Addendum to Using the Multi-extensible Debugger (Mxdb for DG/UX™ and INTERACTIVE UNIX® Systems)

086–000203–00

This addendum updates manual 093–000710–03
See Updating Instructions on the reverse.

Printed in the United States of America
Rev. 00, September 1992
Licensed Material – Property of Data General Corporation
Ordering No. 086–000203

Notice

Addendum to Using the Multi-extensible Debugger (Mxdb for DG/UX™ and INTERACTIVE UNIX® Systems) 086–000203–00

To update your copy of 093–000710–03, please remove manual pages and insert addendum as follows:

Remove	Insert	Remove	Insert
Title/Notice	Title/Notice	12–99/12–106	12–99/12–106
1–1/1–6	1–1/1–6	13–13/13–14	13–13/13–14
2–7/2–11	2–7/2–11	14–3/14–28	14–3/14–28
3–1/3–14	3–1/3–14	14–43/14–61	14–43/14–61
3–23/3–30	3–23/3–30	15–5/15–22	15–5/15–22
10–19/10–20	10–19/10–20	17–3/17–14	17–3/17–14
11–3/11–10	11–3/11–10	17–23/17–32	17–23/17–32
12–13/12–18	12–13/12–18	18–1/18–10	18–1/18–10b
12–31/12–50	12–31/12–50	B–1/B–24	B–1/B–31
12–71/12–78	12–71/12–78		

Insert this sheet immediately behind the new Title/Notice page.

A vertical bar in the margin of a page indicates substantive technical change from the previous revision.

Using the Multi-Extensible Debugger (Mxdb for DG/UX™ and INTERACTIVE UNIX® Systems)

093–000710–03

For the latest enhancements, cautions, documentation changes, and other information on this product, please see the Release Notice (085–series) and/or Update Notice (078–series) supplied with the software.

Printed in the United States of America
Rev. 03, April 1992
Licensed Material – Property of Data General Corporation
Ordering No. 093–000710

Notice

Using the Multi-Extensible Debugger (Mxdb for DG/UX™ and INTERACTIVE UNIX® Systems)
093–000710–03

Revision History:

Original Release – December, 1989
First Revision – June, 1990
Second Revision – July, 1991
Third Revision – April 1992
Addendum 086–000203–00 – September, 1992

Effective with:

Mxdb, Rev. 2.11

A vertical bar in the margin of a replacement page indicates substantive technical change from the previous revision.

Chapter 1
Introduction to the Mxdb Debugger

The Multi-extensible debugger (Mxdb) is a high-level debugger that runs on AViiON® computers with the DG/UX™ operating system and on INTERACTIVE UNIX® systems (Interactive COBOL only); you can debug executable files, core files, and running processes. This debugger supports the C, C++, FORTRAN, Pascal, and Interactive COBOL programming languages, and is well-suited for both technical and commercial applications. Mxdb provides an on-line help system, easy-to-remember command names, consistent command syntax, expression parsing and description in the syntax of the supported languages, and macro and alias capabilities.

This chapter shows how to do the following tasks:

- Invoke and exit from the debugger.
- Use the help system.
- Compile programs to be debugged.
- Identify debugging situations.
- Load for execution the program to be debugged.
- Set a breakpoint.
- Resume executing the program being debugged.
- View source code.
- Display a variable's value.
- Reset a variable's value.
- Use language expressions.

Invoking and Exiting from Mxdb

To invoke the debugger, issue the **mxdb** command from a DG/UX or INTERACTIVE UNIX shell:

```
$ mxdb ↵
Mxdb Version x.x (month day, year) for AViiON Systems
Type "help" for further information.
(debug)
```

If you are using an INTERACTIVE UNIX system, the message above will say "`INTERACTIVE UNIX.`" The `(debug)` prompt means that you are in the debugger realm; if you are using Interactive COBOL, you will see the `(icobol)` prompt instead. Mxdb organizes commands into groups called realms; see Chapter 2 for more information.

To exit from Mxdb, type **quit**:

```
(debug) quit ↵
$
```

You can initialize each debugging session with a set of commands. Mxdb looks for a file named **.mxdb_init**, first in the working directory, then in your home directory. Whichever file the debugger finds first, it executes.

Getting Help

Mxdb offers three ways to use its help system: a **help** command, an **xhelp** command, and command prompting.

The help Command

The **help** command displays information about a command, argument, realm, or topic. To use this command, type **help** after invoking Mxdb. Then, if you want general information, press the New Line key. If you want information about a specific command, argument, realm, or topic, type that name after **help** and press the New Line key. For example, if you type the following:

```
(debug) help event-status ↵
```

Mxdb displays the summary portion of the **event-status** command's help message, which defines the command and its arguments, and shows examples.

To get a more detailed message, add a **,verbosity** argument. For example, type this command:

```
(debug) help event-status, v ↵
```

Mxdb then displays the entire **event-status** help message, which also elaborates the definitions and examples.

The xhelp Command

If you are using Mxdb on an X window display, you can use the **xhelp** command to browse through this manual and the "Using the Command Processor" manual online in a separate window, with hypertext capabilities.

Command Prompting

The command prompting facility helps you to enter commands interactively. Any Mxdb command will prompt you for input if you type the command followed by a comma and no argument. The **directory-list** command sets or displays your source directory search path; to be prompted for the arguments to this command, type the following:

```
(debug) directory-list, ↵
```

Mxdb responds with

```
Type ",help" for help.
directories () =
```

If you want directory **d** to be on your search path, reply

```
dirs () = d↵
Execute? (Yes) =
```

Enter New Line to accept the default answer. To verify that directory **d** is now on your search path, type

```
(debug) directory-list ↵
```

Mxdb will display the following:

```
 d
(debug)
```

Compiling Programs to Be Debugged

The Mxdb debugger handles programs written in C, C++, FORTRAN, Interactive COBOL, and Pascal.

Interactive COBOL

When compiling Interactive COBOL programs, use the "-d" option.

DG/UX

Compile your programs from a UNIX® shell. The ease of producing Mxdb debugging information varies according to the operating system revision that is being used.

LEGENDS Environment Variable

A LEGENDS environment variable can be used to modify the behavior of the debugging information generation utility. It contains a list (separated by blanks) of options, among which the most notable are "-external" and "-compress."

It is recommended that the "-external" option be used because it significantly saves disk space as well as increasing linker performance by avoiding duplication and unnecessary copying of the debugging information.

The "−compress" option produces a very compact form of the debugging information. At debug time, this causes the debugger to take more time when a particular module's information is being read in. (Mxdb never reads in information until it is necessary.) See the legend(5) man page for complete details.

When you compile a program to be debugged, use one of the options described below, which provide Mxdb debugging information.

DG/UX 5.4x

Porting and Developing Applications on the DG/UX System discusses the multiple development environment strategy employed in this version of the operating system. Refer to it for complete details.

In the ELF environment (m88kdgux / m88kdguxelf), Mxdb debugging information is the default. This means that the only option needed to compile for debugging is "−g".

In the COFF environment (m88kdguxcoff), the LEGENDS environment must be defined in order to have "−g" produce Mxdb debugging information.

The "+legend" and "+external−legend" switches must be supplied to the Cfront C++ compiler so the compiler can produce Mxdb debugging information. Also, for improved C++ debugging, use the Cfront switch "+a1." The Cfront switch "+d" improves Mxdb's descriptive behavior, but may degrade program performance.

The standard Mxdb debugging options documented in the next section will always work as well. They avoid any dependencies on environment variables but do not offer all of the functionality available via the LEGENDS environment variable. Also, these options do not preclude the use (or usefulness) of a LEGENDS environment variable. One example of this is that a LEGENDS variable can be used to specify a "compression" environmental option even if command line options are used to cause the information to be generated.

Language/Command	Provide Mxdb Debugging Information
Green Hills	−g −Xlegend
GNU	−g −mlegend
Interactive COBOL	−d
cc	−g −Wc,−fix−bb (Note that there is no space after the comma.)
as	−Wc
Cfront	−g +legend

On AViiON computers with the DG/UX operating system, Mxdb supports the debugging of both ELF and COFF executable files. You can use Mxdb with any COFF-generating compiler (Common Object File Format). For COFF-generating compilers not listed above, you must use the **ctl** (COFF-to-Legend) translation utility provided with Mxdb. This can be invoked as an assembler option to the compiler; see the appropriate assembler man page and the **ctl** man page.

If you want to place the Mxdb debugging information in an external file, use one of the following in addition to the options described above:

Language/Command	Place Mxdb Debugging Information in an External File
Green Hills	-Xexternal-legend
GNU	-mexternal-legend
cc	-Wc,-external,-fix-bb (No space after comma)
as	-Wc,-external (No space after comma; append "-fix-bb" as above if the assembler was produced by cc or gcc)
Cfront	_external-legend

An external legend file will save disk space and link time; instead of the debugging information being duplicated in the object file, executable file, any incrementally-linked objects, and libraries, all the information is stored in one file, which has a **.lg** suffix.

If you want to keep COFF debugging symbol table information (so you can also debug with sdb or dbx), use one of the following in addition to the options for providing Mxdb debugging information:

Language/Command	Keep COFF Symbol Table Information
Green Hills	-Xkeep-coff
GNU	-mkeep-coff
cc	-Wc,-keep-coff,-fix-bb (No space after comma)
as	-Wc,-keep-coff (No space after comma)

Option Mapping to LEGENDS setting:

Option: -mexternal-legends	Environment: LEGENDS "-external"
Option: -mkeep-coff	Environment: LEGENDS "-keep-coff"

This chapter uses the following program to demonstrate some basic debugging techniques. This C program, named **lastarg.c**, accepts arguments and displays the last argument.

```
/* Sample program: lastarg */
/* Display last argument on command line */

#include <stdio.h>

int i;

main( argc, argv )
int argc;
char *argv[];
{
     (void) printf( "The last argument is %s.\n", argv[i] );
     return 0;
}
```

To compile this program and name the output file **lastarg**, type this command:

```
$ gcc -g -mlegend -o lastarg lastarg.c ↵
$
```

Identifying Debugging Situations

This section discusses how to identify two kinds of debugging situations: program core dumps and erroneous program output.

Program Core Dumps

Debugging a core dump requires a strategy different from that of fixing erroneous program output. You need to find out what caused the core dump; core dumps are discussed in Chapter 8. Note that you cannot debug core dumps (since none are produced) if you are using Interactive COBOL.

Erroneous Program Behavior

Once you successfully compile and link your program, it still may not work the way it should; some bugs do not appear until execution time.

The sample program has such a bug. If you type these arguments:

```
$ lastarg a b c ↵
```

the program displays the following instead of displaying the last argument, which is "c":

```
The last argument is lastarg.
```

To systematically locate and fix a program's bugs, you may want to look at the values of certain variables and reset those values to see what happens, as in the sample debugging session for the **lastarg** program in the following sections.

backquote	Display a backquote: ‘
comma	Display a comma: ,
double-quote	Display a double quote: ”
form-feed	Write a form feed (Ctrl-L)
left-curly-brace	Display a left brace: {
left-parenthesis	Display a left parenthesis: (
left-square-bracket	Display a left bracket: [
new-line	Write a new line (Ctrl-J)
right-curly-brace	Display a right brace: }
right-parenthesis	Display a right parenthesis:)
right-square-bracket	Display a right bracket:]
semicolon	Display a semicolon: ;
single-quote	Display a single quote: ’
space	Display a space character
tab	Write a horizontal tab (Ctrl-I)

debugger-toolkit realm

The debugger-toolkit realm contains commands of interest to macro writers.

elf-debug-rtld	Directs the debugger to do minimal shared object initialization.
elf-stop-for-link-map-changes	Directs the debugger to stop for link-map resynchronized events.
event-list	Displays event-names matching a regular expression.
position-frame	Writes the position's frame number.
position-has-debug-info	Writes if the position has debugging information.
position-line	Writes the position's line number.
position-module	Writes the position's module (legend) name.
position-pc	Writes position's program counter.
position-return-address	Writes the position's return address.
position-routine	Writes the position's routine name.
position-scope-pathname	Writes the position's scope-pathname.
position-source-file	Writes the position's source filename.
process-corefile	Displays or sets the corefile for a process.
process-identifier	Displays the runtime identifier for a process.
process-shared-objects	Writes the pathnames of shared-objects.
process-signal	Displays or sets the current continuation signal for a process.
process-stop-reasons	Writes the reasons why the process stopped.
program-command-line	Writes the current command line that was used to create the process.
program-entry-point	Writes the numeric address of the program entry point.
program-name	Writes the program's pathname.
resolve-filename	Resolves a filename via the debugger's directory-list.

icobol realm

The icobol realm is the default realm for Interactive COBOL users. With the commands in this realm, you can debug your Interactive COBOL programs, dynamically create debugger variables, and modify commands.

Execution

continue	Start or resume execution.
debug	Begin debugging a program.
next	Execute statements between valid break locations, not following CALLs or PERFORMs.
step	Execute statements between valid break locations.
terminate	Terminate the Interactive COBOL Interpreter.

Flow

if	Execute commands if a condition is true.
while	Execute commands while a condition is true.

Event

breakpoint	Set a breakpoint at a valid break location.
delete-events	Delete one or more currently set events.
disable-events	Disable one or more currently set events.
enable-events	Enable one or more currently set events.
event-status	Display or modify information about events.
process-status	Print where and why the run unit stopped.
watch-reference	Monitor a data item.
watchpoint-print	Print a history of watchpoint values.

Language

assign	Assign a value to a destination.
convenience-variables	Display convenience variable names or restart the list.
define-variable	Create a dynamic debugger variable.
delete-variable	Delete a dynamic debugger variable.
describe	Print a description of a program entity.
evaluate	Display the result of an expression.
names	Display all visible names.
variable	Obtain information about debugger variables.

Source

directory-list	Display or set the source directory search path list.
file	Display or set the current source file.
find	Search for a regular expression.
frame	Display or set the current frame number.
list	Display a region of lines.
position	Display or set the current position.
routine	Display or set the current program-name.
view	View text around the current or specified line.

Misc.

event-list	Display event-names matching a regular expression.
perform-walkback	Display the perform stack for CALL stack frames.
refresh-screen	Refresh the Interactive COBOL screen display.
walkback	Display the CALL stack.
version	Display the debugger's version number.

graphical-interface realm

The graphical-interface realm contains commands for graphical interface users.

button-status	Set or display button information
button-pane-status	Display button pane information
clear-messages	Clear the Message Pane
define-button	Create a button in a button pane
define-button-pane	Define a button pane
delete-button	Delete a button from a button pane
delete-button−pane	Delete a button pane
graphics-available	Write "true" if Mxdb's graphical user interface is active
selection	Write the value of any selected text
synchronize-display	Center the Source Pane around the debugger position
xhelp	Execute the xhelp-view program

options realm

The options realm contains commands that control various display and operational options for commands in other realms. These commands are not usually invoked directly by the user. The **c-p:option-status** command invokes these commands in order to display current option settings or to reset an option's value.

bit-format	Display or set bit display format.
character-format	Display or set character display format.
command-history	Display or set the number of commands saved in the command line history mechanism.
convenience-variables	Display or set whether a debugger variable is created whenever an expression is evaluated via the evaluate command.
convenience-variables-limit	Display or set how many convenience variables the debugger will remember.
elide-arrays	Display or set whether same-valued array elements are elided.
floating-point-format	Display or set floating-point display format.
language	Display or set the expression evaluation language.
message-history	Display or set the maximum number of lines of text saved in the Message Pane.
mismatched−legends−allowed	Display or set whether mismatched external debugging information files are allowed when debugging a process.

pager-lines	Display or set the number of lines used by the CP paging facility (page and help).
pointer-dereference-level	Display or set how many times a top-level pointer will be automatically dereferenced and displayed by the debugger.
signed-character-format	Display or set signed-character display format.
signed-integer-format	Display or set signed-integer display format.
source-lines	Display or set the number of lines used by the screen-related source commands (list and view).
stop-commands	Display or set commands to execute when the target process stops for any reason.
string-display	Display or set whether string-like objects are automatically displayed in a string-like or an array-like manner.
string-display-limit	Display or set the number of characters that will be displayed in a string-like manner before elision occurs.
unknown-type-format	Display or set unknown-type display format.
unsigned-character-format	Display or set unsigned-character display format.
unsigned-integer-format	Display or set unsigned-integer display format.
windowed-terminal-emulator	Creates a windowed terminal emulator for a live debugged process, even when the graphical user interface is not active

Comparing Mxdb with sdb and dbx

The Mxdb debugger's simple syntax and semantics contrast with those of sdb, dbx, and many other debuggers. In an Mxdb command line, the command name is always the first element on the command line.

The sdb debugger uses single-character commands that must be at the beginning, middle, or end of the command line, depending on the command and upon what arguments you are specifying. Table 2-2 compares sdb and Mxdb commands:

Table 2-2 Comparison of sdb and Mxdb Commands

sdb	Mxdb Unabbreviated	Mxdb Abbreviated
r prog1	debug prog1, again; continue	deb prog1, a;c
12 c	continue 12	c 12
x!19	assign x 19	as x 19
14 b	breakpoint 14	b 14

The dbx debugger uses longer command names, each of which is a word, an abbreviation of a word, a modification of a word, or two words. Table 2-3 compares dbx and Mxdb commands:

Table 2-3 Comparison of dbx and Mxdb Commands

dbx	Mxdb Unabbreviated	Mxdb Abbreviated
run prog2	debug prog2, again; continue	deb prog2, a; c
cont	continue	c
stepi	step, instructions	st, i
stop at 14	breakpoint 14	b 14

Mxdb can debug more than one executable file in a single session, while sdb cannot. The Mxdb debugger's **debug** command can specify a new executable file, while the sdb debugger's **r** command cannot.

End of Chapter

Chapter 3
Using the Graphical Interface (DG/UX Systems on AViiON Computers)

Mxdb for AViiON computers with the DG/UX operating system has an option, **−g**, that provides a graphical user interface. To use the graphical interface, you must have an X Window System ™ server running on the display machine; see the X man page for more information. Use this option when you invoke Mxdb:

```
$ mxdb −g ↵
```

The Mxdb graphical interface uses both the keyboard and the mouse as input devices. All of the command line Mxdb functionality is still available using the keyboard.

When using the graphical user interface, you can use the following X Toolkit switches (see the X man page for more information):

−name *name*
−display *display*
−geometry *geometry*
−bg *color* or **−background** *color*
−fg *color* or **−foreground** *color*
−iconic
−selectionTimeout
−synchronous
+synchronous
−title *string*
−xrm *resourcestring*

These switches are ignored by the graphical user interface: **−bd** *color*, **−bordercolor** *color*, **−bw** *width*, **−borderwidth** *width*, **−fn** *font*, **−font** *font*, **−rv**, **+rv**, and **−reverse**.

If you are debugging an X application and wish to use X Toolkit switches on the Mxdb command line, place any X Toolkit options for Mxdb before the program name, and any X Toolkit options for the program after the program name. The following command line starts Mxdb with a blue foreground and the application **test_program** with a gray foreground:

```
$ mxdb −g −fg blue test_program −fg gray ↵
```

Chapter 15 describes the graphical-interface realm commands.

Main Window

The main window is made up of these areas:

- Menu Bar
- Source Pane
- Message Pane
- Command Line Pane
- Several Button Panes

Figure 3-1 shows the layout of the window.

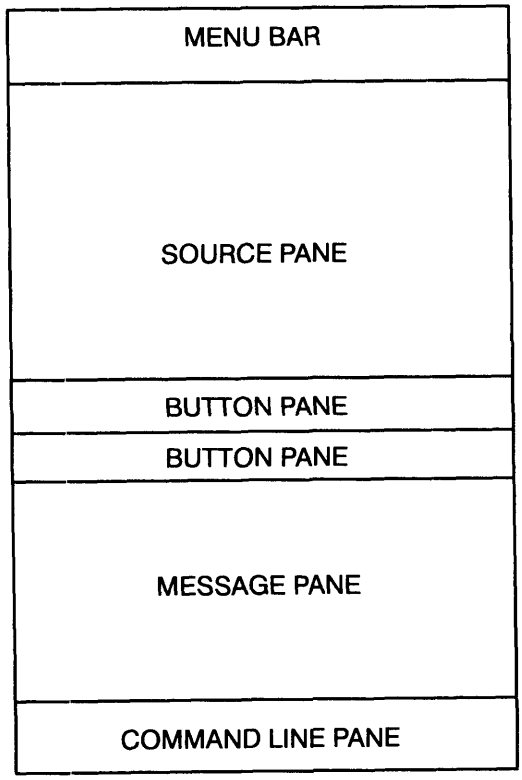


Figure 3-1 The Main Window Layout

The Menu Bar provides access to a row of pull-down menus. The Source Pane displays the source file containing the current debugging position. The Message Pane and the Command Line Pane work together to provide a method of entering commands to, and receiving output from, the debugger. In effect, the Message and Command Line Panes behave as the graphical equivalent of a conventional terminal screen.

The Button Panes provide an alternate method of command entry. These panes contain sets of buttons that, when selected with the mouse, behave as short cuts to the keyboard entry of commands. You can access several of the most commonly used commands by using button panes.

The Source Pane and the Message Pane are resizable. The other panes are fixed in height. You can resize the panes by dragging the square sash with your mouse to the desired new location. The sash is present on the borderline between the panes.

Note that some buttons can be hidden by making the window too narrow.

A more detailed description of each pane is presented below.

Menu Bar

The Menu Bar is a horizontal strip of menu titles. When you select the menu title with the mouse, the corresponding pull-down menu becomes visible. Currently, four menus are available: File, Edit, View, and Help. The File menu contains options that are related to processes. The Edit menu contains the options for manipulating text within the window.

The View menu is used to show or hide button panes and to show or raise viewers for specialized data in the debugger. Currently, there are six options: Stack Frames, Modules, Names, Symbols, Registers, and Button Panes. These will each be explained in detail later.

The Help menu contains various help options, including information about the help system itself, "On Help." For more information about the graphical interface help system, see Appendix A.

File Menu

This menu contains options that manipulate processes. The menu mnemonic is "F."

Exit

This option terminates the application. Selecting this option causes a dialog box to appear, requesting confirmation. Within the dialog are two buttons: "OK" and "Cancel"; the default button is "OK." If you select the "OK" button, the application terminates; otherwise, if you select the "Cancel" button, the application continues execution. The option mnemonic is "x"; no accelerator is available.

Edit Menu

This menu contains options that modify text within the application window. This includes copying, cutting, pasting, and deleting text from pane to pane, as well as from window to window. The menu mnemonic is "E."

Cut

This option removes any selected text in the Command Line Pane and places it in the clipboard. If no text is selected in the Command Line Pane, this option is disabled. The option mnemonic is "t";the accelerator is "Shift+Del."

Copy

This option copies any selected text from any X window that permits text selection into the clipboard; this includes all the panes within the Mxdb window. If no text is selected anywhere, this option is disabled. The option mnemonic is "C"; the accelerator is "Ctrl+Ins."

Paste

This option pastes the clipboard text, if available, into the current text location of the destination cursor in the Command Line Pane. The destination cursor shows the last place that text was inserted, edited, or selected; it is shown as a caret when separate from the insertion cursor. If no text has been placed in the clipboard, this option is disabled. The option mnemonic is "P"; the accelerator is "Shift+Ins."

Delete

This option deletes any selected text in the Command Line Pane without first placing the text in the clipboard. If no text is selected in the Command Line Pane, this option is disabled. The option mnemonic is "D"; no accelerator is available with this option.

View Menu

The View menu is used to show or raise viewers for specialized data in the debugger and to show or hide button panes.

Viewer Menu Entries

The first five entries in the View menu are used to create different viewers. Each viewer is in its own window, and can be moved, resized, and iconified just like the main Mxdb window. To remove a viewer window, use the window manager's Close action.

The core of every viewer is its display area. The format and contents of the display depend on the type of data displayed in that viewer. The right mouse button accesses a popup for debugger operations on that data. Some viewers have a display format button pane with radio buttons (diamond shaped toggle buttons, only one of which can be selected at a time) to control the display format of the data. Some viewers have a regular button pane for debugger command shortcuts related to the displayed data.

There is some performance penalty for using the viewers since they are kept consistent with the debugger and debugged process. The amount of delay depends on the viewer, how large a data set it needs to redisplay, and how often it needs to be updated.

Since the viewers depend on the debugger for information, if you ask for a new viewer or a reformat while the debugger is busy, the redisplay will not occur until after the debugger gets to the request.

Names
This option creates a Names viewer. It displays identifiers defined in the debugged program at the current debugger position. See the debugger realm command **names** for more information.

The popup performs its actions on the name whose line the menu is popped up from. So selecting 'evaluate' from the popup menu (after bringing it up using button 3) anywhere on the third line will evaluate the third name displayed. If an entry in the popup is insensitive, then that means the data on that line is inappropriate for that operation.

The buttons at the bottom of the viewer select the type of names displayed: Routines, Variables, Types, Enumeration Constants, or all of these together. The option mnemonic is "N."

Modules
This option creates a Modules viewer. It displays the module names of all the debuggable compilation unit source files in the target process. See the **modules** command for more information.

The column to the left of the module names is very similar to the first column of the source pane. It contains an arrow pointing to the module containing the current debugger position. Similarly, the process position is indicated by the bold module name. You can position the debugger to any module by clicking with the left mouse button in the first (debugger position arrow) column.

The popup performs its actions on the module whose line the menu is popped up from. So selecting "break all" from the popup menu after bringing it up using button 3 anywhere on the third line will set breakpoints at all the routines in the module named on that line. If an entry in the popup is insensitive, that means the data on that line is inappropriate for that operation. The option mnemonic is "M."

Symbols
This option creates a Symbols viewer. It displays linker symbols from the debugged program. See the debugger realm command **symbols** for more information.

The popup performs its actions on the symbol whose line the menu is popped up from. So selecting 'breakpoint' from the popup menu after bringing it up using button 3 anywhere on the third line will set a breakpoint at the address of the third symbol displayed. If an entry in the popup is insensitive, that means the data on that line is inappropriate for that operation.

The buttons at the bottom of the viewer select how to sort the symbols and whether to display their addresses with them. The option mnemonic is "S."

Stack Frames
This option creates a Stack Frames viewer. It displays the program stack of the debugged process. It is updated whenever the process stops. See the **walkback** command for more information.

The column to the left of the module names is very similar to the first column of the source pane. It contains an arrow pointing to the frame containing the current debugger position. Similarly, the process position is indication by the bold frame description (this is always frame 0). You can position the debugger to any frame by clicking with the left mouse button in the first (debugger position arrow) column.

The popup performs its actions on the frame whose line the menu is popped up from. So selecting 'arguments' from the popup menu (after bringing it up using button 3) anywhere on the third line will send an appropriate command to Mxdb to display frame 2 with its arguments in the Message Pane. The option mnemonic is "F."

Registers
This option creates a Registers viewer. It displays the names and values of the entire register set. See the debugger realm **machine-state** command for more information.

The popup actions operate on the selection. If an entry in the popup is insensitive, then that means there is currently no selection.

The buttons at the bottom perform register-specific debugger operations, and allow you to toggle serialization of the processor. (See the watchpoints-(88k) topic for more information). The option mnemonic is "R."

Button Panes Cascade Menu

The Button Panes menu is used to show or hide panes in the Mxdb main window. User defined button panes will appear on this menu in addition to the predefined entries below.

Process Buttons (P)
This option toggles the visibility of the Process Buttons button pane. The continue, next, step, debug again, terminate, detach, and event status buttons are shortcuts to the commands of those names. This button pane is displayed by default.

The interrupt button can be used to interrupt Mxdb or the target process. It is only sensitive when Mxdb is busy processing a command or the target process is executing. If the target process is running, the interrupt button will interrupt it; otherwise, the interrupt button will only interrupt whatever command Mxdb is running. All pending commands that have been typed (or button-pressed) ahead will be flushed by the interrupt. The option mnemonic is "P."

Machine Buttons (M)
This option toggles the visibility of the Machine Buttons button pane. The next i, step i, disassemble, machine state, address−map, and symbols buttons are shortcuts to the commands of those names. This button pane is not displayed by default. The option mnemonic is "M."

Stack Buttons (S)
This option toggles the visibility of the Stack Buttons button pane. The walkback, finish, frame +1, frame −1, and top frame buttons are shortcuts to the commands of those names. This button pane is displayed by default. The option mnemonic is "S."

Help Menu

This menu contains various reference options that start the graphical interface help system or reposition the help system if it is already started. For more information about the graphical interface help system, see Appendix A. Selecting an option causes a non−modal window (a window that does not need to be closed before further interaction can occur in the Mxdb window) to appear with scrollable (if applicable) help text within it. You can close the window by selecting the "Close" button at the bottom left corner of the help window. The menu mnemonic is "H."

On Help

This option positions the graphical interface help system to a description of the help system. The option mnemonic is "H."

On GUI

This option positions the graphical interface help system to a description of Mxdb's graphical user interface. The option mnemonic is "G."

On Mxdb

This option positions the graphical interface help system to the information contained in *Using the Multi-extensible Debugger*. The option mnemonic is "M."

On CP

This option positions the graphical interface help system to the information contained in *Using the Command Processor*. The option mnemonic is "C."

Tutorial

This option positions the graphical interface help system to graphical interface demos, which show you how to use Mxdb's major features. The option mnemonic is "T."

Table of Contents

This option opens the graphical interface help system Table of Contents dialog. You can browse through the online manual contents and select a topic if you would like more information about it. The option mnemonic is "b."

Source Pane

The Source Pane provides a constant display of the source file you are debugging and provides a visual means of setting or removing breakpoints. The pane is composed of a filename label, which shows the name of the source file being displayed, and a horizontally and vertically scrolled display area below the filename label. The display area shows numbered lines of the source code in which the current debugger position is located. The source file associated with the current debugger position is always visible in the display if the source file is available.

An arrow icon (->) to the left of the line number denotes the debugger position (the static position). A single click in the first column of the Source Pane (where the arrow icon is displayed) repositions the debugger to that line.

A bold source line indicates the program position (the point at which program execution is pending). An enabled breakpoint is denoted by a solid stop sign icon, while a disabled breakpoint is denoted by a hollow stop sign icon. Two or more breakpoints (all enabled) on a line are shown with two solid stop signs, side by side. Two or more breakpoints (all disabled) on a line are shown with two hollow stop signs, side by side. If you have two or more breakpoints on a line, at least one of which is enabled and one of which is disabled, you will see one solid stop sign and one hollow stop sign, side by side.

When breakpoints are added, deleted, disabled, or enabled with the **event-status**, **delete-events**, **enable-events**, and **disable-events** commands, the icons change accordingly.

Position/Breakpoint	Denoted By
Debugger position	Arrow icon before the line number (->)
Program position	Bold line number and source line
One enabled breakpoint	Stop sign icon before the line number
One disabled breakpoint	Hollow stop sign icon before the line number
Multiple enabled breakpoints	Two stop sign icons before the line number
Multiple disabled breakpoints	Two hollow stop signs before the line number
Enabled and disabled breakpoints on the same line	One solid and one hollow stop sign before the line number

The Source Pane provides a short cut to setting and clearing breakpoints. You can single-click the mouse on the line number of the line where you want to set or clear a breakpoint. When you are on a line number, the cursor will be an arrow pointing to the upper right of the screen. Single-clicking on a line without a breakpoint causes a breakpoint to be set on that line; conversely, single-clicking on a line with a breakpoint causes the breakpoints on the line to be cleared.

Any portion of the displayed source text may be selected with the mouse. The Source Pane is read-only, however, so you cannot cut from or paste into the displayed source.

Source Pane Popup

The actions in the popup menu for the Source Pane operate on the current selection. The 'source pane' cascade contains search..., edit, sync pane, and dir list options. The search option raises a dialog that performs regular expression searches on the contents of the Source Pane. See the **debugger:edit**, **g−i:synchronize−display**, and **debugger:directory−list** for more information on the other options.

Message Pane

The Message Pane serves as a display area for previous command line interactions. The commands you have entered, as well as the debugger outputs and error messages, are displayed in a scrolled sub-window with horizontal and vertical scroll bars. A specified number of lines are stored by the Message Pane to serve as a partial session history. You can specify this number of lines with the **option-status** command's **message-history** option.

