# Cray XMT™ Debugger Reference Guide

S–2467–20

RECORD OF REVISION

S–2467–20 Published May 2011 Supports release 2.0 GA running on Cray XMT compute nodes and on Cray XT 3.1UP02 service nodes. This release uses the System Management Workstation (SMW) version 5.1UP03.

1.4 Published December 2009 Supports release 1.4 running on Cray XMT compute nodes and on Cray XT 2.2.UP01 service nodes. This release uses the System Management Workstation (SMW) version 4.0.UP02.

1.3 Published March 2009 Supports release 1.3 running on Cray XMT compute nodes and on Cray XT 2.1.5HD service nodes. This release uses the System Management Workstation (SMW) version 3.1.09.

1.2 Published August 2008 Supports release 1.2 running on Cray XMT compute nodes and on Cray XT 2.0.49 service nodes. This release uses the System Management Workstation (SMW) version 3.1.04.

# Contents

*Page*

**Overview [1]** **7**
1.1 Prerequisites . . . 7
1.1.1 Loading the Module . . . 7
1.1.2 Compiling for Debugging . . . 8
1.1.3 Working Directories . . . 9
1.1.4 Environment Variables . . . 10
1.2 Getting Started . . . 10
1.2.1 Selecting a Program to Debug . . . 11
1.2.1.1 File Commands . . . 11
1.2.1.2 Module Commands . . . 13
1.2.1.3 Object Directory Commands . . . 14
1.2.1.4 Shared Library Directory Commands . . . 15
1.2.2 Running the Program . . . 16
1.2.2.1 Working Directory . . . 16
1.2.2.2 Program I/O . . . 17
1.2.2.3 Environment Variables . . . 17
1.2.2.4 Runtime Arguments . . . 19
1.3 Debugging a Currently Running Job . . . 19
1.4 Ending a Debugging Session . . . 20

**Breakpoints and Watchpoints [2]** **21**
2.1 Breakpoints and Watchpoints . . . 21
2.1.1 Setting Breakpoints . . . 23
2.1.1.1 Special Breakpoint Situations . . . 24
2.1.2 Setting Watchpoints . . . 26
2.1.3 Deleting Breakpoints and Watchpoints . . . 26
2.1.4 Disabling Breakpoints and Watchpoints . . . 27
2.1.5 Break Conditions . . . 28
2.1.6 Commands Executed on Breaking . . . 30
2.2 Continuing . . . 31

*Page*

2.3 Stepping . . . . . . . . . . . . . . . . . . . . . . . . . 32

**Understanding Multithreading [3]** **35**

3.1 Thread Names . . . . . . . . . . . . . . . . . . . . . . . . 35

3.2 Thread States . . . . . . . . . . . . . . . . . . . . . . . . 36

3.3 Focus Thread . . . . . . . . . . . . . . . . . . . . . . . . 37

**Examining the Stack [4]** **41**

4.1 Stack Frames . . . . . . . . . . . . . . . . . . . . . . . . 41

4.2 Backtraces . . . . . . . . . . . . . . . . . . . . . . . . . 42

4.3 Selecting a Frame . . . . . . . . . . . . . . . . . . . . . . . 43

4.4 Information on a Frame . . . . . . . . . . . . . . . . . . . . . 45

**Examining Source Files [5]** **47**

5.1 Printing Source Lines . . . . . . . . . . . . . . . . . . . . . 47

5.2 Searching Source Files . . . . . . . . . . . . . . . . . . . . . 49

5.3 Specifying Source Directories . . . . . . . . . . . . . . . . . . . 50

5.4 Examining Instructions . . . . . . . . . . . . . . . . . . . . . 51

**Examining Data [6]** **53**

6.1 Expressions . . . . . . . . . . . . . . . . . . . . . . . . 53

6.2 Program Variables . . . . . . . . . . . . . . . . . . . . . . 54

6.3 State Bits . . . . . . . . . . . . . . . . . . . . . . . . . 55

6.4 Artificial Arrays . . . . . . . . . . . . . . . . . . . . . . . 56

6.5 Format Options . . . . . . . . . . . . . . . . . . . . . . . 57

6.6 Output Formats . . . . . . . . . . . . . . . . . . . . . . . 59

6.6.1 Examining Memory . . . . . . . . . . . . . . . . . . . . . 60

6.7 Automatic Display . . . . . . . . . . . . . . . . . . . . . . 62

6.8 Value History . . . . . . . . . . . . . . . . . . . . . . . . 64

6.9 Convenience Variables . . . . . . . . . . . . . . . . . . . . . 65

6.10 Registers . . . . . . . . . . . . . . . . . . . . . . . . . 66

6.11 Register Examples . . . . . . . . . . . . . . . . . . . . . . 67

**Examining Symbols [7]** **69**

7.1 Archive Symbol Visibility . . . . . . . . . . . . . . . . . . . . 70

**Altering Execution [8]** **73**

8.1 Assignment to Variables . . . . . . . . . . . . . . . . . . . . . 73

8.1.1 Altering Variables Kept in Registers . . . . . . . . . . . . . . . . 74

8.2 Changing the Full/Empty Bit . . . . . . . . . . . . . . . . . . . 75

*Page*

**Stored Sequences of Commands [9]** **79**
9.1 User-defined Commands . . . . . . . . . . . . . . . . . . . . . . . 79
9.2 Command Files . . . . . . . . . . . . . . . . . . . . . . . . 80
9.3 Commands for Controlled Output . . . . . . . . . . . . . . . . . . . . 81

**Options and Arguments for `mdb` [10]** **83**
10.1 Mode Options . . . . . . . . . . . . . . . . . . . . . . . . 83
10.2 File-specifying Options . . . . . . . . . . . . . . . . . . . . . . 84
10.3 Communication Options and Variables . . . . . . . . . . . . . . . . . . 84
10.4 Breakpoint-behavior Options . . . . . . . . . . . . . . . . . . . 85
10.5 Miscellaneous Options . . . . . . . . . . . . . . . . . . . . . . 85
10.6 Other Arguments . . . . . . . . . . . . . . . . . . . . . . . 85

**Appendix A GNU General Public License** **87**
A.1 Preamble . . . . . . . . . . . . . . . . . . . . . . . . . 87
A.2 Terms and Conditions . . . . . . . . . . . . . . . . . . . . . . 88
A.3 How to Apply These Terms to Your New Programs . . . . . . . . . . . . . . 90

**Appendix B Using `mdb` under GNU Emacs** **93**

**Appendix C `mdb` Input and Output Conventions** **95**

**Glossary** **99**

# Overview [1]

This guide is for application programmers who develop code that runs on Cray XMT systems.

The `mdb` debugger is based on the Free Software Foundation's GDB debugger (version 3.5), as adapted for use on Cray XMT systems. The `mdb` debugger provides both source-level and machine-level debugging.

While using the `mdb` debugger you can:

- Launch your program and specify conditions that might affect the behavior of the program
- Set *breakpoints* and *watchpoints* to make the program either suspend execution at a specified point in the code or upon meeting a specified condition
- Examine program information after such a stop to determine exactly what is happening at this point in the program execution
- Modify conditions and resume program execution as desired

Additional information is in the `mdb`(1) man page and in the `mdb` online help system, which you can view by entering `help` at the `mdb` command line prompt.

## 1.1 Prerequisites

Before using the `mdb` debugger, verify that the programming environment module is loaded and that your application was compiled for debugging, as described in Compiling for Debugging on page 8.

### 1.1.1 Loading the Module

The `mdb` debugger is included as part of the Cray XMT programming environment, which is available by loading the `mta-pe` module. Programs intended for use on Cray XMT systems cannot be compiled or debugged in a cross-compiler environment. You must be logged into the Cray XMT system in order to use the programming environment.

```
workstation% ssh -X XMT_system
Password:

XMT_system> module load mta-pe
```

To see which modules are available on your system, type **`module avail`**.

To see which modules are loaded in your user environment, type **`module list`**.

### 1.1.2 Compiling for Debugging

When you debug a program, the level of detail `mdb` provides depends on the level of debugging information that was generated and stored in the program library when you compiled the program. This information includes the location and data type of each variable or function, as well as the relationship between lines in the source code and addresses in the executable code.

There is an inverse relationship between debugging information and compiler optimization. The more debugging information that you request when compiling the code, the less optimization can be performed by the compiler. As a result, code compiled for debugging runs more slowly than code compiled for normal execution.

Conversely, as the level of compiler optimization is increased, the speed of program execution increases, but the correlation between your source code and the resulting executable code decreases. In some cases, this means that `mdb` may not be able to perform those debugging operations that depend on a close correspondence between source and executable code. In these cases, the debugger typically generates a message warning that due to compiler optimizations the effect of the debugging operation cannot be guaranteed. However, there may be situations where the debugger may lack sufficient information to determine that this problem even exists.

Use the following compiler options to determine the level of debugging information generated and compiler optimizations performed when compiling your program.

`-g2`
: Most debugging information; least optimization. The compiler generates parallelism for future statements but does not automatically parallelize loops or other code. The debugger supports basic debugging operations at statement boundaries. At a breakpoint, you may read or modify any visible variable or memory state, and you may resume execution in ways that are fully consistent with your source code.

`-g1 | -g`
: Moderate debugging information and optimization. The compiler automatically parallelizes code, with some restrictions, based in part on only volatile-qualified data being updated from outside the program. At a breakpoint, you may read any visible variable or memory state, and you modify any volatile-qualified data and resume execution in ways that are fully consistent with your source code.

(omit `-g` option)
: Least debugging information; most optimization. If you omit the `-g` option, the compiler places no restrictions on optimizations and retains no debugging information. You can set breakpoints and pause and resume program execution, but the source and executable code may diverge substantially. You can read global data, but the debugger may not be able to find other variables.

For example, to compile your program and generate the most debugging information and perform the fewest compiler optimizations, use the following command:

```
XMT_system> cc -g2 myprogram.c
```

**Note:** As part of the optimization process, the compiler may *inline* a routine by replacing a statement that calls a routine with the actual body of that routine, if the compiler determines that the resulting code will be more efficient. This optimization can occur at all debugging levels, and each inlined routine retains the debugging level that was specified when it was compiled. For routines inlined by the compiler, in most debugging operations, `mdb` presents the debugging information as if the routine was invoked normally, and not inlined.

### 1.1.3 Working Directories

The `mdb` debugger inherits its working directory from the shell used to launch the debugging session. This working directory also serves as the default location for debugger commands that specify files.

After you launch `mdb`, use the `cd` command to change the default directory. For more information, see Working Directory on page 16.

### 1.1.4 Environment Variables

Environment variables are typically used to define such things as your user name, home directory, terminal type, search path, and other conditions that affect program operation. The `mdb` debugger inherits its environment variable settings from the shell used to launch the debugging session.

After you launch `mdb`, use the `info environment` and `set environment` commands to view and change environment variable settings. For more information, see Environment Variables on page 17.

## 1.2 Getting Started

To begin using `mdb`:

1. Log on to the Cray XMT system.

   ```
   workstation% ssh -X XMT_system
   ```

2. Load the Programming Environment module.

   ```
   XMT_system> module load mta-pe
   ```

3. If needed, change to your working directory.

   ```
   XMT_system> cd workdir
   ```

4. Enter the shell command `mdb`.

   ```
   XMT_system/workdir> mdb [-mtarun-args arguments] [program]
   ```

   Use the `-mtarun-args` option to pass runtime arguments through to `mtarun`. For more information, see Running the Program on page 16.

   If you specify a program name with the `mdb` command, the debugger reads the symbol table in the named file. If you do not enter a program name here, you can specify it later. For more information, see Selecting a Program to Debug on page 11.

   After the `mdb` copyright information displays, you will have an `mdb` prompt:

   ```
   (mdb)
   ```

   **Note:** The default debugger prompt is `(mdb)`. If desired, you can change this to a user-defined prompt string. For more information, see Appendix C, `mdb` Input and Output Conventions on page 95.

   In addition to the `-mtarun-args` and *program* arguments, the `mdb` command supports many other options, including the use of a configuration file. For more information, see Chapter 9, Stored Sequences of Commands on page 79.

At this point, you are ready to begin debugging a program.

**Note:** In addition to the `mdb`(1) man page, the `mdb` debugger includes an online help system which you can access by entering **`help`** at the `(mdb)` prompt. The `help` command recognizes many keywords; for example, to find file-related help content, enter the following command:

```
(mdb) help files
```

For more information, see the online help system.

## 1.2.1 Selecting a Program to Debug

There are two ways to specify the program you want to debug.

- Enter the name of the executable file when you start `mdb`.

  ```
  XMT_system/workdir> mdb a.out
  ```

  In this case, `mdb` reads the executable file and symbol table in as part of the startup process. If you have done this, skip to Running the Program on page 16.

- Alternatively, you can use `mdb` commands to load or swap files after `mdb` is running. For example:

  ```
  (mdb) file a.out
  ```

  In this case, `mdb` takes less time to start up and you have more control over which files are loaded and when they are loaded.

  The following subsections describe the commands used to load or swap files after `mdb` is running.

### 1.2.1.1 File Commands

Use the following commands to load files or change files during a debugging session.

`attach`[*pid*] [*device_filename*]

Attach to a process that was started up outside of `mdb`. This command may take as argument a process id or a device filename. For a process id, you must have permission to send the process a signal, and it must have the same effective `uid` as the debugger. For a device filename, the file must be a connection to a remote debug server.

Before using `attach` you must use the `exec-file` command to specify the program running in the process, and the `symbol-file` command to load its symbol table. Alternatively, use the `file` command, which performs both functions.

```
(mdb)file filename
(mdb)attach filename or pid
```

`info files` Print the path and name of the executable file currently loaded into `mdb` and the path and name of the file from which the symbol table was loaded. For example:

```
(mbd) info files
Executable file "/XMT_system/users/smith/a.out"
Symbols from "XMT_system/users/smith/a.out"
(mdb)
```

`file` *filename*

Load the specified executable file into `mdb`, along with the associated symbol table. If you do not specify a directory and the file is not in the current working directory, `mdb` uses the environment variable `PATH` as a list of directories to search for the file.

If a file is already resident, you are asked to confirm that you want to load the new file. For example:

```
(mbd) file b.out
Load new executable from "b.out"? (y or n) y
Reading executable from XMT_system/users/smith/b.out
Reading symbols from XMT_system/users/smith/b.out
done
(mdb)
```

`exec-file` *filename*

Load only the executable file. Do not load the symbol table.

`symbol-file` *filename*

Load the symbol table from the specified file. If you do not specify a directory and the file is not in the current working directory, `mdb` uses the environment variable `PATH` as a list of directories to search for the file.

The `symbol-file` command does not actually read the symbol table in full. Instead, it scans the symbol table quickly to determine which source files and symbol tables are present. The details are read later, one source file at a time, as needed.

The purpose of this two-stage reading strategy is to make `mdb` start up faster. For the most part, it is invisible to the user, except for occasional messages indicating that the symbol table details for a particular source file are being read. Use the `set verbose` command to control whether these messages are printed; for more information, see Appendix C, `mdb` Input and Output Conventions on page 95.

`symbol-file`

To clear the symbol table, enter `symbol-file` without specifying a *filename*.

**Note:** While the `file`, `exec-file`, and `symbol-file` commands accept both absolute and relative file names as arguments, the file names are always stored as absolute file names.

Using the `symbol-file` command causes `mdb` to purge the contents of its convenience variables, value history, and all breakpoints and auto-display expressions. This is done because these values may contain pointers to the internal data recording symbols and data types that are part of the old symbol table which is being discarded.

