partition id

RETURN VALUE

If *db_get_id* succeeds, the *item_id* for the item in the *class* is returned in *ret_id,* and 0 is returned.

If *db_get_id* succeeds, and there are no more items of the appropriate class, then an error code is returned.

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

ERROR CONDITIONS

No more items in this class.

Invalid *class* identifier.

Item_id not in class.

Invalid *arg*.

NOTES

For example, to process a queue, the *get_id* function is called first with a 0 *item_id* to get the first item in the queue. Subsequent calls use the last value of *item_id* in order to get the next item in the queue.

1.3.3 DB_GET_ITEM

NAME

db_get_item -- "Get Information About an Item"

SYNOPSIS

uint db_get_item (item_id, class, buffer, &size)

```
uint item_id;    /* Item_id */
uint class;      /* Class of item */
char *buffer;    /* address of buffer for returned data */
uint size;       /* Size of item - returned by this call */
```

DESCRIPTION

Db_get_item copies an item description into *buffer,* and returns the size of the item description in *size.* The exact format of the data in *buffer* depends on the *class* parameter.

Item_id is a unique identifier for the item within the *class.*

Class specifies the type of item. Valid *classes* are:

Class	returned data
GENERAL	general info block
TASK	task info block
MESSAGE_QUE	message queue info block
MESSAGE	message info block
SEMAPHORE	semaphore info block
REGION	region info block
SEGMENT	segment info block
PARTITION	partition info block
BUFFER	buffer info block

RETURN VALUE

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

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

Buffer is filled in with various structures depending on the *class* parameter. The following information block structures are used:

```
struct   gib    {
         uint   num_tasks;        /* Total number of tasks */
         uint   num_mque;         /* Total number of message queues */
         uint   num_sema;         /* Total number of semaphores */
         uint   num_regions;      /* Total number of regions */
         uint   num_partitions;   /* Total number of partitions */
         uint   num_ready;        /* Size of ready list */
         uint   num_calls;        /* Total number of RTEID calls made */
         uint   num_inter;        /* Total number of v_returns */
         uint   ticks;            /* Number of ticks on clock */
         uint   min_level;        /* Minimum Processor Mask */
}
```

Figure 1. General Info Block

```
struct   tib    {
         uint   name;             /* Task's name */
         uint   id;               /* Task's Task id */
         uint   mode;             /* Task's current mode */
         uint   prio;             /* Task's current priority */
         uint   stat;             /* Task's current status */
         uint   events_pending;   /* Events pending for the task */
         uint   events_waiting;   /* Task's event condition from ev_receive */
         uint   signals;          /* Task's pending signals */
         uint   timeout;          /* Task's current timeout value */
         ptf    asr_addr;         /* Task's ASR address */
}
```

Figure 2. Task Info Block

```
struct   mqib   {
         uint   name;         /* Message Queue's name */
         uint   id;           /* Message Queue's id */
         uint   num_mess;     /* Number of messages in queue */
         uint   num_tasks;    /* Number of tasks waiting on messages */
         uint   total_mess;   /* Total messages ever placed in this queue */
         uint   total_urg;    /* Total number of urgent messages */
}
```

Figure 3. Message Queue Info Block

```
struct   message   {
         long      text[4];      /* Message text (16 bytes) */
}
```

Figure 4. Message Info Block

```
struct   smib   {
         uint   name;          /* Semaphore's name */
         uint   id;            /* Semaphore's id */
         uint   value;         /* Semaphore's current value */
         uint   num_tasks;     /* Number of tasks waiting on this Semaphore */
         uint   total_v;       /* Total number of sm_v operations */
         uint   total_p;       /* Total number of sm_p operations */
}
```

Figure 5. Semaphore Info Block

```
struct   rib    {
         uint   name;          /* Region's name */
         uint   id;            /* Region's id */
         uint   page_size;     /* Region's page size */
         uint   paddr;         /* Region's physical start address */
         uint   length;        /* Region's length */
         uint   attributes;    /* Region's attributes */
         uint   num_segs;      /* Number of allocated segments */
         uint   num_tasks;     /* Number of tasks waiting for a segment */
         uint   total_getseg;  /* Total number of rn_getseg */
         uint   total_retseg;  /* Total number of rn_retseg */
}
```

Figure 6. Region Info Block

```
struct   sgib   {
         uint   address;   /* Address of the Segment */
         uint   size;      /* Size of the Segment */
         uint   attrib;    /* Segment Attributes (RDONLY) */
}
```

Figure 7. Segment Info Block

```
struct   pib    {
         uint   name;          /* Name of the Partition */
         uint   id;            /* Id of the Partition */
         uint   bsize;         /* Buffer size */
         uint   bnum;          /* Total number of buffers in the Partition */
         uint   bavail;        /* Number of available buffers */
         uint   paddr;         /* Physical start of the Partition */
         uint   flags;         /* Partitions flags */
         uint   total_getbuf;  /* Total number of pt_getbuf calls */
         uint   total_retbuf;  /* Total number of pt_retbuf calls */
}
```

Figure 8. Partition Info Block

```
struct   bib    {
         uint   addr;    /* Physical address of buffer */

}
```

Figure 9. Buffer Info Block

ERROR CONDITIONS

NOTES