The graphical interface uses different fonts to differentiate input and output visually. By default, prompts and user inputs are displayed in a bold font, non-error outputs are displayed in a regular font, and error messages are displayed in an italic font. You can change these default settings to suit your preferences (see the section "Resources" later in this chapter).

Any portion of the displayed text may be selected with the mouse. The Message Pane is read-only. The actions in the popup menu for the Message Pane operate on the current selection.

Message Pane Popup

Message Pane Popup Cascade Menu

The message pane and all of the viewer display panes share a popup cascade of common useful functions. It is accessed through the cascade button in the last position of their popup menus.

The 'message pane' cascade contains search..., sync pane, and clear options. The search option raises a dialog that performs regular expression searches on the contents of the Message Pane. If there is an arrow column on the left of this message pane (or viewer), and there is an arrow displayed somewhere, the sync pane option scrolls the window to make it visible. The clear option just clears the contents of the message pane.

Search Dialog

The Search Dialog is available by selecting the "Search..." item from the 'source pane' cascade menu, or any 'message pane' cascade. It performs searches in the pane it was popped up from. It takes a regular expression pattern to search for, and optionally can search backward or wrap at the end of the pane contents. After it is popped up, the first time it searches it will start at the beginning of the visible text. Subsequently, it searches from the previously found text. Found text is selected, and the pane scrolls to make it visible. If the search fails, the dialog will beep. See the command-processor's regular expression topic for more information.

Command Line Pane

The Command Line Pane is composed of an input prompt and a rectangular text input area. Commands to the debugger are entered in this pane. If a command line becomes longer than the input box, the text scrolls automatically to accommodate the user input. Press the New Line key to complete your command.

The input prompt displays the realm prompt. It also displays the argument prompt during command prompting, the option prompt during option prompting, and the query prompt for the **c-p:query** command. When the debugger is busy, the prompt becomes insensitive.

You can enter commands in the Comand Line Pane while the debugger is busy; they will be put into a queue. Commands that you enter, but have not been executed, are not displayed in the Message Pane until the debugger starts processing them. However, those commands are available in the command history (see below) if you want to see what you've typed ahead.

By default, the Command Line Pane provides key bindings that emulate the editread functionality, described in the next chapter. With these key bindings, you can easily perform operations such as moving backward and forward on the command line and inserting or deleting text. An alternate set of key bindings, which emulate emacs key bindings, is also available (see the section "Keys" later in this chapter).

This pane also features a command history feature. The pane maintains a history list by recording the entered commands. The list behaves as a first-in-first-out (FIFO) queue; when the list reaches its maximum size, the first command in the list is removed before the newest command is appended to the end of the list. The list also behaves circularly when the user moves forward beyond the last command or backward before the first command in the list. By using this command history list, you may conveniently access previously entered commands for perusal or re-entry. The number of entries saved in the command history can be set with the **option-status** command's **command-history** option.

Button Panes

A Button Pane contains one row of user-definable buttons. The buttons and their associated command sequences provide short cuts to entering commands manually. You can create a button by using the **define-button** command in the graphical-interface realm. For convenience, a default set of buttons and button panes exists at startup; you can change these buttons interactively with the **define-button**, **define-button-pane**, **delete-button-pane** , and **delete-button** commands.

Default Button Definitions

By default, the Mxdb default button definitions share the following characteristics:

- Are not suppressed from echoing in the Message Pane (**,suppress-echo no**).
- Do not append an optional or required selection (**,append-selection no ,append-required-selection no**).

Here is a general command invocation, in its most verbose form, for a typical default button:

define−button *"label"* {*commands*} *position*, '
append−selection no, **append−required−selection no**, **suppress−echo no**, *pane*

where *label*, *commands*, *pane*, and *position* vary.

Here is the default set of buttons and their corresponding debugger commands:

Table 3-1 Default Buttons

Label	Command	Position	Button Pane
Interrupt	{interrupt}	1	ProcessButtons
continue	{continue}	2	ProcessButtons
next	{next}	3	ProcessButtons
step	{step}	4	ProcessButtons
debug again	{debug,again}	5	ProcessButtons
terminate	{terminate}	6	ProcessButtons
detach	{detach}	7	ProcessButtons
event status	{event−status}	8	ProcessButtons
process status	{process−status}	9	ProcessButtons
next i	{next,i}	1	MachineButtons
step i	{step,i}	2	MachineButtons
disassemble	{disassemble}	3	MachineButtons
instruction view	{instruction−view}	4	MachineButtons
machine state	{machine−state}	5	MachineButtons
address map	{address−map}	6	MachineButtons
symbols	{symbols}	7	MachineButtons
walkback	{walkback}	1	StackButtons
finish	{finish}	2	StackButtons
frame +1	{frame +1}	3	StackButtons
frame −1	{frame −1}	4	StackButtons
top frame	{frame 0}	5	StackButtons

Normally, you do not specify the actual position where the button is to be placed; new buttons default to the end of the specified (or default) button pane. The examples give the exact position so that the buttons are independent of one another.

This is how two buttons could be defined interactively through the **define-button** command:

```
(g-i) define-button "sync display" g-i:synchronize-display 1, append-selection
no, append-required-selection no, suppress-echo no ↵

(g-i) define-button "directory list" debugger:directory-list 2, append-selection
no, append-required-selection no, suppress-echo no ↵
```

Usually, button definitions are very terse:

```
(g-i) define-button "foo" write foo ↵

(g-i) define-button bar {write bar; mymacro} ↵
```

You may wish to create a new button pane to organize new buttons:

```
(g-i) define-button-pane, name MyButtons, label "My Buttons" ↵
```

This creates an empty button pane at the end of the button pane area in the middle of the main window. It also adds a toggle button to the "Panes" menu in the menu bar. This toggle button controls whether or not this new button pane is visible. By default, the toggle is pushed in and the button pane is visible. To make it invisible, simply click on the toggle button, which then pops out.

You may add buttons to this new button pane with the **define-button** command, specifying the pane argument with the name specified:

```
(g-i) define-button my_button {write foo}, pane MyButtons ↵
```

Execution Window

A separate window is always provided for debugger process input/output when the graphical user interface is active. Note that this option is also available in command-line mode as well. See the windowed-terminal-emulator option for complete details.

Interrupts

If you're in the graphical user interface, you can issue interrupts in the Command Line Pane or in the execution window. Issuing a SIGINT or SIGQUIT to the Command Line Pane interrupts Mxdb when the debugged process is *not* executing. When the debugged process is executing, the SIGINT or SIGQUIT signal is sent to the debugged process instead. This applies to attached processes as well. Note that any queued commands will be discarded.

Issuing any interrupt to the execution window, which is the controlling tty of the process being debugged, will interrupt the debugged process. If you have attached to a process, interrupts must be issued on the tty from which the process started execution or issued via the UNIX command **kill**.

For more information about signals, see "Signal Debugging" in Chapter 7.

Customization

You can change the default settings of various behavioral and visual aspects of the graphical user interface. However, use caution; changing default settings can produce unexpected behavior. Either use the **.Xdefaults** file in your home directory or use the standard command line X Toolkit switches to override the default settings supplied in the Mxdb file in **/usr/lib/X11/app-defaults**. The **app-defaults/Mxdb** file contains defaults for monochromatic, gray-scale, and color systems that can be transferred to your **.Xdefaults** file. Be aware that changing a machine's **app-defaults/Mxdb** file will affect all Mxdb graphical interfaces started on that machine. See X documentation for X Toolkit switches and the section "Resources" in this chapter for more information on modifiable settings.

Fonts

You can specify different fonts for the various text components used by the graphical user interface and its help system. They use the Motif XmFontList resource type, which associates a specific font of some font family and point size to a character set name. View the supplied Mxdb **app-defaults** file for an example of XmFontList usage; see the X and XmFontList man pages for more information.

The graphical user interface recognizes three character set names: "normal," "bold," and "italic."

The Source Pane Text widget uses the **normal** character set for the program source text display, and the **bold** character set for the program position.

The Source Pane Numbers widget uses the **normal** character set to display the line numbers, and the **bold** character set for the program position.

The Source Pane Tags and Source Current widgets use only the **normal** character set for the debugger position arrow and breakpoint symbols.

```
Mxdb (ApplicationShell)
|
+- MainForm (XmForm)
   |
   +- MainVerticalPane (XmPanedWindow)
   |  |
   |  +- DebuggerPositionPane (XmRowColumn)
   |  |  |
   |  |  +- DebuggerPositionDisplay (XmPushButton)
   |  |
   |  +- SourcePane (XmForm)
   |  |  |
   |  |  +- SourcePaneSW (XmScrolledWindow)
   |  |  |  |
   |  |  |  +- SourcePaneBB (XmBulletinBoard)
   |  |  |  |  +- SourcePaneCurrent (PdeTextWidget)
   |  |  |  |  +- SourcePaneCurrentSeparator (XmSeparator)
   |  |  |  |  +- SourcePaneTags (PdeTextWidget)
   |  |  |  |  +- SourcePaneNumbers (PdeTextWidget)
   |  |  |  |  +- SourcePaneNumbersSeparator (XmSeparator)
   |  |  |  |  +- SourcePaneText (PdeTextWidget)
   |  |  |  |
   |  |  |  +- SourcePaneHSB (XmScrollBar)
   |  |  |  |
   |  |  |  +- SourcePaneVSB (XmScrollBar)
   |  |  |
   |  |  *- SourcePopupMenu (XmRowColumn)
   |  |     |
   |  |     +- EvaluateSourcePopup (XmPushButton)
   |  |     |
   |  |     +- DescribeSourcePopup (XmPushButton)
   |  |     |
   |  |     +- AddressSourcePopup (XmPushButton)
   |  |     |
   |  |     +- BreakpointSourcePopup (XmPushButton)
   |  |     |
   |  |     +- PositionSourcePopup (XmPushButton)
   |  |     |
   |  |     +- WatchReferenceSourcePopup (XmPushButton)
   |  |     |
   |  |     +- HelpSourcePopup (XmPushButton)
   |  |     |
   |  |     +- SourcePopupCascadeCascade (XmCascadeButton)
   |  |     *- SourcePopupCascade (XmRowColumn)
   |  |        +- SearchSourceCascade (XmPushButton)
   |  |        +- SynchronizeSourceCascade (XmPushButton)
   |  |        +- EditSourceCascade (XmPushButton)
   |  |        +- DirListSourceCascade (XmPushButton)
   |  |        +- HelpSourceCascade (XmPushButton)
   |  |
```

```
|   +- ProcessButtons (XmRowColumn)
|   |   +- Interrupt (XmPushButton)
|   |   +- ProcessButtons-continue-Button (XmPushButton)
|   |   +- ProcessButtons-next-Button (XmPushButton)
|   |   +- ProcessButtons-step-Button (XmPushButton)
|   |   +- ProcessButtons-debug again-Button (XmPushButton)
|   |   +- ProcessButtons-terminate-Button (XmPushButton)
|   |   +- ProcessButtons-detach-Button (XmPushButton)
|   |   +- ProcessButtons-event status-Button (XmPushButton)
|   |   +- ProcessButtons-process status-Button (XmPushButton)
|   |
|   +- MachineButtons (XmRowColumn)
|   |   +- MachineButtons-next i-Button (XmPushButton)
|   |   +- MachineButtons-step i-Button (XmPushButton)
|   |   +- MachineButtons-disassemble-Button (XmPushButton)
|   |   +- MachineButtons-instruction view-Button (XmPushButton)
|   |   +- MachineButtons-machine state-Button (XmPushButton)
|   |   +- MachineButtons-address map-Button (XmPushButton)
|   |   +- MachineButtons-symbols-Button (XmPushButton)
|   |
|   +- StackButtons (XmRowColumn)
|   |   +- StackButtons-walkback-Button (XmPushButton)
|   |   +- StackButtons-finish-Button (XmPushButton)
|   |   +- StackButtons-frame +1-Button (XmPushButton)
|   |   +- StackButtons-frame -1-Button (XmPushButton)
|   |   +- StackButtons-top frame-Button (XmPushButton)
|   |
|   +- MessagePane (XmForm)
|   |   |
|   |   +- MessagePaneSW (XmScrolledWindow)
|   |   |   |
|   |   |   +- MessagePaneBB (XmBulletinBoard)
|   |   |   |   +- MessagePaneText (PdeTextWidget)
|   |   |   |
|   |   |   +- MessagePaneHSB (XmScrollBar)
|   |   |   |
|   |   |   +- MessagePaneVSB (XmScrollBar)
|   |   |
|   |   *- MessagePopupMenu (XmRowColumn)
|   |       |
|   |       +- EvaluateMessagePopup (XmPushButton)
|   |       |
|   |       +- DescribeMessagePopup (XmPushButton)
|   |       |
|   |       +- AddressMessagePopup (XmPushButton)
|   |       |
|   |       +- BreakpointMessagePopup (XmPushButton)
|   |       |
|   |       +- PositionMessagePopup (XmPushButton)
|   |       |
|   |       +- WatchReferenceMessagePopup (XmPushButton)
|   |       |
|   |       +- HelpMessagePopup (XmPushButton)
```

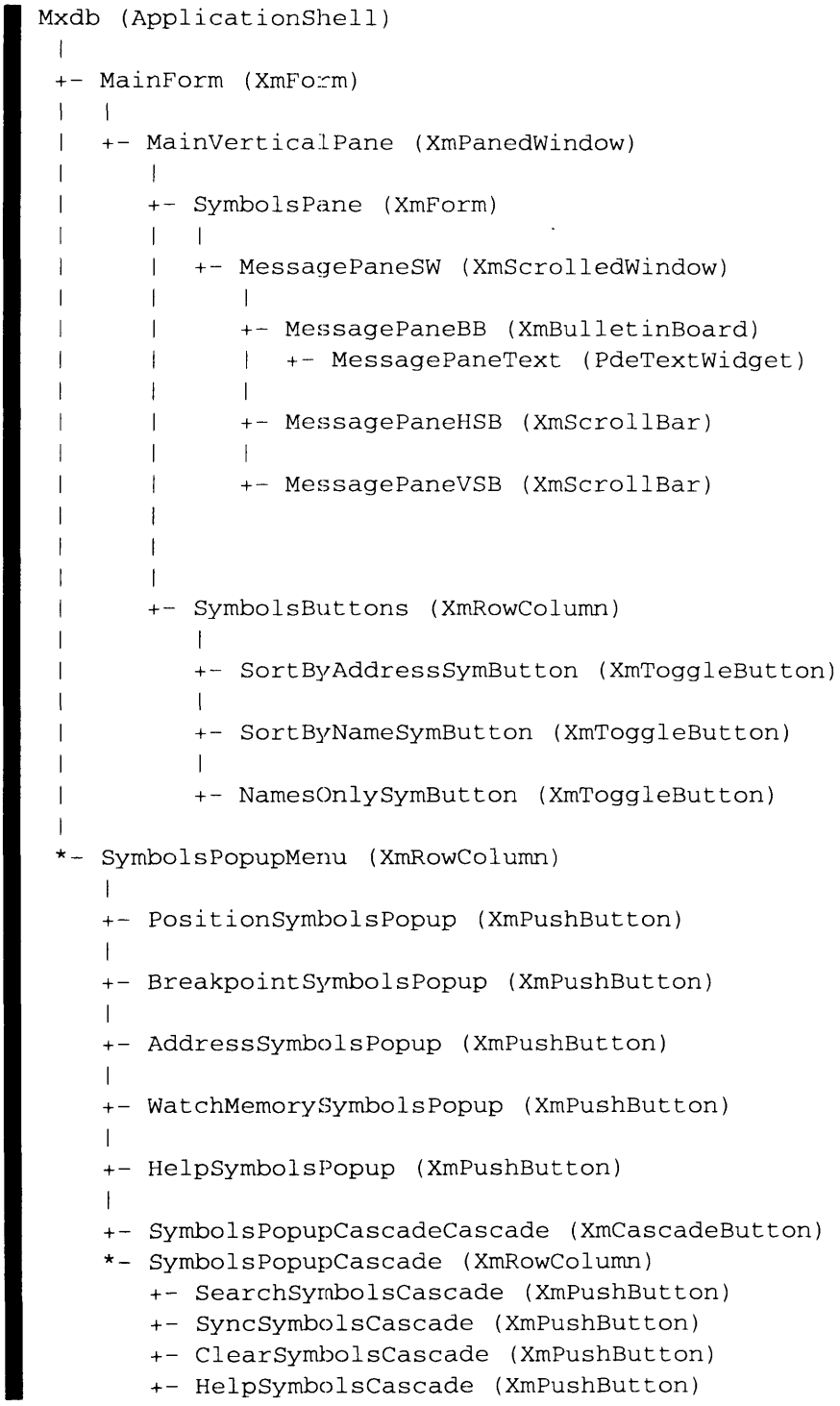
```
|   |      |
|   |      +- MessagePopupCascadeCascade (XmCascadeButton)
|   |      *- MessagePopupCascade (XmRowColumn)
|   |         +- SearchMessageCascade (XmPushButton)
|   |         +- SyncMessageCascade (XmPushButton)
|   |         +- ClearMessageCascade (XmPushButton)
|   |         +- HelpMessageCascade (XmPushButton)
|   |
|   +- CommandLine (XmForm)
|      +- CommandLinePrompt (XmLabel)
|      +- CommandLineText (XmText)
|
+- MenuBar (XmRowColumn)
   |
   +- FileCascade (XmCascadeButton)
   *- File (XmRowColumn)
   |  +- Exit (XmPushButton)
   |
   +- EditCascade (XmCascadeButton)
   *- Edit (XmRowColumn)
   |  +- Cut (XmPushButton)
   |  +- Copy (XmPushButton)
   |  +- Paste (XmPushButton)
   |  +- EditSeparator (XmSeparator)
   |  +- Delete (XmPushButton)
   |
   +- ViewCascade (XmCascadeButton)
   *- View (XmRowColumn)
   |  +- NamesViewerItem (XmPushButton)
   |  +- ModulesViewerItem (XmPushButton)
   |  +- StackFramesViewerItem (XmPushButton)
   |  +- RegistersViewerItem (XmPushButton)
   |  +- SymbolsViewerItem (XmPushButton)
   |  +- Button PanesCascade (XmCascadeButton)
   |  *- Button Panes (XmRowColumn)
   |     +- ProcessButtonsToggle (XmToggleButton)
   |     +- MachineButtonsToggle (XmToggleButton)
   |     +- StackButtonsToggle (XmToggleButton)
   |
   +- HelpCascade (XmCascadeButton)
   *- Help (XmRowColumn)
      +- On Help (XmPushButton)
      +- On GUI (XmPushButton)
      +- On Mxdb (XmPushButton)
      +- On CP (XmPushButton)
      +- Tutorial (XmPushButton)
      +- Command Summary (XmPushButton)
      +- Table of Contents (XmPushButton)
```

```
Mxdb (ApplicationShell)
 |
 +- MainForm (XmForm)
 |  |
 |  +- MainVerticalPane (XmPanedWindow)
 |     |
 |     +- ModulesPane (XmForm)
 |        |
 |        +- MessagePaneSW (XmScrolledWindow)
 |           |
 |           +- MessagePaneBB (XmBulletinBoard)
 |           |  +- MessagePaneCurrent (PdeTextWidget)
 |           |  +- MessagePaneCurrentSeparator (XmSeparator)
 |           |  +- MessagePaneText (PdeTextWidget)
 |           |
 |           +- MessagePaneHSB (XmScrollBar)
 |           |
 |           |
 |           +- MessagePaneVSB (XmScrollBar)
 |
 *- ModulesPopupMenu (XmRowColumn)
    |
    +- BreakAllModulesPopup (XmPushButton)
    |
    +- UnbreakAllModulesPopup (XmPushButton)
    |
    +- HelpModulesPopup (XmPushButton)
    |
    +- ModulesPopupCascadeCascade (XmCascadeButton)
    *- ModulesPopupCascade (XmRowColumn)
       +- SearchModulesCascade (XmPushButton)
       +- SyncModulesCascade (XmPushButton)
       +- ClearModulesCascade (XmPushButton)
       +- HelpModulesCascade (XmPushButton)
```

```
Mxdb (ApplicationShell)
 |
 +- MainForm (XmForm)
 |  |
 |  +- MainVerticalPane (XmPanedWindow)
 |     |
 |     +- StackFramesPane (XmForm)
 |        |
 |        +- MessagePaneSW (XmScrolledWindow)
 |           |
 |           +- MessagePaneBB (XmBulletinBoard)
 |           |  +- MessagePaneCurrent (PdeTextWidget)
 |           |  +- MessagePaneCurrentSeparator (XmSeparator)
 |           |  +- MessagePaneText (PdeTextWidget)
 |           |
 |           +- MessagePaneHSB (XmScrollBar)
 |           |
 |           +- MessagePaneVSB (XmScrollBar)
 |
 *-StackFramesPopupMenu (XmRowColumn)
   |
   +- ArgumentsStackFramesPopup (XmPushButton)
   |
   +- LocalsStackFramesPopup (XmPushButton)
   |
   +- FinishStackFramesPopup (XmPushButton)
   |
   +- HelpStackFramesPopup (XmPushButton)
   |
   +- StackFramesPopupCascadeCascade (XmCascadeButton)
   *- StackFramesPopupCascade (XmRowColumn)
      +- SearchStackFramesCascade (XmPushButton)
      +- SyncStackFramesCascade (XmPushButton)
      +- ClearStackFramesCascade (XmPushButton)
      +- HelpStackFramesCascade (XmPushButton)
```

```
Mxdb (ApplicationShell)
 |
 +- MainForm (XmForm)
 |   |
 |   +- MainVerticalPane (XmPanedWindow)
 |      |
 |      +- NamesPane (XmForm)
 |      |   |
 |      |   +- MessagePaneSW (XmScrolledWindow)
 |      |      |
 |      |      +- MessagePaneBB (XmBulletinBoard)
 |      |      |   +- MessagePaneText (PdeTextWidget)
 |      |      |
 |      |      +- MessagePaneHSB (XmScrollBar)
 |      |      |
 |      |      +- MessagePaneVSB (XmScrollBar)
 |      |
 |      +- NamesButtons (XmRowColumn)
 |         +- AllKindsNamesButton (XmToggleButton)
 |         +- RoutinesNamesButton (XmToggleButton)
 |         +- VariablesNamesButton (XmToggleButton)
 |         +- TypesNamesButton (XmToggleButton)
 |         +- EnumsNamesButton (XmToggleButton)
 |
 *- NamesPopupMenu (XmRowColumn)
    |
    +- EvaluateNamesPopup (XmPushButton)
    |
    +- DescribeNamesPopup (XmPushButton)
    |
    +- AddressNamesPopup (XmPushButton)
    |
    +- WatchReferenceNamesPopup (XmPushButton)
    |
    +- PositionNamesPopup (XmPushButton)
    |
    +- BreakpointNamesPopup (XmPushButton)
    |
    +- HelpNamesPopup (XmPushButton)
    |
    +- NamesPopupCascadeCascade (XmCascadeButton)
    *- NamesPopupCascade (XmRowColumn)
       +- SearchNamesCascade (XmPushButton)
       +- SyncNamesCascade (XmPushButton)
       +- ClearNamesCascade (XmPushButton)
       +- HelpNamesCascade (XmPushButton)
```

```
Mxdb (ApplicationShell)
 |
 +- MainForm (XmForm)
 |  |
 |  +- MainVerticalPane (XmPanedWindow)
 |     |
 |     +- RegistersPane (XmForm)
 |     |  |
 |     |  +- MessagePaneSW (XmScrolledWindow)
 |     |     |
 |     |     +- MessagePaneBB (XmBulletinBoard)
 |     |     |  +- MessagePaneText (PdeTextWidget)
 |     |     |
 |     |     +- MessagePaneHSB (XmScrollBar)
 |     |     |
 |     |     +- MessagePaneVSB (XmScrollBar)
 |     |
 |     +- RegistersButtons (XmRowColumn)
 |        +- PSRFlagsRegButton (XmPushButton)
 |        +- FPSRFlagsRegButton (XmPushButton)
 |        +- FPCRFlagsRegButton (XmPushButton)
 |        +- SerializeRegButton (XmToggleButton)
 |
 *- RegistersPopupMenu (XmRowColumn)
    |
    +- SymbolicPCRegistersPopup (XmPushButton)
    |
    +- DisassembleRegistersPopup (XmPushButton)
    |
    +- PositionRegistersPopup (XmPushButton)
    |
    +- HelpRegistersPopup (XmPushButton)
    |
    +- RegistersPopupCascadeCascade (XmCascadeButton)
    *- RegistersPopupCascade (XmRowColumn)
       +- SearchRegistersCascade (XmPushButton)
       +- SyncRegistersCascade (XmPushButton)
       +- ClearRegistersCascade (XmPushButton)
       +- HelpRegistersCascade (XmPushButton)
```



This page intentionally left blank.

The "applicationShell" class is an X Toolkit widget. All the other widgets except for "pdeText" are standard Motif widgets. "pdeText" is a customized widget whose parent class is XmPrimitive. See the man page for XmPrimitive for resources that are inherited from XmCore (the superclass of XmPrimitive) and XmPrimitive. A list of modifiable resources for this widget class is provided below. There is no man page for pdeText.

Table 3–8 pdeText Resource Set

NAME	DESCRIPTION	DEFAULT
fontList	Fonts used by the class	STRING_DEFAULT_CHARSET
minLineWidth	Min. line width (pixels)	1
minLineHeight	Min. line height (pixels)	1
tabWidth	Spaces between tab stops	8
selectionArray	See XmText man page	{XmSELECT_POSITION, XmSELECT_WORD, XmSELECT_LINE, XmSELECT_PARAGRAPH}
selectionArrayCount	See XmText man page	4

Setting the following resources will allow Mxdb and the execution window to both fit on the screen without overlap. For mterm:

```
mxdb_ws_term.geometry:      55x12+740+90
Mxdb*geometry:              +20+70
```

For xterm:

```
mxdb_ws_term*geometry:      55x12+740+90
Mxdb*geometry:              +20+70
```

As for key translations, please study the X documentation carefully (as well as the section "Keys" in this chapter) before attempting modification.

Keyboard traversal support is available on a limited basis using resources. By default, traversal is available from the Command Line to the sashes for resizing the Source Pane and Message Pane using the up and down arrow keys. To activate traversal from the Command Line, press Control+TAB for forward motion and Shift+TAB for backward motion. The TAB key alone inserts a tab into the Command Line.

In order to allow the Command Line to accept tab characters, the following resource is set by default:

```
Mxdb*CommandLineText.editMode: MULTI_LINE_EDIT
```

Evaluating Function Calls

If you use function call syntax in a command that accepts a language expression (such as **evaluate, assign, breakpoint, define-variable, if,** and **while**), the Mxdb debugger attempts to invoke the function when it evaluates the expression. Note that Mxdb will not try to invoke a function that is an argument to the **describe** command since expressions supplied to **describe** are not evaluated.

If the function invocation is successful, the function is then called whenever the expression involving the call is evaluated. Thus, a function appearing in a **breakpoint** command will be called every time the condition is tested. Here are some commands that result in calling a function:

```
(debug) evaluate some_function() ↵
(debug) assign some_var some_function() ↵
(debug) break somewhere, if (some_function() != 0) ↵
(debug) define-variable debugger_var some_function ↵
(debug) if ( some_function() ) {write something} ↵
```

All events are active when the invoked process is being executed. If the invoked routine gets an event that is not the expected return address, the process-status is noted and a note is issued saying that the process stopped in a debugger-invoked context before returning to the top level.

You can debug as you wish in the invoked routine context. Your stack is terminated at this point by a frame at this address: __debug_info+<*N*>. Here is an example:

```
(debug) b one_param ↵
(debug) eval one_param(4) ↵
Stopped at frame 0, line 10, scope \test4\one_param, pc 0x101dc
 breakpoint "2"
Note: Process stopped in debugger-invoked routine context.
(debug) walk, a ↵
frame 0, line 10, scope \test4\one_param, pc0x101dc
    i           = 4
frame 1, pc __debug_info+24
```

These invocation contexts will nest so that a user can invoke routines even in invoked contexts and the state of the process will be reinstated appropriately as each invocation returns (if it returns).

When you invoke a routine, any signal that stopped the target process is discarded.

Modifying the Expression Language

To change the current expression language, set the **language** option by using the **option-settings** argument of the **option-status** command. The valid values are c, c++, pascal, fortran, and icobol. Note that any modification stays in effect until the debugger's position changes. The debugger position is changed when the **position** or **view** command is supplied with arguments or the debugged process is continued (and it stops).

You can enable Mxdb to evaluate a C expression while you are debugging a FORTRAN program by typing the following:

```
(debug) option-status la c↵
(debug)
```

The following command restores FORTRAN as the expression language:

```
(debug) option lang fortran ↵
(debug)
```

End of Chapter

The following command displays the values for the next-to-top stack frame:

```
(debug) mach top+1 ↵
$pc   0x0001019c __start+116
$r0   0x00000000 $r1   <invalid>  $r2  <invalid>  $r3  <invalid
>
$r4   <invalid>  $r5   <invalid>  $r6  <invalid>  $r7  <invalid
>
$r8   <invalid>  $r9   <invalid>  $r10 <invalid>  $r11 <invalid
>
$r12  <invalid>  $r13  <invalid>  $r14 0x00013e00 $r15 0x00013e
00
$r16  0x00000001 $r17  0x0041ab64 $r18 0x0041ade0 $r19 0x0041bd
40
$r20  0x0041ab60 $r21  0x0041c480 $r22 0x00000001 $r23 0x0051e1
00
$r24  0x0051d900 $r25  0xeffffe38 $r26 0x00000000 $r27 0x000000
00
$r28  0x004084f0 $r29  0x00000000 $r30 0x00000000 $r31 0xeffffd
f0
$fpsr 0x00000000 $fpcr 0x0000001a $psr 0x000003f0
$cfa  0xeffffe38
(debug)
```

Note that registers $r1–$r13 are displayed with a value of "<invalid>"; their contents are only defined in a top stack frame context. Also note that the instruction pointer registers ($sxip, $snip, and $sfip) are not displayed; their values are always invalid for frames other than the top frame.

Creating a Disassembly Listing (disassemble, instruction-view)

The display of an instruction listing is generally known as "disassembly" because it is the inverse of the process generally performed by assemblers or programs taking machine instructions specified in a source file and encoding them into actual machine code that can be executed. In the debugger, the machine code in the process image is decoded back into this source-like format.

To create an instruction listing, use either the **disassemble** or **instruction-view** commands. Both commands provide instruction listing in the same format. The **instruction-view** command generally provides more contextual information than **disassemble** by attempting to output a window of instructions around the current or specified position.

Note that the disassembly listings will differ slightly between COFF and ELF programs. In ELF programs all registers and reserved operands are prefixed by "#" as the version 03.00 and later assembler syntax mandates. Since this assembler syntax is only for ELF, these pound signs are not included in the COFF display.

To display all the instructions associated (if debugging information exists) with the current line, beginning at the program counter associated with the current debugger position, use the **disassemble** command with the **line** argument, which has the value "current":

```
(debug) disas, line current ↵
*    12 0x101e4 _main+24 or.u          r13, r0, 0x40
        0x101e8 _main+28 ld            r12, r13, 0x6850
        0x101ec _main+32 mak           r12, r12, 0<2>
        0x101f0 _main+36 ld            r11, r30, 0x14
        0x101f4 _main+40 addu          r12, r12, r11
        0x101f8 _main+44 or.u          r2, r0, 0x1
        0x101fc _main+48 or            r2, r2, 0x1b0
        0x10200 _main+52 ld            r3, r0, r12
        0x10204 _main+56 bsr           _printf
(debug)
```

Note that the current instruction is highlighted with an asterisk in the same manner as the **view** command highlights the current source line. The highlighted instruction is the actual static (pc) position where a breakpoint is set if you invoke the **breakpoint** command with no arguments. The asterisks displayed at both the source and "machine" levels exactly identify your current debugger position at any time.

If you use the **disassemble** command with no arguments, you will see all instructions associated (if debugging information exists) with the current line, beginning with the current instruction.