#### 1.2.1.2 Module Commands

The `mdb` debugger gets the information it needs to run your program not only from the executable file, but also from any *module* linked with your program. Note that in this context the term module does not refer to a software module, rather to a source file and all of the files it explicitly includes, as well as the corresponding objects that result from compilation of the source module (typically, a portion of a program library or a traditional object file). The information contained in a module includes type definitions, source line mappings, and the locations of local and static variables.

The `mdb` debugger assimilates the information in each module of your program, and reads in a module when your program needs information from that module.

If, during a debugging session, you reach a point where `mdb` has yet to incorporate the module information you want to use (for example, the full definition of a type structure) and you know where the information is located, use the following commands to force `mdb` to load the appropriate module.

`info modules`

Print the names of all modules in the program. If the module is compiled separately, it is listed separately. Constituents of larger files—for example, archives built with `ar` or program libraries produced by whole-program compilation—are listed as part of the parent library.

`load` *modulename*

Read information from *modulename*, including type definitions and source file information.

`mdb` uses the relative path information specified in the root program library to locate *modulename*. If you have moved your program executable after building it, use the `set linkdir` command to provide the necessary path information. For more information, see Specifying Source Directories on page 50.

### 1.2.1.3 Object Directory Commands

The path to each object file used by the linker is recorded in the root program library. If the path is absolute, it is used as-is. If the path is relative, `mdb` appends the path to the *linkdir* variable. (For more information about the *linkdir* variable, see Specifying Source Directories on page 50.) However, if you provide `mdb` with an object file search path, the debugger looks for an object file, first in the directories in the search path, and then in the path used by the linker.

The information from an object file that is already loaded into `mdb` is not affected by later modifications to the object path. When you start `mdb`, the object file search path is empty. Use the following commands to add or change object file search paths.

`info objdirectories`
`info objdir`
`info obj` Print the current object search path as a list of directories.

`objdirectory` *dirname*
`objdir` *dirname*
`obj` *dirname*

Add directory *dirname* to the front of the object search path. You may specify several directory names with this command, by separating the directories with a colon (`:`) or a blank space. If you specify a directory that is already in the source path, it is moved forward and searched earlier.

`objdirectory`

To clear the search path, enter `objdirectory` without specifying a *dirname*. You are asked to confirm that you want to clear the object file search path.

### 1.2.1.4 Shared Library Directory Commands

Each executable file contains a list of shared libraries, along with the set of paths to the shared libraries that was specified when the executable was linked. The `mdb` debugger reads in the symbols for the executable from each of the shared libraries, using these paths to find the libraries. However, the shared libraries or the executable may be moved between compilation and debugging.

The `mdb` debugger uses the *shared library path* to provide a list of directories to search for shared library files. For each shared library, the debugger first tries, in order, the directories in the list, until it finds a file with the desired name. The set of shared library paths from the executable is permanently affixed to the end of the list.

When you start `mdb`, the shared library search path is empty. Use the following commands to add or change shared library search paths.

`info sharedlibpath`

Print the shared library path, and show which directories it contains.

`sharedlibpath` *dirname*

Add directory *dirname* to the front of the shared library search path. You may specify several directory names with this command, by separating the directories with a colon (`:`) or a blank space. If you specify a directory that is already in the path, it is moved forward and searched earlier.

If you know before the start of the debugging session that you need to use the `sharedlibpath` command, start the debugger without using a file name argument. After the debugger is initialized, use the `sharedlibpath` to specify a list of directories, and then use the `file` command to load the executable and read in the symbol table.

`sharedlibpath`

To clear the search path, enter `sharedlibpath` without specifying a *dirname*. You are asked to confirm that you want to clear the shared library search path.

## 1.2.2 Running the Program

After your program is loaded, use the `run` command to execute it.

```
(mdb) run
```

The `run` command creates an inferior process, loads the program into the inferior process, and sets it in motion.

The execution of your program is affected by certain information the inferior process receives from its superior. The `mdb` debugger provides ways to specify this information, which you must do **before** executing the program. (You can change runtime conditions after starting the program, but these changes do not take effect until the program is restarted.)

The following subsections discuss the different runtime conditions.

### 1.2.2.1 Working Directory

Each time you start your program with `run`, it inherits its working directory from the current working directory of `mdb`. The `mdb` debugger in turn inherits its working directory from its parent process, which is typically the shell.

The `mdb` working directory also serves as the default directory for the file-handling commands described in Selecting a Program to Debug on page 11.

Use the following commands to view or reset the working directory.

`pwd` Print the current working directory.

`cd` *directory*

Reset the working directory to *directory*.

**Note:** The `ls` command cannot be used within an `mdb` debugging session.

### 1.2.2.2 Program I/O

By default, a program run under `mdb` pipes I/O to the same terminal that is used by `mdb`. Use `sh`-style redirection commands in the `run` command to redirect input and output. For example, to start your program and redirect its output to the file *outfile*, enter this command.

```
(mdb) run > outfile
```

### 1.2.2.3 Environment Variables

Environment variables are used to specify your user name, home directory, search paths, and so on. The `mdb` debugger inherits its environment variables from the shell session used to start the debugging session.

There is one environment variable that is specific to `mdb`.

`MDB_MTARUN_ARGS`

If this environment variable is set when `mdb` is invoked, the contents of `MDB_MTARUN_ARGS` are passed along as command-line arguments to `mtarun`, to be used when your program is executed.

**Note:** If this environment variable is set and `mdb` is invoked using the `-mtarun-args` option, the arguments listed in the `-mtarun-args` option take precedence.

Use the following commands to view and change the values of environment variables.

`info environment`

> Print the names and values of all environment variables currently set. You can abbreviate this command to `i env`.

`info environment` *varname*

> Print the value of the environment variable *varname*. You can abbreviate this command to `i env` *varname*.

`set environment` *varname* `[`*value*`]`
`set environment` *varname*`=[`*value*`]`

> Set the environment variable *varname* to *value*. The *value* is optional; if it is omitted, the variable is set to a null value. You can abbreviate this command to `set e` *varname value*.
>
> When set from within a debugging session, the environment variable value applies only to the program being debugged. When you exit from the debugging session, the environment variable is restored to its previous value or state.

`unset environment` *varname*
`delete environment` *varname*

> Remove the environment variable *varname* from the environment. This is different from using the `set environment` to set the variable is set to a null value, as it renders the variable undefined. You can abbreviate this command to `d e` *varname*.
>
> When unset from within a debugging session, the environment variable no longer applies to the program being debugged. However, when you exit from the debugging session, the environment variable is restored to the value or state it had before entering the debugging session.

### 1.2.2.4 Runtime Arguments

In normal operations, many programs require the use of runtime arguments appended to the `mtarun` command in order to run. There are several ways to pass these runtime arguments into a debugging session:

- Use the `mdb` command `-mtarun-args` *arguments* option when starting the debugger to specify `mtarun` arguments. If you use this method, the arguments you specify remain in force unless superseded within the debugging session.
- You can set the `MDB_MTARUN_ARGS` environment variable, either before or after starting the debugging session. If it is set before starting the debugging session and you invoke `mdb` using the `-mtarun-args` option, the `-mtarun-args` arguments supersede the environment variable values.

  If you set it after starting the debugging session, the environment variable values override the `-mtarun-args` (if used), but are unset when you exit the debugging session.
- After starting the debugging session, use the `set mtarun-args` command from within `mdb` to specify `mtarun` arguments. The values you set remain in force until superseded or unset, or until the end of the debugging session.
- You can use the Cray Extensions to the `mdb` command line arguments. For more information, see the `mdb`(1) man page.

Additionally, your program may require runtime arguments specific to your program. These can be set from within the debugging session by using the `set args` *arguments* command prior to issuing the `run` command. The arguments you set this way remain in force until superseded or until the end of the debugging session; to unset these arguments, use the `set args` command with no arguments.

## 1.3 Debugging a Currently Running Job

The most straightforward way to debug a running job is to issue the `mdb` with the following arguments:

**`mdb`** ***file pid***

where *file* is the running program and *pid* is its process ID. In this case mdb will attach automatically to a process that was started up outside of `mdb`.

Alternatively, if `mdb` is already running use the following sequence of commands to load the executable file and its associated symbol file, then attach to the running process:

```
(mdb)exec-file filename
(mdb)symbol-file filename
(mdb)attach device_filename or pid
```

The `attach` command takes as an argument either a process ID or a device filename. To use a process ID, you must have permission to send the process a signal, and it must have the same effective `uid` as the debugger. To use a device filename, the file must be a connection to a remote debug server.

## 1.4 Ending a Debugging Session

To end a debugging session and exit `mdb`, enter either **`quit`** or **`q`** at the `(mdb)` prompt. You will exit to your current working directory.

```
(mdb) quit
XMT_system/workdir>
```

The `Ctrl-C` command does not exit from the debugger, but rather terminates the action of any debugger command currently in progress and returns to the `(mdb)` prompt. It is generally safe to use `Ctrl-C` at any time, because the debugger attempts to synchronize the interrupt to a time when it is safe. However, there is a possibility that using `Ctrl-C` during expression evaluation may leave locks in a held state.

To kill the currently running inferior process, use the mdb `kill` command. Be aware that, on large systems, the `kill` command may take some time to complete.

The default time-out for the `kill` command is 100 seconds. Use the `mdb set kill-timeout` option to change this value.

# Breakpoints and Watchpoints [2]

The primary purpose of using a debugger is so that you can stop it before its planned point of normal termination, or if it fails to run that far, so that you can investigate its behavior and find out what went wrong.

When `mdb` stops your program, all threads stop. The state of your entire program is suspended, and you can examine and modify the state, depending on the debugging level with which you compiled the source code.

## 2.1 Breakpoints and Watchpoints

A *breakpoint* stops all the threads in your program whenever some thread reaches a certain point in the program. You set breakpoints explicitly with `mdb` commands, specifying the place where the program should stop by line number, function name, or exact address in the program. You can add other conditions to control whether the program stops.

A *watchpoint* is a data breakpoint that stops all threads in your program when a watched expression changes. A watched expression stops the program when its value is written, though not necessarily changed, or the `full/empty` bit of any constituent memory word changes (see State Bits on page 55). The `full/empty` bit is the only state bit that can toggle as a result of a read; the others change only during writes. For example, an assignment of zero to a variable whose previous value is zero stops the program. This is similar to a read of a watched sync variable because the `full/empty` bit changes from `full` to `empty`. After suspending your program, `mdb` tells you the previous value of the changed watchpoint expression. You can see the new value by printing the expression.

When `mdb` suspends your program due to a watchpoint, the current instruction of a thread that changed a watched expression may be some distance beyond the instruction that triggered the watchpoint. For example, a watchpoint expression may have been changed when control was in the previous stack frame. This long stopping distance is caused by a combination of instruction pipelining by the hardware, multiple operations per instruction, jump operations, and compiler optimizations. To reduce this effect, compile your program at a lower optimization level.

Watchpoints in `mdb` are as efficient to use as breakpoints. The implementation of watchpoints is based on the hardware trap bits associated with each data memory word (see State Bits on page 55).

Watchpoints and breakpoints are differentiated by the commands you use to create them (see Setting Breakpoints on page 23 and Setting Watchpoints on page 26). Most of the commands for enabling, disabling, and deleting breakpoints also apply to watchpoints (see Deleting Breakpoints and Watchpoints on page 26 and Disabling Breakpoints and Watchpoints on page 27).

Each breakpoint and watchpoint is assigned a number when it is created; these numbers are successive integers starting with 1. In many of the commands for controlling various features of breakpoints and watchpoints, you use this number to say which point you want to change. Each breakpoint or watchpoint may be *enabled* or *disabled*; if a point is disabled, it has no effect on the program until you enable it again.

The command `info breakpoints` or `info watchpoints` prints a list of each breakpoint and watchpoint that is set but not deleted: its number, type (breakpoint or watchpoint), disposition (whether the point is marked to be disabled or deleted when reached), whether or not the point is enabled, where in the program it is, and any special features in use for the point (conditions, command sets). Disabled points are included in the list, but marked as disabled (not enabled). You can abbreviate `info breakpoints` as `info break` or even `i b`. `info break` with an integer argument lists only the associated breakpoint or watchpoint.

The following example shows a breakpoint on `main`, and a watchpoint on the variable `foo`.

```
[1] (mdb) info breakpoints
Num Type  Disp En Address    What
  1 break keep y  0x524      in main (/home/users/xxx/main.c line 33)
  2 watch keep y  foo
```

(Unlike GDB, `info break` or `info watch` in `mdb` does not set either the convenience variable `$_` or the default examining-address for the `x` command.)

When your program stops due to a breakpoint, `mdb` prints out the name of the function containing the breakpoint and the function argument values. You can cause `mdb` to omit the argument values by issuing the `set print-function-args off` command (see Format Options on page 57).

When a set of threads hit multiple breakpoints or watchpoints simultaneously, `mdb` displays the names of the threads that hit them. If the thread that previously had the focus is among the stopped threads, it retains the focus. However, if this thread has expired or is not stopped because of a breakpoint, watchpoint, step, or fatal error, `mdb` arbitrarily chooses a stopped thread to be the focus. All breakpoint commands are executed at this time.

## 2.1.1 Setting Breakpoints

Breakpoints are set with the `break` command (abbreviated `b`). You have several ways to say where the breakpoint should go. Two ways require `mdb` to be focused: `break` without any argument and `break` with an offset argument.

`break` *function*

Set a breakpoint at entry to function *function*. (If your program is linked with an archive, the state of the `mdb` `visibility` variable may either affect your ability to access *function* or determine in which of several functions named *function* the breakpoint is set. See Archive Symbol Visibility on page 70 for details.

`break` +*offset*, `break` -*offset*

Within the current source file, set a breakpoint some number of lines forward or back from the position at which the focus thread stopped in the currently selected frame. The focus thread and selected frame determine the current source file: the current source file contains the line the focus thread in which stopped executing in the selected frame. (See Chapter 5, Examining Source Files on page 47.)

`break` *linenum*

Set a breakpoint at line *linenum* in the current source file. The breakpoint stops the program immediately before any thread executes any of the code on that line. You may not be able to set a breakpoint by line number within a file-static function compiled without debugging information. The compiler may inline these functions at every call and not maintain a stand-alone version. If this is the case, mdb has insufficient information to set a breakpoint by line number at an arbitrary, inlined instruction of the function. However, you can set a breakpoint using the function name—which sets a breakpoint at the first line of every inlined instance of the function.

```
(mdb) break 10
No line number 10.
(mdb) break foo
Breakpoint 1 at (0:0x412) (main): foo.c line 10.
```

(In the last line above, `(main)` is an artifact of the program not having a stand-alone implementation of `foo`.)

`break` *filename*`:`*linenum*

Set a breakpoint at line *linenum* in source file *filename*.

`break` *filename*`:`*functionname*

Set a breakpoint at entry to function *functionname* found in file *filename*. Specifying a file name as well as a function name is superfluous except when multiple files contain similarly named functions.

`break *`*address*

Set a breakpoint at address *address*. You can use this to set breakpoints in parts of the program that do not have debugging information or source files. If the instruction at *address* is inlined code, `mdb` does not set any additional breakpoints in corresponding locations in the stand-alone version or other inlined instances; here `mdb` does not maintain the illusion of normal function call.

`break` Set a breakpoint at the next instruction to be executed by the focus thread in the selected stack frame (see Chapter 4, Examining the Stack on page 41).

`break ...` `if` *cond*

Set a breakpoint with condition *cond*; evaluate the expression *cond* each time the breakpoint is reached, and stop only if the value is nonzero. `...` stands for one of the possible arguments described above (or no argument) specifying where to break. See Break Conditions on page 28, for more information on breakpoint conditions.

`tbreak` *args*

Set a breakpoint enabled only for one stop. *args* are the same as in the `break` command, and the breakpoint is set in the same way, but the breakpoint is automatically disabled the first time it is hit. See Disabling Breakpoints and Watchpoints on page 27.

