Debugging with MULTI® 2000

UNIX®

DISCLAIMER

GREEN HILLS SOFTWARE, INC. MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE. Further, Green Hills Software, Inc. reserves the right to revise this publication and to make changes from time to time in the content hereof without obligation of Green Hills Software, Inc. to notify any person of such revision or changes.

PubID: M32U20NG

Time Stamp: Fri Oct 22, 1999

CONTENTS

CONTENTS

CONTENTS

CONTENTS

Preface

***This chapter contains*:**

- About the MULTI manuals
- Conventions

About the MULTI manuals

This manual systematically documents all the features and commands of the MULTI debugger ("the debugger") which are host and target independent. The comprehensive index will help you locate the information you need.

For information about other components of MULTI, such as the Builder and Editor, and information about configuring and customizing MULTI, refer to *Building and Editing with MULTI 2000*.

For specific target systems, refer to the *Development Guide* for your target.

Conventions

Typographical conventions

Convention	Example	Description
italic text in a command line	**-o** *filename*	place-holder for mandatory user-supplied arguments
square brackets, []	**.macro** *name* [*list*]	encloses optional commands, terms, or arguments
square brackets [] around boldface word "**default"**	Specifies **char** as signed. **[default]**	command or option is the default
menu > item > sub-item...	File > Open...	menu bar, menu items, sub-menu items...
Enter *something*	Enter **cc800 -S hi.c**	Type *something* AND press the Enter key. Compare with "Type *something*" below.
Type *something*	Type foo.c and press Edit	Type *something* WITHOUT pressing the Enter key. Compare with "Enter *something*" above.

For example, in the command description:

gcc [*-processor*] *filename*

the command **gcc** should be entered as given, the word *processor* may optionally be substituted with an appropriate option, and the word *filename* must be replaced with an appropriate file name.

GUI mode conventions

The main MULTI windows in the Builder, Editor, and Debugger contain some or all of the following regions:

Convention	Description
source pane	The portion of the window in which the source code is displayed.
status bar	Displays information, such as the process state and the name of the file being debugged.
command pane	Area to enter commands and display results.
toolbar	Contains buttons for commonly used commands.

GUI conventions

MULTI documentation assumes you have a working knowledge of your operating system and its conventions, including its command-line and GUI interfaces—for example, how to use a mouse and standard menus and commands, and how to open, save, and close files, etc.

Convention	Meaning
First mouse button	Mouse buttons are numbered from the left. The first mouse button is the left-most mouse button.
Shift+Click	Hold down the Shift key while clicking a mouse button.
Ctrl+Click	Hold down the Ctrl key while clicking a mouse button.

Check box conventions

There are two types of check boxes: two-way and three-way.

A two-way check box has two states: either enabled (with a check mark in it) or not (when it’s empty). For example, Config > Options > Colors tab > Build File Coloring.

A three-way check box has three states (for example, Builder > Project > Options > General tab > Automatically use MVC):

- The first state is On. The box has a plus sign (+), indicating that the option is turned on, overriding any previous or inherited settings.

- The second state is Off. The box contains a minus sign (-), indicating that the option is turned off, overriding any previous or inherited settings.

- The third state is Default. The box is empty, indicating that the inherited state, if any, is used.

Chapter

1

Introduction to MULTI

This chapter contains:

- Features
- Embedded programming in MULTI
- Running MULTI from the command line
- Resources

MULTI is a complete interactive software development environment for programs written in Ada, C, C++, FORTRAN, and Pascal, as well as in assembly language for each supported target. Source code from these languages can be compiled and linked into a single executable in virtually any combination.

> **NOTE**: If you are upgrading from a previous version (1.8.9 or older) to 2.0, **DO NOT install your 2.0 release in the same location as your previous (1.8.9 or older) release**.

Features

Some of MULTI's powerful features include:

Debugging

- A **Source Level Debugger** that supports mixed language debugging and all C++ and Ada language constructs.
- A **Profiler** that collects data, provides reports, annotates the source code to find hot spots in your program, and provides mechanisms to feed information back into the development process. See Chapter 7, "The Profiler".
- **Run-Time Error Checking** for different classes of errors, implemented with a combination of compiler checks, libraries, and debugger commands. See Chapter 6, "Run-time error checking".
- **Expression Evaluation** to determine whether your expressions are correct. See "Evaluating expressions" on page 46.
- A **Data Explorer** to monitor variables and evaluate expressions during debugging. See Chapter 5, "The data explorer".
- **Memory Leak Detection** to find chunks of memory that have been allocated but are no longer used. See "Finding memory leaks" on page 171.
- **Conditional Breakpoints** that cause a breakpoint to be active under conditions you specify. See "To make a breakpoint conditional" on page 211.
- A graphical **Ada 95 Type Inheritance** and **C++ Class Browser** to delineate the structure of your classes and of classes you inherit. See Chapter 8, "Browse window".