To display a window of instructions associated (if debugging information exists) with the current line, use the **instruction-view** command:

```
(debug) dis  „display instructions associated with current position ↵
*     55 0x10520 _main+24        or.u          r2, r0, 0x1
         0x10524 _main+28        or            r2, r2, 0x1b4
         0x10528 _main+32        bsr           _printf
(debug) op source 5 „provide a 5 instruction window ↵
(debug) i-vi      „display more context about the current position ↵
      26 0x10518 _main+16        st            r2, r30, 0x4e0
         0x1051c _main+20        st            r3, r30, 0x4e4
*     55 0x10520 _main+24        or.u          r2, r0, 0x1
         0x10524 _main+28        or            r2, r2, 0x1b4
         0x10528 _main+32        bsr           _printf
```

To display the instructions associated with two lines, beginning at the current debugger position, specify the number of lines:

```
(debug) disas 2 ↵
*    12 0x101e4 _main+24 or.u          r13, r0, 0x40
        0x101e8 _main+28 ld            r12, r13, 0x6850
        0x101ec _main+32 mak           r12, r12, 0<2>
        0x101f0 _main+36 ld            r11, r30, 0x14
        0x101f4 _main+40 addu          r12, r12, r11
        0x101f8 _main+44 or.u          r2, r0, 0x1
        0x101fc _main+48 or            r2, r2, 0x1b0
        0x10200 _main+52 ld            r3, r0, r12
        0x10204 _main+56 bsr           _printf
    13  0x10208 _main+60 or            r2, r0, r0
        0x1020c _main+64 br            _main+68
(debug)
```

To display the instructions associated with line 13, use the **line** argument:

```
(debug) disassem, line 13 ↵
    13  0x10208 _main+60 or            r2, r0, r0
        0x1020c _main+64 br           _main+68
(debug)
```

To display the instructions associated with two lines of source code, beginning at program counter 0x000101e4, use the **pc** argument:

```
(debug) disassemble 2, pc 0x000101e4 ↵
*    12 0x101e4 _main+24 or.u          r13, r0, 0x40
        0x101e8 _main+28 ld            r12, r13, 0x6850
        0x101ec _main+32 mak           r12, r12, 0<2>
        0x101f0 _main+36 ld            r11, r30, 0x14
        0x101f4 _main+40 addu          r12, r12, r11
        0x101f8 _main+44 or.u          r2, r0, 0x1
        0x101fc _main+48 or            r2, r2, 0x1b0
        0x10200 _main+52 ld            r3, r0, r12
        0x10204 _main+56 bsr           _printf
    13  0x10208 _main+60 or            r2, r0, r0
        0x1020c _main+64 br            _main+68
(debug)
```

Use the **instructions** argument to display the current instruction:

```
(debug) disassemble, instructions ↵
*    12 0x101e4 _main+24 or.u          r13, r0, 0x40
(debug)
```

To display 6 instructions beginning at program label abc, use the **instruction** and **label** arguments:

```
(debug) disassemble 6, in, label abc ↵
```

Table 11-1 shows a summary of instructions that could appear after you issue the **disassemble** or **instruction-view** commands.

Table 11-1 88100 Instruction Summary

Name	Meaning
add	Add
addu	Add unsigned
and	Logical AND
bb0	Branch on bit clear
bb1	Branch on bit set
bcnd	Conditional branch
br	Unconditional branch
bsr	Branch to subroutine
clr	Clear bit field
cmp	Compare
div	Divide
divu	Divide unsigned
ext	Extract signed bit field
extu	Extract unsigned bit field
fadd	Floating-point add
fcmp	Floating-point compare
fdiv	Floating-point divide
ff0	Find first bit clear
ff1	Find first bit set
fldcr	Load from floating-point control register
flt	Convert integer to floating-point
fmul	Floating-point multiply
fstcr	Store to floating-point control register
fsub	Floating-point subtract
fxcr	Exchange floating-point control register
int	Round floating-point to integer
jmp	Unconditional jump
jsr	Jump to subroutine
ld	Load register from memory
lda	Load address
ldcr	Load from control register
mak	Make bit field
mask	Logical mask intermediate
mul	Multiply
nint	Round floating-point to nearest integer
or	Logical OR
rot	Rotate register
rte	Return from exception
set	Set bit field

(continued)

Table 11-1 88100 Instruction Summary

Name	Meaning
st	Store register to memory
stcr	Store to control register
sub	Subtract
subu	Subtract unsigned
tb0	Trap on bit clear
tb1	Trap on bit set
tbnd	Trap on bounds check
tcnd	Conditional trap
trnc	Truncate floating-point to integer
xcr	Exchange control register
xmem	Exchange register with memory
xor	Logical exclusive OR

(concluded)

Dumping the Contents of Memory (dump-memory)

To display the contents of memory in a specified format, use the **dump-memory** command. Following are the valid types:

Table 11-2 Valid Types for the dump-memory command

Type	Meaning
d	A short word in decimal
D	A long word in decimal
o	A short word in octal
O	A long word in octal
x	A short word in hexadecimal
X	A long word in hexadecimal
b	Two bytes in octal
c	Two bytes as characters
s	A character string terminated by a null byte
f	A single precision real number
g	A double precision real number

The type value is initially set to X. If you specify the **type** argument, subsequent invocations of **dump-memory** default to the last used type value.

This example displays in hexadecimal type the word whose address is 0x406850:

```
(debug) dump-memory 0x406850 ↵
0x406850:        0x00000005
(debug)
```

To display the value of the external integer i (which has the decimal value 5) in octal type, use the **type** argument:

```
(debug) dump _i, type O ↵
0x406850:        000000000005
(debug)
```

Note that the variable i is prefixed with an underscore; this is how COFF external symbols are referenced in the AViiON computer system architecture. ELF symbols are not prefixed this way.

To display the value of i in decimal type, use type D:

```
(debug) dump _i, type D ↵
0x406850:        5
(debug)
```

To display six 16-bit words beginning at i's starting address, use the **count** argument (**dump-memory** uses the last display mode):

```
(debug) dump _i, co 6 ↵
0x406850:        5 0 0 0
0x406860:        0 0
(debug)
```

To display six octal bytes beginning at i's starting address, use the **type** and **count** arguments:

```
(debug) dump _i, type b, co 3 ↵
0x406850:        \000 \000 \000 \005 \000 \000
(debug)
```

Machine-level Debugging on AViiON Computer Systems

This section discusses AViiON computer system-specific information.

88100 Pointers

Pointers are 32 bits in length and denote a byte address in the process address space. If a pointer is used in an instruction context (such as a routine return address stored on the stack), the two lower order bits are reserved for exception and validity indicators; see Figure 11-1.

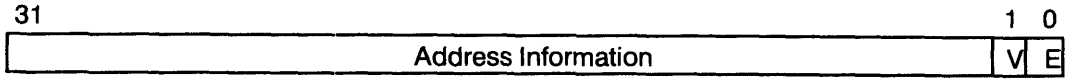


Figure 11-1 88100 Instruction Pointer Format

Registers

Table 11-3 shows the various registers. These special variables can be used in any expression context, unless they are hidden by a declaration in the program being debugged.

Table 11-3 Meaning of Registers Displayed by the machine-state Command

Name	Meaning
$r0	Always equal to zero
$r1	Holds the subroutine return pointer
$r2–$r9	Temporary register set used for parameter passing
$r10–$r13	Temporary registers used for language-specific purposes
$r14–$r25	Preserved registers
$r26 and $r27	Reserved for future use
$r28 and $29	Reserved for the compilation system
$r30	Preserved register
$r31	Contains the stack pointer
$fpsr	Floating-point user status register
$fpcr	Floating-point user control register
$psr	Processor status register
$sxip	Shadow execute instruction pointer
$snip	Shadow next instruction pointer
$sfip	Shadow fetched instruction pointer
$cfa	Canonical frame address pseudo-register
$pc	Program counter pseudo-register

AViiON Computer Systems and Registers

Note that the $pc and $cfa registers are software phenomena and are not defined by AViiON computer systems except by convention. The "$cfa" register performs the function traditionally done by a frame pointer register on other architectures.

When you are positioned in the top stack frame, the "$pc" register denotes the stop position of the debugger. For native processes, this pseudo-register displays either the $sxip or $snip value, depending on the reason the target process last stopped. If it stopped for a breakpoint, single-step, when initially loading a process, or when an imprecise exception has occurred, the $pc register takes on the value of the $snip register. In all other cases, the $pc register takes on the value of the $sxip register.

When you assign a value to the $pc pseudo-register, the $snip and $sfip registers are reset so that all instructions for further execution will be reset correctly. The value placed in the $snip register is the top 30 bits of the specified location, with an OR performed with the constant 2 and the value (the OR operation turns the validity bit on and the exception bit off for the instruction pointer). The $sfip register value is the standardized $snip register value plus 4 (the size of an instruction on AViiON computer systems).

When the $pc register has the $sxip value, attempting to set the $pc's value with an assign statement such as "assign $pc $pc+4" will change the $snip and $sfip registers only. All references to the "$pc" register will reference the $sxip value, which never changes. The $sxip value has nothing to do with the continuation program counter location.

In frames other than the top frame, the "$pc" register value is the return address for its parent frame.

On AViiON computer systems, you can use the alias $fp for the register $r30 and the alias $sp for the register $31. These aliases are not displayed when you use the **machine-state** command, but are available if you access register values by name, such as with the **evaluate** command.

Register variables are used just like any other variables visible in your program, except that register names are case sensitive. To print a register's value, use the **evaluate** command with the name displayed in the "Name" column of Table 11-3. To modify its value, use the **assign** command:

```
(debug) evaluate $pc, format hex ↵
0x000101e4
(debug) assign $pc 0x000101e8, format hex ↵
(debug)
```

If the current expression evaluation language converts identifiers to uppercase, you must explicitly change the language to one that is case sensitive (such as C) to manipulate the registers.

End of Chapter

attach

Debugger Command

Debugs an already executing process.

Summary

Debugs an already executing process.

Syntax

attach *pid* **,executable**

where:

pid	A process identification number
executable	The name of an executable file

Examples

```
at 432, ex foo
```

Description

The **attach** command directs Mxdb to debug an already executing process, possibly giving the location of the associated executable file.

Arguments

pid	Specify the pid of the process to be debugged
executable	Specify the name of the executable file associated with the process to be debugged; the default is **a.out**.

Examples

To debug an executing process with a PID of 432 and the associated filename **foo**, type this command:

```
(debug) at 432, ex foo ↵
```

See Also

Command: **detach**

breakpoint

Debugger Command

Sets a breakpoint at a specified location.

Summary

Sets a breakpoint at a specified location.

Syntax

breakpoint *[position]* **,line ,label ,pc ,scope ,disable ,name ,count ,if ,action**

where:

position	A file, scope, line number, or some combination of the three
line	Line number, or CURRENT or LAST with an optional offset
label	A program label
pc	An address for a program counter
scope	The name of a module or routine
disable	Create a breakpoint and then disable it; yes or no
name	A name to be associated with a breakpoint
count	How many times a location can be reached without a breakpoint occurring
if	An expression that evaluates to true or false
action	A sequence of commands; must be in braces if more than one command is specified

Examples

```
br
break 50
b main
b \foo\bar
b \for:29
break, name rope, scope rou2
breakpoint, pc main+02
breakpoint 23, count 4, if (! i)
br, lab label0, act {wri Back to top-level}
event-status ,,list breakpoints and other events
delete-event 1 2 ,,delete breakpoints named "1" and "2"
br Array::~Array
```

Description

The **breakpoint** command directs the debugger to halt execution of the target process when the specified location is reached and any auxiliary properties are satisfied, and return control to you.

A breakpoint can be set at a position, a line or programming-language label in any specified scope, or just on a scope, or on any valid address (specified symbolically or numerically). A breakpoint can be set on a scope, even if the scope has not been compiled for debugging, if the scope is an external routine name. (This helps when you want to stop in library routines.)

You will receive a warning if an address is not in the text area and an error message if the address is not aligned properly.

You can set multiple breakpoints on a line. One or both of the following messages will be printed to the error output:

```
Note: The following breakpoints have also been set on this
      location: "<breakpoint-name1>" "<breakpoint-name2>" ...
Note: The following DISABLED breakpoints have also been set on
      this location: "<breakpoint-name1>" ...
```

To see a list of currently set breakpoints, use the **event-status** command. To delete a breakpoint, use the **delete-events** command.

Arguments

position

If you specify a value, a breakpoint is set at that position. If you omit the **position** argument and the debugger is at the beginning of a line, the breakpoint is set at the beginning of the line. If you omit the **position** argument and the debugger is not at the beginning of a line, or if no debugging information is present, the breakpoint is set at the current program counter.

You can use the **position** argument in conjunction with the **scope** argument. However, only the line number information from the **position** argument is applied toward the construction of the final position.

line

Create a breakpoint at the beginning of the specified line.

The line number symbolic tags "current" and "last," or their abbreviations, may cause ambiguity if a routine (scope) exists with the same name or abbreviation. If an argument is an alphanumeric character sequence, it will be looked up as a scope (routine or module) first. If this lookup fails, the argument will be processed as a line number. To prevent any ambiguity, supply a leading colon (:) when you use a symbolic tag; the position-type will then process the characters strictly as a line number.

label

Create a breakpoint at the specified program label.

pc	Create a breakpoint at the specified program counter address. You can use a symbolic or numeric expression.
scope	Set a breakpoint at the beginning of the specified scope, or at a specified line or label in that scope.
disable	First create a breakpoint to make sure that it is valid, then disable it. This action retains the breakpoint in the debugger while avoiding the overhead from creating the event in the debugged process.
name	Associate the specified name (character string) with the breakpoint. If you omit the **name** argument, **breakpoint** creates a name. This name is displayed when the breakpoint is taken or when you issue an **event-status** command. The **delete-events** command also uses the name.
count	After the debugger reaches a specified location this many times, a breakpoint will occur the next time the location is reached. The initial default is null (0).
if	The program language at the position where the breakpoint is set defines the syntax of the expression. If the expression evaluates to true, the debugger stops the process and performs the specified action.
action	Execute the specified command(s) when the breakpoint is encountered, the count is 0, and the if predicate, if present, is true. The sequence of commands must be in braces if you specify more than one command. Also, if you specify more than one command, the commands must be separated by semicolons or New Line characters.

Examples

To set a breakpoint at the current debugger position:

```
(debug) br ↵
```

To stop at line 50 of the current module:

```
(debug) break 50 ↵
```

To stop at the local or global routine named "main":

```
(debug) b main ↵
```

To stop at routine "bar" in module "foo":

```
(debug) b \foo\bar ↵
```

To stop at line 29 in module "for":

```
(debug) b \for:29 ↵
```

To stop at line 23 of the module containing routine **bar**, use either of these commands:

```
(debug) breakpoint 23, s bar ↵
(debug) breakpoint bar:23 ↵
```

To create a breakpoint named 'rope' that stops the process at the beginning of routine **rou2**, use either of these commands:

```
(debug) breakpoint, name rope, scope rou2 ↵
(debug) br rou2, name rope ↵
```

To stop at routine **rou2** in module **mod**:

```
(debug) breakpoint, scope \mod\rou2 ↵
```

To stop at line 23 if an expression is true:

```
(debug) breakpoint 23, if (i==j<<2) ↵
```

To stop at line 23 after reaching a location four times if an expression is true:

```
(debug) breakpoint 23, count 4, if (! i) ↵
```

To stop at label 'label0' and print a message:

```
(debug) br, lab label0, act {wri Back to top-level} ↵
```

To stop at the destructor member function of the C++ "Array" class:

```
(debug) br Array::~Array ↵
```

See Also

Commands:	**continue, delete-events, disable-events, enable-events, event-status, signal, watch-memory, watch-reference**
Topics:	events, scopes
Type:	position

continue *Debugger Command*

Continues the current process being debugged.

Summary

Continues the current process being debugged.

Syntax

continue *[line]* **,label ,pc ,signal**

where:

line	Line number, or CURRENT or LAST with an optional offset
label	A programming language label
pc	An address for a program counter
signal	The number or name of a signal (0 through 64, or null, hup, int, quit, ill, trap, abrt, emt, fpe, kill, bus, segv, sys, pipe, alrm, term, usr1, usr2, cld, pwr, winch, poll, stop, tstp, cont, ttin, ttou, urg, io, xcpu, xfsz, vtalrm, prof, or lost)

Examples

```
c
co 23 ,,redirect execution to line 23
cont, sig int
continue, signal sigint
continue, pc _main+240
```

Description

The **continue** command resumes execution of the current process, optionally supplying the signal with which to continue the process. If no signal is supplied and the debugger has intercepted a signal destined for the target process, then that signal is used to continue the process.

The **line** and **pc** arguments redirect execution in the top frame; that is, the debugger resets the program counter in the top frame. Nothing else in the stack changes.

In **event-status** command actions, the **continue** command takes effect at the end of the **action** argument's command-sequence. Multiple **continue** commands have the same effect as one **continue** command, and the last redirection will take effect.

As with the other execution-continuation commands (including **finish** and **step**), the **option:stop-commands** command sequence is executed when the **continue** command completes.

describe *Debugger Command*

Prints a declarative description of an expression.

Summary

Prints a declarative description of an expression.

Syntax

describe *expression* **,meaning-kind**

where:

expression	A name or an expression
meaning-kind	A particular kind of program entity: constant, enumeration, external, field, variable, label, routine, scope, strange, or type

Examples

```
des i
desc sa
descr &i
describe &sa
describe (sa[0]+1)&0377
describe c, meaning type
describe A
```

Description

The **describe** command shows the definition of any program entity that is visible from the current debugger position. The declarative description is in the syntax of the current expression language. The debugger sets the current expression language to the language in which the code is written where possible (some languages may be mapped to other language describers). You can reset the current language with the **c-p:option-status** command.

The name-resolution topic discusses the order in which program- and debugger-defined names are searched. See the individual language topics (c-language, c++-language, fortran-language, and pascal-language) for details on particular language implementations.

Arguments

expression	If you specify a name, the entity is displayed in the declarative syntax of the current expression language. If you specify an expression in C, the result is displayed as a type-cast if possible. If you specify an expression in FORTRAN, the result is always displayed as an anonymous declaration.

meaning-kind Specify the kind of entity of the name to be described if multiple uses of a name exist. This situation can occur in C where the same name exists in different name spaces. With debugger and convenience variables, built-in type names, and register names, **meaning-kind** does not apply during name resolution. This argument only works with a simple name, not a language-specific expression.

Examples

To describe the local integer variable i:

```
(debug) des i ↵
auto int i;
```

To describe the array sa, which is a static array of five short integers:

```
(debug) desc sa ↵
static short sa[5];
```

To describe the address of i:

```
(debug) descr &i ↵
( int * )
```

To describe the address of sa:

```
(debug) describe &sa ↵
( short (*)[5] )
```

To describe an expression:

```
(debug) describe (sa[0]+1)&0377 ↵
( int )
```

To describe a type c:

```
(debug) describe c, meaning type ↵
typedef unsigned int c;
```

To describe a routine (C descriptions follow ANSI–C format):

```
(debug) des main ↵
int main (
    int arg,
    char **argv,
    char **envp);
```

To describe a C++ class:

```
(debug) describe A ↵
class A {
public:
   int foo;
   A (
      int arg);
   A ();
   operator int ();
   int operator+ (
      int arg);
   ~A ();
};
```

See Also

Commands:	**address, assign, evaluate, names, c-p:option-status**
Topic:	name-resolution
Type:	expression

This page intentionally left blank.

detach *Debugger Command*

Stops debugging a process without terminating it.

Summary

Stops debugging a process without terminating it.

Syntax

detach

Examples

```
detach
```

Description

The **detach** command allows a live process that was being debugged to continue execution without further debugging. If the process was started as a child of the debugger, the process will be inherited by init as usual when the debugger terminates. If necessary, the **attach** command can be used to start debugging of the process again.

Note that since a core file is not an active target process, Mxdb will produce an error message if you try to detach from debugging a core image.

Arguments

None

Examples

Here is an example of the **detach** command:

```
(debug) detach ↵
```

See Also

Command: **attach**

directory-list *Debugger Command*

Sets or displays the directory source search path list.

Summary

Sets or displays the directory source search path list.

Syntax

directory-list *[dirs]*

where:

dirs	A sequence of directories

Examples

```
dir
direct b c
directory-list a 'directory d
```

Description

The **directory-list** command sets or displays the directory source search path list. Since the current directory-list can be captured (by using a backquote), you can easily prefix and append directories to the list (as shown in the third example below). More complicated manipulations are possible (see the "See Also" section below).

If you have two or more files with the same name located in separate directories, the debugger may find a source in a directory other than the one you intended. In this case, check the source search path and change it with the **directory-list** command.

Both source files and external legends are located via this mechanism.

Arguments

dirs	Set the source search path list to the specified sequence of directories.

Examples

To display the current directory list:

```
(debug) dir ↵
. foodir bardir
```

To set the directory list to directories **b** and **c**:

```
(debug) direct b c ↵
(debug) direct ↵
b c
```

To add directories **a** and **d** to the list:

```
(debug) directory-list a 'directory d ↵
(debug) directory-list ↵
a b c d
```

See Also

Commands: **c−p:first, c-p:last, c−p:rest**

disable-events *Debugger Command*

Disable one or more currently set events.

Summary

Disable one or more currently set events.

Syntax

disable-events *names* **,all**

where:

names	The name of an event.
all	Disable all currently set events; yes or no.

Examples

```
disable 1 2 3
disable, all
```

Description

The **disable-events** command disables one or more currently set events. Disabling an event causes it to be remembered in the debugger, but it won't affect the debugged process, which can be important for time-critical applications.

For example, to avoid taking a breakpoint, you can place a large count number on it. However, placing a count number on a breakpoint still causes the debugged process to stop so that the debugger can decrement the count and then continue the process. Disabling the same breakpoint avoids all of this overhead.

Disabling an already disabled event has no effect.

Arguments

name	The name of an event.
all	Disable all currently set events; yes or no. The words "yes" and "no" can be abbreviated.

Examples

To disable currently set events named "1," "2," and "3":

```
(debug) disable 1 2 3 ↵
```

To disable all currently set events:

```
(debug) disable, all ↵
```

See Also

Commands: **enable-events, event-status**

disassemble *Debugger Command*

Displays machine instructions in symbolic form.

Summary

Displays machine instructions in symbolic form.

Syntax

disassemble *[count] [position]* **,line ,label ,scope ,pc ,instructions**

where:

count	Number of lines or instructions to be displayed
position	A file, scope, line number, or some combination of the three
line	A source text line number
label	A program label
scope	The name of a module or routine
pc	An address for a program counter
instructions	Use instructions instead of source lines as units; yes or no

Examples

```
disas
disas 2
disassem, line 19
disassemble 2, pc func
disassemble, instructions
disassemble 6, in, label abc
```

Description

The **disassemble** command displays machine instructions in symbolic form. If you do not supply an explicit position, the current debugger position is used.

This command displays instructions in the same format as the **instruction-view** command. **Instruction-view** uses the source-lines option and prints a window of instructions around the current or specified position.

Arguments

count	Display the specified number of lines or instructions (see the **instructions** argument). The default is to display all the instructions associated with the current line, beginning at the program counter associated with the current debugger position.
position	Examine the location indicated by the specified position and display all the associated instructions.

line	Examine the location indicated by the specified line number and display all the associated instructions.
label	Examine the location indicated by the specified programming language label.
scope	Examine the location indicated by the specified module or routine name.
pc	Examine the location indicated by the specified program counter (or address). The current pc is marked by an asterisk (*) at the beginning of its associated instruction line.
instructions	Use instructions instead of source lines as units for the **count** argument. Initially, the unit is lines.

Examples

To display all the instructions associated with the current line, beginning at the pc associated with the current debugger position:

```
(debug) disas ↵
```

To display all instructions associated with the current line even when the pc is not at the beginning of the line:

```
(debug) disas, line c ↵
```

To display the instructions associated with two lines beginning at the current debugger position:

```
(debug) disas 2 ↵
```

To display the instructions associated with line 19:

```
(debug) disassem, line 19 ↵
```

To display the instructions associated with two lines of source code, beginning at program counter func:

```
(debug) disassemble 2, pc func ↵
```

To display the current instruction:

```
(debug) disassemble, instructions ↵
```

To display 6 instructions beginning at program label abc:

```
(debug) disassemble 6, in, label abc ↵
```

See Also

Commands: **dump-memory, instruction–view, machine-state**
Type: position

dump-memory

Debugger Command

Dumps memory in the specified type.

Summary

Dumps memory in the specified type.

Syntax

dump-memory *[start-address]* **,end-address ,count ,type**

where:

start-address	A byte address in memory (a symbolic or numeric specification)
end-address	A byte address in memory
count	The number of units of memory to dump
type	d, D, o, O, x, X, b, c, s, f, or g

Examples

```
du i, type O
dump i, type D
dump i, co 6
dump-memory i, type b, co 3
```

Description

The **dump-memory** command dumps memory to the standard output in the specified type.

Arguments

start-address	Start the memory dump at the specified address. If you omit this argument, the dump starts at the next address after the last address of the previous dump. If the **dump-memory** command has not yet been invoked and you omit all arguments, an error occurs.
end-address	End the memory dump at the specified address. If you specify this argument, you must omit the **count** argument.
count	Dump the specified number of units. Units are determined by the type. If you specify the **count** argument, you must omit the **end-address** argument.

type

Display in one of the following types, using C standard format (hexadecimal numbers are prefixed by "0x," octal numbers are prefixed by "0," and non-graphic characters are printed as a backslash followed by three octal digits):

d	A short word in decimal
D	A long word in decimal
o	A short word in octal
O	A long word in octal
x	A short word in hexadecimal
X	A long word in hexadecimal
b	Two bytes in octal
c	Two bytes as characters
s	A character string terminated by a null byte
f	A single precision real number
g	A double precision real number

The type value is initially set to X. If you specify the **type** argument, subsequent invocations of dump-memory default to the last used type value.

Examples

To display the value of variable i in octal type:

```
(debug) du _i, type O ↵
0x406890:       000000000005
```

To display the value of i in decimal type:

```
(debug) dump _i, type D ↵
0x406890:       5
```

To display six 16-bit words beginning at i's starting address, use the **count** argument:

```
(debug) dump _i, co 6 ↵
0x406890:       5 0 0 0
0x4068a0:       0 0
```

To display six octal bytes beginning at i's starting address, use the **type** and **count** arguments:

```
(debug) dump-memory _i, type b, co 3 ↵
0x406890:       \000 \000 \000 \005 \000 \000
```

See Also

Commands: **disassemble, instruction–view, machine-state**

edit

Debugger Command

Invokes an editor.

Summary

Invokes an editor at the current position or the supplied position.

Syntax

edit *[position]* **,editor−program**

where:

position	A file, scope, line number, or some combination of the three
editor−program	Any syntactically valid system pathname

Examples

```
edit
edit main
edit foo.c
edit \bar\stat
edit foo.c, editor emacs
```

Description

The **edit** command invokes an editor at the current position or the supplied position.

If the graphical user interface is active, a terminal emulator window (Mxdb Edit) is created to handle the edit session; the debugger session continues without pending for edit completion. Otherwise, if the graphical user interface is not active, this command pends the debugger until you exit from the editor.

Note that the **windowed−terminal−emulator** option value, if defined, is used when the graphical user interface is active. If it is not set, **/usr/bin/X11/xterm** is used as the terminal emulator program.

Arguments

position	Select a position other than the current position. The standard editor line positioning directive, *+<line−number>*, is used to position the editor to a specific line. This is followed by an absolute pathname to the source−file located via the debugger's currently set directory−list.
editor−program	Specify the pathname of an editor program that you want to use. If this keyword argument is not supplied, the value of the EDITOR environment variable is used. When the EDITOR environment variable is not set, the pathname **/bin/vi** is used.

Examples

To invoke an editor at the current position:

```
(debug) edit ↵
```

To edit the **main** routine:

```
(debug) edit main ↵
```

To edit the file **foo.c**:

```
(debug) edit foo.c ↵
```

To edit the static routine **stat** in **bar**:

```
(debug) edit \bar\stat ↵
```

To edit the file **foo.c** with the emacs editor:

```
(debug) edit foo.c, editor emacs ↵
```

See Also

Commands: **debugger--toolkit:position-source-file, debugger--toolkit:position-line, debugger-toolkit:resolve-filename options:windowed-terminal-emulator**

enable–events

Debugger Command

Enable one or more currently set events.

Summary

Enable one or more currently set events.

Syntax

enable-events *names* **,all**

where:

names	The name of an event.
all	Enable all currently set events; yes or no.

Examples

```
enable 1 2 3
enable, all
```

Description

The **enable-events** command enables one or more currently set events. Enabling a disabled event causes it to affect the execution of the debugged process again.

Enabling an already enabled event has no effect.

Arguments

name	The name of an event.
all	Enable all currently set events; yes or no. The words "yes" and "no" can be abbreviated.

Examples

To enable currently set events named "1," "2," and "3":

```
(debug) enable 1 2 3 ↵
```

To enable all currently set events:

```
(debug) enable, all ↵
```

See Also

Commands: **disable-events, event-status**

evaluate

Debugger Command

Evaluates a language expression.

Summary

Evaluates a language expression.

Syntax

evaluate *expression* **,format ,length ,array**

where:

expression	A language expression
format	A format: ascii, binary, decimal, hexadecimal, ieee-double, ieee-float, ieee-single, octal, string, symbolic, system-error, or unsigned-decimal
length	The number of bits to be displayed
array	The length of an artificial array

Examples

```
e c
eval c, format string
evaluate c[0]+1, length 10
eval class_obj.length()
```

Description

The **evaluate** command evaluates an expression using the current programming language syntax. By default this is the language in which the source text is written.

You can enhance the display of evaluation results by using the **elide-arrays**, **pointer-dereference-level**, and **string-display options**. See the documentation in the options realm.

The name-resolution topic discusses the order in which program- and debugger-defined names are searched. See the individual language topics (c-language, c++-language, fortran-language, and pascal-language) for details on particular language implementations.

Arguments

expression	The expression must be valid in the current language.
format	Specify a nondefault display format.
length	Modify the number of bits that are interpreted for display. This value may be ignored if it exceeds the length of the expression's value. If the expression is a register name and the length is greater than the register size, you will receive an error.
array	Specifies the length of an artificial array. An artificial array displays data at an expression's address as if it were an array of the expression's type. Note that not all languages support this keyword. This value is only valid when the result of evaluating the **expression** argument is a reference. Also, the **format** and **length** arguments are invalid when the **array** argument is used. For objects that do not reside in your address space, such as debugger and convenience variables, the resulting artificial array cannot extend beyond the length of the object.

Examples

Evaluate the character array c, which has two members:

```
(debug) evaluate c ,, string-display is "no" ↵
{
   [0]        = 'a'
   [1]        = '\000'
}
```

Evaluate c as a string:

```
(debug)  evaluate c, format string ↵
"a"
```

To display the first 10 bits of c in binary format:

```
(debug) evaluate c, format binary, length 10 ↵
0b0110000100
```

To display the integer result of an expression:

```
(debug) evaluate c[0]+1 ↵
98
```

Here is a structure, i2:

```
struct {
  int i;
  int j;
  int k;
} i2;
```

With the **array** argument, you can display the first three elements of artificial arrays i2 and i2.i:

```
(debug) eval i2, ar 3 ↵
{
   [0]        = {
      i           = 99
      j           = 100
      k           = 101
   }
   [1]        = {
      i           = 0
      j           = 0
      k           = 0
   }
   [2]        = {
      i           = 66064
      j           = 0
      k           = 0
   }
}

(debug) eval i2.i, ar 3 ↵
[0]     = 99
[1]     = 100
[2]     = 200
```

In the first example above, the value of i in artificial array element i2[2] is 66064. This illustrates that you can look at arbitrary memory locations as an array.