`mdb` allows you to set any number of breakpoints at the same place in the program. This is useful when the breakpoints are conditional Break Conditions on page 28.

#### 2.1.1.1 Special Breakpoint Situations

If you set a breakpoint by any means other than `break *`*address*, and the breakpoint is within code that has been inlined by the compiler, `mdb` maintains the illusion of normal function call. The breakpoint appears to be set within the body of the function—and thus is set in all other inlined copies of the function. If you set a breakpoint by its address, and the breakpoint is within inlined code, `mdb` creates a single breakpoint.

If you set a breakpoint on a line that contains a future statement, the break occurs at the first statement within the future body, rather than immediately before the future statement is executed.

```
8  foo(){
9    j = 5;
10   future i () {
11    k += 3;
```

The command `break 10` places a breakpoint immediately before the statement in line 11, as opposed to immediately after the statement in line 9.

Setting a breakpoint on a line with a future and another statement, however, can cause the break to take place outside the future body.

`mdb` does not allow breakpoints on a small set of instructions (instructions that contain a `MAC` operation, or any of the following operations: `DATA_OPA_SAVE`, `DATA_OPD_SAVE`, `DATA_OP_REDO`, `LEVEL_ENTER`, `LEVEL_RTN`, `RESULTCODE_SAVE`, `TRAP_RESTORE`, or `TRAP_SAVE`). If you try to set a breakpoint on one of these instructions, `mdb` will ask you to choose a different instruction for the breakpoint.

## 2.1.2 Setting Watchpoints

Use a watchpoint to stop your program immediately after the value of an expression is written, without having to identify the thread updating the expression or the instruction where the modification takes place. Watchpoints are set with the `watch` command (abbreviated `w`) and an expression Expressions on page 53.

`watch` *expression*

Set a watchpoint on *expression*. If *expression* is a data memory address, the address is preceded with an asterisk.

```
[1] (mdb) watch *0x40102bc000
```

`watch ...   if` *cond*

Set a watchpoint with condition *cond*; evaluate the expression *cond* each time the watched expression changes, and stop only if the value is nonzero. `...` stands for an new expression or the number of a previous watchpoint. To ensure that *cond* is in scope, *cond* cannot reference any stack variables. See Break Conditions on page 28, for more information on watchpoint conditions.

`twatch` *expression*

Set a watchpoint enabled only for one stop. The watchpoint is automatically disabled the first time it is hit. See Disabling Breakpoints and Watchpoints on page 27.

`info watchpoints`

This command prints a list of watchpoints and breakpoints. It is the same as `info break`.

## 2.1.3 Deleting Breakpoints and Watchpoints

With the `clear` command you can delete breakpoints according to where they are in the program. With the `delete` command you can delete an individual breakpoint or watchpoint by specifying its number.

If your program stopped because one or more breakpoints were hit, it is not necessary to delete the breakpoints for the breaking threads to proceed past them. `mdb` automatically ignores all breakpoints in the first instruction to be executed by each breaking thread.

`clear` — Delete any breakpoints at the next instruction to be executed by the focus thread in the selected stack frame (see Selecting a Frame on page 43). When the innermost frame is selected, this is a good way to delete the breakpoint at which the focus thread is stopped.

`clear` *function*, `clear` *filename*`:`*function*

Delete any breakpoints set at entry to the function *function*.

`clear` *linenum*, `clear` *filename*`:`*linenum*

Delete any breakpoints set at or within the code of the specified line.

`delete` *bnum(s)*

Delete the breakpoint(s) or watchpoint(s) of the numbers specified as arguments.

**Note:** If you delete or disable a watchpoint while it is being processed, your program may behave in incorrect or undefined ways. Before deleting or disabling watchpoints, check each thread to make sure none is in the vicinity of the use of a watched location. If you find one, try advancing it using `step` or `next`, or continue to hit the watchpoint.

### 2.1.4 Disabling Breakpoints and Watchpoints

You disable and enable breakpoints and watchpoints with the `enable` and `disable` commands, specifying one or more numbers as arguments. Use `info break` or `info watch` to print a list of breakpoints and watchpoints if you do not know which numbers to use.

A breakpoint or watchpoint can have any of four different states:

- Enabled. The point stops the program. A breakpoint made with the `break` command or a watchpoint made with the `watch` command starts out in this state.
- Disabled. The point has no effect on the program.
- Enabled once. The point stops the program, but when it does so, it becomes disabled. A breakpoint made with the `tbreak` command or a watchpoint made with the `twatch` command starts out in this state.
- Enabled for deletion. The point stops the program, but immediately after it does so, it is deleted permanently.

You change the state of a breakpoint or watchpoint with the following commands:

`disable breakpoints` *bnum(s)*
`disable` *bnum(s)*

Disable the specified breakpoint(s) or watchpoint(s). A disabled point has no effect but is not forgotten. All options such as ignore counts, conditions, and commands are remembered in case the breakpoint or watchpoint is enabled again later.

`enable breakpoints` *bnum(s)*`enable` *bnum(s)*

Enable the specified breakpoint(s) or watchpoint(s).

`enable breakpoints once` *bnum(s)*
`enable once` *bnum(s)*

Enable the specified breakpoint(s) and watchpoint(s) temporarily. Each will be disabled again the next time it stops the program (unless you have used one of these commands to specify a different state before that time comes).

`enable breakpoints delete` *bnums*
`enable delete` *bnums*

Enable the specified breakpoints and watchpoints to work once and then die. Each point is deleted the next time it stops the program (unless you have used one of these commands to specify a different state before that time comes).

## 2.1.5 Break Conditions

You can also specify a *condition* for a breakpoint or a watchpoint. A condition is a boolean expression in your programming language. (See Expressions on page 53.) A breakpoint or watchpoint with a condition evaluates the expression each time a thread reaches the breakpoint or modifies the watched expression, and the program stops only if the condition is true. Because the expression *cond* must always be in scope for watchpoints, *cond* cannot reference any stack variables.

**Note:** Avoid break conditions with side effects—for example, printing diagnostics, updating counters, or using generic functions on sync or future variables (see Changing the Full/Empty Bit on page 75). The behavior of break conditions with side effects is unpredictable: A break condition may be evaluated multiple times when the breakpoint or watchpoint is hit. The evaluation order of conditions for several breakpoints sharing the same address or several watchpoints associated with a common memory word is undetermined.

Use breakpoint or watchpoint commands as an alternate to break conditions with side effects. Command sets behave predictably and are usually more convenient and flexible for the purpose of performing side effects when a breakpoint or watchpoint is reached (see Commands Executed on Breaking on page 30).

You can specify break conditions when a breakpoint or watchpoint is set by using `if` in the arguments to the `break` or `watch` command. See Setting Breakpoints on page 23 and Setting Watchpoints on page 26. They can also be changed at any time with the `condition` command.

`condition` *bnum expression*

Specify *expression* as the break condition for breakpoint or watchpoint number *bnum*. From now on, this point stops the program only if the value of *expression* is true (nonzero, in C). *expression* is not evaluated at the time the `condition` command is given. See Expressions on page 53 for more information.

`condition` *bnum*

Remove the condition from breakpoint or watchpoint number *bnum*. It becomes an ordinary unconditional breakpoint or watchpoint.

A special case of a breakpoint or watchpoint condition is to stop only when the point has been reached a certain number of times. Every breakpoint and watchpoint has an ignore count, which is an integer. Most of the time, the ignore count is zero, and therefore has no effect. But if a thread reaches a breakpoint or watchpoint whose ignore count is positive, then instead of stopping the program, it decrements the ignore count by one and continues. As a result, if the ignore count value is *n*, the breakpoint or watchpoint does not stop the program the next *n* times it is reached.

`ignore` *bnum count*

Set the ignore count of breakpoint or watchpoint number *bnum* to *count*. The next *count* times the point is reached, it will not stop. To make the breakpoint or watchpoint stop the next time it is reached, specify a count of zero.

`cont` *count* Continue execution of all threads in the program, setting the ignore count of the breakpoint or watchpoint that the program stopped at to *count* minus one. Thus, the program does not stop at this point until the time defined by *count* is reached. This command is allowed only when the program stopped due to a breakpoint or watchpoint. At other times, the argument to `cont` is ignored. See Continuing on page 31 for more information.

**Note:** If a breakpoint or watchpoint has an ignore count of greater than 0 and a condition, the condition is not checked.

## 2.1.6 Commands Executed on Breaking

You can give any breakpoint or watchpoint a series of commands to execute when the program stops due to that point. For example, you might want to print the values of certain expressions, or enable other breakpoints or watchpoints.

`commands` *bnum*

> Specify commands for breakpoint or watchpoint number *bnum*. The commands themselves appear on the following lines. Type the `end` command to terminate the commands.
>
> To remove all commands from a breakpoint or watchpoint, use the command `commands` and follow it immediately by `end`; that is, give no commands. With no arguments, `commands` refers to the last breakpoint or watchpoint set.

When your program is suspended due to a set of breakpoints or watchpoints being hit simultaneously, for each breaking thread `mdb` executes the command sequence of each point in the set—in some arbitrary order of the command sequences. `mdb` automatically resumes execution only if each breakpoint or watchpoint in the set has a command sequence and each command sequence includes the `cont` command.

For each command sequence, `mdb` ignores all continuation commands other than `cont`, as well as all commands that follow any continuation command, including those after `cont`. For example, `mdb` ignores `step` or `finish` in a command sequence—and any commands that follow.

If you use a stepping command like `step` or `next` to advance a thread to an instruction that has some number of breakpoints, each of which has a command sequence that includes a `cont` command, `mdb` executes each command sequence but does not does not resume execution.

If the first command specified in a breakpoint or watchpoint command sequence is `silent`, the usual message about stopping at a breakpoint is not printed. This may be desirable for points that are to print a specific message and then continue. If the remaining commands too print nothing, you will see no sign that the breakpoint or watchpoint was reached at all. `silent` is not really a command; it is meaningful only at the beginning of the commands for a breakpoint or watchpoint.

The commands `echo` and `output` that allow you to print precisely controlled output are often useful in silent breakpoints or watchpoints. See Commands for Controlled Output on page 81.

For example, here is how you could use breakpoint commands to print the value of `x` at entry to `foo` whenever it is positive.

```
[1] (mdb) break foo if x>0
[1] (mdb) commands
silent
echo x is\040
output x
echo \n
cont
end
```

One application for breakpoint commands is to correct one bug so you can test another. Put a breakpoint immediately after the erroneous line of code, give it a condition to detect the case in which something erroneous has been done, and give it commands to assign correct values to any variables that need them. End with the `cont` command so that the program does not stop, and start with the `silent` command so that no output is produced. Here is an example:

```
[1] (mdb) break 403
[1] (mdb) commands
silent
set x = y + 4
cont
end
```

A similar pseudo-command is `once`. When multiple threads hit a breakpoint at the same time, by default all of the commands for that breakpoint will be executed for each thread. If the first command specified is `once`, then the commands will be executed for only one of the threads. The `silent` and `once` commands may be used together; for example, if `silent` is the first command then `once` may be second.

## 2.2 Continuing

After your program stops, you will mostly likely want it to run some more if the bug you are looking for has not yet occurred.

| | |
|---|---|
| `cont` | Continue running the entire program from the current suspended state; all threads are active (not only the focus thread). |
| `cont/1` | Continue running only the focus thread from the current suspended program state; all other threads remain suspended by `mdb`, regardless of individual thread state. |

If the program stopped because a thread hit a breakpoint, you might expect that continuing would stop immediately at the same breakpoint when it was hit again by the same thread. In fact, `cont` takes special care to prevent that from happening. You do not need to delete the breakpoint to proceed through it after stopping at it.

You can, however, specify an ignore count for the breakpoint that the program stopped at, by means of an argument to the `cont` command. See Break Conditions on page 28.

## 2.3 Stepping

*Stepping* means setting only the focus thread in motion for a limited time, so that control returns automatically to the debugger after the focus thread executes one line of code or one machine instruction. While the focus thread is stepping, all other threads remain suspended.

During a step, the focus thread may toggle the `full/empty` state (see State Bits on page 55) of a sync or future variable on which threads are blocked waiting for the opposite full/empty state. If this happens, the toggling read or write triggers in the same step an action that satisfies one of the blocked threads request to read or write the variable (possibly changing the value of the variable or the full/empty state). This type of read or write by an initially blocked thread is the only non-focus thread activity that occurs during stepping; threads unblocked in this fashion remain suspended while the current focus thread is stepping, though in a state that differs as a result of the read or write assisted by the trap handler.

`mdb` must be focused on a thread (see Focus Thread on page 37) to execute any of the following stepping commands.

| | |
|---|---|
| `step` | Continue running the focus thread until it reaches a different line, then stop the focus thread and return control to the debugger. This command is abbreviated `s`.<br><br>This command may be given when a thread is within a function for which there is no debugging information. In this case, execution proceeds until the thread reaches a different function, or is about to return from this function. An argument repeats this action. |
| `step` *count* | Continue running as in `step`, but do so *count* times. |

`next` Similar to `step`, but any function calls appearing within the line of code are executed by the focus thread without stopping. Execution stops when the focus thread reaches a different line of code at the stack level that was executing when the `next` command was given. This command is abbreviated `n`.

An argument is a repeat count, as in `step`.

`next` within a function without debugging information acts as does `step`; any function calls appearing within the code of the function are executed by the focus thread without stopping.

`finish` Continue running the focus thread until immediately after the selected stack frame returns (or until there is some other reason to stop, such as a fatal signal or a breakpoint). Print value returned by the selected stack frame (if any).

`until` This command is used to avoid single-stepping a thread through a loop more than once. It is like the `next` command, except that when `until` encounters a jump, it automatically continues execution of the focus thread until the program counter is greater than the address of the jump.

This means that when the focus thread reaches the end of a loop after single-stepping though it, `until` causes the program to continue execution until the loop is exited. In contrast, a `next` command at the end of a loop steps the focus thread back to the beginning of the loop, which forces the thread to step through the next iteration.

`until` always stops the focus thread if it attempts to exit the current stack frame.

**Note:** `until` may produce somewhat counter-intuitive results if the order of the source lines does not match the actual, optimized order of execution. For example, in a typical C `for` loop, the third expression in the `for` statement (the loop-step expression) is executed after the statements in the body of the loop, but is written before them. Therefore, the `until` command appears to step the focus thread back to the beginning of the loop when it advances to this expression. However, it has not really done so—not in terms of the actual machine code.

`until` *location*

Continue running the focus thread until either the specified location is reached, or the current (innermost) stack frame returns. This form of the command uses breakpoints, and hence is quicker than `until` without an argument.

`stepi, si` Execute the focus thread for one machine instruction, then stop and return to the debugger.

It is often useful to do `display/i $pc` when stepping by machine instructions. This causes the next instruction to be executed by the focus thread to be displayed automatically at each stop. See Automatic Display on page 62.

An argument is a repeat count, as in `step`.

`nexti, ni` Execute the focus thread for one machine instruction, but if the instruction is a subroutine call, proceed until the subroutine returns.

`nexti` within a trap handler acts as does `stepi`, stopping at the next machine instruction that is outside the trap handler.

An argument is a repeat count, as in `next`.

A typical technique that uses stepping is to put a breakpoint (see Breakpoints and Watchpoints on page 21) at the beginning of the function or the section of the program in which a problem is believed to lie, and then step one or more threads through the suspect area, examining the variables that are interesting, until the problem happens.

You can achieve the effect of some of the stepping commands within a trap handler by setting breakpoints at each line or instruction of the trap handler. Other debugging commands, such as examining and altering memory, work within trap handlers as they normally do.

You can use the `cont/1` command after stepping to resume execution of only the focus thread until the next breakpoint or signal. See Continuing on page 31.

