## 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 $\overset{db}{\cancel{t}}$*_getreg* directive.
3. The debug task sets the trace bit in the status register, and writes it back using the $\overset{db}{\cancel{t}}$*_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.