The next examples show the **elide-arrays**, **string-display**, and **string-display-limit** options. The C character array c_array contains six elements ("hello" and a NUL byte).

```
(debug) opt string--display ↵
yes
(debug) opt elide-arrays ↵
yes
(debug) eval c_array ↵
"hello"
(debug) opt string-display no ↵
(debug) eval c_array ,, shows elision ↵
{
   [0]        = 'h'
   [1]        = 'e'
```

```
   [2..3]     = 'l'
   [4]        = 'o'
   [5]        = '\000'
}

(debug) opt elide-arrays no ↵
(debug) eval c_array ↵
{
   [0]        = 'h'
   [1]        = 'e'
   [2]        = 'l'
   [3]        = 'l'
   [4]        = 'o'
   [5]        = '\000'
}
(debug) opt string-display y ↵
(debug) opt string-display-limit 3 ↵
(debug) eval c_array ↵
"hel"...
(debug)
```

This example shows how the **pointer-dereference-level** option affects the **evaluate** command:

```
(debug) desc a_struct_type ↵
struct a_struct_type
{
   int* intp;
   int** intpp;
};
(debug) desc a_ptr ↵
extern struct a_struct_type *a_ptr;

(debug) opt p-d 0 ↵
(debug) eval a_ptr ↵
0x004071f0

(debug) opt p-d 1 ↵
(debug) eval a_ptr ↵
(struct a_struct_type *) 0x004071f0 ->
   {
      intp        = 0x00404498
      intpp       = 0x004044f4
   }

(debug) opt p-d 2 ↵
(debug) eval a_ptr ↵
(struct a_struct_type *) 0x004071f0 ->
   {
      intp        = (int *) 0x00404498 -> 123
      intpp       = (int **) 0x004044f4
         0x004071f0
   }
```

```
(debug) opt point 3 ↵
(debug) eval a_ptr ↵
(struct a_struct_type *) 0x004071f0 ->
   {
      intp        = (int *) 0x00404498 -> 123
      intpp       = (int **) 0x004044f4
         (int *) 0x004071f0 -> <Previously Displayed >
   }
```

In the last example, the value of ****a_ptr->intpp** is not printed because this field points to the **a_ptr->intp**, which has already been printed.

If the value of the **convenience-variables** option is "yes," a debugger variable is created whenever you evaluate an expression (in this case, an integer with a value of 123) with the **evaluate** command:

```
(debug) opt conv-var yes ↵
(debug) eval an_int ↵
$1     = 123
(debug) eval $1 ↵
$2     = 123
```

To evaluate a member function of a C++ class object "class_obj":

```
(debug) eval class_obj.length() ↵
24
```

See Also

Commands: **address, assign, convenience-variables, describe, c-p:option-status, machine-state, names, variable, c-p:evaluate, options:convenience-variables, options:elide-arrays, options:pointer-dereference-level, options:string-display, options:string-display-limit**

Topics: c-language, c++-language, fortran-language, name-resolution, pascal-language

event-status

Debugger Command

Sets or displays event information.

Summary

Sets or displays event information.

Syntax

event-status *[name]* **,count ,if ,action ,disable ,delete-if**

where:

name	The name of a breakpoint, signal, or watchpoint event
count	A repetition factor
if	An expression that evaluates to true or false
action	A sequence of commands; must be in braces if more than one command is specified
disable	Disable an event; yes or no
delete-if	Make an event unconditional; yes or no

Examples

```
e-s
event-s jig, if ( n > 9 ), action process-status
event-status 1, count 2
e-s 1, disable
```

Description

The **event-status** command displays or modifies information about breakpoints, signals, or watchpoints (defined with the **watch-memory** and **watch-reference** commands) that are set in the current process. The information is printed in a form that is syntactically valid for setting the same events at a later time. Therefore, this command can be redirected to a file and included at a later time in order to reset the events.

Any specified predicate is evaluated as an expression in the language associated with the position where the debugged process stops. For multi-language applications, use the correct syntax for the expected position of the event. For asynchronous events (such as signals), exercise care when you choose what variables are referenced and what language syntax is used.

If a predicate evaluation fails for any reason, the predicate value is assumed to be true, so the event will be taken.

Disabling an event allows it to be saved in the debugger while avoiding the overhead from creating the event in the debugged process.

option-status *Command Processor Command*

Displays or sets an option's value.

Summary

Displays or sets an option's value.

Syntax

option-status *[option-settings]* **,prompt**

where:

option-settings	Any value
prompt	Invoke the prompting facility; yes or no

Examples

```
option-status pager 66
op lang
op
```

Description

The **option-status** command lets you manage the values of options that control the behavior of certain CP-, application-, and user-defined commands. These options are treated as options realm commands that accept exactly one optional argument.

To display the current global options and their values, type **option-status** without an **option-settings** argument.

To set an option's status, use the **option-settings** argument.

If you want to add an option to the options that the **option-status** command manages, create a macro in the options realm that accepts exactly one optional argument (of any type). If you issue the macro's name without supplying the optional value, the macro should print its current value to the standard output. Otherwise it should silently update its current value to the specified value. Once the user-defined command (macro) is added to the options realm, the **option-status** command dynamically and seamlessly manages the new command along with its builtin counterparts. See the Examples section for a sample macro.

Arguments

option-settings

This argument accepts one, two, or a list (enclosed in curly-braces) of tokens. When you supply one token, it must be the name of an options realm command; the command's value will be printed.

If you specify two tokens, the first name must be an options realm command and the second token is the command's new value; if the value is not valid, you will receive an error.

If you supply three or more tokens, they must be paired name–value bindings enclosed in curly-braces.

prompt

Specify this argument to invoke the prompting facility. This facility will prompt for each option using the standard prompting mechanism. See the prompting topic for more information.

Examples

To display all currently set option values:

```
(c-p) op ↵
option-status {
  Pager_Lines                    23,
  Source_Lines                   10,
  Stop_Commands                  ,
  Language                       c,
  Elide_Arrays                   yes,
  String_Display                 yes,
  String_Display_Limit           100,
  Pointer_Dereference_Level      0,
  Convenience_Variables          no,
  Convenience_Variables_Limit    50,
  Mismatched_Legends_Allowed     no,
  Bit_Format                     binary,
  Character_Format               ascii,
  Signed_Character_Format        ascii,
  Unsigned_Character_Format      ascii,
  Floating_Point_Format          ieee-float,
  Signed_Integer_Format          decimal,
  Unsigned_Integer_Format        unsigned-decimal,
  Unknown_Type_Format            hexadecimal,
  Command_History                0,
  Message_History                0,
  Windowed_Terminal_Emulator
  }
(c-p)
```

To set the number of lines used by the pager to 66:

```
(c-p) option-status pager 66 ↵
```

To display the current option value for the expression evaluation language:

```
(c-p) op lang ↵
```

To create a windowed terminal emulator when a live process is debugged, even when the *graphical user interface is not being* used:

```
(c-p) op win xterm ↵
```

To set the number of source lines and set the signed integer format to hexadecimal:

```
(c-p) op {source 15, unsigned_integer_format hex} ↵
```

In this example, command prompting is invoked; since the prompting session is aborted, none of the options are actually changed:

```
(c-p) opt {language fortran}, prompt ↵
          Pager_Lines (23) =  20
          Source_Lines (15) =  ,abort
(c-p) opt lang; opt pager ↵
c
23
(c-p)
```

The next example shows how to you can create a customized command in the options realm:

```
(debug) c-p:assign options:my-vacation-location home „ I'm broke ↵
(debug) define-macro options:my-vacation {,optional location} { ↵
(debug){ c-p:if {location} {„ remember the new location ↵
(debug){{     c-p:assign my-vacation-location 'location ↵
(debug){{ }, else {„ Report the current vacation location ↵
(debug){{     my-vacation-location ↵
(debug){{ }}
```

Now if you use the **option-status** command with no options, the global options will be listed first, and then your user-customization command (**my-vacation**, which contains exactly one optional argument, **location**) in the options realm will be listed:

```
(debug) op ↵
option-status {
  Pager_Lines                    23,
  Source_Lines                   15,
  Stop_Commands                  ,
  .
  .
  .
  Command_History                0,
  Message_History                0,
  Windowed_Terminal_Emulator     ,
  my_vacation                    home
}
```

If a group of tired developers decided to go to Hawaii, they would give the optional argument **location** a new value:

```
(debug) op my-va hawaii „ I wish ↵
(debug) op ↵
option-status {
  Pager_Lines                    23,
  Source_Lines                   15,
  Stop_Commands                  ,
  Language                       c,
  Elide_Arrays                   yes,
  String_Display                 yes,
  String_Display_Limit           100,
  Pointer_Dereference_Level      0,
  Convenience_Variables          no,
  Convenience_Variables_Limit    50,
  Mismatched_Legends_Allowed     no,
  Bit_Format                     binary,
  Character_Format               ascii,
  Signed_Character_Format        ascii,
  Unsigned_Character_Format      ascii,
  Floating_Point_Format          ieee-float,
  Signed_Integer_Format          decimal,
  Unsigned_Integer_Format        hexadecimal,
  Unknown_Type_Format            hexadecimal,
  Command_History                0,
  Message_History                0,
  Windowed_Terminal_Emulator     ,
  my_vacation                    hawaii
}
```

See Also

Topic: c-p:prompting

position *Debugger Command*

Displays or sets the current debugger position.

Summary

Displays or sets the current debugger position.

Syntax

position *[position]* **,line ,label ,pc ,scope ,frame**

where:

position	A file, scope, line number, or a combination of the three
line	Line number, or CURRENT or LAST with an optional offset
label	A program label
pc	An address for a program counter (pc)
scope	The name of a module or routine
frame	A stack frame specification (an integer, or BOTTOM or TOP with an optional offset)

Examples

```
p
p main
p \for\maxnum
p \for:29
pos 21
posi, label here
position, pc foo+24
position, fra top+1
position, scope \doer
pos Array::~Array
```

Description

The **position** command displays or sets the current debugger position. If you omit all arguments, the current debugger position is displayed. When debugging information is present for the associated address (pc), the display has the form:

frame *f,* line *L,* scope *\module\routine,* pc *address*

The *address* is numeric (hexadecimal).

If a group of tired developers decided to go to Hawaii, they would give the optional argument **location** a new value:

```
(debug) op my-va hawaii „ I wish ↵
(debug) op ↵
option-status {
  Pager_Lines                    23,
  Source_Lines                   15,
  Stop_Commands                  ,
  Language                       c,
  Elide_Arrays                   yes,
  String_Display                 yes,
  String_Display_Limit           100,
  Pointer_Dereference_Level      0,
  Convenience_Variables          no,
  Convenience_Variables_Limit    50,
  Mismatched_Legends_Allowed     no,
  Bit_Format                     binary,
  Character_Format               ascii,
  Signed_Character_Format        ascii,
  Unsigned_Character_Format      ascii,
  Floating_Point_Format          ieee-float,
  Signed_Integer_Format          decimal,
  Unsigned_Integer_Format        hexadecimal,
  Unknown_Type_Format            hexadecimal,
  Command_History                0,
  Message_History                0,
  Windowed_Terminal_Emulator     ,
  my_vacation                    hawaii
}
```

See Also

Topic: c-p:prompting

To change the debugger position to the local or global routine named "main":

```
(debug) p main ↵
```

To change the debugger position to the routine "maxnum" in module "for":

```
(debug) p \for\maxnum ↵
```

To change the debugger position to line 29 in module "for":

```
(debug) p \for:29 ↵
```

To change the debugger position to line 21 in the current module:

```
(debug) pos 21 ↵
```

To change the debugger position to program label 'here':

```
(debug) posi, label here ↵
```

To change the debugger position to a text address:

```
(debug) position, pc _foo+24 ↵
```

To change the debugger position to the next-to-top stack frame:

```
(debug) position, fra top+1 „ or frame top+1 ↵
```

To change the debugger position to module **doer**:

```
(debug) position, scope \doer „ or p \doer or file doer.f or file doer ↵
```

To change the debugger position to the static (locally-visible only) routine **donter** in **doer**:

```
(debug) po, s \doer\donter „ or po \doer\donter or routine \doer\donter ↵
```

To change the debugger position to the external routine **HANS_N_FRANZ**:

```
(debug) po HANS_N_FRANZ „ or po, s HANS_N_FRANZ ↵
(debug) „ or routine HANS_N_FRANZ ↵
```

To change the debugger position to a nested routine within module **snl** called **PUMP_YOU_UP**:

```
(debug) po, s \snl\HANS_N_FRANZ\PUMP_YOU_UP ↵
(debug) „ or po \snl\HANS_N_FRANZ\PUMP_YOU_UP ↵
```

To change the debugger position to the destructor member function of the C++ "Array" class:

```
(debug) pos Array::~Array ↵
```

See Also

Commands:	**file, find, frame, routine, view**
Types:	frame, line-number, scope, pc, position

process-status *Debugger Command*

Prints where and why the process last stopped.

Summary

Prints where and why the process last stopped.

Syntax

process-status

Examples

```
p-st
process-status
```

Description

The **process-status** command displays information about the process being debugged: where it stopped and what event(s) stopped it. Events that can stop a process include breakpoints, signals, stepping, initial loading, attaching, or exiting.

The **process-status** command is useful in combination with other commands as a value to an **action** argument on a **breakpoint**, **signal**, **step, watch-memory**, or **watch-reference** command. If you specify another command instead of **process-status**, the command with the **action** argument will not display the program position (the position where the program stopped). If you omit the **action** argument, the program position will be displayed.

The format of the output display is similar to that of the **position** and **walkback** commands:

Stopped at frame *f,* line *n,* scope *\mod\rou*, pc *addr*
why

f	A stack frame number
n	A source code line number
mod	The name of a program module
rou	The name of a program routine or routines
addr	A program counter's address, which is numeric if debugging information is available

why What caused the program to stop; any of the following:

breakpoint *"event-name"*

watch-memory *"event-name"*

watch-reference *"event-name"*

caught signal *n,* SIG*tag, signal-description [arch-info]*

arguments	Print the arguments for every stack frame as each frame is encountered. If the value of this argument is "yes," the parameters of each routine will be evaluated and displayed by the correct language. The same holds true for local (or automatic) variables.
locals	Display the active local variables for each stack frame.

Examples

To display all frames if you are positioned in the top frame:

```
(debug) wal ↵
```

To display four frames, including the current one:

```
(debug) walk 4 ↵
```

To display all frames and their arguments:

```
(debug) walkback, arg ↵
```

To display all frames and their active local variables:

```
(debug) walkback, locals ↵
```

To display the two oldest stack frames and their arguments:

```
(debug) walk, f b-1, a ↵
```

To display the arguments and local variables for the current frame only:

```
(debug) walk 1, a, l ↵
```

See Also

Commands: **finish, machine-state, position, step**

watch-memory *Debugger Command*

Monitors changes to a specified memory region.

Summary

Monitors changes to a specified memory region.

Syntax

watch-memory *low* **,length ,high ,values ,disable ,name ,count ,if ,action**

where:

low	A symbolic or numeric bit address
length	A number of bits (greater than or equal to 1)
high	An addressable expression whose address is greater than *low*
values	The size of the values history queue
disable	Disable a watchpoint; yes or no
name	A name for the watchpoint
count	How many times the region may be changed without a watchpoint occurring
if	An expression that evaluates to true or false
action	A sequence of commands; must be in braces if more than one command is specified

Examples

```
watch-memory _eint
watch-memory _eshcrt, length 16
watch-memory 0xefffcc00
watch-memory _eint:31, length 1
watch-memory 0x400688:2, length 3
watch-memory _arr 8000, values 4
watch-memory _eint, values 1000
```

Description

The **watch-memory** command lets you specify any monitored memory-range addresses directly. This is useful when the address range that is being corrupted is known and there is no programming language variable defined for this range, such as a memory region allocated by a dynamic memory heap allocation software library like malloc(3). When a programming language variable or reference expression (like "A," "A.B" or "A[4]") is needed, then use the **watch-reference** command since the debugger implicitly supplies the correct address and size of the referenced object.

You can specify the monitored memory range with the **low** and **length** arguments (where **length** is a keyword; the default is 32 bits), or with the **low** and **high** arguments. The **length** and **high** arguments are mutually exclusive.

The values to the *low* address and **high** address arguments are recorded exactly as the user specified on the command line to preserve symbolic references so that these watchpoints can be reset reliably across multiple sessions. For example, if the events are written out to a file via **redirect–output** and then included in a subsequent session where the executable may have been relinked, the watchpoints should be able to be reset on those same symbolic locations even though they may have different addresses.

Since the debugger monitors bit-granular instances, same value stores will not trigger the event in Mxdb.

Using the **length** or **high** argument has an impact on the size of the values queue. If you specify a length that's too long or an address that's too high, it is possible that Mxdb will attempt to create a values queue too large for the memory available to Mxdb. If this occurs, Mxdb displays an error message saying that there is not enough memory to execute the command. To correct this, specify a smaller value for the **length** argument or a lower address for the **high** argument.

Arguments

low
If you specify this argument without a **length** or **high** argument, 32 is used as the length.

length
This argument specifies the length, in bits, of the memory range.

high
The address of this addressable expression must be higher than the **low** address. It defines an exclusive boundary for the memory region. The exclusivity of this address differs from the **watch-reference** command's **high** argument because for **watch–reference** the **high** argument is an object that has a known size.

values
This argument sets the size of the values history queue. In other words, it sets the number of unique values for a particular watchpoint that will be retained for later display via the **watchpoint-print** command. At most, the last **values** number of values for the monitored region will be available for display at any time. (Obviously the values queue can contain less than **values** number of historical instances if the region has not been modified that many times.) The initial default value for this argument is 2. This allows for retention of the previous as well as latest values of a monitored region. When a watchpoint is created, the values queue is immediately initialized to contain the current value. When a process is restarted with the command **debug, again**, the values queue is again initialized to contain the current value after the process is reinitialized; sometimes there is no current value, such as a heap or stack address that is not valid when the process is restarted.

disable Disable a watchpoint if the value is "yes." The watchpoint is first created to make sure that it is valid and then it is disabled. While a watchpoint is disabled, no values are added to the history queue.

name A name associated with this watchpoint.

count Reset the count value. This denotes the number of times that the region will change before a watchpoint occurs. Note that all of the unique values will be added to the history queue, even if any conditions evaluate to false (**if**) or if the process is continued (with the **action** argument).

if If the expression evaluates to a language-specific true value, the debugger stops the process and performs the specified action.

action Execute the command sequence whenever the event occurs.

Examples

```
(debug) watch-memory _eint    „ implicit length (32) used ↵
(debug) watch-memory _eshort, length 16 „ monitor 16 bits ↵
(debug) watch-memory 0xefffcc00 „ monitor a 32-bit stack location ↵
(debug) „ Watch the lowest bit of a 32-bit integer ↵
(debug) watch-memory _eint:31, length 1 ↵
(debug) „ Watch a 3-bit sequence starting at 2-bits offset into ↵
(debug) „ a static data address ↵
(debug) watch-memory 0x400688:2, length 3 ↵
(debug) „ Watch 1000 bytes (8000 bits) at "_arr" and retain the ↵
(debug) „ last four values ↵
(debug) watch-memory _arr 8000, values 4 ↵
(debug) „ Keep the last 1000 values of "_eint" ↵
(debug) watch-memory _eint, values 1000 ↵
```

When a watchpoint event occurs, it is not printed by default. To print events, use the **watchpoint-print** command or a command similar to this one:

```
(debug) watch-memory _eint, name EINT, action {p-s; w-p EINT} ↵
```

See Also

Commands: **breakpoint, disable-events, enable-events, signal, watch-reference, watchpoint-print**

Topic: watchpoints-(88k)

watch-reference *Debugger Command*

Monitors storage in a memory region specified by addressable expressions.

Summary

Monitors storage in a memory region specified by addressable expressions.

Syntax

watch-reference *low* **,length ,high ,values ,scope ,line ,disable ,name ,count ,if ,action**

where:

low	Any addressable expression
length	A number of bits (greater than or equal to 1)
high	An addressable expression whose address is greater than *low*
values	The size of the values history queue
scope	The name of a module or a routine
line	Line number, or CURRENT or LAST with an optional offset
disable	Disable a watchpoint; yes or no
name	A name for the watchpoint
count	How many times the region may be changed without a watchpoint occurring
if	An expression that evaluates to true or false
action	A sequence of commands; must be in braces if more than one command is specified

Examples

```
watch-reference a_struct
watch-reference a_struct, len 2
watch-reference ar[2], hi ar[4]
```

Description

The **watch-reference** command lets you monitor the values in memory regions referred to by expressions.

The **length** and **high** arguments are mutually exclusive. If the scope and line arguments are both supplied, then the position is set to that location before evaluating the expressions so that the variables will be evaluated in the correct context.

The values to the *low* expression and **high** expression arguments are recorded exactly as the user specified on the command line in order to preserve symbolic references so that these watchpoints can be reset reliably across multiple sessions. For example, if the events are written out to a file via **redirect-output** and then included in a subsequent session where the executable may have been relinked, the watchpoints should be able to be reset on those same symbolic locations even though they may have different underlying addresses.

When you create a watchpoint, the high-level language used for printing at that time is recorded. That language is always used to print values for that watchpoint.

Arguments

low
: If you specify this argument without a **length** or **high** argument, the length of the reference is used as the length. The output will then look like the result of an **evaluate** command. This **evaluate**-type of display will also occur if a **high** value is supplied, the two types of the expressions are the same, and the addresses of the two objects are such that an "artificial array" display makes sense.

length
: This specifies the length (in bits) that will be monitored of the expression. When the **length** argument is specified, the high-level display is turned off and the watchpoint will be displayed in a primitive display format.

high
: The address of this object must be greater than the address of the **low** object. When both the **low** and **high** arguments are specified, a high-level "artificial array" display is used if the resultant types of evaluating both expressions are the same and the distance between these objects would contain exactly enough space for an integral number of objects of the same type. Otherwise, a low-level display format is used to display the values.

values This argument sets the size of the values history queue. In other words, it sets the number of unique values for a particular watchpoint that will be retained for later display via the **watchpoint-print** command. At most, the last **values** number of values for the monitored region will be available for display at any time. (Obviously the values queue can contain less than **values** number of historical instances if the region has not been modified that many times.) The initial default value for this argument is 2. This allows for retention of the previous as well as latest values of a monitored region. When a watchpoint is created, the values queue is immediately initialized to contain the current value. When a process is restarted with the command **debug, again**, the values queue is again initialized to contain the current value after the process is reinitialized; sometimes there is no current value, such as a heap or stack address that is not valid when the process is restarted.

scope Move the debugger position to the specified scope before a watchpoint is set.

line Move the debugger position to the specified line before a watchpoint is set.

disable Disable a watchpoint if the value is "yes." The watchpoint is first created to make sure that it is valid and then it is disabled. While a watchpoint is disabled, no values are added to the history queue.

name A name associated with this watchpoint.

count Reset the count value. This denotes the number of times that the region will change before a watchpoint occurs. Note that all of the unique values will be added to the history queue, even if any conditions evaluate to false (**if**) or if the process is continued (with the **action** argument).

if If the expression evaluates to a language-specific true value, the debugger stops the process and performs the specified action.

action Execute the command sequence whenever the event occurs.

Examples

```
(debug) watch -reference a_struct   „ monitor a structure ⤵
(debug) eval a_struct ⤵
{
  a      = 2
  b      = 12.01
}
(debug) watchpoint -print 2 ⤵
{
  a      = 2
  b      = 12.01
}
```

```
(debug) watch-reference a_struct, len 2 ,, monitor first 2 bits ↵
(debug) w-p 3 ↵
0x0
(debug) ,, Monitor an array slice (high-level printing format) ↵
(debug) watch-reference ar[2], hi ar[4] ↵
(debug) eval ar[2], ar 3 ↵
{
  [0]    = 11.2
  [1]    = 22.3
  [2]    = 33.4
}
(debug) w-p 4 ↵
{
  [0]    = 11.2
  [1]    = 22.3
  [2]    = 33.4
}
(debug)
```

See Also

Commands: **breakpoint, disable-events, enable-events, signal, watch-memory, watchpoint-print**

Topics: scopes, watchpoints–(88k)

This page intentionally left blank.

watchpoint-print *Debugger Command*

Prints values for a monitored region defined by watch-memory or watch-reference.

Summary

Prints values for a monitored region defined by **watch-memory** or **watch-reference**.

Syntax

watchpoint-print *[name] [value-number]* **,length ,format ,all**

where:

name	The name of a watchpoint event
value-number	An integer greater than or equal to 0
length	A number of bits (greater than or equal to 1)
format	A format: ascii, binary, decimal, hexadecimal, ieee−double, ieee−float, ieee−single, octal, string, symbolic, system−error, or unsigned−decimal
all	Print all recorded instances of watchpoints; yes or no

Examples

```
watchpoint-print EINT
watchpoint-print EINT, all
w-p
w-p, all ,,print all recorded instances of all watchpoints
w-p EINT, all, format dec
```

Description

The **watchpoint-print** command displays recorded instances of watchpoints from the values queue associated with each watchpoint. This queue records changes made to a watchpoint over time.

Arguments

name

This argument specifies the name of a watchpoint-event (an event set with the **watch-memory** or **watch-reference** command). If the **name** argument is not supplied, this command invocation is applied to all watchpoints. By default, this will print the latest value for each watchpoint.

If this argument is not supplied, each event's **name**, **low**, and **high** (if supplied) argument values are printed in the same manner as displayed by the **event-status** command to distinguish between events. This identification is followed by a colon and then the appropriate value.

language *Debugger Type*

Accepts the name of a language supported by Mxdb.

Summary

Accepts the name of a language supported by Mxdb.

Syntax

c | c++ | fortran | icobol | pascal

Description

The language type accepts the name of any language whose expressions Mxdb can evaluate. The names are case-insensitive and can be abbreviated.

Examples

```
C
FOR
```

See Also

Command:	**c-p:option-status**
Topics:	abbreviation, c-language, c++-language, fortran-language pascal-language
Realm:	icobol

line-number

Debugger Type

Specifies a numeric or symbolic line number.

Summary

Specifies a numeric or symbolic line number.

Syntax

number | **current**[+*offset*] | **current**[−*offset*] | **last**[−*offset*]

where:

number	An integer greater than or equal to 1. This can be a hexadecimal (beginning with 0x), octal (beginning with 01 through 07), or decimal (beginning with 1 through 9) number.
offset	Plus or minus a decimal (beginning with 1 through 9), octal (beginning with 01 through 07), or hexadecimal (beginning with 0x) number. If you specify an offset, you must not put any space between the name and the offset.

Description

The line-number type specifies a line number numerically or symbolically. The names "current" and "last" can be abbreviated.

Examples

```
256
0xa
012
last
c
curr+19
L-7
```

See Also

Commands: **list, view**
Types: cardinal, ordinal, position

```
typedef uchar           *_ucharp,       **_ucharpp;
typedef ushort          *_ushortp,      **_ushortpp;
typedef uint            *_uintp,        **_uintpp;
typedef ulong           *_ulongp,       **_ulongpp; /* unsigned int */

typedef unsigned        *_unsignedp,    **_unsignedpp;

typedef float           *_floatp,       **_floatpp;
typedef double          *_doublep,      **_doublepp;
```

See Also

Topic: c-language, c++-language

c-language

Debugger Topic

Using C and Mxdb.

Summary

C is one of the languages supported by the Mxdb debugger. Mxdb supports the complete C syntax except explicit structure assignment using braces and the evaluation of macro expansions. Thus, most expressions appearing in source code can be evaluated in the debugger also.

Mxdb also emulates the semantics and descriptive format of the ANSI C language. To do this, Mxdb implements various conversion operations. However, it does not let you construct fully-general dynamic types such as "(bar_type[2])." Since dynamic construction of most types is not supported, some generally useful type declarations are provided by the debugger (see the c-builtin-types topic). Mxdb also supports type-casts of structures, but no bit conversions are performed.

You can specify casts to dynamic pointer types with one level of indirection, provided that the denotation type is not dynamic. For example, you could have "(struct foo *)expression" or "(foo *)expression," but not "(foo **)expression."

You can have array assignment if the two arrays are of the same type (same shape and element type). The assignment operands must be arrays; no pointer conversions are performed.

You may explicitly typecast structure objects to any arbitrary type if the desired type's size is less than or equal to the structure type's size.

When the debugger evaluates an unsigned expression, it prints an integer literal suffixed with "u" or "U." The evaluation result can then be used as input to another command and the value's sign is preserved. Normally, integer literals with no type suffixes are stored as "unsigned int" only if they are too large to fit into "int."

Description

Description of standard and user–defined scalar types is supported, as well as objects of such types:

```
(debug) describe unsigned int ↵
unsigned int;

(debug) desc natural ↵
typedef int natural;
```

Objects (identifiers) are described using declarative syntax, including storage classes:

```
(debug) desc usint1 ↵
static unsigned short usint1;

(debug) desc dfl1 ↵
extern double dfl1;

(debug) desc int1 ↵
auto int int1;
```

Expressions are described using typecast syntax:

```
(debug) desc int1++ ↵
( int )
```

ANSI literal type specifiers may be used to specify literal "signedness." These specifiers are not currently used to determine short vs. int or float vs. double.

```
(debug) describe 123 ↵
( int )
(debug) describe 123U ↵
( unsigned int )
(debug) describe 1.5l ↵
( double )
```

This describes an integer array type:

```
(debug) desc array_type ↵
typedef int array_type[3];
```

The description of complex types such as structures, unions, and enumerations is also in declarative syntax:

```
(debug) desc fruits ↵
enum fruits {
   apple = 2,
   cherry = -1,
   pear = 0,
   kumquat = 1
};

(debug) desc a_small_union_type ↵
typedef union small_u {
   short (*a)(void);
   int b;
   struct small_struct c;
} a_small_union_type;
```

Enumeration constants are described using declarative syntax, while enumeration expressions are described as typecasts. Enumeration constants are not described using standard C syntax.

```
(debug) desc snack ↵
extern enum fruits snack;

(debug) des kumquat ↵
enum fruits {
   ...
   kumquat = 1
   ...
};
```

Named bit fields can be described, while unnamed bit fields are appropriately inaccessible:

```
(debug) desc t1_tag    „ unnamed field doesn't show up ↵
struct t1_tag {
   unsigned ants_ : 16;
   unsigned cats : 8;
   struct t1_tag *next;
   unsigned dogs_and_cats : 1;
};

(debug) desc foo.ants_ ↵
unsigned  : 16;
```

Functions are described using ANSI C function prototype format:

```
(debug) desc func3 ↵
extern unsigned short func3 (
      char *str,
      int one,
      int two,
      int three);
```

Function calls are described in terms of the function return type (no invocation is done):

```
(debug) desc func3(0,1,2,3) ↵
( unsigned short )
```

Arbitrarily complex types and expressions may be described. This example describes an array of pointers to functions taking no arguments and returning an unsigned integer:

```
(debug) desc funcp_arr ↵
extern unsigned int (*funcp_arr[10])(void);
```

You can describe a pointer to an array of integer pointers:

```
(debug) desc p_to_a_of_p ↵
extern int *(*p_to_a_of_p)[3];
```

Mxdb also annotates constructs for which no debugging information is available, such as external variables and routines in vendor-supplied libraries:

```
(debug) desc printf ↵
extern int printf (...); /* $No_Debug_Info */
(debug) desc errno ↵
extern int errno; /* $No_Debug_Info */
```

This page intentionally left blank.

Global Variable Lookup

When attempting to locate a symbol in the user's program, Mxdb first examines the debugging information for the program module the debugger is positioned to.

If this lookup fails, Mxdb attempts to find the name as a global external symbol in the debugged program. If Mxdb finds the name this way and there is debugging information for it in the defining module, that definition is used.

If Mxdb finds the name where there is no debugging information, it is treated as an "int" variable if it seems to be data; if the name seems to represent code, it is typed as "int (*) (...)" (a function taking any number of arguments of any type and returning an int). For more information on name resolution, refer to the name-resolution topic.

String Literal Operations

Mxdb treats string literals in accordance with ANSI C: they are arrays of characters. To allow operations that are legal on an array to be legal on a string literal also, Mxdb attempts to allocate space for string literals in the user program and copies a string literal into this space. If Mxdb cannot locate an allocation routine definition in the user program, any string literal operation requiring this implicit allocation generates an error.

Allocation can often be avoided. For instance, since every string literal copied into the target process by Mxdb has a unique address, it is never possible for the following expression to have the value "true":

```
(debug) evaluate char_ptr == "string literal" ↵
```

Therefore, Mxdb does not try to allocate space for the literal, and the expression is transformed into one which will correctly evaluate to 0, while still insuring non-literal operand expressions are evaluated (in case there are side-effects):

```
char_ptr && 0
```

A further extension allows assignment of string literals to character arrays:

```
(debug) assign char_array "a string" ↵
```

In this case, Mxdb attempts no allocation for the string literal. Characters are copied into the targeted array either to the end of the array or through the terminating null byte of the literal, whichever comes first. Note that if the assignment stops before the end of the literal because the array's end is reached, the "string" in the array will not be null-terminated; Mxdb issues a note to point this out. To copy characters into an array beyond the array bounds, use **assign**'s format keyword with the value "string."