Your program is probably linked with some number of standard libraries such as `libc`, `libm`, and `librt`. By default, if you step the focus thread through code that contains a call into one of the functions in these libraries, `mdb` will step completely over the function. You can alter this behavior and have `mdb` step into the function by setting the `mdb` variable `enter-stdlib` to true.

```
set enter-stdlib true
```

Setting `enter-stdlib` to false restores the default behavior.

# Understanding Multithreading [3]

The Cray XMT compilers automatically identify sections of source code that can be partitioned into independent and parallel operations. When you execute this compiled code on a Cray XMT system, each program starts as a single thread. As the program executes, it spawns new threads to perform simultaneous and parallel operations, while existing threads may be waiting for a resource or the completion of a memory reference, and yet other threads are completing their tasks and disappearing. As a result, the set of threads executing in your program changes dynamically throughout the life of the program, in both number and nature.

At any intermediate point in the execution of your program, `mdb` knows only about the threads that are currently part of the execution; that is, the threads that are running or waiting to run. When your program is suspended under `mdb`, you can ask `mdb` for information about the current set of threads and perform debugging operations on individual threads (see Focus Thread on page 37).

Each thread originates either from a future statement in your source code or as part of an automatic compiler optimization. Each future statement explicitly determines a primary thread responsible for executing the body of the future statement, while compiler optimizations are performed automatically, when the compiler recognizes an opportunity to improve code performance by executing certain sections of code simultaneously.

For example, the compiler may recognize a loop whose iterations are relatively independent. The compiler in this case partitions the entire execution of the loop (where each resulting component is usually one or more loop iterations) and directs the single thread that encounters the loop to split into a set of threads, or *frays*. Each thread of the fray independently executes some component of the overall execution of the loop, in parallel with the other fray members. If the fray size, which is determined at runtime, is less than the number of execution components, a fray thread may execute more than one component.

## 3.1 Thread Names

The `mdb` debugger learns about the threads in your program progressively and assigns each new thread a unique integer identifier. The `mdb` debugger reassigns the identifiers, starting from 0, each time your program is run. Due to timing issues inherent in parallel programs, the mapping of identifiers to threads may be different from one program run or debugging session to the next.

The runtime system also assigns a name to each thread in your program. They are of the form `a.b` or `a.b.c`, where `a`, `b`, and `c` are non-negative integers, and the names conform to the following rules.

- `0.1` names the initial thread.
- The form `a.b` designates a thread that is usually determined by a future statement in your source code.
- The runtime name form `a.b.c` designates a thread that is generated strictly by the compiler.

For example, a fray thread has a three-component name. The runtime system gives related names to threads in a group generated by the compiler to execute a particular section of code. A fray is a typical example of such a group. The names of all threads in a compiler-generated group differ only in the third component.

You can change the form of thread name `mdb` uses with the `set id-style` command.

`set id-style system`

Use the runtime system names of the form `a.b` or `a.b.c` for thread names.

`set id-style mdb`

Use integers for thread names. This is the default.

You may also start `mdb` using the command-line option `-id-style` with the same argument choice.

On rare occasions, the thread name in the prompt may be a negative integer—indicating a runtime thread running on a dedicated stream. See Focus Thread on page 37 for more details.

## 3.2 Thread States

The state of a thread persists during suspension of the program by `mdb`. For example, if a running thread *t* hits a breakpoint (see Breakpoints and Watchpoints on page 21) and `mdb` stops your program, when `mdb` resumes execution, *t* will be running.

A thread *t* is in one of the following states:

| | |
|---|---|
| `running` | Thread *t* is executing. |
| `startable` | A thread in your program has executed the future statement that establishes thread *t*; *t* has not run; *t* will run when execution resources become available. |
| `blocked` | Thread *t* is waiting for a memory reference to complete and has released the execution resources it was using. |
| `spinning` | Thread *t* is waiting for a memory reference to complete; while it waits, it continues to execute but does not make progress. |
| `resumable` | The memory reference on which thread *t* blocked has completed; *t* will resume running when execution resources become available. |
| `aborted` | Thread *t* has experienced a fatal error and is not resumable. |
| `indeterminate` | Thread *t* is in transition between two of the previous states; `mdb` is unable to determine the recent or impending state of *t*. |

`mdb` retains no knowledge of threads that have completed and disappeared.

## 3.3 Focus Thread

When you run your program under `mdb`, you or `mdb` may use the `thread` command to designate a single thread as the *focus thread*—the thread of particular interest. Many debugging operations refer implicitly to the focus thread. If `mdb` is focused on a single thread, the identifier of the focus thread appears on the same line as, and in front of, the prompt.

```
[1] (mdb)
```

You can change the format of the thread state description with the `set state-length` command. You can explicitly set the focus thread to be one of the threads through the `thread` command. The focus thread is the target thread of any `mdb` command that pertains to a single thread, such as `step` or `backtrace`. If `mdb` is not focused when you issue such a command, `mdb` returns an error message stating the need for a focus.

Once your program has begun running, one or more threads execute your program. When `mdb` suspends execution, you can examine the threads currently comprising the execution of your program by issuing the info threads command.

`info threads`

Print a table of the current set of threads in your program sorted by thread state and breakpoint number. By default, a compressed list of thread ids are printed.

`info threads/l`

Print table in long format (all thread IDs).

`info threads/f`

Sort table by the name of the function where execution is currently stopped as well as thread state and breakpoint number.

`info threads/n`

Print only the total number of threads in each state or function.

`info threads/b` *number*

Print only threads stopped at breakpoint *number*.

`info threads/v`

Print verbose information about the current set of threads, including thread id, system name, and name of function where execution is currently stopped. By default, the number of threads displayed in verbose mode is limited to 500. Use `set info-limit` to change the limit.

`set state-length`

Print the current value of `state-length`.

`set state-length` *length*

Set the length of the state description `mdb` prints before the prompt for threads that are not in state running. The argument *length* may take on one of the following values.

| | |
|---|---|
| `long` | Print the state as a long name. This is the default. |
| `short` | Print the state as the first character of the long name. |
| `none` | Print no state description. |

You can explicitly set the focus thread to be one of the threads through the thread command. The focus thread is the target thread of any `mdb` command that pertains to a single thread, such as step or backtrace. If `mdb` is not focused when you issue such a command, `mdb` returns an error message stating the need for a focus.

`thread` *thread-name*

Set the target thread of subsequent `mdb` commands that pertain to a single thread to *thread-name*.

In the following example, `info threads` lists the names of the threads in the program, as well as the state of each thread. Thread 1 is running, since no state is printed to the left of the thread identifier.

```
[1] (mdb) info threads
Thread      State       Brk
  1        running       1
  2        startable
  3        startable
          .
          .
          .
[1] (mdb)
```

Below, the `thread` command changes the focus thread from 1 to 2 then back to 1. Line 149 is the current source position of startable thread 2. Line 30 is the current source position of thread 1.

```
[1] (mdb) thread 2
149           future $left_done (data, left) {         // fork left
<startable> [2] (mdb) thread 1
30         for (int j = 1;  j < size;  j++) {
[1] (mdb)
```

For the `thread` command, the square brackets (`[]`) around the thread name are optional.

Program execution may be interrupted if your program receives certain UNIX signals or if some set of parallel threads hits a breakpoint or watchpoint, raises a fatal exception, or completes a `mdb` stepping command. After execution is suspended, `mdb` determines if there is a thread of particular interest. If so, `mdb` focuses on that thread; otherwise `mdb` does not set a focus thread.

For example, when you type `Ctrl-C`, `mdb` remembers the previous focus thread, which `mdb` retains as the new focus thread if the thread is still active. If the previous focus thread is blocked or otherwise unable to resume execution, however, `mdb` prints out a message to that effect and leaves the focus thread unset.

On rare occasions, `mdb` may focus on a thread whose name is a negative integer. This is a runtime thread whose underlying stream is dedicated to the runtime, such as a daemon. `mdb` may focus on such a thread if a runtime daemon hits a user-set breakpoint or watchpoint (see Breakpoints and Watchpoints on page 21). These runtime threads executing on dedicated streams are never listed in the output of `info threads`.

If you set the focus thread to be a thread that has just executed an instruction that raised an exception and caused the thread to trap, `mdb` sets the current source position to the line containing the next instruction to execute after the trap is handled, which may be up to eight dynamic instructions beyond the trapping instruction. In this case, any instructions between the trapping instruction and the current source position are executed before the trap is handled. Additionally, if the trapping instruction contains a function return, the current source line may even be in a source file different than the one where the trap occurred.

# Examining the Stack [4]

Each time a thread performs a function call, the information about where in the program the call was made from is saved in a block of data called a *stack frame*. The frame also contains the arguments of the call and the local variables of the function that was called. For each thread, all the stack frames are allocated in a region of memory called the *call stack*.

`mdb` recognizes when the compiler optimizes your code by inlining a function call (substituting the function body for the call statement). For inlined functions, `mdb` maintains the illusion of a normal function call.

When your program stops, the `mdb` commands for examining the stack of the focus thread allow you to see all of the information saved in the stack frame.

When the program stops, `mdb` automatically selects the currently executing frame of the focus thread and describes the frame briefly as the `frame` command does (see Information on a Frame on page 45).

Whenever you ask `mdb` for the value of a variable in the program, the value is found in the selected frame. There are special `mdb` commands to select whichever frame of the focus thread you are interested in.

## 4.1 Stack Frames

The call stack of a thread is divided up into contiguous pieces called *stack frames*, or *frames* for short; each frame is the data associated with one call to one function. The frame contains the arguments given to the function, variables local to that function, and the address at which the function is executing.

When a thread starts, its stack has only one frame. This is called the *initial* frame or the *outermost* frame. Each time a function is called by the thread, a new frame is made. Each time the thread returns from a function, the frame for that function invocation is eliminated. If a function is recursive, there can be many frames for the same function. The frame for the function in which the thread is actually executing is called the *innermost* frame. This is the most recently created of all the thread stack frames that still exist.

Inside your program, stack frames are identified by their addresses. A stack frame consists of many words, each of which has its own address. The address of the first word of the frame serves as the address of the frame itself. For each thread, this address is kept in a register called the *stack pointer register* while the thread is executing in that frame.

For each thread, `mdb` assigns numbers to all existing stack frames, starting with zero for the innermost frame, one for the frame that called it, and so on upward. These numbers do not really exist in your program, but give you a way of talking about stack frames in `mdb` commands.

Many `mdb` commands refer implicitly to one stack frame of one parallel thread. The implied thread can be selected by you or by `mdb` (see Focus Thread on page 37). Once focused, `mdb` records a stack frame that is called the *selected* stack frame; you can select any frame of the focus thread by using one set of `mdb` commands, and then other commands will operate on that frame. When your program stops, if `mdb` focuses on a thread, `mdb` automatically selects the innermost frame for the focus thread.

## 4.2 Backtraces

A backtrace is a summary of how a thread got where it is. It shows one line per frame, for many frames, starting with the currently executing frame (frame zero), followed by its caller (frame one), and on up the stack of the thread. *Backstops* are initial (outermost) frames on the stack: `main` for the first thread in your program, the initial frame for any subsequent thread.

`backtrace, bt`

Print a backtrace of the entire stack of the focus thread: one line per frame for all frames in the stack. You can stop the backtrace at any time by typing `Ctrl-C`.

`backtrace` *n*`, bt` *n*

Similar, but print only the innermost *n* frames.

`backtrace -`*n*`, bt -`*n*

Similar, but print only the outermost *n* frames.

`backtrace/all, bt/all`

Backtrace all current threads. This is equivalent to issuing `backtrace` for each thread in your program.

The names `where` and `info stack` are additional aliases for `backtrace`—and require that `mdb` be focused.

Every line in the backtrace shows the frame number, the function name, and the program counter value.

If the function is in a source file whose symbol table data has been fully read, the backtrace shows the source file name and line number. The program counter value is omitted if it is at the beginning of the code for that line number.

If the symbol data in the source file has only been scanned and not fully read, this extra information is replaced with an ellipsis. You can force the symbol data for that frame's source file to be read by selecting the frame. (See Selecting a Frame on page 43).

Here is an example of a backtrace.

```
[1] (mdb) backtrace
#0  foobar2()(hello.c line 11)
#1  foobar1()(hello.c line 18)
#2  foo()(hello.c line 28)
#3  main(hello.c line 60)
```

## 4.3 Selecting a Frame

Most commands for examining the stack and other data in the program work on whichever stack frame is selected at the moment. Here are the commands for selecting a stack frame, usually of the focus thread.

| | |
|---|---|
| `frame` *n* | Select frame `number` *n* of the focus thread. Recall that frame zero is the innermost (currently executing) frame. Frame one is the frame that called the innermost one, and so on. The highest-numbered frame is the initial frame of the focus thread. |
| `frame` *addr* | Select the frame at address *addr*. This is useful mainly if the chaining of stack frames has been damaged by a bug, making it impossible for `mdb` to assign numbers properly to all frames. In addition, this can be useful when a thread has multiple stacks and switches between them. |
| `up` *n* | Select frame *n* frames up from the frame previously selected. For positive numbers *n*, this advances toward the outermost frame, to higher frame numbers, to frames that have existed longer. *n* defaults to one. |
| `down` *n* | Select the frame *n* frames down from the frame previously selected. For positive numbers *n*, this advances toward the innermost frame, to lower frame numbers, to frames that were created more recently. *n* defaults to one. |

`upto` *regexp*

Select the first frame in the calling stack whose function name matches *regexp*. For instance, if a backtrace shows functions `sprint`, `print1`, `print` and `main`, and the current frame is at `sprint`, the command `upto print` would select the frame at `print1`. `upto print$` would go to the frame at `print`. `upto` also functions as a boolean expression and can be used as the condition for the `if` or `while` commands. When used in this manner, it must be the only expression within the condition. Also when used as a condition, no frame information is printed; use the `frame` command with no argument within the body of the `if` or `while` to print out the frame information, if necessary.

`downto` *regexp*

Select the last frame in the calling stack whose function name matches *regexp*. For instance, if a backtrace shows functions `sprint`, `print1`, `print` and `main`, and the current frame is at `print`, the command `downto print` would select the frame at `print1`. `downto print$` would go to the frame at `sprint`. `downto` also functions as a boolean expression and can be used as the condition for the `if` or `while` commands. When used in this manner, it must be the only expression within the condition. Also when used as a condition, no frame information is printed; use the `frame` command with no argument within the body of the `if` or `while` to print out the frame information, if necessary.

All of these commands (except `upto` and `downto` when used as a condition for `if` or `while`) end by printing some information on the frame that has been selected: the frame number, the function name, the arguments, the source file and line number of execution in that frame, and the text of that source line. For example:

```
#3  main (argc=3, argv=??, env=??) at main.c, line 67
67        read_input_file (argv[i]);
```

After such a printout, the `list` command with no arguments will print ten lines centered on the point of execution in the frame. Printing Source Lines on page 47.

## 4.4 Information on a Frame

There are several other commands to print information about a stack frame, usually the selected frame of the focus thread.

`frame`
: Print a brief description of the selected stack frame. You can abbreviate it `f`. With an argument, this command is used to select a stack frame; with no argument, it does not change which frame is selected, but still prints the same information.

`info frame`
: Print the stack level, the address of the frame, and the program counter of the selected stack frame. This description is useful when something has gone wrong that has made the stack format fail to fit the usual conventions.

`info frame` *addr*
: Print the address of the selected frame along with its program counter, function name, and source location (if known).

`info args`
: Print the arguments of the selected frame, each on a separate line. Also, see the `set print-function-args` command in Format Options on page 57.

`info locals`
: Print the local variables of the selected frame, each on a separate line. Every variable declared static or automatic in the current scope is printed.