For more information on the following MULTI features, consult your *Building and Editing with MULTI 2000* manual.

Project Management

- A **Program Builder** for creating, assembling, and controlling your programming projects.
- A **Progress Window** to keep you informed at all times as you construct your project.

Version Control

- An automatic **Version Control System** with features for managing revision levels and program branches, and for tracking the origins of suspicious code.
- The capability to **Merge Two or Three Versions** of a file.
- **Highlighted Diff Windows** to see the difference between two files.

Editing

- A built-in **Editor** that is fully configurable, enhanced with special features to support some of the advanced capabilities of MULTI.

Embedded programming in MULTI

MULTI supports embedded development for the 32- and 64-bit microprocessor families listed in the following table:

Processor families supported by MULTI
680x0/683xx
ARM / Thumb
ColdFire
i960
MCore
MIPS
PowerPC
RH32
SPARC
SH
TriCore
V800
x86 / Pentium

Embedded programming is the programming of microprocessors which are incorporated into an embedded product. PCs and Workstations are used as host computers on which programs are edited and compiled. The programs are then downloaded into a target system to be debugged and executed.

MULTI interfaces to embedded targets by connecting to a debug server. The debug server may reside on the same host as MULTI, or on any other host on your network. The debug server communicates with the target under development. Green Hills supplies servers for many common target systems and real time operating systems:

- **Instruction set simulators**: Simulators can test programs before target hardware is ready and are available for most processor models. Instruction set simulators incorporate an integrated debug server as a front end.
- **ROM Monitors**: Monserv and the ROM monitor specific to your target support basic debug features, host I/O, a command window, and profiling.

- **Emulation Probes**: Available for In-Circuit Emulators and On-Chip Debugging Probes. Emulator servers communicate with the probe using a serial port, parallel port, or your network.
- **ROM Emulator**: NetROM provides debugging capabilities with only a single connection to the ROM socket.
- **RTOS (real-time operating system) servers**: Available for several real time operating systems including INTEGRITY from Green Hills Software, ThreadX from Express Logic, OSE from Enea Systems, and VxWorks from Wind River. RTOS servers use ethernet and serial communication to communicate with a debug process running under the RTOS. Commands from MULTI's various debug windows are combined into a single command stream by the RTOS server; the debug process interprets these messages and performs the proper action on the appropriate task.

MULTI allows you to use the same tools for both embedded and native development. The same MULTI program can debug both native and embedded code; the only difference is that MULTI uses a different host processor when communicating with an embedded target.

Running MULTI from the command line

When you start MULTI, it attempts to use the host system windowing package by default. If you start MULTI on a color monitor, it defaults to color. If you start MULTI from a non-windowing monitor or if MULTI encounters problems with the window interface, it comes up in non-GUI mode. If MULTI is incorrectly coming up in non-GUI mode, check that the **DISPLAY** environment variable is set, or set it from the command line with the **-display** option.

If MULTI is in your path, then the command line syntax is:

multi [*options*] [*filename*]

If *filename* is an executable program file that has had some (or all) of its component modules compiled for debugging (with Green Hills compiler's **-G** or **-g** options), then the Debugger starts up. For a list of command line options to use when opening the Debugger, see "Command line options" on page 6.

If *filename* is a build file, then the Builder starts up with the build file loaded. Note that some options are specific to the Debugger and are not applicable to the Builder. If you specify a build file, it can either be a main project or a subproject. To open a subproject directly with the inherited options from a

particular main project, specify the main project name followed by the subproject's name in quotes. For example:

```
multi "main.bld sub.bld"
```

This opens the subproject, **sub.bld**, with the options inherited from **main.bld**.

See the following table of examples.

How to open MULTI	
Example	**Description**
multi	The Builder is invoked on the last build file that was open.
multi default.bld	The Builder is invoked on **default.bld**. If **default.bld** is not found, MULTI will create it.
multi foo.bld	The Builder is invoked on the file **foo.bld**. The build file may be a main project or a subproject.
multi "parent.bld child.bld"	The Builder is invoked on the subproject **child.bld** directly with the inherited options from the main project **parent.bld**.
multi a.out	The Debugger is invoked on the executable **a.out**
multi -remote simppc	The Builder is invoked, and is connected to the simulator **simppc**. In this syntax, it is a function of the debug server whether the Builder window or the Debugger window is invoked. See the example below with **multi -remote 5emon**.
multi -remote simppc a.out	The Debugger is invoked on the executable **a.out** and the Debugger is connected to the simulator **simppc**.
multi -remote 5emon	The Debugger is invoked and is connected to the debug server **5emon**. In this syntax, it is a function of the debug server whether the Builder window or the Debugger window is invoked. See the example above with **multi -remote simppc**.
multi foo.c	The Editor is invoked on the file **foo.c**.