Ambiguity Resolution

There are several possible sources of name ambiguity in a C program. Some ambiguities inherent in C, while others are introduced because Mxdb does not place type tags in a separate namespace from other type names in the same scope.

1. **Type tag vs. typedef names.**
   If a program contains a typedef and a type tag of the same name in the same scope, it is not clear whether the typedef or the tag is desired when the name is used in an expression. Mxdb always resolves this ambiguity in favor of the typedef name; if the tagged type is desired, the tag name should be preceded by "struct," "union," or "enum," as appropriate. For instance:

```
struct ambig1 { int i; };
typedef int ambig1;

(debug) describe ambig1 ↵
typedef int ambig1
(debug) describe struct ambig1 ↵
struct ambig6 {
int i;
};
```

2. **Type tags vs. local identifier/constant/function names.**
   If a program contains both a type tag and an object, constant, or function of the same name in the same scope, Mxdb always resolves this ambiguity in favor of the object, constant, or function; Mxdb informs you of this choice:

```
enum trouble {ambig = 1};
struct ambig { int i; };

(debug) desc ambig ↵
Note:'ambig' is both a type and an object/constant/function
name.
     Using non-type definition.
enum trouble {
   ...
   ambig = 1
   ...
};
```

   You can use the "struct," "union," or "enum" specifier to make it clear the type you want Mxdb to use:

```
(debug) evaluate (struct ambig*) 0 ↵
(struct ambig *) 0x00000000
```

   In addition, on some commands, you can use the meaning-kind keyword to choose your desired definition:

```
(debug) describe ambig, meaning-kind type ↵
struct ambig {
        int i;
}

(debug) desc ambig, meaning-kind enumeration ↵
enumeration constant, value is 1;
```

3. **Label name vs. type/object/constant/function name.**
   If a label has the same name as a type (tag or typedef) or an object, constant, or function declared in the same scope, Mxdb resolves the ambiguity in favor of the name that is not the label name. Mxdb selects an object, constant, or function name over a type name, but chooses a type name over a label name. You can only access a label with the meaning-kind keyword:

```
(debug) list ↵
     5 main()
     6 {
     8    int ambig;
*   11 ambig:
    12    ambig = 6;
(debug) desc ambig ↵
Note:'ambig' is both a label and an
     object/constant/function name.
     Using non-label definition.
auto int ambig;
(debug) desc ambig, m label ↵
ambig: /* label at line "test.c":11 */;
```

4. **Type tags vs. global identifier/function names.**
   Mxdb selects a local object, function constant, or typedef name over a global (external) object or function of the same name. Typedef names are not hidden by global names.

   However, if Mxdb detects an ambiguity between a type tag and a global variable or function, Mxdb chooses the global definition; this applies only to type tags. To force Mxdb to select the type tag in such cases, use the "struct," "union," or "enum" specifier or the meaning-kind keyword.

   To access a global name that is hidden by a local typedef, move your debugger position outside the scope of the typedef.

Evaluation

Literals in expressions may be entered in any legal C format, including decimal, hexadecimal, octal, scientific notation, and binary (similar to hexadecimal, but begins with "0b" or "0B" instead of "0x" or "0X"):

```
(debug) eval 123 ↵
123
(debug) eval 0x7b ↵
123
(debug) eval 0173 ↵
123
(debug) eval 1.5 ↵
1.5
(debug) eval 0.15e1 ↵
1.5
(debug) eval 0b10 ↵
2
```

Unsigned values are output with an attached unsigned specifier for clarity:

```
(debug) eval int1 ↵
123
(debug) eval uint1 ↵
123u
```

Values may be displayed in a variety of formats by using the **evaluate** command's **format** argument (see the **evaluate** command for more information):

```
(debug) eval 123, f hex ↵
0x0000007b
(debug) eval 123, f octal ↵
000000000173
(debug) eval (short) 123, f binary ↵
0b0000000001111011
```

Note that the hexadecimal format outputs a "0x" prefix, and the binary format outputs a "0b" prefix to the value.

Array values are printed in a format similar to that used for array initialization:

```
(debug) desc int_arr1 ↵
static int int_arr1[3];
(debug) eval int_arr1 ↵
{
   [0]        = 1
   [1]        = 2
   [2]        = 3
}
```

The following examples evaluate string literals:

```
(debug) eval "abc"[1] „with malloc() linked in ↵
'b'
(debug) eval "abc"[1] „without malloc() linked in ↵
Error: The process does not contain a "malloc" memory
allocation routine.
(debug) eval char_ptr == "some literal" ↵
0
(debug)
```

Array slices may be displayed by using the **evaluate** command's **array** argument. The **array** keyword causes data at the given address to be displayed as an array of the specified length. See the **evaluate** command for more information.

```
(debug) eval int_arr1[1], array 2 ↵
{
   [0]        = 2
   [1]        = 3
}
```

Pointer values are printed in hexadecimal format. All standard C operations on pointers are available. The example below demonstrates pointer dereferencing and pointer arithmetic.

```
(debug) desc intp ↵
extern int *intp;
(debug) assign intp int_arr1 ↵
(debug) eval intp ↵
0x00404498
(debug) eval *intp ↵
1
(debug) eval *(intp + 2) ↵
3
```

Once again, the **array** keyword may be used to display data in array format when the default behavior uses another format:

```
(debug) assign intp int_arr1 ↵
(debug) eval *(intp+1) ↵
2
(debug) eval *(intp+1), array 2 ↵
{
   [0]        = 2
   [1]        = 3
}
```

In addition to the standard arithmetic operations, post- and pre-increment and post- and pre-decrement are supported:

```
(debug) eval *++intp „ see example above ↵
2
(debug) eval int_arr1[1]++ ↵
2
(debug) eval int_arr1 „ demonstrates elide-arrays option ↵
{
   [0]        = 1
   [1..2]     = 3
}

(debug) eval --int1 ↵
122
(debug) eval int1-- ↵
122
(debug) eval int1 ↵
121
```

Assignment may be done using either the **assign** command or the standard C assignment operators:

```
(debug) assign int1 12 ↵
(debug) eval int1 += int2 ↵
135
(debug) eval int1 ↵
135
(debug) eval int1 /= -5 ↵
-27
```

This is an example of a string literal being assigned to an array of characters:

```
(debug) assign array "abracadabra" ↵
```

Structure and union values are printed out in a quasi-declarative format (refer to a previous example to see a declaration of small_u):

```
(debug) desc sut_obj1 ↵
extern union small_u sut_obj1;
(debug) eval sut_obj1 ↵
{
   a          = 0xeffffb90
   b          = 23
   c          = {
      i          = -4
      j          = 0
      p          = "a"
   }
}

(debug) desc small_struct ↵
struct small_struct {
   int i;
   int j;
   char p[2];
};
(debug) desc sst_obj1 ↵
extern struct small_struct sst_obj1;
(debug) eval sst_obj1 ↵
{
   i          = 343567
   j          = 2
   p          = "b"
}
```

Extended functionality allows structure values to be cast to any type where sizeof(struct-type) >= sizeof(target-type):

```
(debug) eval (int) sst_obj1 ↵
343567
```

Enumeration values are printed as comma list expressions so that it is possible to show both the enumeration constant name and the associated integer value. Since the result is a valid C expression with the correct value and type, it is suitable for cut–and–paste operations. It can also be used as input to another command.

```
(debug) eval snack ↵
(pear, (fruits) 0)
(debug) desc '{eval snack} ↵
( enum fruits )
```

Evaluated expressions may be arbitrarily complex:

```
(debug) assign sut_ptr &sut_obj1 ↵
(debug) assign sut_ptr_ptr &sut_ptr ↵
(debug) eval (*sut_ptr_ptr) -> b += 2 ↵
125
(debug) eval (*sut_ptr_ptr) -> c.p ↵
"a"
```

If you use function call syntax in a command that accepts a language expression (such as **evaluate**, **assign**, **breakpoint**, **define-variable**, **if**, and **while**), the Mxdb debugger makes an attempt to invoke the function when it evaluates the expression; ANSI C specifications for parameter matching and conversions are applied. Note that Mxdb will not try to invoke a function that is an argument to the **describe** command since expressions supplied to **describe** are not evaluated.

The debugger allows a routine to be invoked with fewer or more arguments than the debugging information records. When this situation occurs, warnings are issued by the debugger, but the invocation is permitted. The default argument value promotions are made for the argument values passed for which no information is available.

This differs from the case where the routine interface information is "(...)," indicating that any number and type of arguments are acceptable. Default argument value promotions are made but no warning is issued.

If the function invocation is successful, the function is then called whenever the expression involving the call is evaluated. Thus, a function appearing in a **breakpoint** command will be called every time the condition is tested. The result of evaluating a function call is printed out in the same manner as any other expression:

```
float a_function(int base, float incr);
void another_function(void); /* has no return value */

(debug) eval a_function(1, 0.25) ↵
1.25
(debug) eval another_function() ↵
(void)
(debug) „ in the next example, printf has no debugging info so it ↵
(debug) „ is implicitly given the "(...)" interface by the debugger ↵
(debug) eval printf("hi = %d, level = %u\n", hi, $$level); ↵
hi = 10, level = 200
21
```

All events are active when the invoked process is being executed. If the invoked routine gets an event that is not the expected return address, the process-status is noted and a note is issued saying that the process stopped in a debugger-invoked context before returning to the top level.

You can debug as you wish in the invoked routine context. Your stack is terminated at this point by a frame at this address: __debug_info+<*N*>. Here is an example:

```
(debug) b one_param ↵
(debug) eval one_param(4) ↵
Stopped at frame 0, line 10, scope \test4\one_param, pc 0x101dc
 breakpoint "2"
Note: Process stopped in debugger-invoked routine context.
(debug) walk, a ↵
frame 0, line 10, scope \test4\one_param, pc0x101dc
    i           = 4
frame 1, pc __debug_info+24
```

These invocation contexts will nest so that a user can invoke routines even in invoked contexts and the state of the process will be reinstated appropriately as each invocation returns (if it returns).

When you invoke a routine, any signal that stopped the target process is discarded.

Assignment of integer to pointer and expressions comparing pointers and integers are allowed. A note is generated because this type of interoperability is non-standard.

```
(debug) assign a_ptr 1 ↵
Note: Assignment of integer to pointer without a cast allowed,
      but not standard.
(debug) eval a_ptr == 1 ↵
Note: '==' comparison between pointer and integer allowed,
      but not standard.
1
(debug) eval a_ptr < 1 ↵
Note: '<' comparison between pointer and integer or literal 0
      allowed, but not standard.
0
(debug) desc a_ptr_param ↵
extern void a_ptr_param (
      char *c);
(debug) eval a_ptr_param(1) ↵
Note: Assignment of integer to pointer without a cast allowed,
      but not standard.
(void)
```

Options

Several language-specific options are supported that modify the way expression values are displayed: **elide-arrays**, **string-display**, and **pointer-dereference-level**.

The **elide-arrays** option, which is turned on by default, causes same−valued array elements to be elided. For example,

```
(debug) opt elide-array no ↵
(debug) eval int_arr1 ↵
{
   [0]        = 1
   [1]        = 3
   [2]        = 3
}
(debug) opt e-a y ↵
(debug) eval int_arr1 ↵
{
   [0]        = 1
   [1..2]     = 3
}
(debug) eval a_large_array ↵
{
   [0]        = -5
   [1]        = 45
   [2..150]   = 0
   [151]      = 8
}
```

This page intentionally left blank.

The **string-display** option, also turned on by default, applies to the display of character array and character pointer data. Character arrays are displayed as strings, instead of as arrays. This option overrides the **elide-arrays** option in that character arrays are always displayed as strings rather than arrays (elided or otherwise) if **string-display** is turned on.

```
(debug) desc c_arr3 ↵
static char c_arr3[6];
(debug) opt s-d n ↵
(debug) eval c_arr3 ↵
{
   [0]        = 'h'
   [1]        = 'e'
   [2..3]     = 'l'
   [4]        = 'o'
   [5]        = '\000'
}
(debug) opt s-d y ↵
(debug) eval c_arr3 ↵
"hello"
```

The string is terminated when a NUL byte is encountered or the end of the array is reached. If the end of the array is reached before an NUL byte is encountered, a warning is printed:

```
(debug) desc c_arr1 ↵
extern char c_arr1[10];
(debug) eval c_arr1 ↵
"0123456789"
Warning: Array not terminated by null byte.
```

String length can also be controlled by setting the value of the **string-display-limit** option:

```
(debug) opt s-d-l 5 ↵
(debug) eval c_arr1 ↵
"01234"...
```

Normally, character pointer values are printed in hexadecimal format and not dereferenced. If **string-display** is turned on, character pointers are dereferenced and their contents are displayed as strings. The strings displayed are terminated either by a NUL byte or by reaching the limit set by **string-display-limit**.

```
(debug) assign cp c_arr3 ↵
(debug) opt s-d n ↵
(debug) eval cp ↵
0x004044ec
(debug) opt s-d y ↵
(debug) eval cp ↵
0x004044ec -> "hello"

(debug) desc c_arr2 ↵
extern char *c_arr2[3];
(debug) eval c_arr2 ↵
{
   [0]        = 0x000101b0 -> "goodbye"
```

```
    [1]         = 0x000101b8 -> "so long"
    [2]         = 0x000101c0 -> "farewell"
}
```

The **pointer-dereference-level** option allows the user to specify the number of levels of pointers to automatically dereference. This option is initially set zero, meaning pointers are not automatically dereferenced. Repeated pointer values are not fully displayed again;instead, a message is printed indicating the dereferenced value has already been printed.

```
(debug) desc a_struct_type ↵
struct a_struct_type
{
   int* intp;
   int** intpp;
};
(debug) desc a_ptr ↵
extern struct a_struct_type *a_ptr;

(debug) opt p-d 0 ↵
(debug) eval a_ptr ↵
0x004071f0


(debug) opt p-d 1 ↵
(debug) eval a_ptr ↵
(struct a_struct_type *) 0x004071f0 ->
   {
      intp          = 0x00404498
      intpp         = 0x004044f4
   }


(debug) opt p-d 2 ↵
(debug) eval a_ptr ↵
(struct a_struct_type *) 0x004071f0 ->
   {
      intp          = (int *) 0x00404498 -> 123
      intpp         = (int **) 0x004044f4 ->
         0x004071f0
   }


(debug) opt point 3 ↵
(debug) eval a_ptr ↵
(struct a_struct_type *) 0x004071f0 ->
   {
      intp          = (int *) 0x00404498 -> 123
      intpp         = (int **) 0x004044f4 ->
         (int *) 0x004071f0 -> <Previously Displayed >
   }
```

In the last example, the value of ****a_ptr->intpp** is not printed because this field points to the **a_ptr->intp**, which has already been printed.

When dereferencing pointers, scalars are printed on the same line as the pointer; non−scalar objects such as structs and pointers are displayed starting on a new line. When automatic pointer dereferencing fails on a NULL pointer, no warning is produced.

For additional information on any of these options, refer to the options realm.

See Also

Commands: **assign, describe, evaluate, c-p:option-status, options:elide-arrays, options:pointer-dereference-level, options:string-display, options:string-display-limit**

Topics: c-builtin-types, c++-language, fortran-language, pascal-language

c++-language

Debugger Topic

Using C++ and Mxdb.

Summary

C++ is one of the languages supported by the Mxdb debugger. C++ is a relatively new language; the first widely available C++ compiler appeared in 1985. An ANSI-sanctioned language standardization effort commenced in 1989 and is scheduled to deliver a standard in 1995. Draft revisions of the standard are based on the "Annotated C++ Reference Manual" (ARM) by Ellis and Stroustrup (1990). Currently no compiler fully supports this language definition.

Description

Mxdb supports the C++ language as defined by Release 2.1 of the AT&T C++ Language System (better known as Cfront 2.1) to the extent that the implementation allows it. Cfront 2.1 implements all of the major language features defined in the ARM except templates and exceptions. However, its design does not lend itself to the generation of high quality debugging information. This means that the debugger cannot take full advantage of the expressive power of C++.

Mxdb relaxes certain C++ semantics to promote unfettered debugging:

- Non-const and non-volatile objects are allowed to invoke const and volatile member functions, respectively.
- Non-const and non-volatile expressions may be assigned to const and volatile lvalues, respectively.
- Pointers to const and volatile objects are implicitly converted to non-const and non-volatile pointer types, respectively, and to type void*.
- Implicit conversions from integral types to enumeration types are performed on assignments.
- For objects of classes not defining an operator=() member function, memberwise assignment is performed wherever possible and bitwise assignment is performed everywhere else.

For practical purposes, C++ can be considered a superset of ANSI C. Mxdb users can debug C or C++ modules with the C++ language interface. Users cannot debug C++ modules with the C language interface. The restrictions and extensions noted for C debugging apply to C++ debugging as well, with one notable exception: the C++ LP more fully supports dynamic type casts. For more information, refer to the release notice or the on-line c-language help topic.

C++ differs mainly from C in its introduction of the concept of a class. Mxdb allows users to describe class types and to describe, evaluate, and assign objects of those types.

For class type description, Mxdb uses declarative syntax to describe salient features such as as inheritance relationships, member access specifiers and special member functions:

```
(debug) describe D ↵
class D : public B, virtual public C {
public:
   int d;
   D ();
   ~D ();
   operator int ();
};
```

Static and const/volatile members are also described:

```
(debug) describe E ↵
class E {
public:
   E (
      int arg);
   int value () const;
   static int static_value;
};
```

For a class type having virtual functions, some compilers (such as Cfront 2.1) will generate a virtual function table as a member. The structure of this table is implementation-specific. Mxdb indicates this fact by giving the member a stylized and annotated name:

```
(debug) describe Base ↵
class Base {
public:
   signed char x;
   Base ();
   double bar (
      unsigned int i);
   $vtbl // Compiler-generated
};
```

In general, Mxdb will annotate compiler-generated ("artificial") constructs. Another example is the implicit "this" pointer for non-static member functions:

```
(debug) position ↵
frame 0, line 150, scope \const\Three\Three, pc 0x101274
(debug) describe this ↵
class Four *this; // Compiler-generated
```

Mxdb also annotates constructs for which no debugging information is available, such as external variables and routines in vendor-supplied libraries:

```
(debug) describe printf ↵
extern int printf (...); // $No_Debug_Info
(debug) describe errno ↵
extern int errno; // $No_Debug_Info
```


086-000203 updates
093-000710-03

When evaluating a class object, Mxdb displays the names and values of immediate data members first, then displays the names and values of base class data members with annotations to help identify them as such:

```
(debug) describe Three ↵
class Three : public Two {
public:
   double z;
   int x;
   int y;
   Three ();
   int hi (
      signed char *str);
};
(debug) describe m ↵
auto class Four m;
(debug) describe Four ↵
class Four : public Three {
public:
   ~Four ();
   signed char *x;
   int *h;
   const int w;
   class One one;
   class Two two;
   Four ();
};
(debug) evaluate m ↵
{
   x          = 0x00101be0 -> "initial value"
   h          = 0xeffffa70
   w          = 18
   one        = {
      y           = 0
      yy          = -3
   }
   two        = {
      x           = 106
         // Base class One
         y           = 0
         yy          = -3
   }
      // Base class Three
      z          = 0
      x          = 99
      y          = 876
         // Base class Two
         x           = 106
            // Base class One
            y           = 0
            yy          = -3
}
```

Member functions are not displayed during class object evaluation since they have no meaningful "values" in this context. Additionally, due to limitations in Cfront–derived debugging information, there is no guarantee that evaluated (or described) class members will appear in source code order, or that they will appear at all if not used.

The contents of virtual function table members are not displayed during evaluation:

```
(debug) evaluate n ↵
{
   x          = 'j'
   $vtbl      // Virtual function table contents not displayed.
}
```

If a virtual base subobject appears more than once in an evaluated object, only the first subobject is elaborated. This output format reflects the fact that virtual base subobjects are unique and are shared among objects:

```
(debug) describe L ↵
class L {
public:
   double l;
};
(debug) describe A ↵
class A : virtual public L {
public:
   int a;
};
(debug) describe B ↵
class B : virtual public L {
public:
   signed char *b;
};
(debug) describe C ↵
class C : public A, public B {
public:
   unsigned int c;
};
(debug) describe c1 ↵
auto class C c1;
(debug) evaluate c1 ↵
{
   c          = 335555u
      // Base class A
      a          = 335555
         // Virtual base class L
         l          = -56.46
      // Base class B
      b          = 0x00113eb0 -> "base class B value"
}
```

Mxdb provides for the description, evaluation and assignment of individual class members in a manner consistent with their usage in C++. Any necessary implicit conversions and/or qualifications are performed to permit resolution of the name with respect to the indicated scope. Here are some examples manipulating data members at file scope:

```
(debug) position ↵
frame 0, line 184, scope \const\main, pc 0x101560
(debug) describe One ↵
class One {
public:
   int yes (
      const double arg) const;
   int y;
   double yy;
   const signed char *bye () const;
   ~One ();
   One ();
   One (
      int i);
   void hi ();
};
(debug) describe one ↵
auto class One one;
(debug) describe one.y ↵
( int )
(debug) evaluate one.y ↵
-8
(debug) describe one.yy ↵
( double )
(debug) assign one.yy 34.3 ↵
(debug) evaluate one.yy ↵
34.3
```

The next examples manipulate data members at member function scope. Note that explicit qualification of member names with the "this" object pointer is not required. Observe also that Mxdb understands the use of scope qualifier syntax to distinguish between conflicting member names:

```
(debug) position ↵
frame 0, line 150, scope \const\Three\Three, pc 0x101274
(debug) describe this ↵
class Four *this; // Compiler-generated
```

```
(debug) evaluate *this ↵
{
   z          = 22.3
   x          = 4
   y          = 11
      // Base class Two
      x          = 9999
         // Base class One
         y          = -36
}
(debug) describe this->z ↵
double Three::z;
(debug) evaluate this->z ↵
22.3
(debug) evaluate y ↵
11
(debug) evaluate One::y ↵
-36
(debug) evaluate this->Two::x ↵
9999
(debug) evaluate x ↵
4
(debug) evaluate *(class One *) this ↵
{
   y          = -36
}
```

Anonymous unions are unnamed objects that may contain only publicly accessible data members. Anonymous union members share the same address, but otherwise are used like ordinary (nonmember) variables. Mxdb will describe these members with respect to the associated union:

```
(debug) describe a ↵
union {
   int a;
   char *p;
};
(debug) evaluate a ↵
36
(debug) address a ↵
0xefff910:0
(debug) address p ↵
0xefff910:0
```

Mxdb users can describe member functions and invoke them on objects. C++ has no implicit conversion from member functions to pointers to those functions; you must use the declarative pointer-to-member function syntax instead. This means that you can evaluate a non-member function name but not a member function name, unless it is static:

```
(debug) position ↵
frame 0, line 169, scope \const\Four\Four, pc 0x101430
(debug) describe this->echo ↵
int Three::echo (
      signed char *str);
(debug) evaluate echo( "there\n" ) ↵
there
6
(debug) describe echo( "now" ) ↵
( int )
(debug) evaluate echo ↵
Error: No conversion to pointer allowed for member functions.
(debug) describe One::hi ↵
static void One::hi (
      signed char *str);
(debug) evaluate One::hi( "yes" ) ↵
yes
(void)
(debug) evaluate One::hi ↵
0x00101060
```

C++ allows function names and most operators to be overloaded. With ordinary language expression evaluation, Mxdb can apply function argument matching rules to disambiguate overloaded names. However, extra-language debugger activities such as setting breakpoints may require the user to interactively assist in the resolution of ambiguous names. In those cases, Mxdb will present the user with a menu of possible name matches, and will proceed with the user's original request after a satisfactory selection is supplied. For example,

```
(debug) describe One ↵
class One {
public:
   int yes (
      const double arg) const;
   int y;
   const signed char *bye () const;
   ~One ();
   One ();
   One (
      int i);
   static void hi (
      signed char *str);
};
(debug) breakpoint One::One ↵
'One' is an ambiguous reference:

(1) One::One ()
(2) One::One (int i)

Please enter the number of the
correct resolution.(Default is 1.)
1 ↵
(debug) continue ↵
Stopped at frame 0, line 92, scope \const\One\One, pc 0x100fa4
 breakpoint "0"
```

Remember that the overloaded name must be unambiguously found (with respect to multiple base classes) for overload resolution to be performed. Otherwise Mxdb will print an error stating that the name is ambiguous:

```
(debug) describe A ↵
class A {
public:
   short f ();
   signed char a;
   int f (
      int arg);
};
(debug) describe B ↵
class B {
public:
   int f ();
   float a;
};
(debug) describe C ↵
class C : public A, public B {
};
(debug) evaluate c1.f() ↵
Error: Member name 'f' is ambiguous; it occurs in multiple base

classes:
   C derived from B
   C derived from A
```

Mxdb supports operator functions:

```
(debug) describe MyClass::operator+ ↵
int MyClass::operator+ (
      int arg);
(debug) describe obj ↵
auto class MyClass obj;
(debug) evaluate obj ↵
{
   foo       = 1
}
(debug) evaluate obj + 4 ↵
5
(debug) evaluate 4 + obj ↵
5
```

The careful reader may deduce from the last example that Mxdb also supports user–defined conversions. User–defined conversions, which may be specified by conversion constructors and conversion functions, are used implicitly in addition to standard conversions. Like standard conversions, they may be applied to function arguments, function return values, expression operands and explicit type conversions.

```
(debug) describe B ↵
class B {
public:
   double bar;
   B (
      double arg);
   operator int ();
   operator A ();
};
(debug) describe b ↵
auto class B b;
(debug) assign b 34.5 ↵
(debug) evaluate b ↵
{
   bar        = 34.5
}
(debug) evaluate b - 4    „ B::operator int() invoked implicitly ↵
30
(debug) evaluate (int) b  „ B::operator int() invoked implicitly ↵
34
(debug) evaluate char( b ) „ B::operator int() invoked implicitly ↵
'\"'
```

Mxdb detects ambiguities that arise from multiple choices of user-defined conversions and from multiple choices between user-defined and built-in conversions. It also detects ambiguities resulting from multiple choices between conversion constructors and conversion functions. Consider this continuation of the previous example:

```
(debug) describe A::A ↵
'A' is an ambiguous reference.

(1) A::A (int arg)
(2) A::A (const class B *rhs)

Please enter the number of the
correct resolution.(Default is 1.)

1 ↵
A::A (
      int arg);
(debug) evaluate a = b ↵
Error: User-defined conversion cannot be applied unambiguously
to
expression:
     A::A (const class B *rhs)
     B::operator A ()
(debug) breakpoint One::One ↵
```

Mxdb complains since it does not know whether to invoke A(b) or b.operator A() in this case. In the following example, the ambiguity is between multiple choices of conversion functions:

```
(debug) des C ↵
class C : public B {
public:
   C (
      double arg);
   operator double ();
   operator void* ();
};
(debug) describe c ↵
auto class C c;
(debug) evaluate (char) c ↵
Error: User-defined conversion cannot be applied unambiguously
to
expression:
     B::operator int ()
     C::operator double ()
```

The ambiguity is between B::operator int() and C::operator double().

Mxdb adheres to the semantics of C++ concerning initialization and assignment of objects via special member functions. If a class defines the operator=() member function, Mxdb will invoke that function to perform assignments to objects of the class type. Similarly, if a class defines a copy constructor, Mxdb will invoke it to perform initialization of arguments of the class type that occur in an invoked function.

The result of the combined use of operator functions, user-defined conversions, and assignment and initialization member functions in an evaluated expression is multiple implicit function calls. The user should be aware that debugger events such as breakpoints and watchpoints that are set on these functions will be activated in the process. Continuing with the previous example we observe:

```
(debug) event-status ↵
breakpoint, name 1, line 31, scope \conv\B\int
(debug) evaluate c - 1 ↵
Stopped at frame 0, line 31, scope \conv\B\int, pc 0x1010f4
 breakpoint "1"
Note: Process stopped in debugger-invoked routine context.
```

Here the user has activated the breakpoint set on the conversion function B::operator int(), which is invoked to convert "b" into an integer prior to the specified subtraction operation. In this case, the user can recover from the routine context by finishing the routine. After that he/she can delete the interloping event and re-evaluate the original expression:

```
(debug) finish ↵
Run until exit from frame 0, line 31, scope \conv\B\int, pc
0x1010f4
Stopped at frame 0, line 98, scope \conv\main, pc 0x101590
 debugger-invoked routine returned
(debug) delete-event 1 ↵
(debug) evaluate c - 1 ↵
44
```

See Also

Commands:	**assign, describe, evaluate, c-p:option-status,**
Topics:	c-builtin-types, c-language, fortran-language, pascal-language

debugger-variables

Debugger Topic

Using debugger variables.

Summary

A debugger variable is a dynamic object created by the user to hold an expression's value in the current expression evaluation language. These variables can be used in expressions like identifiers are used in programs.

Description

You can create debugger variables with the **define-variable** command. Debugger variables may take on the value of any expression acceptable to the **evaluate** command.

Since variable names may be used in expressions, the names must conform to the identifier syntax of the current language. If the current language changes, it might not be possible to reference variables if their names are not syntactically valid under the new language.

Debugger variable values are bound at the time the variable is created. The values are initially stored in the debugger's address space, but are moved into the target process by some commands. See the migration topic for more information.

Use the **variable** command to obtain information about a particular variable, or about all debugger variables.

Debugger variables persist until you explicitly remove them with the **delete-variable** command.

Convenience variables are variables that are automatically created by the debugger when the **evaluate** command is used. See the **options:convenience-variables** command for more information.

See Also

Commands: **define-variable, delete-variable, variable, c-p:option-status, options:convenience-variables**

Topic: migration

```
 breakpoint "1"
(debug) address i1 ↵
0x4051a0:0
(debug) address ch1 ↵
0x4051a0:0
(debug) assign ch1 "hello" ↵
(debug) evaluate ch1 ↵
'hello'
(debug) evaluate i1 ↵
1751477356
(debug) evaluate i1 ,format ascii ↵
hell
```

5. **Statement Labels:** Statement labels are used to identify and reference individual statements in a program. Statement labels in FORTRAN 77 are lexically indistinguishable from integer constants, so the user must instruct the debugger to resolve such tokens as labels:

```
(debug) list 1 13 ↵
     1  PROGRAM LABELS
     2          INTEGER*4 I1
     3          INTEGER*2 I2
     4
*    5          DO 10,I1=1,10
     6          I2 = 1
     7          GO TO 200
     8 10       CONTINUE
     9 200      I2 = 3
    10          ASSIGN 10 TO I1
    11          GO TO I1
    12          END
    13
(debug) breakpoint 9; continue ↵
Note: There is no code at line 9.  Line 10 will be used
instead.
Stopped at frame 0, line 10, scope \topic9\main, pc 0x101d8
 breakpoint "1"
(debug) describe 10 ↵
INTEGER*4
(debug) describe 10 ,meaning-kind label ↵
10 ! LABEL
(debug) address 10 ↵
Error: The address of the expression can not be taken.
(debug) address 10 ,meaning-kind label ↵
0x101c4:0
(debug) describe 200 ,meaning-kind label ↵
200 ! LABEL
(debug) address 200 ,meaning-kind label ↵
0x101d8:0
```

In debugging a program it might be useful to modify flow control by assigning a label to an integer variable and then transferring control to the associated statement. To do this the user must assign the address of the label to the variable:

```
(debug) list ↵
*    10          ASSIGN 10 TO I1
     11          GO TO I1
     12          END
     13
(debug) assign i1 Z'101c4' ↵
(debug) continue ↵
Stopped at frame 0, line 10, scope \topic9\main, pc 0x101d8
 breakpoint "1"
(debug) view ↵
      4
      5          DO 10,I1=1,10
      6          I2 = 1
      7          GO TO 200
      8 10       CONTINUE
      9 200      I2 = 3
*    10          ASSIGN 10 TO I1
     11          GO TO I1
     12          END
```

Mxdb also annotates constructs for which no debugging information is available, such as external variables and routines in vendor-supplied libraries:

```
(debug) desc printf ↵
INTEGER*4 FUNCTION printf() ! $No_Debug_Info
(debug) desc errno ↵
INTEGER*4 errno ! $No_Debug_Info
```

See Also

Commands: **assign, describe, evaluate, c-p:option-status, options:elide-arrays**

Topics: c-language, c++-language, pascal-language

initialization

Debugger Topic

Initializing Mxdb.