# Examining Source Files [5]

`mdb` knows which source files your program was compiled from, and can print parts of their text. When your program stops, if `mdb` automatically determines the current focus thread, then `mdb` spontaneously prints the line the focus thread stopped in. Likewise, when you select a stack frame (see Selecting a Frame on page 43), `mdb` prints the *current source line* in which the focus thread stopped executing in that frame. The *current source file* contains the line in which the focus thread stopped executing in the selected stack frame. If the program has not yet been run, the current source file is that of `main` for C/C++ programs.

`mdb` only knows about source files encountered during the course of running your program. If you wish to access source information yet to be seen by `mdb`, use the `load` command with the pertinent module name as an argument. (See Module Commands on page 13.)

You can also print parts of source files by explicit command.

## 5.1 Printing Source Lines

To print lines from a source file, use the `list` command (abbreviated `l`). There are several ways to specify what part of the file you want to print.

Here are the forms of the `list` command most commonly used:

`list` *linenum*

Print ten lines centered around line number *linenum* in the current source file.

`list` *function*

Print ten lines centered around the beginning of function *function*.

`list`

Print ten more lines. If the last lines printed were printed with a `list` command, this prints ten lines following the last lines printed; however, if the last line printed was a solitary line printed as part of displaying a stack frame (see Chapter 4, Examining the Stack on page 41), this prints ten lines centered around that line.

`list -`

Print the ten lines immediately before the lines last printed.

Repeating a `list` command with `RET` discards the argument, so it is equivalent to typing only `list`. This is more useful than listing the same lines again. An exception is made for an argument of `-`; that argument is preserved in repetition so that each repetition moves up in the file.

In general, the `list` command takes zero, one, or two *linespecs* as arguments. A linespec is a way in which a particular line in the source file can be specified; there are several ways of writing them but the effect is always to specify some source line. Here is a complete description of the possible arguments for `list`:

| | |
|---|---|
| `list` *linespec* | Print ten lines centered around the line specified by *linespec*. |
| `list` *first*`,`*last* | Print lines from *first* to *last*. Both arguments are linespecs. |
| `list,` *last* | Print ten lines ending with *last*. |
| `list` *first*`,` | Print ten lines starting with *first*. |
| `list +` | Print the ten lines immediately after the lines last printed. |
| `list -` | Print the ten lines immediately before the lines last printed. |
| `list` | As described in the preceding table. |

Here are the ways of specifying a single source line--all the kinds of linespec.

| | |
|---|---|
| *linenum* | Specifies line *linenum* of the current source file. When a `list` command has two linespecs, this refers to the same source file as the first linespec. |
| `+`*offset* | Specifies the line *offset* lines after the last line printed. When used as the second linespec in a `list` command that has two, this specifies the line *offset* lines down from the first linespec. |
| `-`*offset* | Specifies the line *offset* lines before the last line printed. |
| *filename*`:`*linenum* | Specifies line *linenum* in the source file *filename*. |
| *function* | Specifies the line that begins the body of the function *function*. |

*filename*`:`*functionname*

Specifies the line that begins the body of the function *functionname* in the file *filename*. The file name is needed with a function name only for disambiguation of identically named functions in different source files.

`*`*address*

Specifies the line containing the program address *address*. *address* may be any expression.

Two commands relate source lines and program addresses.

`info line` *linenum*

Print the starting and ending addresses of the compiled code for source line *linenum*. `mdb` reports an address range for each inlined instance of the source line *linenum*. Unlike GDB, `info line` in `mdb` does not set either the default examine address for the `x` command or the convenience variable `$_`.

`info pc` *address*

Print the source lines from which the operations in the instruction at *address* are derived. Because of compiler optimizations, `mdb` may not be able to identify the source lines for the single given instruction. When this happens, `mdb` prints the source lines for a small range of instructions that includes the instruction at *address*.

```
(mdb) info pc 0x401
Source lines for pc range: 0x401..0x403
main.c:11 (foo())
11     {
12   for (int i=0; i<20; i++) {
```

The default *address* argument for `info pc` is the instruction at which the focus thread is stopped.

## 5.2 Searching Source Files

There are two commands for searching through the current source file for a regular expression.

The command `forward-search` *regexp* checks each line, starting with the one following the last line listed, for a match for *regexp*. It lists the line that is found. You can abbreviate the command name as `for`.

The command `reverse-search` *regexp* checks each line, starting with the one before the last line listed and going backward, for a match for *regexp*. It lists the line that is found. You can abbreviate this command with as little as `rev`.

## 5.3 Specifying Source Directories

The path to the source file passed to the compiler or calculated by the front end (for include files) is recorded in the corresponding program library. If the executable moves or if any directories move between the compilation and your debugging session, you must tell `mdb` where to find the source files for your program. `mdb` has a list of directories to search for source files; this is called the *source path*. Each time `mdb` wants a source file, it tries in order the directories in the source path, until it finds a file with the desired name. Note that the executable search path is not used for this purpose. The current working directory is always the last item in the source path, and is displayed as `$cwd`.

If `mdb` cannot find a source file in the source path, and the program library records a directory, `mdb` tries that directory too. If the source path is empty, and there is no record of the compilation directory, `mdb` looks in the current directory as a last resort.

Whenever you reset or rearrange the source path, `mdb` clears out any information it has cached about where source files are found and where each line is in the file.

When you start `mdb`, its source path is empty. To add other directories, use the `directory` command.

`directory` *dirname*`, dir` *dirname*

Add directory *dirname* to the front of the source path. Several directory names may be given to this command, separated by `:` or white space. You may specify a directory that is already in the source path; this moves it forward, so `mdb` searches it sooner. You can use the string `$cwd` to refer to the current working directory. `$cwd` is not the same as `.`—the former tracks the current working directory as it changes during your `mdb` session, while the latter is immediately expanded to the current directory at the time you add an entry to the source path.

`directory` Reset the source path to `$cwd` again. This requires confirmation.

`info directories`

Print the source path: show which directories it contains.

`set linkdir` *dir*

If you moved your executable after it was linked, tell `mdb` that your executable was linked from directory *dir*. This enables `mdb` to find the modules for your program based on the information in your root program library. Note that the root program library should be in its original directory.

If your source path is cluttered with directories that are no longer of interest, `mdb` may sometimes cause confusion by finding the wrong versions of source. You can correct the situation as follows:

1. Use `directory` with no argument to reset the source path to `$cwd`.
2. Use `directory` with suitable arguments to reinstall the directories you want in the source path. You can add all the directories in one command.

## 5.4 Examining Instructions

Sometimes it is useful to examine the low-level machine instructions generated by the compiler. The specialized command `disassemble` dumps a range of memory as machine instructions.

`disassemble`

Disassemble the function surrounding the program counter of the selected frame of the focus thread.

`disassemble` *function*

Disassemble the specified *function*.

`disassemble` *pc*

Disassemble the function surrounding the specified program counter.

`disassemble` *start_pc end_pc*

Disassemble the range of memory locations between *start_pc* and *end_pc*.

When a program gets a data exception such as a data protection violation or data alignment error, the `info opa` command can be used to try to determine the offending machine instruction. The `info opa` command prints out the list of instructions that may be responsible for the trap.

The `info opa` command takes as an argument the value of the `opa` register, the contents of the `t1` register, and the program counter where the data exception occurred. All three are printed out when an exception is encountered.

If invoked without arguments, `info opa` uses the current values of the `opa` and `t1` registers and the program counter.

# Examining Data [6]

The usual way to examine data in your program is with the `print` command (abbreviated `p`). It evaluates and prints the value of any valid expression of the language the program is written in. Enter:

```
[1] (mdb) print exp
```

where *exp* is any valid expression, and the value of *exp* is printed in a format appropriate to its data type.

You may need to provide `mdb` with type information if your program has yet to encounter the type name or type definition you wish to use in *exp*. Use the `load` command to inform `mdb` about type and other information contained in a module yet to be assimilated by `mdb` (see Module Commands on page 13).

If you use a function or variable name from a linked archive in an expression as part of a `mdb` command, the state of the `mdb visibility` variable determines whether you can access the symbol, as well as which one of several entities with the same name is being used. See Archive Symbol Visibility on page 70 for details.

A more low-level way of examining data is with the `x` command. It examines data in memory at a specified address and prints it in a specified format.

## 6.1 Expressions

Many different `mdb` commands accept an expression and compute its value. Any kind of constant, variable, or operator defined by the programming language you are using is legal in an expression in `mdb`. This includes conditional expressions, function calls, casts, and string constants. It unfortunately does not include symbols defined by preprocessor `#define` commands.

For parsing expressions and formatting printed data, `mdb` uses by default either the language of the current module in your executable program or the most recently known language, if the language information for the module cannot be found. This default language mode is called `auto`. If you want `mdb` to use a specific language regardless of the current module, use the `set language` command with either `C` or `C++`. The command `set language auto` returns the `mdb`language mode to the default. For the purposes of parsing expressions and formatting data, `mdb` considers C and C++ to be the same language. The `info language` command returns the current `mdb` expression language.

If evaluating an expression involves calling a function in your program, any side effects of the call are realized. In particular, any data references as a result of the call change state bits as if the references were executed by a thread in your program.

If you type `Ctrl-C` while `mdb` is evaluating an expression, `mdb` tries to interrupt the evaluation at a point where no locks are held. It may fail however, and locks may be left in an abnormal state on return from the interrupt. Typically, interrupting the printing of a large array or structure can be done safely.

`mdb` does not currently support calling a function defined in your program that contains a future statement. If you call such a function from `mdb`, `mdb` may hang.

Casts are supported in C and C++. It is often useful to cast a number into a pointer so as to examine a structure at that address in memory.

`mdb` supports three kinds of operators, in addition to those of programming languages:

`@`
: `@` is a binary operator for treating parts of memory as arrays. See Artificial Arrays on page 56 for more information.

`::`
: `::` allows you to specify a variable in terms of the file or function it is defined in. This use is in addition to its use when specifying class or namespace membership in C++. See Program Variables on page 54.

`{`*typename*`}` *addr*
: Refers to an object of type *typename* stored at address *addr* in memory. *addr* may be any expression whose value is an integer or pointer (but parentheses are required around nonunary operators, as with a cast). This construct is allowed regardless of what kind of data is officially supposed to reside at *addr*.

## 6.2 Program Variables

The most common kind of expression to use is the name of a variable in your program.

Variables in expressions are understood in the selected stack frame. See Selecting a Frame on page 43 and Focus Thread on page 37.) Variables must either be global (or static) or be visible according to the scope rules of the programming language from the point of execution in that frame. This means that in the function:

```
foo (int 2);
{
  bar (a);
  {
    int b = test ();
    bar (b);
  }
}
```

the variable `a` is visible whenever the focus thread is executing within the function `foo`, but the variable `b` is visible only while the focus thread is executing inside the block in which `b` is declared.

## 6.3 State Bits

Every physical data memory cell contains a 64-bit (word) value and has associated with it four access state bits: trap 0, trap 1, forward, and `full/empty`. Use the `x` command to view the value of these memory state bits for a particular word. See Examining Memory on page 60.

When `mdb` prints the value of a variable in your program, `mdb` may also print the state bits associated with the variable. Variables whose types occupy less than a word may be packed several to a memory word. Each packed variable shares its memory word state bits with other variables packed into the same word. Variables whose type occupies one, two, or four words have a corresponding number of sets of state bits. The examples and descriptions assume that each variable occupies no more than a word and has a single set of state bits unless stated otherwise.

If the variable is *normal*—that is, if it is not qualified as being *sync* or *future*, `mdb` ignores the `full/empty` bit. For each of the trap 0 and trap 1 bits, `mdb` prints the names of the trap bits that are on, for each word the variable occupies.

When printing the value of a sync or future variable, `mdb` always lists the state of the `full/empty` bit (`full` or `empty`), as well as any of the trap 0 and trap 1 bits that are on, for each word the variable occupies.

For variables whose value is determined by following an address chain defined by one or more set forward bits, `mdb` prints the value at the end of the chain. When printing a forwarded variable, `mdb` gives no indication of the set forward bits. Use the `x` command on the address of the word where the variable is stored to see the state of forward bits (see Examining Memory on page 60).

When `mdb` prints a variable, `mdb` leaves the state bits unchanged. In particular, `mdb` does not change the `full/empty` bit from full to empty when printing a sync or future variable. Rather than "consuming" the value, `mdb` looks at it.

Suppose `a$` and `b$` are sync variables.

```
[1] (mdb) print a$
$1 = 5 (full)
[1] (mdb) print a$
$2 = 5 (full)
[1] (mdb) print b$
$3 = 2 (empty)
[1] (mdb) print b$
$4 = 2 (empty)
```

You may change the `full/empty` bit of a sync or future variable, thereby perhaps unblocking any threads that happen to be blocked on that variable, using one of the Cray XMT generic functions to simulate an "active" read or write of a sync or future variable. (See Changing the Full/Empty Bit on page 75.)

Similarly, when `mdb` assigns a value to variables, no state bits are changed.

```
[1] (mdb) print b$
$3 = (empty) 2
[1] (mdb) set b$ = 10
[1] (mdb) print b$
$5 = (empty) 10
```

If any of the forward bits or the trap bits of a variable are set, the actual value of the variable may be only indirectly accessible from its nominal address (see Examining Memory on page 60). The ability of `mdb` to print the correct value of the variable and state is not affected.

When `mdb` calls a function in your program as part of evaluating an expression, any resulting data references that normally change state bits do indeed change state bits.

## 6.4 Artificial Arrays

It is often useful to print out several successive objects of the same type in memory; a section of an array, or an array of dynamically determined size for which only a pointer exists in the program.

You can do this by constructing an *artificial array* with the binary operator `@`. The left operand of `@` should be the first element of the desired array, as an individual object. The right operand should be the length of the array. The result is an array value whose elements are all of the type of the left argument. The first element is actually the left argument; the second element comes from bytes of memory immediately following those that hold the first element, and so on. Here is an example. If a program contains:

```
int *array = (int *) malloc (len * sizeof (int));
```

you can print the contents of `array` with:

```
[1] (mdb) p *array@len
```

The left operand of `@` must reside in memory. Array values made with `@` in this way behave as other arrays in terms of subscripting—they are coerced to pointers when used in expressions. (It would probably appear in an expression using the value history, after you had printed it out.)

## 6.5 Format Options

`mdb` provides a few ways to control how arrays and structures are printed.

`info format`

Display the current settings for the format options.

`set prettyprint on`

Cause `mdb` to print structures in an indented format with one member per line, like this:

```
$1 = {
  next = 0x0,
  flags = {
    sweet = 1,
    sour = 1
  },
  meat = 0x54 "Pork"
}
```

`set prettyprint off`

Cause `mdb` to print structures in a compact format, like this:

```
$1 = {next = 0x0, flags = {sweet = 1, sour = 1}, meat = 0x54 "Pork"}
```

This is the default format.

`set unionprint on`

Tell `mdb` to print unions that are contained in structures. This is the default setting.

`set unionprint off`

Tell `mdb` not to print unions that are contained in structures. For example, given the declarations:

```
typedef enum {Tree, Bug} Species;
typedef enum {Big_tree, Acorn, Seedling} Tree_forms;
typedef enum {Caterpillar, Cocoon, Butterfly} Bug_forms;

struct thing {
  Species it;
  union {
    Tree_forms tree;
    Bug_forms bug;
  } form;
};

struct thing foo = {Tree, {Acorn}};
```

with `set unionprint on` in effect `p foo` prints:

```
$1 = {it = Tree, form = {tree = Acorn, bug = Cocoon}}
```

and with `set unionprint off` in effect it prints:

```
$1 = {it = Tree, form = {...}}
```

`set stringprint on`

Tell `mdb` to automatically print the value of character strings. This is the default setting.