Command line options

When you start MULTI from the command line on an executable program file (i.e. when you want to use the Debugger directly), then the following options may be used. Some of these options should not be used when starting MULTI on a build file, or when starting MULTI without a file.

-c *file*

Reads configuration information from file.

-C *corefile*
Sets core file. *corefile* is assumed to be a core image of *objectfile*.

-D
Ignores all currently specified alternate directories.

-data *offset*
Offsets for all data addresses. This is for position independent data. The offset is entered in decimal by default. A hexadecimal number may be specified by preceding the number with **0x**. For example, **0x10000**.

-dotciscxx
Treats files ending in **.c** as C++ files instead of C files.

-e *entry*
Specifies entry label. The default is **main**. In C++ mode, the entry must be specified in such a way that it may be demangled.

-E *file*
Tells MULTI to debug more than one file. Use this option for each file you wish to debug at the same time. For example, if you want to debug **foo**, **bar**, and **rin**, then type:

```
multi foo -E bar -E rin
```

-help
Runs MULTI and opens the on-line help system with the MULTI manual.

-I *directory*
Names an alternate directory where files are searched for. Alternate directories are searched in the order given. If a file is not found in an alternate, the current directory is searched.

-L[cpfC]
Sets language type (C, Pascal, FORTRAN, or C++ respectively). By default, MULTI uses the file name extension to determine the language.

-m *file*
Uses *file* as default specification file. See “Specification file” on page 8 for more information.

-nocfg
Does not read any of MULTI’s .cfg files on startup.

-norc
Does not run any .rc files on startup.

-noshared
Does not debug shared libraries.

-nosplash
Does not open the About banner. See “About MULTI...” on page 41 for more information.

-p *file*
Startup with command playback from *file*.

-P *pid*

Attaches to process with process id *pid*. This option is currently for Solaris only.

-r *file*
Startup with commands recording to *file*.

-R *file*
Startup with commands and output recording to *file*.

-rc *file*
Reads *file* as a command script when the first debugger window appears. The file is read after the global and user script files.

-remote *target*
Attaches to remote debug server with name *target.*

-text *offset*
Offsets for all text addresses. This is for position independent code. The offset is entered in decimal by default. A hexadecimal number may be specified by preceding the number with **0x**. For example, **0x10000**.

-V

Prints debugger version information.

Specification file

The specification file allows you to set up a default set of command line arguments that may be used with any given executable you want to debug. However, not all command line options are available for use in a specification file. If you want a set of default arguments for each program, put the program name at the beginning of a line followed by a space and then a set of command line arguments. The arguments may be continued on the next line if the first character in that line is a tab. When MULTI runs with the **-m** option, the file listed is checked and if there is an entry that matches the name of the executable being debugged, then that list of command line arguments are used. For example, a specification file named **albatross** might look like this:

```
foo -norc -I /usr/joebob -I /usr/foodir
```

```
bar -text 10000 -data 10000
```

If you then type:

```
multi -m albatross foo
```

the file **albatross** is searched and the arguments found after **foo** are used. This is equivalent to typing:

```
multi foo -norc -I /usr/joebob -I /usr/foodir
```

Resources

To install MULTI, see the *MULTI 2000 Installation & Licensing Guide*.

This manual describes the features of the MULTI debugger.

The *Building and Editing with MULTI 2000* manual describes the features of the MULTI Development Environment other than the debugger.

The *Quick Reference Card* summarizes the most common Debugger and Editor commands.

For assistance or additional information about the use of Green Hills Software, contact our Technical Support:

North America Mountain/Pacific time, Australia, and New Zealand:

Tel: (805) 965-6044, Fax: (805) 965-6343

email: support-west@ghs.com

North America Eastern/Central time, South America:

Tel: (781) 862-2002, Fax: (781) 863-2633

email: support-east@ghs.com

Europe, Africa, India:

email: support-nl@ghs.com

Japan, Taiwan, and South Korea:

Tel: +81-3-3576-6805, Fax: +81-3-3576-0106

email: support@adac.co.jp

Chapter 2

Debugger GUI

This chapter contains:

- Main debugger window
- Debugger menus
- Debugger toolbar
- Generic debugger window features
- Other window topics

This chapter shows you how to run and use the MULTI debugger (“the debugger”). It includes a description of all the debugger buttons and menus. It is implied in these Chapters that the program you’re debugging has been built with Debugging Level set to MULTI (equivalent to the **-G** compiler option).

Main debugger window

To open the debugger, do one of the following:

- From the Builder, click Debug ().
- From the command line of your host system, start MULTI on a program (e.g. **multi a.out**)
- From the command line of your host system, start MULTI attaching to a target where the program is already running.