Summary

The Mxdb debugger lets you initialize each debugging session with a set of commands. Mxdb looks for a file named **.mxdb_init**, first in the working directory, then in your home directory. Whichever file the debugger finds first, it includes as a c-p command file.

See Also

Command: **c-p:include**

machine-registers

Debugger Topic

Accessing machine registers.

Summary

Registers are accessed by using the names displayed by the **machine-state** command. These names cannot be abbreviated.

Description

The machine registers are as follows:

Register	Function
$r0	Zero
$r1	Subroutine return pointer
$r2–$r9	Called procedure parameter registers
$r10–$r13	Called procedure temporary registers
$r14–$r25	Calling procedure reserved registers
$r26	Linker
$r27	Linker
$r28	Linker
$r29	Linker
$r30	Calling procedure reserved register
$r31	Stack pointer
$fpsr	Floating-point status register
$fpcr	Floating-point control register
$psr	Processor status register
$sxip	Shadow execute instruction pointer
$snip	Shadow next instruction pointer
$sfip	Shadow fetched instruction pointer
$cfa	Canonical frame address pseudo-register
$pc	Program counter pseudo-register

To print any register's value, use the **evaluate** command. To modify its value, use the **assign** command. (These variables are used just like any other variables visible in your program.)

Each register's type is "unsigned integer" when used in language expression evaluation.

Note that on AViiON computer systems, the registers $r1–$r13 are valid only in the top frame; $r1–$r13 are invalid in other frames since they are temporary registers, which are assumed to be destroyed across every call boundary.

See Also

Commands: **assign, continue, evaluate, c-p:option-status, machine-state**

migration *Debugger Topic*

How and why debugger–resident objects move to the target process.

Summary

Mxdb automatically moves debugger–variables, convenience–variables and sometimes literals (such as strings) from the debugger's address space to the target process.

Description

Normally, debugger–variables and convenience–variables are stored in the debugger's address space. However, Mxdb will move them into the target process whenever their address is required. There are two cases when this happens:

1) when the address command is used:

```
(debug) def–var j 22 ↵
(debug) eval j ↵
22
(debug) address j ↵
0x111d10:0
```

and 2) when the address of an object is computed within an expression:

```
(debug) def–var f 23.4 ↵
(debug) eval &f ↵
0x00111d30
```

When a variable is migrated, **malloc** is called in the user's program to create space for the object. If **malloc** is not linked into the user's program, an error will result:

```
(debug) def–var x 2332 ↵
(debug) address x ↵
Error: The process does not contain a "malloc" memory
allocation routine.
(debug)
```

Migration has two main purposes. It allows pointers to what are usually debugger−resident objects to be assigned to variables in the target process:

```
(debug) def−var s "hello" ↵
(debug) eval argv[0] ↵
0xeffff768 -> "/pdd/pde/dtl/test/war"
(debug) assign argv[0] s ↵
(debug) eval argv[0] ↵
0x00111d10 -> "hello"
```

and it allows pointers to debugger−resident objects to be passed as arguments to functions in the user's program:

```
(debug) desc print_square ↵
extern void print_square (
      int *x);
(debug) def−var y 8 ↵
(debug) eval print_square(&y) ↵
64
```

Note that once an object is migrated to the target process it is never deallocated by the debugger. This is because the debugger does not know if the object is actively being used by that process.

See Also

Commands:	**address, convenience−variables, options:convenience−variables**
Topics:	debugger−variables, c-language, c++-language, fortran-language, pascal-language

name-resolution *Debugger Topic*

Resolving names in expressions.

Summary

When attempting to resolve a name used in an expression, the debugger tries to match the name, in order, to one of the following:

1. Program-defined names
   a. A visible name in the current module of the program being debugged.
   b. A global external name in the program being debugged.
2. Debugger-defined names
   a. A built-in type name.
   b. A debugger variable.
   c. A convenience variable.
   d. A register name.

The debugger uses the first match it encounters. For example, a program variable "foo" hides a debugger variable of the same name, and a debugger variable "$r2" hides register 2.

For details about how a particular language resolves a name used in an expression, see the appropriate language topic.

See Also

Commands:	**assign, define-variable, describe, evaluate, options:convenience-variables**
Topic:	c-builtin-types, c-language, c++-language, fortran-language, pascal-language
Types:	variable-name, expression

pascal-language

Debugger Topic

Using Pascal with Mxdb.

Summary

Pascal is one of the languages supported by the Mxdb debugger. Most expressions appearing in source code can be evaluated in the debugger. However, there are several deficiencies in the COFF debugging information available from the Green Hills Pascal compiler. See the release notice for details on known compiler and language processor restrictions.

Description

Identifiers are described using declarative syntax. Description of variables, literals, procedures, functions, and both standard and user–defined types is supported:

```
(debug) describe integer ↵
type integer;

(debug) desc a_var ↵
type a_var = integer;

(debug) desc double_ptr ↵
type double_ptr = ^double;

(debug) desc bool ↵
bool : boolean; external;

(debug) desc int1 ↵
int1 : integer; external;

(debug) desc color ↵
type color =
(
   red {0},
   orange {1},
   yellow {2},
   green {3},
   blue {4},
   indigo {5},
   violet {6}
);
```

Description of structured types such as arrays and records is also in declarative syntax:

```
(debug) desc array_type ↵
type array_type = array[0..2] of integer;

(debug) desc medical_rec ↵
type medical_rec = record
   id : integer;
   name : array [0..20] of char;
   age  : integer;
   height : double;
   weight : real;
   vaccinations : array [0..5] of boolean;
end;
```

Enumeration constants, however, are described using non-standard syntax:

```
(debug) desc paint ↵
paint : (
   red {0},
   orange {1},
   yellow {2},
   green {3},
   blue {4},
   indigo {5},
   violet {6}
); external;

(debug) desc indigo ↵
indigo, enumeration constant, value {5};

(debug) desc true ↵
true, boolean, value {1};
```

Functions and procedures are described using standard Pascal syntax. The use of both value and variable parameters is supported:

```
(debug) desc f1 ↵
   function f1 (
      var aaaaa : integer;
      b : float) : integer; external;

(debug) desc proc1 ↵
procedure proc1 (
   var a_var_param : integer;
   a : integer); external;
```

Expressions are described with the expression's type:

```
(debug) desc int1+5 ↵
integer

(debug) desc 'abcde' ↵
^char
```

Describing an expression containing a function or a procedure call will describe its return value. Since Pascal procedures have no return value, no return type should be printed when a procedure call is described (presently, "void" is returned):

```
(debug) desc f1(buf,2) ↵
integer

(debug) desc proc1(buf,1) ↵
void
```

Green Hills Pascal extensions for types float and double and for the visibility directives, static and external, are supported.

Mxdb also annotates constructs for which no debugging information is available, such as external variables and routines in vendor-supplied libraries:

```
(debug) desc printf ↵
function printf () : integer; external; (* $No_Debug_Info *)
(debug) desc errno ↵
errno : integer; external; (* $No_Debug_Info *)
```

Evaluation

Expressions containing variables, literals and function/procedure calls can be evaluated. For expressions of simple types, a single value is printed:

```
(debug) eval buf ↵
'start'
(debug) eval a_float ↵
1.5
```

Pointer evaluation prints the address contained in the pointer in hexadecimal format. A pointer may be dereferenced according to standard Pascal syntax:

```
(debug) desc a ↵
a : ^integer; external;
(debug) eval a ↵
0x0050f000
(debug) eval a^ ↵
1
```

Array values are printed showing each array element index and the value found at that index. Note that array elision is used in the following example. See the **options:elide-arrays** command for details.

```
(debug) desc int_arr ↵
int_arr : array [0..99,0..199] of integer; external;
(debug) eval int_arr ↵
{
   [0..98]    = {
      [0..199]   = 0
   }
   [99]       = {
      [0]         = 5
      [1..199]   = 0
   }
}
```

When a record is evaluated, the name of each of the record's fields is printed out, followed by its value:

```
(debug) desc a_rec ↵
a_rec : packed record
   int1 : packed array [0..2] of integer;
   rec1 : packed record
      int2 : integer;
      real1 : double;
   end;
   arr1 : packed array [0..9] of char;
end; external;

(debug) eval a_rec ↵
{
   int1       : {
      [0]         = 1
      [1]         = 2
      [2]         = 3
   }
   rec1       : {
      int2        : 5
      real1       : 5.5
    }
   arr1       : 'abc'
}
```

The standard arithmetic operations (+,−,/,*) and integer division operations (div,mod) are available for use in both expression evaluation and description. The relational (<,>,<=,>=,<>) and logical (not,and,or) operators can be used as well. Pascal's operator precedence rules and arithmetic conversion rules are adhered to for complex expressions:

```
(debug) eval an_int ↵
4
(debug) eval an_int+2*5 ↵
14
(debug) eval (an_int+1) mod 2 ↵
1
(debug) eval not true or (true and false) ↵
false
(debug) eval 'abc' > 'abcd' ↵
false
(debug) desc red <> blue ↵
boolean
(debug) desc 1.5 −1 ↵
double
```

Assignment

Assignment may be done using either the **assign** command or Pascal's assignment operator:

```
(debug) assign int1 12 ↵
(debug) eval int1 ↵
12
(debug) eval int1 := int2 ↵
135
(debug) eval int1 ↵
135
(debug) assign a_rec.rec1.int2 5 ↵
(debug) assign bool not bool ↵
(debug) assign p_arr p_arr2 ↵
(debug) eval p_arr = p_arr2 ↵
true
```

String literals may be assigned to character arrays using the **assign** command. If a string literal is longer than the targeted array, a warning will result:

```
(debug) desc p_arr ↵
p_arr : array [0..4] of char; external;
(debug) assign p_arr 'string literal' ↵
Warning: String literal is longer than targetted character
array.
Only 5 characters assigned.
(debug) eval p_arr ↵
'strin'
```

Options

All of the debugger's language−specific options for controlling expression evaluation are supported under Pascal. For additional information on any of these options, refer to the **c-p:option-status** command.

See Also

Commands: **address, assign, describe, evaluate, c-p:option-status, options:elide-arrays, options:pointer-dereference-level, options:string-display, options:string-display-limit, options:language**

Topics: fortran-language, c++-language, c-language

scopes *Debugger Topic*

Affecting programming language name visibility.

Summary

A scope is a location where some programming language names are visible and others are not. A named scope can be a module, program block, or routine. In the C language, if you declare names within a pair of braces, the range of the pair of braces is a scope (unnamed); the names declared within the braces are not visible outside that scope.

Description

Scopes influence how programming language names are resolved. In the programming languages that Mxdb currently supports, scopes are lexical in nature; i.e., local names will be found before more widely visible names.

One-level source files are really just scopes because each source file defines a certain set of names that will be visible in that file. This level is usually described as the module level.

Within modules, routines are visible. Routines define other name visibility scopes within themselves (including other routines in languages such as Pascal).

To specify a file where the debugger is looking for a scope, use the simple filename without the .c, .f, or .p extension, preceded by a backslash (to indicate that this is a module name).

To specify a routine you usually just use the name of the routine. In some languages, there are cases where routine names are not visible outside of their associated file or module. One example of this is "static" functions in the C language. To access these local functions from another module, you must specify both the module name and the routine name.

Examples

To indicate file **foo.c**, specify module **foo**:

```
\foo
```

If file **foo.c** contains a function named **func** that is static, you can access that function from another module:

```
\foo\func
```

To indicate the external routine (system call) **sleep**:

```
sleep
```

See Also

Commands: **file, position, routine**

source-files

Debugger Topic

Viewing source files.

Summary

Mxdb has three commands for viewing your program's source code: **view**, **list**, and **find**. With these you can do the following tasks:

Look at the current source text line, plus surrounding lines:	**view**
Look at the next screen:	**list**
Reposition and look at the next screen:	**view, down**
Reposition 3 screens back and view source:	**view, up 3**
See only the current line:	**list current**
Find a line containing a pattern:	**find** *<pattern>*
Find several occurrences of a pattern:	**fi** *<pattern>*, **count** *<n>*
Search backward for a pattern:	**find** *<pattern>*, **back**
Search backward and wrap if not found:	**fi** *<pat>*, **back, wrap**
Reposition and see the new current line, plus lines before and after:	**view** *<number>*
See a range of source lines without changing the current source position:	**list** *<n1>* *<n2>*

Description

The **view** command displays a window of source text centered on your current source position. Generally, when your process stops, you will want to type **view** immediately (unless you are using the graphical interface, where that is done automatically). The up and down keywords move your source text window either up toward the beginning of the file or down toward the end of the file.

The **list** command is used when you want to print either a single line or some other region of code without changing your current source position.

The **find** command searches for regular expressions in your source code. It is usually invoked as "**find** <pattern>."

To view the code in another source file, you must reset your position. To do this, use the **file** macro and specify the source file's filename or the filename without the usual language extension. For example, to view the source code for file **foo.c**, type either of these commands:

```
(debug) file foo ↵
(debug) file foo.c ↵
```

When your source code or external legends file (debugging information files) resides in more than one directory in the file system, you must tell the debugger where to search for your files. Every **debug** command invocation that does not reload the current executable resets the directory list to look in the current directory "." and in the directory prefix part of the executable file that you specified. The **directory-list** command displays or resets the search path used to find source files.

See Also

Commands: **debug, directory-list, file, find, list, view**
Topic: scopes

stack-frames

Debugger Topic

Accessing stack frames.

Summary

To access a frame in your program's call stack, use the following commands:

To view the call stack:	**walkback**
To view each routine's parameters on the call stack:	**walk, arg**
To view each routine's local variables on the call stack:	**walk, local**
To view five frames, including the current one	**walk 5**
To view the topmost five frames:	**walk 5, f top**
To view the bottommost five frames:	**walk, f b-4**
To view the current frame's parameters and local variables:	**walk 1, arg, local**
To position to another frame:	**frame** *<number>*
To position to the top frame:	**frame top**
To position to oldest frame:	**frame bottom**

Description

Mxdb lets you access any frame that is linked into the program's call stack. The **walkback** command can be used to view the stack. Each frame on the stack is displayed on a separate line with a number identifying the frame (0 is the most recent; higher numbers indicate older frames). The number indicates the number of frames away from the top frame that the current frame resides. A short description of the source position and/or instructional position associated with that frame is also displayed.

When your process stops, you are positioned to the top frame of the stack. To move to some other frame, look at the stack frames displayed by the **walkback** command and type "frame *<f>*" where *<f>* is the number of the stack frame that you are interested in viewing. To quickly get to the newest or oldest stack frame substitute the tag "top" to get to the newest or "bottom" to get to the oldest. Now you can access variables that are local to the procedure and/or view the source code associated with your new position.

The **walkback** command has options for displaying additional information. To view the arguments for each stack frame, use the **arguments** argument. To see the active local variables for each stack frame, use the **locals** argument.

When the graphical user interface is active, you may view and traverse the stack with the Stack Frames viewer. Use the Stack Frames option in the View menu to create one.

See Also

Commands: **frame, position, walkback**

watchpoints–(88k)

Debugger Topic

Watching memory locations.

Summary

Because the MC 88000 is a pipelined architecture, several instructions may be in various stages of execution at any particular time. In another mode of operation, processor serialization, each instruction completes execution before the next instruction begins execution.

Processor serialization is an atypical mode of operation for the MC 88000. Mxdb automatically sets this mode of execution when a watchpoint is defined so the machine will stop at exactly the instruction that is modifying a watched memory region. This behavior benefits most users.

The rest of this topic discusses the pipeline architecture and how serialization can be enabled and disabled. This is only of interest to users who have time-critical applications that cannot execute correctly in serial execution mode.

The data unit pipeline of the MC 88100 has three stages, so up to three memory–related instructions can be in progress at once.

When Mxdb is watching a memory location and the MC 88100 is not serialized, the instruction that caused the memory modification can be ***any*** of the three possible instructions in the data pipeline. The instruction that caused the modification may or may not be one of the instructions denoted by the instruction pointers where the process finally halted and notified the debugger of the modification.

These data instructions are also interleaved with other instructions that do not "touch" the data unit so the process can even be in a different routine when it halts if one of the intervening instructions was a BSR/JSR.

If you have trouble locating the instruction that caused the watchpoint to be modified, you may explicitly serialize the processor using the procedure discussed below. This will cause the "sxip" to denote the instruction that caused the memory modification.

A macro, **serialize–processor**, is provided in the Mxdb macros directory (in the file **88k.cp**) that will serialize the processor for you. To do this, type:

```
(debug) include /usr/opt/mxdb/macros/88k.cp ↵
(debug) serialize–processor „ "ser" should be a unique prefix ↵
```

When you wish to resume pipelined execution mode then type:

```
(debug) ser, deactivate ↵
```

Note that this macro only serializes the machine when your process is executing; it should not directly impact the execution of other users' processes.

When the graphical user interface is active, you may access the macro **serialize−processor** with the Serialize Processor toggle button, located in the Registers viewer.

See Also

Commands: **watch−memory**, **watch−reference**, **watchpoint−print**
Topic: events

End of Chapter

Arguments

label	Label of the button to be displayed or modified.
command	A new command sequence that will be interpreted by the debugger when the button is pushed.
position	Position of the button to be displayed or modified.
pane	This keyword specifies which pane will be searched for button information.
all	If this argument is specified, information for all buttons in all button panes is displayed.
append-selection	If you specify this keyword, the current text selection, if any exists, is appended to the end of the command. The default behavior is not to append the current text selection.
append-required-selection	If you specify this keyword, the current text selection is appended to the end of the command. Since a selection is required, an error occurs if no text is currently selected when the button is pressed.
suppress-echo	This keyword controls whether the command is echoed to the Message Pane when the button is pressed. If you modify a button so that this keyword's value is "no" (the default value), the command associated with that button press will be echoed after the prompt in the Message Pane. If you modify a button so that this keyword's value is "yes" (the implied value), the command associated with that button press (and the prompt) will not be echoed in the Message Pane. However, any output will still be output to the Message Pane. Note that the associated actions will still be recorded in a log file if the **c-p:log** command has been used.

Examples

The following command displays information for all buttons:

```
(g-i) button-status, all ↵
```

This command displays information for a button labeled "walkback" in the default button pane:

```
(g-i) button-status "walkback" ↵
```

The following command displays information for the fourth button from the left of the default button pane:

```
(g-i) button-status, position 4 ↵
```

This command changes the command associated with the button "foo" in the button pane named "MyButtons":

```
(g-i) button-status foo, pane MyButtons, command {write Foo!} ↵
```

This command writes the information displayed for all buttons to a file named **my−button−definitions**. You could use this file as an include file at a later date.

```
(g-i) redirect−output {button-status, all} my−button−definitions ↵
```

See Also

Command: **button-pane-status, define-button**

button-pane-status *Graphical-Interface Command*

Displays button pane information.

Summary

Displays button pane information.

Syntax

button-pane-status *[name] [position]*

where:

name	Name of button pane to be displayed
position	Position of the button pane to be displayed

Description

The **button-pane-status** command displays information about button panes. The information is printed in a form that is syntactically valid for defining the same set of button panes at a later time. Therefore, this command can be redirected to a file and included at a later time in order to recreate the same button panes.

You can specify the button pane that you want to display with either the **name** or **position** arguments. The **name** and **position** arguments cannot be specified concurrently.

By default, there are three button panes. They are named "ProcessButtons," "MachineButtons," and "StackButtons." The "StackButtons" pane is the default button pane for button-related operations, such as **define-button**, **button-status**, and **delete-button**.

The "MachineButtons" pane is hidden from view initially, but may be made visible by toggling the "Machine Buttons" entry in the "Button Panes" submenu of the "View" menu. The other button panes may be hidden/shown through their respective entries in the "Button Panes" submenu of the "View" menu. The **button-pane-status** command currently does not include the visibility status of panes.

Arguments

name	Name of the button pane to be displayed.
position	Position of the button pane to be displayed. The position is relative to the other button panes in the main window of the GUI. The first is position 1.

Examples

The following command displays information for all button panes:

```
(g-i) button-pane-status ↵
```

This command displays information for a button pane named "StackButtons":

```
(g-i) button-pane-status "StackButtons" ↵
```

The following command displays information for the second button pane:

```
(g-i) button-pane-status, position 2 ↵
```

See Also

Command: **button-status, define-button, define−button−pane**

clear-messages

Graphical-Interface Command

Clears the Message Pane.

Summary

Erases all the saved lines in the Message Pane and clears the pane.

Syntax

clear-messages

Description

The **clear-messages** command erases all the lines stored by the Message Pane. This command also clears the display in the Message Pane.

Arguments

None

Examples

These equivalent commands clear all messages:

```
(g-i) clear-messages ↵
(g-i) c-m ↵
```

See Also

Command: **option-status (message-history** option)

define-button *Graphical-Interface Command*

Creates and sets the attributes of a button.

Summary

Creates and sets the attributes of a button in a button pane.

Syntax

define-button *label command [position]* **,pane ,append-selection ,append-required-selection ,suppress-echo**

where:

label	Label to be displayed on the button
command	Command sequence to be interpreted by the debugger when the button is pushed
position	Button position (0 <= position <= last button position+1)
pane	The name of a button pane; default is "StackButtons"
append-selection	Yes or no
append-required-selection	Yes or no
suppress-echo	Yes or no

Description

The **define-button** command creates a button in the specified button pane (the default button pane is StackButtons). The buttons and their associated command sequences serve as short cuts to entering commands manually. Since clicking a button is functionally equivalent to typing a command sequence into the Command Line Pane at the keyboard, by creating a button and associating it with a frequently used command, the user can save time and effort by simply pushing the button instead of re-entering the same command. Thus, buttons can be viewed as a limited graphical macro facility.

The names of buttons for the purposes of X resource specification are of the form *<pane>-<button-label>*-Button. For example, the continue button in the ProcessButtons button pane is referred to as ProcessButtons-continue-Button. So if you wish to change its background color to green, the resource may be specified like this:

Mxdb*ProcessButtons-continue-Button.background: green

The only exception to this rule is the Interrupt button. It is still referred to simply as Interrupt.

Arguments

label	Label to be displayed on the button.
command	Command sequence to be interpreted by the debugger when the button is pushed.
position	The position of the button to be placed in the specified button pane.
pane	Unless this argument is specified, the default button pane, "StackButtons," is used.
append-selection	If you specify this keyword, the current text selection, if any exists, is appended to the end of the command. The default behavior is not to append the current text selection.
append-required-selection	If you specify this keyword, the current text selection is appended to the end of the command. Since a selection is required, an error occurs if no text is currently selected when the button is pressed.
suppress-echo	This keyword controls whether the command is echoed to the Message Pane when the button is pressed. When a button is created, if this keyword has the value "no" (the default), the command associated with that button press will be echoed after the prompt in the Message Pane. If this keyword has the value "yes" (the implied value) when a button is defined, the command associated with that button press (and the prompt) will not be echoed in the Message Pane. However, any output will still be output to the Message Pane. Note that the associated actions will still be recorded in a log file if the **c-p:log** command has been used.

Examples

This example defines a button named "exit" at position 1 in the default button pane:

```
(g-i) define-button exit {write So long!; quit} 1 ↵
```

The following command creates a button labeled "debug again" that, when pushed, invokes the "debug, again" command at position 3. Note that this command will be entered automatically, but will not be added to the command history list. Generally, any button-entered command should not be added to the command history list, since it can be readily duplicated.

```
(g-i) define-button "debug again" {debug, again} 3 ↵
```

The next example shows the simplest way to create a button. The command creates a button labeled "selection" at the bottom-most position in the panel.

```
(g-i) def-b selection {'selection} ↵
```

The example below creates a button labeled "next" that is placed at the left-most position.

```
(g-i) def-button next {step, here} 1 ↵
```

See Also

Command: **delete-button, button-status, define-button-pane**
option-status (**command-history** option)

define-button-pane

Graphical-Interface Command

Defines a button pane.

Summary

Defines a button pane.

Syntax

define-button-pane *name label [position]*

where:

name	Name of the button pane to be defined
label	Label to be displayed as a toggle button in the "Button Panes" submenu of the "View" menu
position	Position of the button pane to be defined

Description

The **define-button-pane** command defines a button pane. The button pane must have a name which is used in other button and button pane related commands that require a pane specification. The button pane also requires a label, which is placed in a toggle button in the "Button Panes" submenu of the "View" menu. The toggle button turns the associated button pane off and on.

If no position is specified, the position defaults to be the last position.

Three button panes are initially defined: "ProcessButtons", "MachineButtons" and "StackButtons". The second is hidden by default and can be made visible with its associated toggle button in the "Button Panes" submenu of the "View" menu.

Arguments

name	Name of the button pane to be defined. References to panes in button and button pane related commands expect this name as an identifier.
label	Label to be displayed as a toggle button in the "Button Panes" submenu of the "View" menu.
position	Position of the button pane to be defined. The position is relative to the other button panes in the main window of the GUI. The first is position 1. The last may be specified as position 0.

Examples

This example defines a button pane named "MyButtons" as the second button pane:

```
(g-i) define-button-pane "MyButtons" "MyButtons", position 2 ↵
```

This next example defines a button pane named "YourButtons" as the last button pane:

```
(g-i) def-but-pane "YourButtons" "YourButtons", position 0 ↵
```

See Also

Command: **button-pane-status, button-status, define-button**

delete-button *Graphical-Interface Command*

Deletes a button in the specified button pane.

Summary

Deletes a button in the specified button pane.

Syntax

delete-button *[label]* **,pane ,position ,all**

where:

label	Label displayed on the button
pane	The name of a button pane
position	Position of button to be deleted (0 <= position <= last button position)
all	Yes or no

Description

The **delete-button** command deletes a button from the specified button pane. The button to be deleted may be specified either (or both) in terms of its position or its label. If both are specified, the value of the label and the position for the specified button must match. If neither is specified, the last button is deleted by default.

Due to the new implementation of the GUI, the "Interrupt" button in the "ProcessButtons" button pane cannot be deleted. It is a required status indicator that, when pressed, interrupts a debugger operation or the target process (if it is executing).

Arguments

label	The label of the button or button separator to be deleted.
pane	The name of a button pane. If this value is not specified, the default button pane, "StackButtons," is used.
position	The position of the button to be deleted from the specified button pane. If this value is not specified, then the default value of 0 is used to imply the position of the bottom-most button in the column.
all	Delete all buttons in the specified button pane.

Examples

The following example deletes a button by position:

```
(g-i) delete-button, position 3 ↵
```

This example deletes a button by label:

```
(g-i) delete-button next ↵
```

This example deletes all buttons in the default button pane:

```
(g-i) del-button, all ↵
```

This example deletes a button at the last position:

```
(g-i) del-but ↵
```

See Also

Command: **define-button**

delete-button-pane

Graphical-Interface Command

Deletes a button pane.

Summary

Deletes a button pane.

Syntax

delete-button-pane *[name] [position]*

where:

name	The name of the button pane to be deleted
position	Position of the button pane to be deleted

Description

The **delete-button-pane** command deletes a button pane. The button pane to be deleted may be specified by its name or its position. The name and position arguments cannot be set concurrently.

The buttons contained in the button pane are deleted.

The toggle button entry associated with the specified button pane in the "Button Panes" submenu of the "View" menu is also deleted.

Due to the new implementation of the GUI, the "ProcessButtons" button pane and its "Interrupt" button cannot be deleted. If you wish to hide the "ProcessButtons" pane, simply use the "Process Buttons" toggle button entry of the "Button Panes" submenu of the "View" menu to turn it off and on. Every other button within "ProcessButtons" may be deleted.

Arguments

name	The name of the button pane to be deleted.
position	The position of the button pane to be deleted. The position is relative to the other button panes in the main window of the GUI. The first is position 1. Even if a button pane is hidden, it still occupies a position. To check the positon, use the **button-pane-status**.

Examples

The following example deletes a button pane by position:

```
(g-i) delete-button-pane, position 3 ↵
```

This example deletes a button pane by name:

```
(g-i) delete-button-pane "MachineButtons" ↵
```

See Also

Command: **button-pane-status, delete-button**

graphics-available *Graphical-Interface Command*

Writes "true" if Mxdb's graphical user interface is active.

Summary

Writes the value "true" if Mxdb's graphical user interface is active.

Syntax

graphics-available

Description

The **graphics-available** command writes the value "true" with a New Line appended to the standard output if the graphical user interface of Mxdb is active. If the graphical user interface is not active, nothing is output.

This command is useful for writing macros that behave differently when the graphical user interface is enabled.

Arguments

None

Examples

This example assumes that Mxdb was invoked with the −g option.

```
(debug) graphics-available ↵
true
```

selection *Graphical-Interface Command*

Writes the value of selected text,

Summary

Writes the value of any selected text from any window on the screen.

Syntax

selection ,required

where:

required The value "yes" or "no"

Description

The **selection** command writes the value of any selected text from any window on the screen. This command is generally used in a backquoted context to supply arguments to other debugger commands.

Arguments

required A selection must exist or an error message is issued.

Examples

If the currently selected text is the expression "1 + 2 * 3," the following application is possible:

```
(g-i) selection ↵
1 + 2 * 3
(g-i) evaluate '{selection, required} ↵
7
```

synchronize-display *Graphical-Interface Command*

Centers the Source Pane around the current debugger position.

Summary

Centers the Source Pane around the current debugger position.

Syntax

synchronize-display

Description

The **synchronize-display** command redisplays the Source Pane, with the current debugger position located at the middle of the pane. Use this command to return quickly to the debugger position after you scroll through the Source Pane.

This command is available in the Source Pane popup cascade menu.

Arguments

None

Examples

These equivalent commands move the Source Pane to the debugger position:

```
(g-i) synchronize-display ↵
(g-i) s-d ↵
```

xhelp *Graphical-Interface Command*

Execute the xhelp-view program.

Summary

Executes the xhelp-view program.

Syntax

xhelp ,display

where:

display The display on which the help system will appear

Examples

```
xhelp, display bigbootie:0.0
xhelp
```

Description

The **xhelp** command executes the xhelp-view program, which is a driver for the graphical user interface help system. If you are a command-line Mxdb user, this command allows you to use the graphical user interface help system if you have access to an X Window System server running on a display machine. This command is also accessible from the graphical user interface Command Line Pane.

Arguments

display Specify the display on which the help system window will appear. The implied value is "unix:0.0," the machine from which you invoke the command.

Examples

This example specifies that the help system window should be created on screen 0 of a machine named Bigbootie:

```
(g-i) xhelp, display bigbootie:0.0 ↵
```

End of Chapter

Description

Following is a list of Mxdb options commands; capital letters indicate the shortest unique abbreviation:

Command Name	Action
Bit-format	Display or set bit display format.
CHaracter-format	Display or set character display format.
Command-History	Display or set the number of commands saved in the command line history mechanism.
Convenience-Variables	Display or set whether a debugger variable is created whenever an expression is evaluated via the evaluate command.
Convenience-Variables-Limit	Display or set how many convenience variables the debugger will remember.
ELide-arrays	Display or set whether same-valued array elements are elided.
Floating-point-format	Display or set floating-point display format.
Language	Display or set the expression evaluation language.
Message-History	Display or set the maximum number of lines of text saved in the Message Pane.
Mismatched-Legends-allowed	Display or set whether mismatched external debugging information files are allowed when debugging a process.
Pager-Lines	Display or set the number of lines used by the CP paging facility (page and help).
POinter-dereference-level	Display or set how many times a top-level pointer will be automatically dereferenced and displayed by the debugger.