`set stringprint off`

Tell `mdb` not to print the value of character strings. C arrays of characters not on the heap are unaffected.

`set print-function-args off`

Turn off printing of function argument values when displaying function information. By default, `mdb` prints function argument values.

You can start a `mdb` session with argument printing turned off by invoking `mdb` with the command-line option `-no-function-args`.

`set print-function-args on`

Turn on printing of function argument values when displaying function information. This is the default.

`set array-max` *number-of-elements*

If `mdb` is printing a large array, it stops printing after it has printed the number of elements set by the `set array-max` command. This limit also applies to the display of strings. The default number of array elements printed is 200.

## 6.6 Output Formats

`mdb` normally prints all values according to their data types. Sometimes this is not what you want. For example, you might want to print a number in hex, or a pointer in decimal. Or you might want to view data in memory at a certain address as a character string or an instruction. You can do these things with *output formats*.

The simplest use of output formats is to say how to print a value already computed. This is done by starting the arguments of the `print` command with a slash and a format letter. The format letters supported are:

| | |
|---|---|
| `x` | Regard the bits of the value as an integer, and print the integer in hexadecimal. |
| `d` | Print as integer in signed decimal. |
| `u` | Print as integer in unsigned decimal. |
| `o` | Print as integer in octal. |
| `a` | Print as an absolute address in hex. |
| `c` | Regard as an integer and print it as a character constant. |
| `f` | Regard the bits of the value as a floating-point number and print using typical floating-point syntax. |

For example, to print the program counter of the focus thread in hex (see Registers on page 66), type:

```
[1] (mdb) p/x $pc
```

Note that no space is required before the slash; this is because command names in `mdb` cannot contain a slash.

To reprint the last value in the value history with a different format, you can use the `print` command with only a format and no expression. For example, `p/x` reprints the last value in hex.

See Expressions on page 53 for details of the `set language` command, which directs `mdb` to format printed data in a specific programming language.

## 6.6.1 Examining Memory

You can use the command `x` to examine data memory without reference to the data types within the program. The format in which you wish to examine memory is instead explicitly specified. The allowable formats are a superset of the formats described in the previous section.

You cannot specify a qualified format for the `x` command. In particular, the `x` command examines data memory without regard to whether the program considers the data to be qualified as sync or future, or to be unqualified. `x` always prints the data value of the actual memory word, as well as the value of the `full/empty` bit associated with the examined word (even if you are looking at only a part of the word), and any of the trap 0, trap 1, and forward bits that are on.

The `x` command prints the value of the state bits on the examined memory if the state bits are not in what is considered the default state. This description comes after the value. With the `/v` option, for `verbose`, the state bits are printed without regard for their values. The default states print as `~trap0`, `~trap1`, `empty`, `~fwd`, while the non-defaults are `trap0`, `trap1`, `full` and `fwd`.

If the trap 0 bit is set for a word *w*, then the value printed for *w* is not the value of the variable your program associates with the variable stored in *w*. Instead, *w* holds an identifier used by the runtime to locate the actual value of the variable associated with *w*. To see the value of the variable, use the `mdb print` command (see Chapter 6, Examining Data on page 53) in conjunction with info address *symbol* (see Chapter 7, Examining Symbols on page 69).

If the trap 1 bit is set for a word, the value of that word may not be the value of the variable your program associates with the word. As in the case for a set trap 0 bit, the word may instead contain an identifier.

If the forward bit is on for the word *w*, you can examine the forwarded value of *w* by examining the memory location stored in *w*.

`x` is followed by a slash and an output format specification, followed by an expression for an address. The expression need not have a pointer value (though it may). It is used as an integer, as the address of a byte of memory. See Expressions on page 53 for more information on expressions. For example, `x/4xw $sp` prints the four words of memory above the stack pointer in hexadecimal.

The output format in this case specifies how big a unit of memory to examine and how to print the contents of that unit. It is done with one or two of the following letters.

These letters specify the size of unit to examine:

| | |
|---|---|
| `b` | Examine individual bytes. |
| `h` | Examine halfwords (4 bytes each). |
| `w` | Examine words (8 bytes each). |
| `g` | Examine giant words (16 bytes each). |

These letters specify the way to print the contents:

| | |
|---|---|
| `x` | Print as integers in unsigned hexadecimal. |
| `d` | Print as integers in signed decimal. |
| `u` | Print as integers in unsigned decimal. |
| `o` | Print as integers in unsigned octal. |
| `a` | Print as an absolute address in hex. |
| `c` | Print as character constants. |
| `f` | Print as floating point. This works only with sizes `w` and `g`. |
| `s` | Print a null-terminated string of characters. The specified unit size is ignored; instead, the unit is however many bytes it takes to reach a null character (including the null character). |
| `i` | Print a machine instruction in assembler syntax (or nearly). The specified unit size is ignored; the number of bytes in an instruction varies depending on the type of machine, the opcode and the addressing modes used. |

If either the manner of printing or the size of unit fails to be specified, the default is to use the same one that was used last. If you do not want to use any letters after the slash, you can omit the slash as well.

You can also omit the address to examine. Then the address used is immediately after the last unit examined. This is why string and instruction formats actually compute a unit-size based on the data: so that the next string or instruction examined will start in the right place. The `print` command sometimes sets the default address for the `x` command; when the value printed resides in memory, the default is set to examine the same location.

When you use `RET` to repeat an `x` command, it does not repeat exactly the same: the address specified previously (if any) is ignored, so that the repeated command examines the successive locations in memory rather than the same ones.

You can examine several consecutive units of memory with one command by writing a repeat count after the slash (before the format letters, if any). The repeat count must be a decimal integer. It has the same effect as repeating the `x` command that many times except that the output may be more compact with several units per line. For example,

```
[1] (mdb) x/10i $pc
```

The previous command prints ten instructions, starting with the one to be executed next, by the focus thread in the selected frame. After doing this, you could print another ten instructions using the following command:

```
[1] (mdb) x/10
```

in which the format and address are allowed to default.

The addresses and contents printed by the `x` command are not put in the value history because there are often too many of them. Instead, `mdb` makes these values available for subsequent use in expressions as values of the convenience variables `$_` and `$__`.

After an `x` command, the last address examined is available for use in expressions in the convenience variable `$_`. The contents of that address, as examined, are available in the convenience variable `$__` .

If the `x` command has a repeat count, the address and contents saved are from the last memory unit printed; this is not the same as the last address printed if several units were printed on the last line of output.

## 6.7 Automatic Display

To print the value of an expression frequently (to see how it changes), you can add the expression to the *automatic display list*, a list of expressions that are displayed each time the program stops. Each element in the list is numbered; to remove an expression from the list, you specify that number. The automatic display looks like this:

```
2: foo = 38
3: bar[5] = (struct hack *) 0x3804
```

showing item numbers, expressions and their current values.

If the expression refers to local variables, then it does not make sense outside the lexical context for which it was set up. Such an expression is printed only when execution is inside that lexical context. For example, if you give the command `display name` while inside a function with an argument `name`, this argument is displayed whenever the program stops inside that function, but not when it stops elsewhere (because this argument does not exist elsewhere).

`display` *exp*

Adds the expression *exp* to the list of expressions to display each time the program stops. See Expressions on page 53.

`display/` *fmt exp*

Specifies a display format and not a size or count for *fmt*, adds the expression *exp* to the auto-display list, and arranges to display *exp* each time in the specified format *fmt*.

`display/` *fmt addr*

Adds the expression *addr* as a memory address to be examined each time the program stops for *fmt* `i` or `s`, including a unit-size or a number of units. Examining means in effect doing `x/`*fmt addr*. See Examining Memory on page 60.

`display/i $pc`

Displays the next instruction to be executed by the focus thread.

`undisplay` *dnum(s)*`, delete display` *dnum(s)*

Removes the item number(s) *dnums* from the list of expressions to display.

`disable display` *dnum(s)*`...`

Disables the display of item number(s) *dnum(s)*. A disabled display item is not printed automatically, but is not forgotten. It may be reenabled later.

`enable display` *dnum(s)*

Enables display of item number(s) *dnum(s)*. It becomes effective once again in auto display of its expression, until you specify otherwise.

`display` Displays the current values of the expressions on the list, as is done when the program stops.

`info display`

Prints the list of expressions previously set up to display automatically, each one with its item number, but without showing the values. This includes disabled expressions, which are marked as such. It also includes expressions that are not displayed right now because they refer to automatic variables not currently available.

## 6.8 Value History

Every value printed by the `print` command is saved for the entire session in the `mdb` *value history* so that you can refer to it in other expressions.

The values printed are given *history numbers* for you to refer to them by. These are successive integers starting with 1. `print` shows you the history number assigned to a value by printing `$`*num* `=` before the value; here *num* is the history number.

To refer to any previous value, use `$` followed by the history number of the value. The output printed by `print` is designed to remind you of this. A single `$` refers to the most recent value in the history, and `$$` refers to the value before that.

For example, to see the contents of a structure to which you have printed a pointer:

```
[1] (mdb) p *$
```

If you have a chain of structures where the component `next` points to the next one, you can print the contents of the next one:

```
[1] (mdb) p *$.next
```

It might be useful to repeat this command many times by typing `RET`.

Note that the history records values, not expressions. If the value of `x` is 4 and you type this command:

```
[1] (mdb) print x
[1] (mdb) set x=5
```

then the value recorded in the value history by the `print` command remains 4 even though the value of `x` has changed.

By extension, the type of a history value does not change when circumstances are altered. For example, by continuing to a breakpoint in a different module or, in a multi-threaded context, by focusing on a new thread, you may find that the new context harbors a type whose name is identical to that of the history value type but whose structure differs; however, printing the value continues to produce the same result as in the original context.

`info values`

Print the last ten values in the value history, with their item numbers. This is like `p $$9` repeated ten times, except that `info values` does not change the history.

`info values` *n*

Print ten history values centered on history item number *n*.

`info values +`

Print the ten history values immediately after the values last printed.

## 6.9 Convenience Variables

`mdb` provides *convenience variables* that you can use within `mdb` to hold on to a value and refer to it later. These variables exist entirely within `mdb`; they are not part of your program, and setting a convenience variable has no effect on further execution of your program. That is why you can use them freely.

Convenience variables have names starting with `$`. You can use any name starting with `$` for a convenience variable, unless it is one of the predefined set of register names (see Registers on page 66).

You can save a value in a convenience variable with an assignment expression, as you would set a variable in your program. Example:

```
[1] (mdb) set $foo = *object_ptr
```

saves in `$foo` the value contained in the object pointed to by `object_ptr`.

Using a convenience variable for the first time creates it; but its value is `void` until you assign a new value. You can alter the value with another assignment at any time.

Convenience variables have no fixed types. You can assign a convenience variable any type of value, even if it already has a value of a different type. The convenience variable as an expression has whatever type its current value has.

`info convenience`

Print a list of convenience variables used so far, and their values. Abbreviated `i con`.

One of the ways to use a convenience variable is as a counter to be incremented or a pointer to be advanced. For example:

```
[1] (mdb) set $i = 0
[1] (mdb) print bar[$i++]->contents ...repeat that command by typing RET.
```

Some convenience variables are created automatically by `mdb` and given values likely to be useful.

`$_` The variable `$_` is automatically set by the `x` command to the last address examined (see Examining Memory on page 60).

`$__` The variable `$__` is automatically set by the `x` command to the value found in the last address examined.

## 6.10 Registers

Each thread in your program has an identically named set of machine registers. `mdb` tracks the full set of registers for only the innermost frame. You can refer to register contents of the focus thread in expressions as variables with names starting with `$`. Use `info registers` to see the names and values of the focus thread registers.

| | |
|---|---|
| `exception` | Identifies raised exceptions. |
| `resultcode` | Further refines the value of the `exception` register. |
| `mslots_flag` | Used by the trap handler. |
| `dc` *[0,1,...,7]*, `dv`*[0,1,...,7]* | Used by the trap handler. |
| `t`*[0,1,...,7]* | Target registers. `t0` always holds the value of the primary trap handler. |
| `r`*[0,1,...,31]* | General purpose registers. `r0` always holds the value 0. |
| `instcount` | One use by `mdb` is to step a thread some number of instructions. |
| `ssw` | Holds the program counter (`$pc`), various condition codes, and trap masks. The value is for the innermost frame, regardless of the selected frame. |

In addition, `mdb` recognizes aliases for certain registers.

| | |
|---|---|
| `$pc` | Program counter. Lower bits of `ssw`. |
| `$sp` | Stack pointer. Points to the current stack frame. Same as `r1`. |
| `$er` | Exception register. Same as `exception`. |
| `$eps` | Pointer to end of memory block allocated for stack. Same as `r5`. This use of `$eps` is valid only when the focus thread begins executing the function; the compiler may use `r5` as a general purpose register during execution of the function body. |
| `$ccb` | Pointer to the control block of the focus thread. Same as `r2`. |

Register values are relative to the selected stack frame (see Selecting a Frame on page 43). This means that you get the value that the register would contain if all stack frames farther in were exited and their saved registers restored. Registers that were not saved may hold values irrelevant to the selected stack frame. In order to see the real contents of all registers, you must select the innermost frame (with `frame 0`).

**Note:** Currently, `mdb` only provides correct values for registers in the innermost frame.

`info registers`

Print the names and values of all registers for the focus thread relative to the selected frame. If your are not at the frame where execution is currently stopped (that is, in a frame that is not innermost), some registers may not be tracked and can retain values from lower frames.

`info registers` *regname*

Print the value of register *regname* for the focus thread. *regname* may be any valid register name, with or without the initial `$`. When `mdb` recognizes that a general-purpose register contains a named variable from your program (as opposed to a compiler-generated temporary or some other value), it prints the name of the variable.

## 6.11 Register Examples

You could print the program counter of the focus thread in hex with:

```
[1] (mdb) p/x $pc
```

or print the instruction to be executed next.

```
[1] (mdb) x/i $pc
```

You can assign registers directly, but if the register holds the value of a variable, the variable may exist in multiple locations (that is, in other registers and memory). In this case, the compiled code does not have to keep the values in these locations consistent or even use the locations in subsequent branching decisions if it can obtain information about the current value of the variable from analysis of the code.

# Examining Symbols [7]

The commands described in this chapter allow you to make inquiries for information about the symbols (names of variables, functions and types) defined in your program. This information is found by `mdb` in the program symbol table, one or more program libraries, or one or more object files. This symbol information is inherent in the text of your program and does not change as the program executes.

`whatis` *exp* Print the data type of expression *exp*. *exp* is not actually evaluated, and any side-effecting operations (such as assignments or function calls) inside it do not take place. See Expressions on page 53.

`whatis` Print the data type of `$`, the last value in the value history.

`info address` *symbol*

Describe where the data for *symbol* is stored. For a register variable, this says which register it is kept in. For a non-register local variable, this prints the stack-frame offset at which the variable is always stored.

Note the contrast with `print &`*symbol*, which does not work at all for a register variables, and for a stack local variable prints the exact address of the current instantiation of the variable.

`ptype` *typename*

Print a description of data type *typename*. *typename* may be the name of a type, or for C code it may have the form `struct` *struct-tag*, `union` *union-tag* or `enum` *enum-tag*.

`info sources`

Print the names of all source files in the program. For standard shared libraries such as `libc`, `libm`, and `librt`, only the names of the source files referenced by the program are printed.

`info modules`

Print the names of all object files in the program. Each object file is listed either as a stand-alone fat `.o` file or as one of several components of a program library.

`info functions`

Print the names and data types of all defined functions.

`info functions` *regexp*

Print the names and data types of all defined functions whose names contain a match for regular expression *regexp*. Thus, `info fun step` finds all functions whose names include `step`; `info fun ^step` finds those whose names start with `step`.