The following shows the main debugger window:

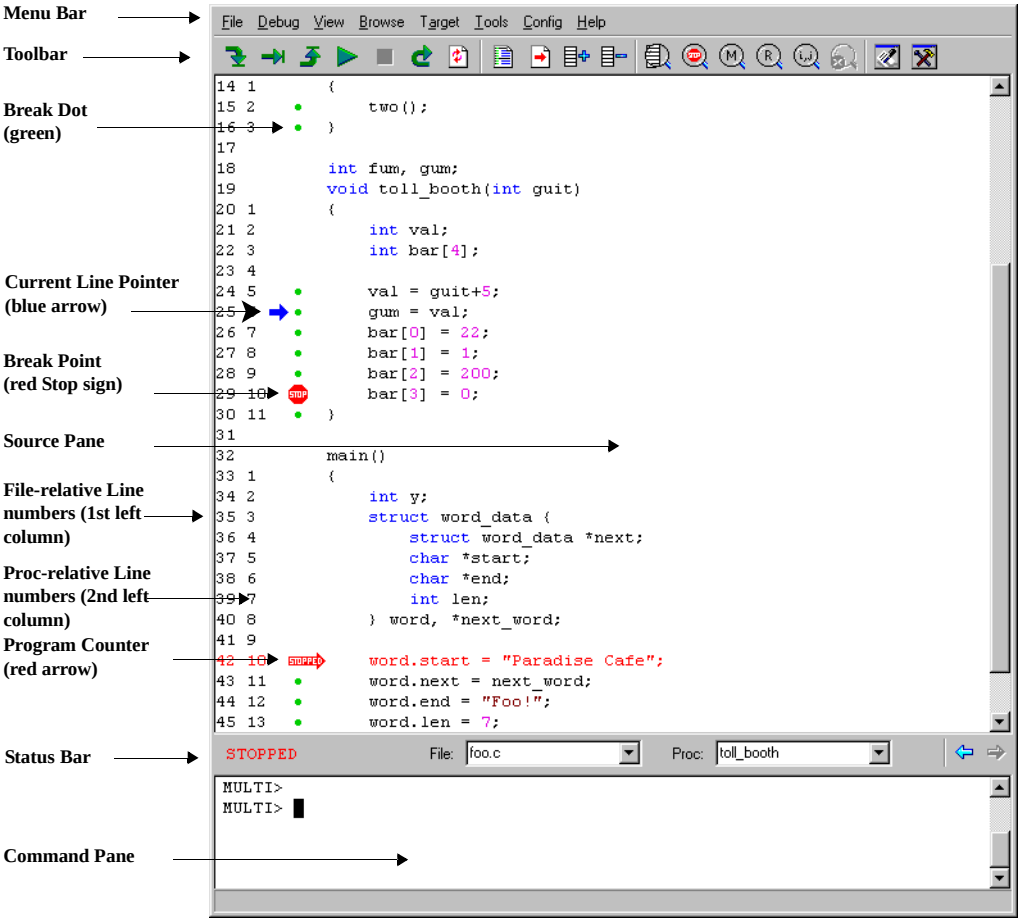


Figure 1 **Main debugger window**

Source pane

Below the menu bar is the debugger tool bar, and below that is the source pane. The source pane shows the source code which can be in C, C++, FORTRAN, Pascal, Ada, or assembly language. When you're debugging, the source pane normally displays the code where the program has stopped, indicated by the red STOPPED arrow. There are several ways to change the source pane to view other locations. For instance, you can click a procedure name in the source pane

to display source code for that procedure in the source pane. You can also use the status bar (discussed below) to load a particular procedure or file into the source pane.

When you right click anywhere in the source pane, a pop-up menu appears which shows some information about the clicked object, and which shows a list of operations you can perform. We will discuss the right-click pop-up menus in a separate section later in this chapter.

In the debugger window, all key strokes go to the command pane (discussed below), unless the focus is in the File or Procedure drop-down list boxes (discussed below). Some key strokes affect the source pane; the following is a list of the frequently used ones:

To do this	Press
Scroll the source pane up by one page.	PageUp
Scroll the source pane down by one page.	PageDown
Scroll the source pane up by one line.	Shift+UpArrow
Scroll the source pane down by one line.	Shift+DownArrow
Search forward in the source pane.	Ctrl+f
Search backward in the source pane.	Ctrl+b
Erase the current searched pattern if the debugger is in incremental search mode.	Ctrl+u
Run the debugged program.	F5
Step out of the current procedure.	F9
Run to the next statement (skip over function call).	F10
Run to the next statement (step into function call).	F11
Go up one level on the stack call.	Ctrl+ + (plus sign)
Go down one level on the stack call.	Ctrl+ – (minus sign)

You can configure the functions for both the keystrokes and mouse clicks.

Breakdots

A breakdot (•) is a small green dot. Breakdots appear directly to the left of certain source lines. These dots indicate lines of source code that correspond to executable instructions; you can set breakpoints on these lines. Lines without breakdots are source lines which do not correspond to executable instructions (for example, source lines that did not generate any instructions).