Command Name	Action
SIgned-Character-format	Display or set signed-character display format.
Signed-Integer-format	Display or set signed-integer display format.
SOurce-lines	Display or set the number of lines used by the screen-related source commands (list and view).
Stop-Commands	Display or set commands to execute when the target process stops for any reason.
String-Display	Display or set whether string-like objects are automatically displayed in a string-like or an array-like manner.
String-Display-Limit	Display or set the number of characters that will be displayed in a string-like manner before elision occurs.
UNKnown-type-format	Display or set unknown-type display format.
Unsigned-Character-format	Display or set unsigned-character display format.
Unsigned-Integer-format	Display or set unsigned-integer display format.
Windowed-terminal-emulator	Creates a windowed terminal emulator for a live debugged process, even when the graphical user interface is not active.

The initial values for these commands are as follows:

Command Name	Initial Value
bit-format	Binary
character-format	ASCII
command-history	100 (0 if GUI is disabled)
convenience-variables	no
convenience-variables-limit	50
elide-arrays	yes

Command Name	Initial Value
floating-point-format	IEEE-float
language	Varies per debugging position
message-history	1000 (0 if the graphical user interface is disabled)
mismatched-legends-allowed	no
pager-lines	Value of the LINES environment variable (23 if LINES is not set)
pointer-dereference-level	0
signed-character-format	ASCII
signed-integer-format	Decimal
source-lines	Half the value of the LINES environment variable (10 if LINES is not set)
stop-commands	No command
string-display	yes
string-display-limit	0
unknown-type-format	Hexadecimal
unsigned-character-format	ASCII
unsigned-integer-format	Unsigned decimal

See Also

Commands: **c-p:option-status, debug:assign, debug:evaluate, debug:list, debug:view**
Types: debug:format, debug:language

bit-format *Options Command*

Displays or sets the bit display format.

Summary

Displays or sets the bit display format.

Syntax

bit-format *[format]*

where:

format	One of the following:

ascii	binary
decimal	hexadecimal
ieee-double	ieee-float
ieee-single	octal
string	symbolic
system-error	unsigned-decimal

Description

Displays or sets the bit display format.

Examples

```
(options) bit „write display format for the bit type ⤵
binary
(options) bit hex „set display format for the bit type ⤵
(options)
```

See Also

Type: debug:format

character-format

Options Command

Displays or sets the character display format.

Summary

Displays or sets the character display format.

Syntax

character-format *[format]*

where:

format	One of the following:

ascii	binary
decimal	hexadecimal
ieee-double	ieee-float
ieee-single	octal
string	symbolic
system-error	unsigned-decimal

Description

Displays or sets the character display format.

Examples

```
(options) character „write display format for the character type ↵
ascii
(options) character string „set display format for the character type ↵
(options)
```

See Also

Type: debug:format

command-history *Options Command*

Displays or sets the maximum number of commands saved in the history mechanism.

Summary

Displays or sets the maximum number of commands saved in the command line history mechanism (used in graphical user interface).

Syntax

command-history *[limit]*

where:

limit	An integer greater than or equal to 0

Description

When the graphical user interface is not active, this option's value is undefined and is displayed as zero. Any attempts to set the option in this situation will raise an error.

Examples

```
(options) command-history „write maximum number of commands ↵
100
(options) command-history 150 „set maximum number of commands ↵
(options)
```

See Also

Command: **message-history**

convenience-variables *Options Command*

Displays or sets whether to create a debugger variable when an expression is evaluated.

Summary

Displays or sets whether a debugger variable will be created whenever an expression is evaluated with the **evaluate** command.

Syntax

convenience-variables *[on]*

where:

on	The value "yes" or "no"

Description

When convenience variable generation is turned on (the optional argument is set to "yes"), a debugger variable is created whenever the user evaluates an expression with the **evaluate** command. These variables will be named according to the following convention:

$<*variable-number*>

where <*variable-number*> is 1 at the beginning of the debugging session and increases by 1 for each convenience variable created. Creation of debugger variables via the **define-variable** command has no effect on the convenience variable count; convenience variables do not affect the **variable** command and cannot be removed by using the **delete-variable** command. However, any restrictions applying to debugger variables apply to convenience variables as well.

The most recently created convenience variable has the symbolic notation "$$."

Only the last *n* convenience variable values are remembered by the debugger, where *n* is a user-settable value; see the **options:convenience-variables-limit** command.

You can use convenience variable names in expressions. However, convenience variable names may be hidden by objects with the same name in a program, by regular debugger variables, or by built-in type names.

Examples

```
(debug) opt conv-var yes „turn convenience variable generation on ⤵
(debug) evaluate foo_ptr ⤵
$1      = 0x12345678
(debug) „ reference value of $1 via $$ notation ⤵
(debug) evaluate *$$ ⤵
$2      = {
     i      = 1
     fp     = 0x01234567
}
(debug) „ Convenience variable names are still available when ⤵
(debug) „ generation is off ⤵
(debug) opt conv-var no ⤵
(debug) evaluate *$2-->fp ⤵
{
     i      = 2
     fp     = 0x00000000
}
```

See Also

Command: **convenience-variables-limit, debugger:convenience-variables**
Topic: debugger-variables, migration

convenience-variables-limit *Options Command*

Displays or sets how many convenience variables to remember.

Summary

Displays or sets how many convenience variables the debugger will remember.

Syntax

convenience-variables-limit *[limit]*

where:

limit	An integer greater than or equal to 1

Description

The **convenience-variables-limit** command determines the number of convenience variables remembered by the debugger. If fewer than the **limit** argument's number of convenience variables have been created, all convenience variables are active. Once the number of active variables reaches the value of the **limit** argument, the oldest convenience variable in the list is deleted each time a new convenience variable is created.

Initially, the limit is set to 50. If the user sets the limit to a value less than the current number of active variables, the oldest variables are deleted until the new limit is reached. If the user sets the limit to a value larger than the current limit, no existing convenience variables are deleted.

Examples

```
(debug) evaluate expr ↵
$25     = 2
(debug) opt conv-var-lim ↵
10
(debug) „ limit is 10, so active variables are $16 through $25 ↵
(debug) opt c-v-l 5 ↵
(debug) „ active variables are now $21 through $25 ↵
(debug) opt conv-var-li 6 ↵
(debug) „ no values lost, active variables are still $21 through $25 ↵
```

See Also

Command: **convenience-variables**

elide-arrays *Options Command*

Displays or sets whether same-valued array elements are elided.

Summary

Displays or sets whether same-valued array elements are elided.

Syntax

elide-arrays *[on]*

where:

on	The value "yes" or "no"

Description

When the optional argument is set to "yes," consecutive array elements that have the same value are displayed in a compressed manner. Compression is attained through subscript elision for the same-valued array element range. Ellipsis, a sequence of consecutive periods, is used to denote that value printing compression has been done. The initial value of this option is "yes."

If the **string-display** option's value is "yes," character arrays are printed as strings and no elision is performed.

This option is not supported for all languages; a language that does not support this feature will ignore the option setting. For information about support in a particular language, see the topic for that language.

This option may affect the output from the **evaluate**, **walkback**, **watchpoint-print**, and **variable** debugger commands.

Examples

For example, the array **arr** has the following description:

```
int arr[4] = { 1, 0, 0, 0 };

(debug) opt elide-arrays ↵
yes
(debug) eval arr ↵
{
   [0]        = 1
   [1..3]     = 0
}
(debug)
```

See Also

Commands:	**string-display, string-display-limit**
Topics:	c-language, c++-language, fortran-language, pascal-language

floating-point-format *Options Command*

Displays or sets the floating-point display format.

Summary

Displays or sets the floating-point display format.

Syntax

floating-point-format *[format]*

where:

format	One of the following:

ascii	binary
decimal	hexadecimal
ieee-double	ieee-float
ieee-single	octal
string	symbolic
system-error	unsigned-decimal

Description

Displays or sets the floating-point display format.

Examples

```
(options) float „write display format ⤵
ieee-float
(options) float ieee-double „set display format ⤵
(options)
```

See Also

Type: debug:format

language *Options Command*

Displays or sets the expression evaluation language.

Summary

Displays or sets the expression evaluation language.

Syntax

language *[language]*

where:

language One of these languages: c, c++, fortran, pascal, or icobol

Description

Note that any modification stays in effect until the debugger's position changes. The debugger position is changed when the **position** or **view** command is supplied with arguments or the debugged process is continued (and it stops).

Examples

```
(options) language „display the current expression evaluation language ↵
c
(options) language fort „set the current expression evaluation language ↵
(options)
```

See Also

Type: debug:language

stop-commands *Options Command*

Displays or sets commands to execute when the target process stops.

Summary

Displays or sets commands to execute when the target process stops for any reason.

Syntax

stop-commands *[new-commands]*

where:

new-commands A valid command sequence

Description

The **stop-commands** command lets you conveniently specify a sequence of commands that will be executed every time that the target process stops. Supplying a **new-commands** argument replaces any previous commands.

To unset any stop commands, supply the value "{}":

opt stop {}

Examples

```
(options) opt stop-commands {wri hi} ↵
(options) opt stop-commands ↵
wri hi
(options) „ Append to the current commands; Notice that two ↵
(options) „ backquotes must be used to invoke the command ↵
(options) „ inside the command−sequence argument braces ↵
(options) opt stop−commands {"{opt stop−commands}; write bye} ↵
(options) opt stop−commands ↵
wri hi; write bye
(options) „ Append to the current commands ↵
(options) „ via the primitive command ↵
(options) opt stop−commands {"opt:stop−commands; write again} ↵
(options) opt stop−commands ↵
wri hi; write bye; write again
(options)
```

See Also

Type: debug:command-sequence

string-display *Options Command*

Displays or sets whether string-like objects are automatically displayed in a string-like or an array-like manner.

Summary

Displays or sets whether string-like objects are automatically displayed in a string-like or an array-like manner.

Syntax

string-display *[on]*

where:

on	The value "yes" or "no"

Description

When the **on** argument has a value of "yes," string-like objects such as character arrays are displayed in a string-like or an array-like manner. The current expression evaluation language determines the exact format. The initial value of this option is "yes."

By setting the **string-display-limit** option, you can control the number of characters in the string.

When the value of **on** is "no," character arrays will be displayed in an array-like manner that will be affected by the **elide-arrays** option.

This option is not supported for all languages; a language that does not support this feature will ignore the option setting. For information about support in a particular language, see the topic for that language.

This option may affect the output from the **evaluate**, **walkback**, **watchpoint-print**, and **variable** debugger commands.

Examples

The following examples evaluate the character pointer **strp** and the C character array **str**, described here:

```
char str[2000] = "abcdef"
char *strp = "abcdef"

(debug) opt string-display ↵
yes
(debug) eval str ↵
"abcdef"
(debug) eval strp ↵
0x000101b0 -> "abcdef"
(debug) opt string-display no ↵
(debug) eval str ↵
{
   [0]        = 'a'
   [1]        = 'b'
   [2]        = 'c'
   [3]        = 'd'
   [4]        = 'e'
   [5]        = 'f'

   [6..1999] = '\000'
}
(debug) eval strp ↵
0x000101b0
(debug)
```

See Also

Commands: **elide-arrays, string-display-limit x**
Topics: c-language, c++-language, fortran-language, pascal-language

string-display-limit *Options Command*

Displays or sets the number of characters displayed in a string-like manner.

Summary

Displays or sets the number of characters that will be displayed in a string-like manner.

Syntax

string-display-limit *[limit]*

where:

limit	An integer greater than or equal to 0

Description

This option defines an upper limit for the number of characters that will be displayed in a string-like manner. If the **limit** argument is 0, there is no upper limit and the string display terminates in a language-specific manner (such as when a null byte is encountered in a C string).

When a string is longer than the set limit, trailing periods outside of the string delimiters are appended. The display of a C character pointer (if the pointer is valid) shows both the pointer value and the associated string.

This option is not supported for all languages; a language that does not support this feature will ignore the option settting. For information about support in a particular language, see the topic for that language.

This option may affect the output from the **evaluate**, **walkback**, **watchpoint-print**, and **variable** debugger commands.

Examples

The following examples evaluate the character pointer **strp** and the character array **str**, described here:

```
char str[2000] = "abcdefghijklmnopqrstuvwxyz"
char *strp = "abcdef"
```

```
(debug) opt string-display-limit ↵
0
(debug) eval str ↵
"abcdefghijklmnopqrstuvwxyz"
(debug) eval strp ↵
0x000101b0 -> "abcdef"
(debug) opt string-display-limit 5 ↵
(debug) eval str ↵
"abcde"...
(debug) eval strp ↵
0x000101b0 -> "abcde"...
(debug)
```

See Also

Commands:	**elide-arrays, string-display**
Topics:	c-language, c++-language, fortran-language, pascal-language

unknown-type-format *Options Command*

Displays or sets the unknown-type display format.

Summary

Displays or sets the unknown-type display format.

Syntax

unknown-type-format *[format]*

where:

format One of the following:

ascii	binary
decimal	hexadecimal
ieee-double	ieee-float
ieee-single	octal
string	symbolic
system-error	unsigned-decimal

Description

Displays or sets the unknown-type display format.

Examples

```
(options) unknown „write display format ⤵
hexadecimal
(options) unknown binary „set display format ⤵
(options)
```

See Also

Type: debug:format

unsigned-character-format *Options Command*

Displays or sets the unsigned-character display format.

Summary

Displays or sets the unsigned-character display format.

Syntax

unsigned-character-format *[format]*

where:

format	One of the following:

ascii	binary
decimal	hexadecimal
ieee-double	ieee-float
ieee-single	octal
string	symbolic
system-error	unsigned-decimal

Description

Displays or sets the unsigned-character display format.

Examples

```
(options) un-char „write display format ⤵
ascii
(options) un-char string „set display format ⤵
(options)
```

See Also

Type: debug:format

unsigned-integer-format *Options Command*

Displays or sets the unsigned-integer display format.

Summary

Displays or sets the unsigned-integer display format.

Syntax

unsigned-integer-format *[format]*

where:

format	One of the following:

ascii	binary
decimal	hexadecimal
ieee-double	ieee-float
ieee-single	octal
string	symbolic
system-error	unsigned-decimal

Description

Displays or sets the unsigned-integer display format.

Examples

```
(options) un-int „write display format ↵
ascii
(options) un-int binary „set display format ↵
(options)
```

See Also

Type: debug:format

windowed-terminal-emulator *Options Command*

Creates a windowed terminal emulator for a live debugged process even when the graphical user interface is not active.

Summary

Creates a windowed terminal emulator for a live debugged process, even when the graphical user interface is not active.

Syntax

windowed-terminal-emulator *[new-terminal-emulator]*

where:

new-terminal-emulator	Pathname for windowed terminal emulator

Description

A windowed terminal emulator is a program that is expected to create a window to which the input, output and error streams for a debugged process are bound by mxdb. This avoids the problems associated with intermingling the input and output of the debugger and debugged process. (This also makes debugging curses–based applications much easier.)

The terminal emulator also creates a new controlling tty for the debugged process. Thus interrupts issued in this execution window are only sent to the debugged process. (Note that interrupting the debugged process will usually "wake up" the debugger unless it has been told to ignore these signals.)

The windowed terminal emulator option accepts the pathname of the terminal emulator program. (Typically xterm or mterm are used.) If this user–specified pathname fails to execute properly when starting a target process then the default terminal emulator pathname, **/usr/bin/X11/mterm**, is tried as well automatically.

When the graphical user interface is active, it resets this option value if the Mxdb.executionWindow X resource has been set. Otherwise the user–specified option is used. Since a windowed terminal emulator is requiredfor the graphical user interface, the default terminal emulator is used if both of the previous methods are not used to set the terminal emulator pathname.

When this option is set and no terminal emulator process currently exists, the debug command will print a message trying to create the terminal emulator process during the creation of a live debugged process:

Note: Attempting to start a windowed terminal emulator on display 'unix:0.0'.

Whenever you restart the execution of a process with the command **debug, again** or create a new process, a sequence of dashes is output to the execution window. These dashes serve to separate the output from various debugged processes that have used the window.

When this option's value is reset it will not take effect until the next time that the debugger must create a windowed terminal emulator, ie. if one currently exists then it must be terminated and the debugged process must be restarted for the option value to be used.

The value *none* is reserved to indicate that no windowed terminal emulator program should be used. When it is supplied the option value becomes the empty string. This does not destroy any extant terminal emulator. It will just cause the debugger to not create any subsequent ones.

Note that any terminal emulator created by the debugger is destroyed when the debugger exits.

Also note that all X options specified on the mxdb command line are supplied to the terminal emulator program as well.

Examples

```
(options) op win xterm ,, use xterm ↵
(options) op win *none* ,, do not use a windowed terminal emulator ↵
```

See Also

Command: **command-processor:option-status**

End of Chapter

Chapter 18
Debugger-toolkit Commands

This chapter contains the on-line help messages for the debugger-toolkit realm and for the commands in that realm. The realm help message is first, followed by the help messages for the individual commands, listed in alphabetical order. The debugger-toolkit realm contains special debugging commands.

debugger-toolkit

Realm

Contains special debugging commands.

Summary

The debugger-toolkit realm contains special debugging commands.

Here is how to perform some common tasks:

To get	A list of debugger-toolkit commands:	**help, command, r d-t**
	A list of all commands:	**help ,command ,realm**
	Help on a specific command:	**help** *<command-name>*
	A list of help topics:	**help, topic**
	A list of all help topics:	**help ,topic ,realm**
	Help on a specific topic:	**help** *<topic-name>*
	Debugger-toolkit commands summarized:	**help, r d-t, v**
	A list of all realms:	**help ,realm**
	Help on a specific realm:	**help** *<realm-name>*
To go back to the debugger realm		**realm debugger**

Description

Following is a list of debugger-toolkit commands; you can abbreviate character names using standard CP abbreviation rules.

Command Name	Action
elf-debug-rtld	Directs the debugger to do minimal shared object initialization.
elf-stop-for-link-map-changes	Directs the debugger to stop for link-map resynchronized events.
event-list	Displays event-names matching a regular expression.
position-frame	Writes the position's frame number.
position-has-debug-info	Writes if the position has debugging information.
position-line	Writes the position's line number.
position-module	Writes the position's module (legend) name.
position-pc	Writes position's program counter.
position-return-address	Writes the position's return address.
position-routine	Writes the position's routine name.
position-scope-pathname	Writes the position's scope-pathname.
position-source-file	Writes the position's source filename.
process-corefile	Displays or sets the corefile for a process.
process-identifier	Displays the runtime identifier for a process.
process-shared-objects	Writes the pathnames of shared-objects.

process−signal	Displays or sets the current continuation signal for a process.
process−stop−reasons	Writes the reasons why the process stopped.
program−command−line	Writes the current command line that was used to create the process.
program−entry−point	Writes the numeric address of the program entry point.
program−name	Writes the program's pathname.
resolve−filename	Resolves a filename via the debugger's directory-list.

elf–debug–rtld *Debugger-toolkit Command*

Directs the debugger to do minimal shared object initialization.

Summary

Directs the debugger to do minimal shared object initialization.

Syntax

elf–debug–rtld *[yes|no]*

where:

yes|no	A value of yes or no

Examples

```
elf-d yes
elf-d
```

Description

The **elf–debug–rtld** command directs the debugger to load the debugging information for the ELF a.out file and the program interpreter, and to leave the process at the initial load position for shared ELF processes, _rt_boot. This is *not* the default action.

This option is only useful debugging the runtime linker and thus is not the default mode of operation of the debugger. "Yes" is printed when rtld debugging mode is in effect; "no" is printed otherwise.

Arguments

yes|no	Accepts a value of yes or no. The initial value of this argument is "no."

Examples

To list the current value of this command:

```
(debug) elf–d ↵
no
```

To direct the debugger to do minimal shared object initialization:

```
(debug) elf–d yes ↵
```

elf−stop−for−link−map−changes *Debugger-toolkit Command*

Directs the debugger to stop for link-map resynchronized events.

Summary

Directs the debugger to stop for link-map resynchronized events.

Syntax

elf−stop−for−link−map−changes *[yes|no]*

where:

yes|no	A value of yes or no

Examples

```
elf-stop
elf-stop no
```

Description

The **elf−stop−for−link−map−changes** command controls whether the debugger will stop for shared-object link-map resynchronized event occurrences. The default action is to stop when this event occurs.

This command prints "yes" or "no" when invoked with no arguments. "Yes" is printed when the debugger will stop for link-map resynchronized events; "no" is printed otherwise.

Arguments

yes|no	Accepts a value of yes or no. The initial value of this argument is "yes."

Examples

To list the current status of this debugger directive:

```
(debug) elf−stop ↵
yes
```

To prohibit stopping at link-map resynchronized event occurrences:

```
(debug) elf−stop no ↵
```

event-list *Debugger-toolkit Command*

Displays event-names matching a regular expression.

Summary

Displays event-names matching a regular expression.

Syntax

event-list *[regex]*

where:

regex	A regular expression

Examples

```
e-l
e-l o
```

Description

The **event-list** command displays all event names that contain the supplied regular expression. If you do not supply a regular expression, all event names are printed.

Arguments

regex	A regular expression.

Examples

To list all event names:

```
(debug) e-l ↵
```

To list event names containing the letter "o":

```
(debug) e-l o ↵
```

position-frame

Debugger-toolkit Command

Writes the position's frame number.

Summary

Writes the position's frame number.

Syntax

position-frame *[position]* **,frame**

where:

position	A file, scope, line number, or a combination of the three
frame	A stack frame specification

Examples

```
position-frame, frame 5
position-frame
```

Description

The **position-frame** command writes the frame number for the specified position or frame. When no debugging information is present for the location, an error results.

You can specify either the position or frame argument if you wish, but you cannot supply both. If you do not supply any argument, the current debugger position is used.

Arguments

position	Position to a file, scope, routine, line number, or any combination thereof.
frame	Position to a stack frame.

Examples

To write the current frame number:

```
(debug) position-frame ↵
```

To write the frame number of a specified frame:

```
(debug) position-frame, frame 5 ↵
```

See Also

Commands: position-has-debug-info, position-line, position-module, position-pc, position-return-address, position-routine, position-scope-pathname, position-source-file

position-has-debug-info *Debugger-toolkit Command*

Writes if the position has debugging information.

Summary

Writes "yes" if the position has debugging information.

Syntax

position-has-debug-info *[position]* **,frame**

where:

position	A file, scope, line number, or a combination of the three
frame	A stack frame specification

Examples

```
position-has, frame 0
pos-has
```

Description

The **position-has-debug-info** command writes "yes" if the position has debugging information. When no debugging information is present for the location, nothing is printed.

You can specify either the position or frame argument if you wish, but you cannot supply both. If you do not supply any argument, the current debugger position is used.

Arguments

position	Position to a file, scope, routine, line number, or any combination thereof.
frame	Position to a stack frame.

Examples

To write if the current debugger position contains debugging information:

```
(debug) position-has ↵
yes
```

To write if a specified frame contains debugging information:

```
(debug) position-has, frame 0 ↵
yes
```

See Also

Commands: position-frame, position-line, position-module, position-pc, position-return-address, position-routine, position-scope-pathname, position-source-file

position-line

Debugger-toolkit Command

Writes the position's line number.

Summary

Writes the position's line number.

Syntax

position-line *[position]* **,frame**

where:

position	A file, scope, line number, or a combination of the three
frame	A stack frame specification

Examples

```
position-line
position-line main
```

Description

The **position-line** command writes the line number for the specified position or frame. When no debugging information is present for the location, an error results.

You can specify either the position or frame argument if you wish, but you cannot supply both. If you do not supply any argument, the current debugger position is used.

Arguments

position	Position to a file, scope, routine, line number, or any combination thereof.
frame	Position to a stack frame.

Examples

To write the current line number:

```
(debug) position-line ↵
```

To write the line number of a specified position:

```
(debug) position-line main ↵
```

See Also

Commands: position-frame, position-has-debug-info, position-module, position-pc, position-return-address, position-routine, position-scope-pathname, position-source-file

position-module

Debugger-toolkit Command

Writes the position's module (legend) name.

Summary

Writes the position's module (legend) name.

Syntax

position-module *[position]* **,frame**

where:

position	A file, scope, line number, or a combination of the three
frame	A stack frame specification

Examples

```
position-module
position-module, frame 1
```

Description

The **position-module** command writes the module name for the specified position or frame. If the position does not have debugging information associated with it, an error results.

You can specify either the position or frame argument if you wish, but you cannot supply both. If you do not supply any argument, the current debugger position is used.

Arguments

position	Position to a file, scope, routine, line number, or any combination thereof.
frame	Position to a stack frame.

Examples

To write the current module name:

```
(debug) position-module ↵
```

To write the module name of a specified frame:

```
(debug) position-module, frame 1 ↵
```

See Also

Commands: position−frame, position−has−debug−info, position−line, position−pc, position−return−address, position−routine, position−scope−pathname, position−source−file

086-000203 updates
093-000710-03

This page intentionally left blank.

position-pc *Debugger-toolkit Command*

Writes the position's program counter.

Summary

Writes the position's program counter.

Syntax

position-pc *[position]* **,frame**

where:

position	A file, scope, line number, or a combination of the three
frame	A stack frame specification

Examples

```
position-pc printf
position-pc, frame 3
```

Description

The **position-pc** command writes the program counter (instruction address) for the specified position or frame in a non-symbolic form.

You can specify either the position or frame argument if you wish, but you cannot supply both. If you do not supply any argument, the current debugger position is used.

Arguments

position	Position to a file, scope, routine, line number, or any combination thereof.
frame	Position to a stack frame.

Examples

To write the program counter for a specified position:

```
(debug) position-pc printf ↵
```

To write the program counter for a specified frame:

```
(debug) position-pc, frame 3 ↵
```

See Also

Commands: position−frame, position−has−debug−info, position−line, position−module, position−return−address, position−routine, position−scope−pathname, position−source−file

position-return-address

Debugger-toolkit Command

Writes the position's return address.

Summary

Writes the position's return address.

Syntax

position-return-address *[position]* **,frame**

where:

position	A file, scope, line number, or a combination of the three
frame	A stack frame specification

Examples

```
position-return-address
pos-ret, frame 3
```

Description

The **position-return-address** command writes the return address for the specified position's frame in a non-symbolic form. The return address is the program counter (instruction address) for the parent frame (where execution will resume after return from the specified position's frame).

This command is useful when stack walkback problems occur for helping to diagnose if the return address for a particular frame has been corrupted.

You can specify either the position or frame argument if you wish, but you cannot supply both. If you do not supply any argument, the current debugger position is used.

Arguments

position	Position to a file, scope, routine, line number, or any combination thereof.
frame	Position to a stack frame.

Examples

To write the return address for the current position:

```
(debug) position-return-address ↵
```

To write the return address for a specified frame:

```
(debug) pos-ret, frame 3 ↵
```

See Also

Commands: position−frame, position−has−debug−info, position−line, position−module, position−pc, position−routine, position−scope−pathname, position−source−file

Appendix B
Mxdb Graphical Interface Demos

Notation used in these demonstrations is as follows:

<popup> refers to pressing and holding mouse button 3 until the appropriate choice from the popup menu is highlighted

<click> refers to pressing down and releasing the left mouse button

<double−click> refers to pressing down and releasing the left mouse button two times in quick succession.

<CR> refers to pressing and releasing the Enter or Carriage Return key

X−select refers to holding down the left mouse button while dragging the cursor over a selected area and then releasing the left mouse button.

Enter refers to typing in the succeeding command line at the keyboard.

Note that it is possible to abbreviate Mxdb commands and arguments names if the abbreviations are unique. Each Mxdb command will be given in its entirety with its shortest abbreviation in square brackets directly below it for comparison. If the command is already as terse as possible, [" "] will be used.

To run the demos in this section, you must have loaded the second optional package on the Mxdb release media.

Load in the Mxdb macros. Enter

include /usr/opt/mxdb/macros/all.cp <CR>
[in /usr/opt/mxdb/macros/all.cp <CR>]

The programs used in these demonstrations exist in the /demos subdirectory of the Mxdb release area (/usr/opt/mxdb). cd into this directory. Enter

cd /usr/opt/mxdb.demos <CR>
[" "]

These demos are independent of each other and may be run in any sequence. Each may be run from any live debugger session.

C/X–window Demo

This demo uses a tictactoe program to demonstrate:

* attaching to / detaching from an already extant process
* machine– and source–level debugging integration
* debugging a Motif application written in C
* C data types support
* the command line prompting facility
* the string display option
* the display format option

Before starting this demo, you must merge some app–defaults into your resource manager. To do this, at the Mxdb command line enter

```
shell /usr/bin/X11/xrdb –rnerge app–defaults/Tictactoe <CR>
[sh /usr/bin/X11/xrdb –merge app–defaults/Tictactoe <CR>]
```

To start this demo, enter

```
shell cdemo/tictactoe& <CR>
[sh cdemo/tictactoe& <CR>]
```

to bring up the tictactoe program from the shell as a background job.

Make note of the pid number. This will be the number found after the [<job number>] prompt; you will need it later. Note! The [<job number>] prompt is specific to certain shells. The shells **sh** and **bash** do not print the pid number in Mxdb's usage, while **csh** does print the pid number. You can always find the pid number by using following command:

```
shell ps –fu <your user name> | grep tictactoe <CR>
[sh ps –fu <your user name> | grep tictactoe <CR>]
```

A line of the form:

```
<your user name> <pid> <ppid> ..... cdemo/tictactoe
```

will be displayed. Get the pid number from there.

The tictactoe program is now active, but its execution was not started from within Mxdb. You can attach to it and place it under the control of Mxdb. At the (debug) prompt from Mxdb, attach to the tictactoe program:

```
attach, <CR>
[at, <CR>]
```

Whenever you end a command with a comma, Mxdb prompts for argument values, so you don't have to memorize the order in which any command's arguments are defined. Now, Mxdb will prompt you for

"pid () ="

Enter the tictactoe pid you noted earlier and hit <CR>. Mxdb will then prompt you for

"executable () ="

Enter

cdemo/tictactoe <CR>

and respond to the

"Execute? (Yes) = "

prompt with a <CR>. Now Mxdb will load linker symbols and modules from the executable file specified earlier.

The source pane will contain no information because you will be stopped inside the X Toolkit, which was not compiled for debugging. You still can see that Mxdb has oriented itself:

<click> walkback (for a frame walkback)
<click> machine state (to get pc and register values)

Enter

disassemble 10 <CR>
[dis 10 <CR>]

to show the disassembled code. The first argument to the disassemble command refers to the number of lines to disassemble or instructions to display. Note that since there is no debugging information at the current position, ten instructions are displayed instead of displaying the instructions associated with the succeeding ten lines of source code.

Now set a breakpoint on code which was compiled for debugging. You can specify a breakpoint in a particular module or function. In this case, you want to stop in the procedure position_changed. To do this, enter

breakpoint position_changed <CR>
[b position_changed <CR>]

and continue execution with:

<click> continue

Note that the graphical user interface's prompt is insensitive, which indicates that the debugger is busy.

You must click on the "Play" button in the tictactoe window to start the tictactoe game.

Now select a square from the tictactoe game with a mouse click. Mxdb will show the source in the source pane and highlight the line reading

"Arg args[1]"

This highlighting indicates the current position in the source code of the executing program. The static position of the debugger is represented graphically by an arrow and is used to indicate the source code corresponding to the current debugging information. Use the vertical scroll bar in the source pane to scroll until line 101 is visible. Move the cursor over the line number of line 101 and

<click> (on the line number for line 101)

This sets a breakpoint on line 101. A stop sign icon will appear to the left of the line number as a visual reminder of this breakpoint. Then

<click> continue

Evaluate args[0] by X-selecting the text "args[0]" on line 100 (be careful not to miss) and then

<popup> evaluate SEL

Note that the name field of args[0] is a string. In addition to displaying the starting address of this string, the pointer has been dereferenced to actually display the string, "labelPixmap". The displaying of strings is under the control of the boolean global option, "String_Display". The **option-status** command displays the current settable global options and their present values. Enter

option-status <CR>

to see this. The option "String_Display" defaults to "yes," its more useful mode. Try setting this option to "no" to see how this changes what is displayed when strings are evaluated. Enter

option-status string-display no <CR>
[op s-d n <CR>]

and then evaluate the selection again using the popup:

<popup> evaluate SEL

There are several things to note here. First note that you can refer to "String_Display" as string-display on the Mxdb command line. The Mxdb command-processor is case insensitive; this includes hyphen/underscore case insensitivity as well. Also, with string-display set to "no," only the starting address of the field "name" is displayed. Note also that you did not have to X-select "args[0]" once again in order to do this evaluation (unless you had selected something else in the meantime).