`info variables`

Print the names and data types of all variables that are declared outside of functions (that is, except for local variables).

`info variables` *regexp*

Print the names and data types of all variables (except for local variables) whose names contain a match for regular expression *regexp*.

`info types` Print all data types defined in the program.

`info types` *regexp*

Print all data types that are defined in the program whose names contain a match for regular expression *regexp*.

`printsyms` *filename*

Write a complete dump of the debugger symbol data into the file *filename*.

See also the `info files` command in File Commands on page 11 and the `info modules` command in Module Commands on page 13.

## 7.1 Archive Symbol Visibility

On the Cray XMT, when your program is linked with an archive, your program may or may not see a particular global symbol in the archive. If `annotate` has been run on the archive, the linker allows your program to use only those global archive symbols explicitly made available to user programs. If `annotate` has not been run, all global symbols within the archive are available. Using `annotate` to hide symbols provides for a measure of safety, analogous to that provided by static symbols in multiple-module user programs, and allows identical names in multiple archives, as well as in freestanding object files (not contained within an archive), to represent separate entities.

Consider the global symbols in an archive with which your program is linked, where `annotate` has been run on the archive. The program treats the global archive symbols exported by `annotate` as *visible*, and those global archive symbols that are not exported as *hidden*.

Symbols appear in your source code in one of two contexts: as a definition or as a use. When the linker encounters a use of a global symbol within a freestanding module, it locates the symbol definition by searching the visible symbols defined in freestanding modules and archives. When the linker encounters the use of a hidden global symbol within an archive, symbols defined within the archive take precedence over external names.

If you request information from `mdb` about a hidden archive symbol or try to set a breakpoint on a hidden function, `mdb` uses the internal variable `visibility` to determine whether to grant access to the symbol and to resolve the ambiguity if there are multiple hidden symbols by that name.

`set visibility` *value*

Determine the way `mdb` resolves conflicts between visible and hidden global symbols.

The possible states for *value* are:

| | |
|---|---|
| `auto` | This is the default state. Given the choice between a global exported symbol and a hidden symbol of the same name in an archive, `mdb` selects the hidden symbol if the current stack frame belongs to a function within that archive. When the choice is between multiple hidden symbols, `mdb` selects the local symbol rather than the one residing in another archive. In this case, if no local symbol exists, `mdb` chooses one of the symbols arbitrarily. |
| `hidden` | `mdb` resolves all conflicts in favor of hidden symbols. When multiple hidden symbols with the same name exist, `mdb` displays a menu. Consider using this mode when there are potential conflicts between exported and hidden symbols in expressions involving several global variables. |
| `any` | `mdb` makes no attempt to resolve ambiguities. When multiple global symbols of the same name are present, you can choose the symbol you want from a menu. |
| `visible` | `mdb` resolves all conflicts in favor of the exported symbol. If none exists, an error message is issued. |

You can change visibility either strictly for archive variables or strictly for archive function names by setting one of the subsidiary variables `code-visibility` or `data-visibility` to `auto`, `any`, `visible`, or `hidden`.

You may also use abbreviated forms for all these variables and values. The variables may be abbreviated as `v`, `cv`, and `dv`, respectively, and the values as any unambiguous prefix. Thus, the following command sets the visibility for archive function names to the value `any`.

```
[1] (mdb) set cv an
```

# Altering Execution [8]

Once you think you have found an error in the program, you might want to find out for certain whether correcting the apparent error leads to correct results in the rest of the run. You can find the answer by experiment, using the `mdb` features for altering execution of the program.

For example, you can store new values into variables or memory locations, give the program a signal, restart it at a different address, or even return prematurely from a function to its caller.

## 8.1 Assignment to Variables

The ability of `mdb` to change the value of a variable depends on the debugging level with which you compiled your code (see Compiling for Debugging on page 8), as well as on the nature of the variable.

Evaluating an assignment expression is one way to alter the value of a variable. See Expressions on page 53. For example,

```
[1] (mdb) print x=4
```

stores the value 4 into the variable `x` and then prints the value of the assignment expression (which is 4).

All the assignment operators for C are supported, including the increment operators ++ and --, and combining assignments such as += and <<=.

If you are not interested in seeing the value of the assignment, use the `set` command instead of the `print` command. `set` is really the same as `print` except that the value of the expression value is not printed and is not put in the value history (see Value History on page 64). The expression is evaluated only for side effects.

Whenever the beginning of the argument string of the `set` command appears identical to a `set` subcommand, it may be necessary to use the `set variable` command. This command is identical to `set` except for its lack of subcommands. For example, the first of the following two commands sets the `mdb` variable `rw` (equivalently `register-warning`), and the second sets the program variable `rw`.

```
[1] (mdb) set rw 0
[1] (mdb) set variable rw 0
```

If the value of a variable is kept in a register, mdb may not always be able to update the variable in ways that are fully consistent with normal execution. See Altering Variables Kept in Registers on page 74 for a discussion of how mdb handles such an assignment.

If the use of a Cray XMT generic to assign a sync or future variable value changes the full/empty state to full (see Changing the Full/Empty Bit on page 75), and there are blocked threads waiting for the variable to become full, the write is accompanied by a subsequent trap handling action that satisfies the request of one of the blocked threads to write or read the variable (possibly changing the value of the variable value or re-setting the variable to empty). Thus, if you print a sync or future variable after writing to it, without resuming any threads after the write operation, the value or full/empty state of the variable may be different than you expected. The unblocked thread remains suspended, though in a state that differs as a result of the read or write assisted by the trap handler, until the entire program resumes or the particular thread is explicitly set in motion from the mdb command line.

See Changing the Full/Empty Bit on page 75, for another means of changing the full/empty bit.

mdb allows more implicit conversions in assignments than C does; you can freely store an integer value into a pointer variable or vice versa. You can also convert any structure to any other structure that is the same length or shorter.

To store values into arbitrary places in memory, use the {...} construct to generate a value of specified type at a specified address (see Expressions on page 53). For example, {int}0x401033b918 refers to memory location 0x401033b918 as an integer (which implies a certain size and representation in memory), and:

```
[1] (mdb) set {int}0x401033b918 = 20
```

stores the value 20 into that memory location.

## 8.1.1 Altering Variables Kept in Registers

Under certain compiler optimizations, the value of a variable is sometimes kept in one or more registers. If you change the value of the variable from mdb, mdb may change only one of the copies. The multiple copies of the variable may not have identical values, and further execution may have unexpected behavior. You can prevent the compiler from performing this kind of optimization by compiling your program at the -g2 debugging level (see Compiling for Debugging on page 8).

For example, when the following program is compiled at the `-g1` debugging level, setting `i` to a new value at line 4 from `mdb` has no visible effect, because the parameter to `printf` is in a different register than the one changed by `mdb`. Also, the loop control is in yet a third register.

```
main() {
   int i;
   for (i=0; i<20; i++) {
      printf("%d\n", i); // line 4
   } }
```

When you change the value of variable stored in a register, `mdb` issues a warning that the new value may not be propagated to other copies of the variable.

```
[1] (mdb) set foo = 10
Warning: Variable in register. New value may not
 propagate to all copies.
```

You can adjust the frequency of these warning messages either by invoking `mdb` with a command line option:

```
% mdb -register-warning frequency
% mdb -rw frequency
```

or by setting a `mdb` variable during your debugging session.

```
set register-warning frequency
set rw frequency
```

In either method, *frequency* is one of `never`, `first`, or `always` (equivalently, `0`, `1`, or `2`, respectively). The default warning frequency is `always`. A value of `first` means the warning is given the first time an assignment is made to a variable on a per-variable basis.

## 8.2 Changing the Full/Empty Bit

You can use the generic functions to manipulate the `full/empty` bit or bits of a variable `v` while simultaneously reading from or writing to `v`. See *Cray XMT Programming Environment User's Guide* for more information. You cannot directly manipulate any of the other state bits through `mdb` commands (see State Bits on page 55). You can change the other state bits by writing an appropriate function, compiling it into your program, and calling the function from `mdb`.

If multiple threads are blocked on a sync variable, all are either waiting for the variable to become full or waiting for the variable to become empty. When the state of the variable changes, one thread resumes running. If multiple threads are blocked on a future variable, then the variable is empty. When the state of the variable changes from empty to full, all waiting threads resume running.

mdb prints the value of a variable without changing any of the state bits. In particular, when mdb prints the value of a sync or future variable, the full/empty state does not change. Suppose you want to cause resumption of a thread that is blocked (see Thread States on page 36) waiting to write the one-word sync variable v$ when v$ becomes empty. To simulate an emptying read of v$, use the generic function readfe.

```
[1] (mdb) print v$
$1 = 4 (full,trap0)
[1] (mdb) print readfe(&v$)
$2 = 4
[1] (mdb) print v$
$3 = 9 (full, trap0)
```

The set trap 0 bit in the second line indicates that there is at least one thread blocked on v$, waiting for the full/empty bit to become empty. The generic function readfe reads v$ when the full/empty bit is full, simultaneously changing the full/empty bit to empty. Because there are threads blocked on v$, immediately following the readfe a trap handler also satisfies one of the blocked threads--in this example, a writer changes the value of v$ to 9, also toggling the full/empty bit to full. Because the trap 0 bit of v$ is still set after the readfe operation and subsequent trap handling, more than one thread was initially blocked on v$. If only one thread were waiting for v$ to become empty, the last line of the example above would read $3 = (Full)9. When you continue your program, the thread whose write request (value 9) was satisfied by the trap handling resumes running; any other threads blocked on v$ remain blocked.

In normally executing code, if v$ had been empty, the readfe operation would have blocked until v$ was set to full. If v$ is empty and you issue a readfe of v$ from mdb, mdb returns a message saying the operation was not done because it would have blocked. When mdb halts execution of your program, as long as all the threads are suspended, it makes little sense for you to issue a generic operation on an empty variable or memory location that depends on the variable or location being full. Similarly, generic operations that require an object to be empty hang indefinitely on a full object, in the absence of running threads in the program.

The generic functions are intended for use on only sync or future variables, although neither mdb nor the compilers enforce this. Your program may behave incorrectly if you change the full/empty state of a normal (not sync or future) variable with one of these generics: mdb prints a warning message if you access a normal variable with one of these functions as part of a mdb command. Also, if you change the full/empty bit of a normal variable whose size is less than a word, because the full/empty bit actually belongs to the memory word containing the variable, the full/empty bit is changed for any other variables contained in the same word (see State Bits on page 55).

If a watchpoint is set on the variable, that is, the `x` command reveals that the trap 1 bit is set (see Examining Memory on page 60), the generics will not manipulate the `full/empty` bit properly, nor are they guaranteed to wake up any thread blocked on that variable. Try disabling the watchpoint before proceeding.

In your program, generics may be used on sync or future variables of types such as `long double` in C, which are composed of multiple words. This functionality is not yet implemented in `mdb`.

# Stored Sequences of Commands [9]

`mdb` provides two ways to store sequences of commands for execution as a unit: user-defined commands and command files.

## 9.1 User-defined Commands

A *user-defined command* is a sequence of `mdb` commands to which you assign a new name as a command. This is done with the `define` command.

`define [`*commandname $arg1 $arg2 ...*`]`

Define a command named *commandname* with optional arguments *$arg1*, *$arg2*, .... If there is already a command by that name, you are asked to confirm that you want to redefine it.

The definition of the command is made up of other `mdb` command lines, which are given following the `define` command. The end of these commands is marked by a line containing `end`.

Formal arguments must each start with a dollar sign (`$`) and may contain letters, digits, and underscores. The arguments may be used within the command lines that follow. When you invoke *commandname*, you must supply the same number of actual arguments as there are formals. The text of the actual argument is substituted for that of the formal argument before execution of each command line. Arguments may be contained within double quoted material. To avoid substitution, prefix a backslash (`\`) before the dollar sign.

`document` *commandname*

Give documentation to the user-defined command *commandname*. The command *commandname* must already be defined. This command reads lines of documentation the same way that `define` reads the lines of the command definition, ending with `end`. After the `document` command is finished, `help` on command *commandname* prints the documentation you have specified.

You may use the `document` command again to change the documentation of a command. Redefining the command with `define` does not change the documentation.

When user-defined commands are executed, the commands of the definition are not printed. An error in any command stops execution of the user-defined command.

Commands that ask for confirmation if used interactively proceed without asking when used inside a user-defined command. Many `mdb` commands that normally print messages to say what they are doing omit the messages when used in a user-defined command.

## 9.2 Command Files

A command file for `mdb` is a file of lines that are `mdb` commands. Comments (lines starting with #) may also be included. An empty line in a command file does nothing; it does not mean to repeat the last command, as it would from the terminal.

When `mdb` starts, it automatically executes its *init files*—command files named `.mdbinit`. `mdb` reads the init file (if any) in your home directory and then the init file (if any) in the current working directory. (The init files are not executed if `mdb` is invoked with the `-nx` option.) You can also request the execution of a command file with the `source` command:

`source` *filename*

Execute the command file *filename*.

The lines in a command file are executed sequentially. They are not printed as they are executed. An error in any command terminates execution of the command file.

Commands that ask for confirmation if used interactively proceed without asking when used in a command file. Many `mdb` commands that normally print messages to say what they are doing omit the messages when used in a command file.

## 9.3 Commands for Controlled Output

During the execution of a command file or a user-defined command, the only output that appears is what is explicitly printed by the commands of the definition. This section describes three commands useful for generating exactly the output you want.

`echo` *text*
: Print *text*. You can include non-printing characters in *text* using C escape sequences, such as `\n` to print a newline. No newline is printed unless you specify one. In addition to the standard C escape sequences a backslash followed by a space stands for a space. This is useful for outputting a string with spaces at the beginning or the end, because leading and trailing spaces are trimmed from all arguments. Thus, to print `" and foo = "`, use the command `echo "\ and foo =\ "`. You can use a backslash at the end of *text*, as in C, to continue the command onto subsequent lines. For example,

```
echo This is some text\n\
that is continued\n\
onto several lines.\n
```

produces the same output as:

```
echo This is some text\n
echo that is continued\n
echo onto several lines.\n
```

`output` *expression*

Print only the value of *expression*—no newlines, no `$`*nn* `=`. The value is not entered in the value history either. See Expressions on page 53 for more information on expressions.

`output/` *fmt* *expression*

Print the value of *expression* in format *fmt*. See Output Formats on page 59 for more information.

`printf` *string*`,` *expressions*`...`

Print the values of the *expressions* under the control of *string*. The *expressions* are separated by commas and may be either numbers or pointers. Their values are printed as specified by *string*, in general exactly as if the program were to execute:

```
[1] (mdb) printf (string, expressions...);
```

(One minor exception is that integers are currently treated as 32 bit numbers by `mdb`.) As an example, you can print two values in hex like this:

```
[1] (mdb) printf "foo, bar-foo = 0x%x, 0x%x\n", foo, bar-foo
```

The only backslash-escape sequences that you can use in the string are the simple ones that consist of backslash followed by a letter.

# Options and Arguments for `mdb` [10]

When you invoke `mdb`, you can specify arguments telling it what files to operate on and what other things to do.

## 10.1 Mode Options

| | |
|---|---|
| `-nx` | Do not execute commands from the init files `.mdbinit`. Normally, the commands in these files are executed after all the command options and arguments have been processed. See Command Files on page 80. |
| `-q` | Quiet. Do not print the usual introductory messages. |
| `-batch` | Run in batch mode. Exit with code 0 after processing all the command files specified with `-x` (and `.mdbinit`, if not inhibited). Exit with nonzero status if an error occurs in executing the `mdb` commands in the command files. |
| `-fullname` | This option is used when Emacs runs `mdb` as a subprocess. It tells `mdb` to output the full file name and line number in a standard, recognizable fashion each time a stack frame is displayed (which includes each time the program stops). |

## 10.2 File-specifying Options

All the options and command line arguments given are processed in sequential order. The order makes a difference when the `-x` option is used.