To set a breakpoint on a source line, click the breakdot. See Chapter 11, “Breakpoints window” for additional methods to set breakpoints.

You can define the color of these dots. The default color is green.

Breakpoint markers

A breakpoint () is denoted by a small red stop sign. To set a breakpoint, click a breakdot; the breakpoint marker will replace the breakdot. To remove a breakpoint, click it; the breakpoint marker will revert to a breakdot.

Current line pointer

The current line pointer () is a small blue arrow directly to the left of the breakdots. The pointer is strictly a debugger tool, unrelated to the current program counter (PC) of the program. Many debugger commands, such as the breakpoint command, use the pointer as the default and these commands execute at that location in the program. For example, the breakpoint command **b** with no specified line number sets the breakpoint at the current line pointer location. When a running program halts, the pointer is set to the line where the program’s execution halts.

The current line pointer is always associated with source code; that is, it always appears in the source pane whenever there is source code. If the source line at the current line pointer disappears from the source pane due to scrolling, the debugger will relocate the pointer to either the top or bottow source line in the source pane.

Line numbers

Previous versions of the debugger (1.8.9 and earlier) only have one column of line numbers on the left, and they are file-relative. This version of the debugger continues to support file-relative line numbers. In addition, it now also supports procedure-relative line numbers. You can configure the debugger to display one, both, or none of them.

When both file-relative and procedure-relative line numbers are displayed, you can choose which to display left-most with one of the following methods:

- Choose Config > Options... > Debugger tab, and use the check box: Use procedure relative line number (vs. file relative).
- Use the **configure** command with the option **procRelativeLines**. That is:
  - configure procRelativeLines true
  - configure procRelativeLines false.

See “Procedure-relative vs file-relative line numbers” on page 67.

Note: configuration options are not case sensitive, thus procRelativeLines and procrelativelines are the same option, but for clarity we use procRelativeLines here. (System variables are also not case sensitive.)

PC pointer / Highlighted line

The PC (program counter) pointer () is a larger red arrow with the word STOPPED. When the program stops at some point in the code, the PC pointer is directly over the breakdot for that line. The PC pointer indicates the position to execute next if you perform a **go** () or **single step** ( or ) command.

C++ Templates and Ada Generics

When debugging C++ templates or Ada generics, the source pane contains some lines of source code with and without breakdots. Ada generic instantiations introduce another physical source file, that of the source of the generic, into the symbol table at the point of the instantiation. C++ template instantiation may also introduce other physical source files into the symbol table. That is, the symbol table for the current source file is fragmented by the instantiation. If you encounter this problem, use the **e** command to view different procedures or tasks within a single piece of source. (See the command **e** on page 101.)

dblink

The debugger only reads the symbolic debug information contained in **.dnm** and **.dla** symbol files. **dblink** is a utility that generates such symbol files. It extracts the symbol table from the host system object code and translates it into an independent representation that the debugger uses. The output files are the name of the executable file, with a **.dnm** extension and a **.dla** extension respectively. For example, **a.out** will generate **a.dnm** and **a.dla**.

If the existing **.dnm** file is older than the executable, or if the **.dnm** file was previously generated on a different host machine, the debugger will open a dialog box to ask you to choose whether to regenerate the symbolic debug information. If you choose Translate, **dblink** will be launched to generate the new symbolic debug information.

If you move or copy the executable to another location, be sure to also move or copy the associated **.dnm** and **.dla** files. Then, either preserve the original time stamps of the files, or make sure that the .dnm file has an earlier time stamp than the executable’s.

Interlaced source view

The source pane may display either high level source code alone or source code interlaced with disassembly instructions. The interlaced view shows you the machine instructions that correspond to each source line.

To get an interlaced view, do one of the following:

- Click the Assembly button (). This toggles between high level source code only and interlaced view.
- In the command pane, enter _DISPMODE:=1. To return to high level source code only, enter _DISPMODE:=0. Here _DISPMODE is a debugger system variable. See “Special variables” on page 51.

Assembly code view

When assembly code is displayed, the address, in hexadecimal, of each assembly instruction is shown in the source pane. When the program stops at a line in the high-level source code, the PC pointer is placed at the first assembly instruction associated with the high-level source line. Not all high-level source lines generate executable code.

To set a breakpoint on an assembly instruction, click the breakdot.

Assembly-only view

If no high-level source code is available or if the module was not compiled with debugging information (the **-G** or **-g** option in the compiler), the debugger will display assembly code only.

When viewing only assembly code in the source pane, you can scroll through all of memory in either direction. In this mode, you cannot drag the scroll thumb. See “Infinite scrolling” on page 39 for more information.

Status bar

The status bar is between the source pane and the command pane:

It displays various state information about the currently debugged program. To change the relative sizes of the source and command panes, drag the status bar up or down.