Get the description and address of "args[0]":

<popup> describe SEL
<popup> address SEL

Now X-select the text "position" on line 101 (you can double-click on the text to select an entire word) and evaluate this variable:

<popup> evaluate SEL

"position" is an int and therefore the format in which it is displayed is governed by the "Signed_Integer_Format" option. The default value of this option is decimal. Change the value of this option to binary by entering

option-status signed-integer-format binary <CR>
[op s-i b <CR>]

Evaluate "position" once again:

<popup> evaluate SEL

Note the change in display format. Try changing the display format to hexadecimal:

option-status signed-integer-format hexadecimal <CR>
[op s-i h <CR>]

Evaluate "position" once again:

<popup> evaluate SEL

Set the display format back to decimal:

option-status signed-integer-format decimal <CR>
[op s-i d <CR>]

Query some other data types found in this program. Enter

evaluate moves <CR>
[e moves <CR>]
describe moves <CR>
[des moves <CR>]

in order to evaluate and describe "moves." Note that "moves" is described as an array of "auto struct move." Describe the type "move." Enter

describe move <CR>
[des move <CR>]

Enter

```
describe sessions <CR>
[des sessions <CR>]
```

Note that "sessions" is described as an array of "auto struct ttt_board." You can also try to describe "ttt_board." Enter

```
describe ttt_board <CR>
[des ttt_board <CR>]
```

Enter

```
address sessions[0].map <CR>
[ad sessions[0].map <CR>]
```

in order to get the starting address of the map field of sessions[0]. Enter

```
address sessions[0].x_pix <CR>
[ad sessions[0].x_pix <CR>
```

in order to get the starting address of the x_pix field of sessions[0]. Note that these two addresses differ by eight, which is expected since the field "map" is a union and therefore allocates enough space for its largest member, a double. Enter

```
describe sessions[0].map <CR>
[des sessions[0].map <CR>]
```

in order to see a description of the individual members of "map."

To release execution of the tictactoe program from under the control of Mxdb, detach from the tictactoe process. Enter

```
detach <CR>
[det <CR>]
```

and you may then continue playing.

Note that all buttons, except "debug again," are insensitive. This is because there is no target process to debug. If you wish to reattach to the same tictactoe process, click on "debug again."

To exit from the tictactoe game, click on the "Quit" button from the tictactoe window.

FORTRAN–77 Demo

This demo highlights several interesting FORTRAN–77 features supported by Mxdb for use with the Edinburgh Portable Compilers FORTRAN compiler. Features demonstrated include:

* records (evaluating, describing, assigning to)
* array elision (using both single– and multi–dimensional arrays)
* COMMON blocks (plus overloaded COMMON blocks)
* C–interoperability with FORTRAN–77 syntax
* EQUIVALENCE statements
* star–extents

To begin, enter

debug epcf77demo/records <CR>
[deb epcf77demo/records <CR>]

This program demonstrates the handling of records and the use of array elision. Scroll down to line 62 in the source pane and

<click> on the line number of line 62

in order to set a breakpoint.

<click> continue

to begin execution. At the breakpoint, describe the variable "grad." Enter

describe grad <CR>
[des grad <CR>]

It will be a record of structure "student_file." Now evaluate this record to see the values of its specific fields. Enter

evaluate grad <CR>
[eval grad <CR>]

Note that you can also describe and/or evaluate a field of a record as well. Enter

describe grad.address.city <CR>
[des grad.address.city <CR>]

to describe the sub–field "city" of the field "address" and enter

evaluate grad.address.city <CR>
[eval grad.address.city <CR>]

to display the value of this field.

Try changing the value of a field in the record. Enter

```
assign grad.address.street_addr '1 Main Street' <CR>
[as grad.address.street_addr '1 Main Street' <CR>]
```

to assign the new value and enter

```
evaluate grad <CR>
[eval grad <CR>]
```

to display the new contents of the record. Also try assigning a value to the variable "dossier." "dossier" is a large, empty array of structure "student_file." Enter

```
evaluate dossier <CR>
[eval dossier <CR>]
```

to see this. Note that array elision is in effect here. The notation

```
(1..1000) =
```

is used to indicate that the first 1,000 adjacent records in "dossier" are the same and therefore elided when displayed. Enter

```
assign dossier(500) grad <CR>
[as dossier(500) grad <CR>]
```

to assign "grad" to the 500th record of "dossier" and enter

```
evaluate dossier <CR>
[e dossier <CR>]
```

to see how "dossier" is now displayed, with (0..499) and (501..999) being elided.

Array elision can occur with multi-dimensional arrays as well. Enter

```
evaluate young_alumni <CR>
[e young_alumni <CR> ]
```

to see this. The notation

```
(1..10,1..250) =
```

is used to specify that all 1,000 records of this two-dimensional array are the same. Note that evaluating such a large, complex array may take a few moments.

086-000203 updates
093-000710-03

Now try assigning a value to an entry in this array. Enter

```
assign young_alumni(5,125) grad <CR>
[as young_alumni(5,125) grad <CR>]
```

and then

```
evaluate young_alumni <CR>
[eval young_alumni <CR>]
```

Entry (5,125) is uniquely different from the other entries in "young_alumni" and therefore causes the elision to occur in the following order:

```
(1..10,1..125)
(1..5,125)
(5,125)
(6..10,125)
(1..10,126..250)
```

To complete execution of the program,

```
<click> continue
```

Start debugging the next program. Enter

```
debug epcf77demo/c_inter <CR>
[deb epcf77demo/c_inter <CR>]
```

This program demonstrates the handling of common blocks and interoperability with GNU C. Set a breakpoint at line 52 by entering

```
breakpoint 52 <CR>
[b 52 <CR>]
```

and

```
<click> continue
```

to start execution. Try to describe and evaluate the common blocks used in this program. Enter

```
describe num_area <CR>
[des num_area <CR>]
```

Note that Mxdb realizes that "num_area" is a common block and lists all the variables found within that common block. Enter

```
evaluate num_area <CR>
[eval num_area <CR>]
```

Each variable found within the common block "num_area" is evaluated. Now describe "com_area" by entering

```
describe com_area <CR>
[des com_area <CR>]
```

Note that besides a common block called "com_area," there is variable called "com_area," which was described with the preceding command. Since the intent was to describe the common block and not the variable, the

```
$$COMMON(common block)
```

construct must be used. This construct must be used explicitly to distinguish a common block from a variable when overloading of their names occurs. It is not needed, but can be used when overloading does not occur as well. Enter

```
describe $$COMMON(com_area) <CR>
[des $$COMMON(com_area) <CR>]
```

to describe "com_area."

Now describe a variable of a common block. When a variable found in a common block is described, its common block is displayed as well. Enter

```
describe real1 <CR>
[des real1 <CR>]
```

to see this. Assignment to variables found in common blocks is possible also. Enter

```
assign real1 −5.5 <CR>
[as real1 −5.5 <CR>]
```

to assign a value to "real1" and

```
evaluate real1 <CR>
[eval real1 <CR>]
```

to display its new value.

Now

<click> step

in order to step into a C module. Enter

breakpoint 245 <CR>
[b 245 <CR>]

to set a breakpoint on line 245 and

<click> continue

to proceed. To show that the language is indeed C, enter

option-status <CR>
[op <CR>]

The fourth option line displays

Language c,

If necessary, use the vertical scroll bar of the message pane to scroll back up to where this line is visible.

Describe variables and data types in this C module using FORTRAN syntax instead of C syntax with the command processor. This feature is useful for FORTRAN programmers who may call a routine written in C but would feel more comfortable in seeing commands processed using FORTRAN rather than C syntax. Enter

option-status language fortran <CR>
[op l f <CR>]

to change the language to FORTRAN. Note that this change remains in effect only up until the next change of the debugger position. Note also that some C constructs cannot be described or evaluated using FORTRAN syntax because parallel constructs do not exist in FORTRAN-77.

Enter

describe c_inter <CR>
[des c_inter <CR>]

to describe the procedure "c_inter" using FORTRAN syntax. Note that the parameters of "c_inter" are described using FORTRAN syntax as well.

Enter

```
describe at <CR>
[des at <CR>]
```

to describe the variable "at." It is described as a RECORD of "my_struct."

Enter

```
describe my_struct <CR>
[des my_struct <CR>]
```

to describe this structure. To evaluate "at," enter

```
evaluate at <CR>
[eval at <CR>]
```

Enter

```
describe anut <CR>
[des anut <CR>]
```

to describe the variable "anut." It is described as a RECORD of "another_u." Enter

```
describe another_u <CR>
[des another_u <CR>]
```

to describe this structure. To evaluate "anut," enter

```
evaluate anut <CR>
[eval anut <CR>]
```

Enter

```
describe nums <CR>
[des nums <CR>]

evaluate nums <CR>
[eval nums <CR>]
```

to describe and evaluate the variable "nums."

Enter

describe a_float <CR>
[des a_float <CR>]

evaluate a_float <CR>
[eval a_float <CR>]

to describe and evaluate the variable "a_float."

Enter

describe c_int <CR>
[des c_int <CR>]

evaluate c_int <CR>
[eval c_int <CR>]

to describe and evaluate "c_int."

Enter

describe string <CR>
[des string <CR>]

evaluate string <CR>
[eval string <CR>]

to describe and evaluate the variable "string."

To complete the execution of this program,

<click> continue

Start debugging the next program. Enter

debug epcf77demo/starext_eqv <CR>
[deb epcf77demo/starext_eqv <CR>]

This program demonstrates the handling of equivalence statements and star−extents. Set breakpoints at lines 19 and 20:

<click> on the line number for line 19
<click> on the line number for line 20

and

<click> continue

to reach the first breakpoint. Variables "i3" and "ca5x3" are equivalenced.

To display the address of "i3," first X–select the text "i3" on line 16 and

```
<popup> address SEL
```

Display the address of "ca5x3" as well. X–select the text "ca5x3" on line 16 and

```
<popup> address SEL
```

The addresses should be the same. Try assigning a value to "i3" and then evaluating "i3." Enter

```
assign i3(2) 5 <CR>
[as i3(2) 5 <CR>]
```

and

```
evaluate i3 <CR>
[eval i3 <CR>]
```

Assign a value to its equivalent. Enter

```
assign ca5x3(1) 'astring' <CR>
[as ca5x3(1) 'astring' <CR>]
```

Now evaluate "i3" once again. Enter

```
evaluate i3 <CR>
[eval i3 <CR>]
```

The value of "i3" will have changed because both "i3" and "ca5x3" share the same address space.

Now step into the subroutine "foo":

```
<click> step
<click> next
```

Describe some of the parameters to the subroutine "foo" to see how star–extents used in parameter declarations are handled. Enter

```
describe c <CR>
[des c <CR>]
```

The argument passed to "foo" as "c" was a character*3 c(5). Note that it was determined that "c," declared as a character*(*) c(*), was actually a character*3 c(*).

Change the value of this variable using the assignment command. Enter

```
assign c(1) '5mxdb' <CR>
[as c(1) '5mxdb' <CR>]
```

to assign a string to "c" and

```
evaluate c(1) (2:3) <CR>
[eval c(1) (2:3) <CR>]
```

to display a substring of "c."

Now

```
<click> continue
```

Control returns to the main program. Step into the next call to subroutine "foo":

```
<click> step
<click> next
```

Describe "c" again:

```
describe c <CR>
[des c <CR>]
```

This time, "c" is described as "character*7 c(*)."

To finish,

```
<click> continue
```

GreenHills C and FORTRAN-77 Demo

This demo has two parts. The first part uses a factorial calculation program to demonstrate Green Hills language support (Green Hills C and Green Hills FORTRAN−77) and user extensibility (macro and button definitions for the dbx realm).

The second part uses a shortest path heuristic to demonstrate the following:

* setting/viewing events: breakpoints, watchpoints
* disabling/enabling/deleting events
* defining new buttons
* the stop−commands option
* the array elision option

Mxdb has both its regular command set and macros to make it mimic dbx. It is preferable to have both up at the same time, in separate windows, so that one can see that Mxdb implements a dbx superset. Since the dbx commands and buttons are implemented in Mxdb macros, this demo also demonstrates the power of the Mxdb macro facility.

To start up another Mxdb with the dbx command set, invoke it from the directory **/usr/opt/mxdb.demos**:

shell mxdb −g & <CR>
[sh mxdb −g & <CR>]

Let's go through the first part of the demo for Green Hills comparing the mxdb and dbx command sets:

mxdb	dbx
Start the first part of the demo for Green Hills:	
debug ghdemo/factorial <CR>	c−p:include /usr/opt/mxdb/ui/dbx/dbx.cp <CR>
[deb ghdemo/factorial <CR>]	[" "]
	debug ghdemo/factorial <CR>
	[deb ghdemo/factorial <CR>]

Set a breakpoint and start execution:

<click> (on line number for line 10)	<click> (on line number for line 10)
<click> continue	<click> run

Enter a number between 1 and 10 in each Mxdb execution window.

Determine what the stack frame looks like:

<click> walkback	<click> where

Step into a C routine:

<click> step	<click> step
<click> (on line number for line 9)	<click> (on line number for line 9)
<click> continue	<click> cont

X−select the text "*number * *return_val" and then

<popup> evaluate SEL	<click> print
<popup> describe SEL	<click> whatis

To end the demo:

<click> terminate	quit <CR> [qui <CR>] (this will also get rid of windows associated with the dbx demo)

To start the second part of this demo, enter

```
debug ghdemo/spath <CR>
[deb ghdemo/spath <CR>]
```

Now, set breakpoints in the source code using two different methods. The first method is to enter the break command with arguments. In the command pane, enter

```
breakpoint 83, name dist−mat  <CR>
[b 83, n dist−mat <CR>]
```

The "name" argument keyword of the **breakpoint** command is used here in order to tag this event explicitly instead of having it tagged by default with an integer.

The second method is clicking on the line number of the associated source line. Scroll down to line 98 and set a breakpoint:

<click> (on the line number for line 98)

Scroll down to line 103 and set another breakpoint:

<click> (on the line number for line 103)

To verify that both of these methods are successful in setting breakpoints,

<click> event status

In the message pane, you will see that three events have been set. They are all breakpoints. The first is named "dist−mat," the second is named "2," and the third is named "3."

Now to start execution of the program,

<click> continue

Switch to the Mxdb execution window. At the prompt "Input the # of nodes," enter

5 <CR>

At the prompt for each pair of nodes, enter a number between 1 and 50 representing the length between the pair and hit <CR>.

The breakpoint at line 83 will now have been reached. This breakpoint, "dist−mat," was placed in the middle of the printing of the distance matrix.

<click> continue

If you look in the Mxdb execution window, you will see that this breakpoint allows one to see the distance matrix being printed out row by row. Disable this event to finish the printing of the distance matrix in full:

```
disable-events dist-mat <CR>
[di-e dist-mat <CR>]
```

and then

```
event-status <CR>
[e-s <CR>]
```

to show that this breakpoint has only been disabled. The breakpoint at line 83 now has the added phrase "disabled yes." Note also that the breakpoint icon is still in place on line 83 because the breakpoint has been disabled only and not deleted. Enter

```
evaluate DIST <CR>
[eval DIST <CR>]
```

to evaluate DIST, the distance matrix. Note the unique display of the entries of this two dimensional array. Adjacent entries which have the same values are elided when displayed. For example the entries from DIST[6][0] to DIST[49][49] are all zero and therefore are displayed as

```
[6..49] = {
[0..49] = 0
}
```

This lends towards a much more concise and easily readable display. The displaying of arrays is under the control of the boolean global option, "Elide_Arrays." It defaults to "yes." Setting this option to "no" will print out each entry of the array, regardless if its adjacent entry is the same or not. If you wish, try setting this option to "no." However, DIST is a 50 by 50 matrix, mostly filled with zeros. It will take about a minute to display this array without elisions. Enter

```
option-status elide-arrays no <CR>
[op e-a n <CR>]
```

and

```
evaluate DIST <CR>
[eval DIST <CR>]
```

to see the effect of having no elision of arrays.

To finish the printing of the distance matrix

```
<click> continue
```

At the prompt "Input start and destination nodes" in the Mxdb execution window, enter

4 5 <CR>

We can now re-enable the breakpoint if we wish by entering

enable-events dist-mat <CR>
[en dist-mat <CR>]

Entering

event-status <CR>
[e-s <CR>]

will show that this breakpoint has been re-enabled. Execution will have stopped at the next breakpoint, on line 98.

Try using the global option "stop-commands" to print out the value of a variable whenever this breakpoint is reached. Enter

option-status stop-commands {write LABEL[destination]=; eval LABEL[destination]} <CR>
[op sto {wr LABEL[destination]=; e LABEL[destination]} <CR>]

This will cause the series of commands between the {} to be executed each time an event which stops execution is reached.

<click> continue

to see this. Set the stop_commands option back to null and try to create a button which will perform a similar operation: selected after a stop in execution, it will print out an X-selected expression, its value, and will then continue execution. Enter

option-status stop-commands {} <CR>
[op sto {} <CR>]

to reset the stop_commands option to its initial value. To create a new control panel button, enter

define-button, <CR>
[def-b, <CR>]

At the "label ()= " prompt, enter

"cont/print SEL" <CR>

At the "commands () =" prompt, enter

```
{continue; write '{selection,required}=; evaluate '{selection,required}} <CR>
[{co; wri '{sel,re}=; ev '{sel,re}} <CR>]
```

At the "position (0) = " prompt, enter

```
,execute <CR>
[,e <CR>]
```

in order to execute this command, taking the defaults for all of the succeeding arguments. A button labeled "cont/print SEL" will be created in the StackButtons pane.

Now X−select the text "LABEL[destination]" on line 96 and

```
<click> cont/print SEL
```

The value of LABEL[destination] will be printed and execution will continue until the next breakpoint is reached at line 98.

```
<click> continue
<click> continue
```

Notice that the static position icon will also jump from line 98 to line 103 and remain there because the breakpoint was placed inside of a for−loop. Delete the breakpoint at line 103. The easiest way to delete this breakpoint is to <click> on the line number associated with line 103, but try deleting the breakpoint using the **delete−events** command instead. First execute an **event−status** command to get the name of the breakpoint set at line 103. Enter

```
event−status <CR>
[e−s <CR>]
```

The breakpoint should be event number 3. Enter

```
delete−event 3 <CR>
[de−e 3 <CR>]
```

in order to delete the breakpoint.

Enter

```
event-status <CR>
[e-s <CR>]
```

once again to verify that the breakpoint has been deleted.

```
<click> continue
```

Notice that the static position arrow no longer stops at line 103 and the breakpoint icon disappears because the breakpoint has been deleted.

Try performing a stack trace to determine the values of the local variables at this point. Enter

```
walkback, locals <CR>
[wal, l <CR>]
```

Notice that several of the variables are pointers or have fields that are pointers. Only the starting addresses are displayed for these variables. This is because the "Pointer_Dereference_Level" option defaults to 0. Change the value of this option to 1. Enter

```
option-status pointer-dereference-level 1 <CR>
[op po 1 <CR>]
```

to do this. Now evaluate one of the local variables. Enter

```
evaluate LABEL <CR>
[eval LABEL <CR>]
```

Notice now that the pointer fields have now been dereferenced and their contents displayed.

Click on the continue button until the minimum tour is shown in the Mxdb execution window.

Try to debug this program again and this time use some watchpoints.

```
<click> debug again
```

in order to restart execution of the previous executable.

First delete all of the previous events which had been set,

delete−events, all <CR>
[de−e,a <CR>]

set a breakpoint at line 96,

breakpoint 96 <CR>
[b 96 <CR>]

start execution

<click> continue

and proceed to input data as before until the breakpoint at line 96 is reached.

Two interesting variables to have watchpoints on when running the shortest path heuristic are MILEAGE and nextnode. A watchpoint can be placed using either the watch−reference or watch−memory command. Try both of these methods.

Place a watchpoint on nextnode:

watch−reference nextnode <CR>
[w−r nextnode <CR>]

Place a watchpoint on the memory location associated with MILEAGE. Enter

watch−memory '{address MILEAGE} <CR>
[w−m '{address MILEAGE} <CR>]

The argument, '{address MILEAGE}, makes use of the command processor's command substitution facility. With this syntax, the result of the command, address MILEAGE, will be substituted as the argument to the **watch−memory** command. This is equivalent to "watch-reference MILEAGE."

A break in execution will occur whenever these variables change. At these breaks, the **watch−print** command can be used to print out the value of the watchpoints which have been set.

<click> continue

Execution will stop at the first watchpoint on MILEAGE.

<click> watch−print

in order to see the value of all watchpoints, including the one for MILEAGE, at this point.

Click on the continue and watch–print buttons several more times until the minimum tour is shown in the Mxdb execution window. To terminate the program before then,

<click> terminate.

C++ Demo

This demo highlights Mxdb's support of C++ programs compiled with version 2.1 of the AT&T Cfront compiler. Key features demonstrated include:

* single inheritance of classes
* static members
* multiple inheritance, virtual base classes
* overloaded functions and operators
* user-defined conversions
* anonymous unions

Before beginning, be reminded that a complete discussion of Mxdb's C++ support is available on-line. At any time during the demo session, you can reference the "c++–language" help topic for additional information.

To begin, enter

deb cxxdemo/student_classes <CR>

This program demonstrates classes, objects and inheritance relationships. Set a breakpoint at the last line in routine "main."

b last <CR>

<click> continue

to begin execution. At the breakpoint, describe the object "mary." Enter

des mary <CR>

It will be a local object of class type "Student." Describe this type by entering

des Student <CR>

You will notice that type "Student" contains a member named "$vtbl." This is a compiler-generated member that contains information required to implement virtual functions in C++. The Cfront compiler does not support the debug-time identification of individual virtual functions in a class.

Now evaluate object "mary." Enter

eval mary <CR>

Observe that the contents of the "$vtbl" member are not displayed. This does not prevent you from directly evaluating it just like any other member. Enter

```
eval mary.$vtbl <CR>
```

Also note that the member functions are not displayed during object evaluation since they do not have meaningful "values" in this context.

You can also describe and assign to data members. Enter

```
des mary.my_name <CR>
as mary.my_name "Janet" <CR>
eval mary.my_name <CR>
```

Mxdb relaxes the semantics of const type specifiers. This permits the successful assignment to const data member Student::my_name in the above example.

Now let's demonstrate Mxdb's support of the C++ feature of inheritance. Describe object "egghead" by entering

```
des egghead <CR>
```

Note that "egghead" is a const local object of class type "GradStudent," which can be described by entering

```
des GradStudent <CR>
```

"GradStudent" is a derived class of class "Student." The members defined in the base class "Student" are also defined in class "GradStudent." This can be demonstrated by evaluating "egghead." Enter

```
eval egghead <CR>
```

"GradStudent" is an unambiguous base class of class "GradStudent," so member names in "GradStudent" can be referenced without explicit scope name qualification. Thus the following commands have the same effect:

```
eval egghead.my_name <CR>
eval egghead.Student::my_name <CR>
```

In C++ it is illegal for a const object to invoke a non-const member function. Non-const member functions may directly or indirectly modify the values of data members, which would violate the const property specified for the object. Mxdb however, relaxes this constraint as can be seen by entering:

```
eval egghead.thesis_advisor( "Dr. Brilliant" ) <CR>
eval egghead.my_thesis_advisor <CR>
```

You may have noticed that function "thesis_advisor" is overloaded. Mxdb applies C++ matching rules to the arguments (if any) to an overloaded function to select the correct function to invoke.

Mxdb supports assignment to class objects. If the object is of a class type defining a suitable assignment operator function (operator=), that member function will be invoked to perform the assignment. Otherwise, objects of the same class, or of unambiguous base classes, may be assigned to the object. You can demonstrate the latter rule by entering

```
as mary egghead <CR>
eval mary <CR>
```

C++ permits a class type to define multiple base classes; this is referred to as multiple inheritance. To see an example of multiple inheritance, first describe object "tricky." Enter

```
des tricky <CR>
```

Observe that "tricky" is a local object of class type "NightGradStudent." Describe class "NightGradStudent:"

```
des NightGradStudent <CR>
```

Class type "NightGradStudent" inherits members from classes "GradStudent" and "NightSchoolAttributes." Object "tricky" contains a sub-object for each of the two base classes, as can be seen by entering

```
eval tricky <CR>
```

It is possible to convert a derived class object to any unambiguous base class type. Enter

```
eval (Student) tricky <CR>
```

Mxdb allows the user to set breakpoints on member functions. As an example, set a breakpoint on NightGradStudent::gpa by entering

```
b NightGradStudent::gpa <CR>
```

Now evaluate member function "gpa" for object "tricky":

```
eval tricky.gpa() <CR>
```

When Mxdb hits the breakpoint, it will inform you that the process has stopped in an invoked routine context. Mxdb is now positioned to member function "gpa", which is defined in class type "GradStudent." At this scope you can evaluate unambiguous members without qualification. Enter

```
eval my_thesis_advisor <CR>
```

Within member function scopes, the same result can be obtained by qualifying the member name with the "this" object pointer. Enter

```
eval this->my_thesis_advisor <CR>
```

"this" pointer qualification provides a way for you to evaluate the "GradStudent" sub-object of object "tricky" even though the name "tricky" is not in scope. Enter

```
eval *this <CR>
```

It is legal in C++ to convert a base class object, reference, or pointer to a derived class object, reference, or pointer provided that the base class is unambiguous and the conversion is explicit. This means that you can completely evaluate "tricky" as a "NightGradStudent" object even though Mxdb is positioned to base class member function "gpa." Enter

```
eval *(NightGradStudent *) this <CR>
```

To complete invocation of member function "gpa" and return to the calling context,

```
<click> finish
```

Delete the breakpoint set on "NightGradStudent::gpa" by entering

```
del-ev 1 <CR>
```

You may have noticed that class "NightGradStudent" defines two static members. Static members are shared by all objects of a class type. Static members can be referenced in language expressions without respect to any particular class object, i.e. using only scope qualifier syntax. The following commands have the same effect:

```
eval tricky.our_parking_lot_number = 44 <CR>
eval NightGradStudent::our_parking_lot_number = 44 <CR>
```

Now if you define a variable of class type "NightGradStudent" and assign to its "our_parking_lot_number" data member, the assignment will occur in object "tricky" as well. Enter

```
def-var mystudent tricky <CR>
as mystudent.our_parking_lot_number 55 <CR>
eval tricky.our_parking_lot_number <CR>
```

A C++ class type can be defined that inherits the same base class type via more than one inheritance path. An object of such a type will contain a separate base class subobject for each occurrence of the multiply-inherited base class. However, C++ provides a feature (called virtual base classes) that permits a single occurrence of a base subobject to be shared along inheritance paths. To demonstrate this feature, begin by describing "pele":

```
des pele <CR>
```

"pele" is a local class object of class "ForeignTransferStudent," which can be described by entering

```
des ForeignTransferStudent <CR>
```

As can be seen, "ForeignTransferStudent" has two base classes, "ForeignStudent" and "TransferStudent." Describe these two classes:

```
des ForeignStudent <CR>
des TransferStudent <CR>
```

Both of the base classes of class "ForeignTransferStudent" define class "Student" as a virtual base class. This has the effect that the "Student" subobject in "pele" is shared along the associated inheritance paths. Verify this by entering

```
eval pele <CR>
```

A class may inherit a base virtually along some inheritance paths and non-virtually along others.

Ambiguities can occur in referencing a member name defined in more than one base class. For example:

```
eval pele.gpa() <CR>
```

Mxdb cannot determine whether you intended to invoke "TransferStudent::gpa" or "ForeignStudent::gpa." In this case you must specify scope qualifier syntax to disambiguate the reference. As an example, enter

```
eval pele.TransferStudent::gpa()
```

To demonstrate some other notable C++ features supported by Mxdb, debug the following program:

```
deb cxxdemo/SmallInt <CR>
```

This program highlights the use of user-defined conversions, overloading and anonymous unions. Set a breakpoint at the last line in routine "main."

```
b last <CR>
```

```
<click> continue
```

to begin execution. At the breakpoint, describe the object "sm_int1." Enter

```
des sm_int1 <CR>
```

"sm_int1" is a local object of class type "SmallInt", which can be described by entering

```
des SmallInt <CR>
```

Notice that class "SmallInt" defines two member functions called "operator=." As discussed previously, classes can define "operator=" to specify the semantics of assigning expressions to objects of the class. In class "SmallInt", an assignment operator controls the assignment of integers to "SmallInt" objects. To see this, enter

```
as sm_int1 5 <CR>
eval sm_int1 <CR>
```


086-000203 updates
093-000710-03

An assignment operator also controls the assignment of "SmallInt" objects to "SmallInt" objects. Try

```
des sm_int2 <CR>
eval sm_int2 <CR>
as sm_int1 sm_int2 <CR>
eval sm_int1 <CR>
```

Unlike the assignment operation for integers, the one for "SmallInt" objects would be legal in C++ even if no associated "operator=" member function had been defined. How can you tell that Mxdb did the right thing and invoked the assigment operator on object "sm_int2" in the above example? One way would be to set a breakpoint on the operator and verify that the breakpoint is hit when you perform the assignment. Enter

```
b SmallInt::operator= <CR>
```

"operator=" is an overloaded function in class "SmallInt", so Mxdb displays the member function interfaces for all of the possible matches and prompts you to select the desired member function. Enter

```
2 <CR>
```

to indicate you want to select the second choice. Now try another object-to-object assignment:

```
des const_sm_int3 <CR>
eval const_sm_int3 <CR>
eval sm_int1 = const_sm_int3 <CR>
```

Mxdb stops in the context of the invoked member function. Now that you're satisfied that the function has been appropriately invoked, finish it and then verify the effect of the assignment:

```
<click> finish
eval sm_int1 <CR>
```

Delete the breakpoint set on "SmallInt::operator=" by entering

```
del-ev 1 <CR>
```

Mxdb supports global operator functions as well as class operator functions. To see this, describe the global "operator==" function by entering

```
des ::operator== <CR>
```

You can logically compare two "SmallInt" objects with this operator function.As an example, enter

```
eval sm_int1 == sm_int1 <CR>
```

Operator functions are indeed functions, so nothing prohibits you from invoking them just like non-operator functions. For instance, you can try:

```
eval ::operator==(sm_int2,const_sm_int3) <CR>
```

In addition to their utility in defining the semantics of unary and binary operators for class operands, operator functions can be used to define the semantics of type conversion between objects and values of other types. Along with conversion constructors, these conversion functions make it possible to implicitly (or explicitly) convert objects to and from other types. Class "SmallInt" defines an operator conversion function, which can be described by entering

```
des SmallInt::operator int <CR>
```

This member function converts a "SmallInt" object to its associated integer value. Try these varying explicit and implicit uses of this operator function:

```
eval (int) sm_int1 <CR>
eval (char) sm_int1 <CR>
eval int(const_sm_int3) <CR>
des a <CR>
eval a = sm_int2 <CR>
eval sm_int1 + sm_int1 <CR>
```

Ambiguities can occur between two or more user-defined conversions, or between user-defined conversions and builtin conversions. In these cases Mxdb reports the problem and identifies conflicting user-defined conversions. Here's an example of an ambiguity between the global "operator==" function and the builtin "==" operator:

```
eval sm_int1 == 4 <CR>
```

It is possible to recast the syntax in the above example to force one or the other of the possible interpretations. These next evaluations take decidedly different routes to effect the comparison:

```
eval ((int) sm_int1) == 4 <CR>
eval ::operator==(sm_int1,4) <CR>
```

The second example makes use of the implicit conversion of integers to "SmallInt" objects via a conversion constructor defined by class "SmallInt."

The last Mxdb-supported C++ feature to be demonstrated is anonymous unions, which are unnamed objects containing only data members. Anonymous union members share the same address, but otherwise are used like ordinary (nonmember) variables. The "SmallInt" program defines an anonymous union containing members "a" and "b." Enter

```
des a <CR>
eval &a <CR>
eval a <CR>
des b <CR>
address b <CR>
eval b <CR>
```

To verify that members "a" and "b" share the same data, enter

```
as a 9 <CR>
eval b <CR>
```

Congratulations, you have successfully completed the Mxdb C++ demo!

End of Appendix