`-s` *filename*

Read symbol table from *filename*.

`-e` *filename*

Use *filename* as the executable file to execute when appropriate, and for examining pure data.

`-se` *filename*

Read symbol table from *filename* and use it as the executable file.

`-x` *filename*

Execute `mdb` commands from *filename*.

`-d` *directory*

Add *directory* to the path to search for source files.

`-cd` *directory*

Use *directory* as the working directory for `mdb`.

## 10.3 Communication Options and Variables

In each of the following pairs, the first item is the command-line option form, the second item is the variable setting that will evoke the option behavior for subsequent `run` commands.

`-rm`, `set remote-manual`

Start in remote-manual mode. `mdb` does not start the inferior—it waits until the inferior is started manually.

`-open-socket`, `set communication open-socket`

Use a socket as the communication channel between `mdb` and the target program. `mdb` creates the socket.

`-socket` *hostname*,*portnumber*, `set communication socket` *host*,*port*

Use the socket from *hostname* using *portnumber* as the communication channel between `mdb` and the target program.

## 10.4 Breakpoint-behavior Options

`-ox` Execute instructions at breakpoints by creating and calling a pseudo-function that simulates the behavior of the original instruction. The other option, restoring the original instruction and executing it in place, allows other activities to proceed past the breakpoint without stopping. `-ox` is the default. The `-out-of-line-execution` option is identical to this option.

`-ix` Execute instructions at breakpoints by restoring the original instruction to its rightful address, single-stepping across it and then restoring the breakpoint. You can use this if a bug is suspected in the pseudo-function created with `-ox`, but it is not recommended for general use. Conditional breakpoints cannot be used with this option. The `-inline-execution` option is identical to this option.

## 10.5 Miscellaneous Options

`-ximm` *command*

Execute *command* immediately.

## 10.6 Other Arguments

If there are arguments to `mdb` that are not options or associated with options, the first one specifies the symbol table and executable file name (as if it were preceded by `-se`). A second unassociated argument should be a decimal number which is treated as the process id (PID) of the running process to which `mdb` should attach.

When `mdb` attaches to a process, the process halts until you enter the `run` command. After you enter `run`, `mdb` resumes execution of the process until either the program exits, you type `Ctrl-C`, or the process reaches the next breakpoint.

# GNU General Public License [A]

## A.1 Preamble

The license agreements of most software companies try to keep users at the mercy of those companies. By contrast, our General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. The General Public License applies to the Free Software Foundation's software and to any other program whose authors commit to using it. You can use it for your programs, too.

When we speak of free software, we are referring to freedom, not price. Specifically, the General Public License is designed to make sure that you have the freedom to give away or sell copies of free software, that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of a such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must tell them their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license that gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

The precise terms and conditions for copying, distribution and modification follow.

## A.2 Terms and Conditions

1. This License Agreement applies to any program or other work that contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The Program, below, refers to any such program or work, and a work based on the Program means either the Program or any work containing the Program or a portion of it, either verbatim or with modifications. Each licensee is addressed as you.

2. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this General Public License and to the absence of any warranty; and give any other recipients of the Program a copy of this General Public License along with the Program. You may charge a fee for the physical act of transferring a copy.

3. You may modify your copy or copies of the Program or any portion of it, and copy and distribute such modifications under the terms of Paragraph 1 above, provided that you also do the following:

   - Cause the modified files to carry prominent notices stating that you changed the files and the date of any change; and

   - Cause the whole of any work that you distribute or publish, that in whole or in part contains the Program or any part thereof, either with or without modifications, to be licensed at no charge to all third parties under the terms of this General Public License (except that you may choose to grant warranty protection to some or all third parties, at your option).

   - If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the simplest and most usual way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this General Public License.

   - You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

   Mere aggregation of another independent work with the Program (or its derivative) on a volume of a storage or distribution medium does not bring the other work under the scope of these terms.

4. You may copy and distribute the Program (or a portion or derivative of it, under Paragraph 2) in object code or executable form under the terms of Paragraphs 1 and 2 above provided that you also do one of the following:

   - Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Paragraphs 1 and 2 above; or,

   - Accompany it with a written offer, valid for at least three years, to give any third party free (except for a nominal charge for the cost of distribution) a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Paragraphs 1 and 2 above; or,

   - Accompany it with the information you received as to where the corresponding source code may be obtained. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form alone.)

     Source code for a work means the preferred form of the work for making modifications to it. For an executable file, complete source code means all the source code for all modules it contains; but, as a special exception, it need not include source code for modules that are standard libraries that accompany the operating system on which the executable file runs, or for standard header files or definitions files that accompany that operating system.

5. You may not copy, modify, sublicense, distribute or transfer the Program except as expressly provided under this General Public License. Any attempt otherwise to copy, modify, sublicense, distribute or transfer the Program is void, and will automatically terminate your rights to use the Program under this License. However, parties who have received copies, or rights to use copies, from you under this General Public License will not have their licenses terminated so long as such parties remain in full compliance.

6. By copying, distributing or modifying the Program (or any work based on the Program) you indicate your acceptance of this license to do so, and all its terms and conditions.

7. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein.

8. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. Each version is given a distinguishing version number. If the Program specifies a version number of the license that applies to it and any later version, you have the option of following the terms and conditions either of that version or

of any later version published by the Free Software Foundation. If the Program does not specify a version number of the license, you may choose any version ever published by the Free Software Foundation.

9. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software that is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

   **NO WARRANTY**

10. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM AS IS WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

11. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

## A.3 How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to humanity, the best way to achieve this is to make it free software that everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the copyright line and a pointer to where the full notice is found.

```
one line to give the program's name and a brief idea of what it does.
Copyright (C) 19yy  name of author

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 1, or (at your option)
any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) 19yy  name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type `show w'.
This is free software, and you are welcome to redistribute it
under certain conditions; type `show c' for details.
```

The hypothetical commands `show w` and `show c` should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than `show w` and `show c`; they could even be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a copyright disclaimer for the program, if necessary. Here a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright interest in the
program `Gnomovision' (a program to direct compilers to make passes
at assemblers) written by James Hacker.

signature of Ty Coon, 1 April 1989
Ty Coon, President of Vice
```

That's all there is to it!

# Using mdb under GNU Emacs [B]

A special interface allows you to use GNU Emacs to view (and edit) the source files for the program you are debugging with `mdb`. To get the interface with under Emacs 19.25, put the following line in your `.emacs` file:

```
(autoload 'mdb "gud" "grand unified debugging mode" t)
```

To use this interface, use the command `M-x mdb` in Emacs. Give the executable file you want to debug as an argument. This command starts `mdb` as a subprocess of Emacs, with input and output through a newly created Emacs buffer. If Emacs produces an error message instead of starting `mdb`, you may be using an older file. Remove the autoload line from your `.emacs` file and use `M-x gdb`. Then, substitute `mdb` for `gdb` in the minibuffer.

Using `mdb` under Emacs is like using `mdb` normally except for two things:

- All terminal input and output goes through the Emacs buffer. This applies to `mdb` commands and their output, and to the input and output done by the program you are debugging. This is useful because it means that you can copy the text of previous commands and input them again; you can even use parts of the output in this way. All the facilities of the Emacs Shell mode are available for this purpose.
- `mdb` displays source code through Emacs. Each time `mdb` displays a stack frame, Emacs automatically finds the source file for that frame and puts an arrow (`=>`) at the left margin of the current line. Explicit `mdb list` or search commands still produce output as usual, but you probably have no reason to use them.

In the `mdb` I/O buffer, you can use these special Emacs commands:

| | |
|---|---|
| `C-c C-s` | Execute to another source line, like the `mdb step` command. |
| `C-c C-n` | Execute to next source line in this function, skipping all function calls, like the `mdb next` command. |
| `C-c C-i` | Execute one instruction, like the `mdb stepi` command. |
| `C-c C-b` | Set a breakpoint on the current line, like the `mdb break` *linenum* command, where *linenum* corresponds to the position of (`=>`) in the source file buffer. |
| `C-c C-f` | Execute until exit from the selected stack frame, like the `mdb finish` command. |
| `C-c C-r` | Continue execution of the program, like the `mdb cont` command. |
| `C-c C-p` | Evaluate the expression immediately following the cursor, like the `mdb print` *exp* command where *exp* is the expression immediately following the cursor in the `mdb` buffer. |
| `C-c <` | Go up the number of frames indicated by the numeric argument, like the `mdb up` command. |
| `C-c >` | Go down the number of frames indicated by the numeric argument, like the `mdb down` command. |

In any source file, the Emacs command `C-x SPC` (`mdb-break` ) tells `mdb` to set a breakpoint on the source line point is on.

The source files displayed in Emacs are in ordinary Emacs buffers that are visiting the source files in the usual way. You can edit the files with these buffers if you wish; but keep in mind that `mdb` communicates with Emacs in terms of line numbers. If you add or delete lines or characters from the text, the line numbers that `mdb` knows will cease to correspond properly to the code.

# mdb Input and Output Conventions [C]

To invoke `mdb`, enter the shell command `mdb`. Once started, `mdb` reads commands from the terminal until you tell it to exit.

A `mdb` command is a single line of input. There is no limit on how long it can be. It starts with a command name, which is followed by arguments whose meaning depends on the command name. For example, the command `step` accepts an argument that is the number of times to step, as in `step 5`. You can also use the `step` command with no arguments. Some command names do not allow any arguments.

`mdb` command names may always be abbreviated if the abbreviation is unambiguous. Sometimes even ambiguous abbreviations are allowed; for example, `s` is specially defined as equivalent to `step` even though there are other commands whose names start with `s`. Possible command abbreviations are often stated in the documentation of the individual commands.

A blank line as input to `mdb` means to repeat the previous command verbatim. Certain commands do not allow themselves to be repeated this way; these are commands for which unintentional repetition might cause trouble and that you are unlikely to want to repeat. Certain others (`list` and `x`) act differently when repeated because that is more useful.

A line of input starting with # is a comment; it does nothing. This is useful mainly in command files (see Command Files on page 80).

`mdb` indicates its readiness to read a command by printing a string called the *prompt*. This string is normally (`mdb`). If a thread currently has the focus (see Focus Thread on page 37), the focus thread ID is printed in square brackets to the left of (`mdb`). If the thread is in a non-running state (see Thread States on page 36), `mdb` prints the state of the focus thread leftmost in the prompt, within angle brackets.

Use the `set prompt` command to change the prompt string. You can also include system information in the prompt.

`set prompt` *newprompt*

Directs `mdb` to use *newprompt* as its prompt string henceforth. In addition to a literal prompt string, *newprompt* may include any of the following two-character specifications for system information:

| | |
|---|---|
| `%T` | focus thread name |
| `%S` | focus thread state |
| `%R` | focus thread state, if not running |
| `%F` | function name |
| `%U` | source file name |
| `%L` | line number |
| `%M` | module name (`.o` file) |
| `%P` | program library name (`.a` or `.pl` file) |
| `%C` | program counter |
| `%H` | history number |

Many of the specifications described above result in an empty string if the relevant information is unknown or unavailable. For instance, when you initially start `mdb`, no thread has the focus, so `%T` results in an empty string. You can use the following three-character specifications to control the printing of characters near the resulting strings.

| | |
|---|---|
| `%p`*X* | Immediately precedes a two-character specification %*?* from the list above, where *X* is a single character of your choice. If %*?* is printed, *X* is printed to its left; otherwise *X* is omitted. |
| `%s`*X* | Immediately succeeds a two-letter specification, where *X* is a single character of your choice. If %*?* is printed, *X* is printed to its right; otherwise *X* is omitted. |

If `mdb` cannot determine system information included in the prompt, `mdb` prints nothing. The default prompt specification is `%p<%R%s>%s %p[%T%s]%s (mdb)`.

To exit `mdb`, use the `quit` command (abbreviated `q`). `Ctrl-C` does not exit from `mdb`, but rather terminates the action of any `mdb` command that is in progress and returns to `mdb` command level. It is generally safe to type `Ctrl-C` at any time because `mdb` attempts to synchronize the interrupt to a time when it is safe. However, there is the possibility that `Ctrl-C` during expression evaluation may leave locks in a held state.

Certain commands to `mdb` may produce large amounts of information output to the screen. To help you read all of it, `mdb` pauses and asks you for input at the end of each page of output. Press `Enter` when you want to continue the output. Normally `mdb` knows the size of the screen from the termcap database together with the value of the `TERM` environment variable. To change the screen size use the `set screensize` command:

`set screensize` *lpp*`, set screensize` *lpp cpl*

Specify a screen height of *lpp* lines and (optionally) a width of *cpl* characters. If you omit *cpl*, the width does not change.

If you specify a height of zero lines, `mdb` will not pause during output no matter how long the output is. This is useful if output is to a file or to an editor buffer.

Also, `mdb` may at times produce more information about its own workings than is of interest to the you. You can turn some of these informational messages on and off with the `set verbose` command:

`set verbose on`

Re-enables `mdb` output of certain informational messages.

`set verbose off`

Disables `mdb` output of certain informational messages.

Currently, the messages controlled by `set verbose` are those that announce that the symbol table for a source file is being read (see File Commands on page 11), in the description of the command `symbol-file`).

# Glossary

**blade**

1) A Cray XMT compute blade consists of Threadstorm processors, memory, Cray SeaStar chips, and a blade control processor. 2) From a system management perspective, a logical grouping of nodes and blade control processor that monitors the nodes on that blade.

**blade control processor**

A microprocessor on a blade that communicates with a cabinet control processor through the HSS network to monitor and control the nodes on the blade. See also *blade*, *L0 controller*, *Hardware Supervisory System (HSS)*.

**cabinet control processor**

A microprocessor in the cabinet that communicates with the HSS via the HSS network to monitor and control the devices in a system cabinet. See also *Hardware Supervisory System (HSS)*.

**CLE**

The operating system for Cray XMT systems.

**fork**

Occurs when processors allocate additional streams to a thread at the point where it is creating new threads for a parallel loop operation.

**future**

Implements user-specified or explicit parallelism by starting new threads. A future is a sequence of code that can be executed by a newly created thread that is running concurrently with other threads in the program. Futures delay the execution of code if the code is using a value that is computed by a future, until the future completes. The thread that spawns the future uses parameters to pass information from the future to the waiting thread, which then executes. In a program, the term future is used as a type qualifier for a synchronization variable or as a keyword for a future statement.

**Hardware Supervisory System (HSS)**

Hardware and software that monitors the hardware components of the system and proactively manages the health of the system. It communicates with nodes and with the management processors over the private Ethernet network. See also *system interconnection network*.

**logical machine**

An administrator-defined portion of a physical Cray XMT system, operating as an independent computing resource.

**login node**

The service node that provides a user interface and services for compiling and running applications.

**metadata server (MDS)**

The component of the Lustre file system that manages Metadata Targets (MDT) and handles requests for access to file system metadata residing on those targets.

**node**

For CLE systems, the logical group of processor(s), memory, and network components acting as a network end point on the system interconnection network. See also *processing element*.

**phase**

A set of one or more sections of code that the stream executes in parallel. Each section contains an iteration of a loop. Phases and sections are contained in control flow code generated by the compiler to control the parallel execution of a function.

**processing element**

The smallest physical compute group. There are two types of processing elements: a compute processing element consists of an AMD Opteron processor, memory, and a link to a Cray SeaStar chip. A service processing element consists of an AMD Opteron processor, memory, a link to a Cray SeaStar chip, and PCI-X or PCIe links.

**System Management Workstation (SMW)**

The workstation that is the single point of control for system administration. See also *Hardware Supervisory System (HSS)*.