The status bar, from left to right, consists of the Status, the File drop-down list (“File:”), the Procedure drop-down list (“Proc:”), the Back button, and the Forward button.

Status

Messages about the state of the debugger appear in the Status section on the far left of the status bar. Different messages will appear based on the priority of the messages. Program state messages have the lowest priority, error messages are next, and informational messages have the highest priority.

Program state message	Description
NO PROCESS	The program to debug has not started.
STOPPED	The program being debugged is stopped.
RUNNING	The program being debugged is currently executing.
DYING	The program being debugged is hung up and killed
FORKING	The debugged process is being forked (UNIX only).
EXEC'ING	The debugged program performs an **exec**.
CONTINUING	The program is preparing to begin execution.

Informational Message	Description
SSrch: *string*	The source pane is being searched with the incremental search utility for the pattern *string* (see "Incremental search" on page 40).

File drop-down list

The File drop-down list displays the base name of the current file:

If the file name is only partially displayed or you want to see the full name of the file, move the cursor over the drop-down list and the full name will appear in a tooltip after a short period of time.

To browse all the source files in the program:

1. Open the File drop-down list (click the drop-down button).
2. Choose "Browse all source files in program..."
3. Choose from the "Source Files with Procedure" window that appears. See also "Browse window for source files" on page 195.

The File drop-down list also contains up to ten of the most recently displayed files in the source pane. They are sorted according to the time they were loaded into the source pane, with the latest one at the top of the list. When you choose a

file from this list, the debugger will load the file into the source pane, with the current line pointer at the position when the file was last in the source pane.

Procedure drop-down list

The Procedure drop-down list displays the name of the current procedure:

If the procedure name is only partially displayed, move the cursor over the drop-down list and the full name will appear in a tooltip after a short period of time.

To browse all the procedures in the program:

1. Open the Procedure drop-down list (click the drop-down button).
2. Choose “Browse procedures in program...”
3. Choose from the “Procedures” window that appears.

To browse all the procedures in the current file:

1. Open the Procedure drop-down list (click the drop-down button).
2. Choose “Browse procedures in current file...”
3. Choose from the “Procedures: *current_file*” window that appears, where *current_file* is the name of the current file.

See also “Browse window for procedures” on page 191.

The Procedure drop-down list also contains up to ten of the most recently browsed procedures in the source pane. They are sorted according to the time they were loaded into the source pane, with the latest one at the top of the list. When you choose a file from this list, the debugger will load the file into the source pane, with the current line pointer at the position when the procedure was last in the source pane.

History navigation buttons

The debugger not only keeps a history of the objects shown in the source pane according to time, but it also keeps a history of procedures according to your browsing logic. To navigate the procedures according to your browsing logic, press the Back ( ) and Forward ( ) buttons on the status bar.

Let's use an example to illustrate the two different orders, time versus browsing logic, using the following program segment:

file1.c	file2.c	file3.c

current()
{
  son();
  daughter();
}
	son()
	{
	  grandchild();
	}
		grandchild()
		{
		  foo();
		}
		
		daughter()
		{
		  bar();
		}

You do the following:

1. You are viewing "current" in the source pane.
2. Click "son" to examine it.
3. Click "grandchild" to examine it.

Now, the top of the procedure drop-down list and the browsing logic history look like this:

Top of Proc drop-down list	Browsing Logic History
grandchild son current etc.	grandchild son current etc.

So far, the two orders are the same.

Now, you do the following:

1. Click the Back button () to go from "grandchild" to "son".
2. Click the Back button again to go from "son" to "current".
3. Click "daughter" to examine it.

Now “daughter” is in the source pane. The two orders now look like this:

Top of Proc drop-down list	Browsing Logic History
daughter current son grandchild etc.	daughter current etc.

They are different; “son” and “grandchild” have been pruned from the browsing logic list.

See also **indexnext** on page 110 and **indexprev** on page 110.

Command pane

The command pane accepts debugger commands for the process being debugged and displays the output of those commands. It is directly below the status bar. When the debugger window is the active window on your desktop, all key strokes go into the command pane, unless the focus is on one of the menus or one of the drop-down list boxes on the status bar. For a detailed description of the debugger commands, see Chapter 4, “Debugger commands”.

The default prompt in the command pane is: **MULTI >**

To change the prompt, do one of the following:

- Use the **configure** command with the option **prompt**.
- Choose Config > Options... > Debugger Tab. Set the Command pane prompt field.

The debugger keeps a history of all the debugger commands entered.

To execute a command from the history, do one of the following:

- Use the **!** (exclamation point) command.
- Use the UpArrow and DownArrow keys to navigate through the history and choose one to execute.

