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;          /* Sise 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