The following is a list of often used keys in the command pane:

To do this	Press
Bring the previous comand in history into the command buffer.	UpArrow
Bring the next command in history into the command buffer.	DownArrow
Scroll up the command pane.	Ctrl+UpArrow
Scroll down the command pane.	Ctrl+DownArrow
Clear the command buffer if the command pane has the focus.	Ctrl+u

The following is a list of often used mouse clicks in the command pane:

To do this	Do this
Copy a string onto the clipboard.	left-click and drag
Paste the string in clipboard to the command buffer.	middle-click
Paste the string in clipboard to the command buffer.	right-click

Functions for both keys and mouse clicks are configurable. See also the Configuration chapters in the *Building and Editing with MULTI 2000* manual.

Debugger menus

A menu item that is followed by an *ellipsis* (...) means additional information is required before the debugger can execute the operation. When you choose such a menu item, a dialog box will prompt you to supply or confirm the information.

Dimmed out menu items are those inapplicable in the current context. For example, if the debugged program is running, the Go item in the Debug menu will be dimmed out, because Go does not make sense in this context.

The following sections describe each of these menus.

File menu

NOTE: Spaces are not allowed in filenames. This restriction applies throughout the entire MULTI development environment.

File menu (debugger)	
Menu Item	**Description**
Debug Program in New Window...	Opens a dialog box which asks for the name of a program to debug. A new debugger window will be brought up on that program. The new debugged program's resource file (if any) will be executed and the final debugging environment will be shared by the new debugger and the existing debugger window(s). See **dbnew** on page 96.
Debug Program...	Similar to Debug Program in New Window..., except the current debugger window will be used to debug the new program, and the old program will be terminated. The new debugged program's resource file (if any) will be executed to establish the final debugging environment. Any configuration changes made while debugging the previous program will remain in effect unless overridden.
Print...	Opens the Print dialog box to print the current source file. Only ASCII text is printed. The entire source file is printed, including interlaced assembly code if that is the current display mode. If no source file is available (that is, there is only assembly code), then only the content of the source pane is printed.
Print Window...	Opens the Print dialog box to print only the visible, ASCII contents of the source pane.
Print to File...	Similar to the Print... menu item, except the output is directed into a text file.
Attach to Process...	Attaches to a running process. See **attach** on page 80. This command works only with a multi-tasking target and is grayed out otherwise. It opens a new debugger window to debug the specified task.
Detach from Process	Closes the debugger but leaves the debugged process running. See **detach** on page 99.
1,2,3,4	List the most recently debugged programs. To debug any of them in the current debugger window, click it.
Close Debugger	Closes the debugger window. If this is the last MULTI window, MULTI itself also exits.
Exit All	Closes all of MULTI's windows and exits.

The Print dialog box contains the following items:

Print dialog box	
Item	**Description**
Print To:	Choose between printing to the “Printer” or to a “File”.
Print Command	You should enter the command for printing a file on your operating system (such as **lpr** on Unix). You can set it with **configure** command with the option **printCommand**. See **configure** on page 92 and **configurefile** on page 93.
File Name	If you choose “Print To File”, this is the output post-script file.
Browse	If you choose “Print To File”, this button let you select the output file.
Font Name	From this drop-down list box, you choose from a list of available fonts on your system matching the pattern “*-r-normal-*-m-*”. You can also type a font name into this field if it is not on the list.
Font size	Choose the font size used for printing.
Print/Cancel Button	Let you perform/cancel the print request.
Others	Let you define paper size, orientation, and the number of columns to print.

Debug menu

Debug menu (debugger)		
Menu Item	**Press**	**Description**
Set Program Arguments	n/a	Opens a dialog box to enter arguments for your program when it runs. It allows you to control input and output redirection to and from your program. See **setargs** on page 134 and **r** on page 127. See the table “Set Program Arguments dialog box” below.
Go	 F5	Starts running a program which has not been started or continues executing one which has stopped. This command cannot be used to start tasks on VxWorks systems. See **runtask** on page 130 for starting tasks on these systems.
Restart		Starts running or restarts the currently debugged program with preset arguments, if any. See the command **r** on page 127.
Halt		Halts the current program. See **halt** on page 107.
Kill Process	n/a	Kills the current program. See **k** on page 112.

Step	 F11	Executes single statements and steps into procedure calls. See **s** on page 131.
Next	 F10	Executes single statements and steps over procedure calls. See **n** on page 121.
Return	 F9	Continues to the end of the current subroutine and stops in the calling routine after returning to it. See **cu** on page 94.
Send Signal	n/a	Opens a dialog box to let you specify the signal name and then sends a signal to the current program. For signal names, see the **l** command with the **z** option. See **l** on page 113 (lowercase **L**).
Add Assertion	n/a	Opens a dialog box to let you specify a logic expression. When you click OK, the debugger creates an assertion so that whenever the logic expression becomes true, it stops the running process and prints a message to indicate that the assertion is hit. See **a** on page 77 and **assert** on page 79.
Set Watchpoint	n/a	Opens a dialog box to let you create a watch point. See **watchpoint** on page 141.
Remove All Breakpoints	n/a	Removes all software breakpoints from the current program.

The following table shows the components of the Set Program Arguments dialog box:

Set Program Arguments dialog box	
Item	**Description**
Input File	This is a file on the host system which will be used as input to your program.
Output File	This is a file on the host system which will capture output from your program.
Program arguments	These are the arguments passed to the debugged program the next time you run it without specifying any arguments. For example, if you click the Go or Restart button, or if you execute the **r** command, etc. If you specify any arguments in the next execution, for example through the **r** command, then the existing arguments will be replaced by the new ones.
Run	Click this button to run your program with the given arguments.
Set	Click this button to set the given arguments as the default for this session. See **r** on page 127.

View menu

The following is the View > Nagivation sub-menu:

View menu (debugger)		
Menu Item	**Button and/or command**	**Meaning**
Navigation	n/a	The Navigation sub-menu. See “View > Navigation sub-menu” below.
Interlaced Assembly	 **assem**	Toggles between source code only and interlaced source with assembly code.
Breakpoints...	 **breakpoints**	Opens the Breakpoints dialog box. See Chapter 11, “Breakpoints window”.
Call Stack...	 **callsview**	Opens a Call Stack window.
Local Variables...	 **view $locals$**	Opens a Local Variables window.
Registers...	 **regview**	Opens a Registers window.
Memory...	 **memview**	Opens a dialog box for you to specify which memory address to view, then brings up a Memory View window for the address.
Find Memory Leaks...	**findleaks**	Opens a window displaying information regarding memory allocations by the program being debugged.
Profile...	**profile**	Opens the Profile window.
Tasks...	**taskwindow**	Opens the Task window.
Print Expression...	n/a	Opens a dialog box for you to specify an expression to be printed.
View Expression...	n/a	Opens a dialog box for you to enter an expression to evaluate.
List	n/a	The List sub-menu. See “View > List sub-menu” below.
Source Path...	n/a	Opens the Source Path window. To change the source path, make the edits and press OK. To discard the changes, press Cancel.
Refresh Views	**update**	Refreshes all non-frozen data explorers, including the Register windows, Memory View windows, Call Stack windows, etc. (If a data explorer is frozen, this command does not update the window. See also “Title bar” on page 148.)
Close All Views	 **viewdel**	Closes all the data explorers, including the Register windows, Memory View windows, Call Stack windows, etc.

View > Nagivation sub-menu		
Menu Item	**Press**	**Description**
UpStack	 Ctrl+ +	Views the procedure one higher on the call stack.
DownStack	 Ctrl+ –	Views the procedure one lower on the call stack.
Current PC		Views the procedure where the program is currently stopped.
Upstack To Source	n/a	Views the first procedure higher on the call stack which has source code. For example, you can use this feature if you are stopped inside a library function with no source code (such as **printf**), and you wish to return to viewing your program.
Goto Location...	n/a	Opens a dialog box for you to specify a procedure or an address; view the program at the specified location.

The following is the View > List sub-menu:

List sub-menu	
Menu Item	**Description**
Files	Lists all the file names in the program.
Procedures	Lists the basic information of all the procedures, such as their names and addresses.
Mangled Procedures	List the mangled names and other basic information of all the procedures.
Globals	Lists the basic information of all the global variables, such as their names and addresses.
Statics	Lists all static variables.
Locals	Lists all local variables for the procedure you are viewing (i.e., the procedure at the current line pointer) if the procedure is on the stack.
Local Addresses	Lists the addresses of the local variables specified above.
Registers	Lists all registers.
Register Synonyms	Lists register synonyms.
Variables In Procedure...	Lists all parameters and local variables of the specified procedure if it is on the stack.
Defines	Lists all defined macros.
MULTI Variables	Lists MULTI's internal variables.
Processes	Lists processes currently being debugged.
Signals	Lists signals.
Assertions	Lists assertions.
Breakpoints	Lists all breakpoints.
Dialog Boxes	Lists all dialog boxes.
Source Paths	Lists the directories where MULTI looks for source files and scripts.

Browse menu

Browse menu (debugger)	
Menu Item	**Description**
Procedures...	Opens a Browse window to show all procedures of the program. See “Browse window for procedures” on page 191.
Globals...	Opens a Browse window to show all globals of the program. See “Browse window for globals” on page 193.
Files...	Opens a Browse window to show all source files of the program. See “Browse window for source files” on page 195.
Classes...	Opens a Tree Browser to show the the class hierarchy of the debugged program. See Chapter 12, “Tree browser”.
Static Calls...	Opens a Tree Browser to show the static calling relationships of the current procedure, that is, the procedure at the current line pointer, if any. See Chapter 12, “Tree browser”.
Dynamic Calls...	Opens a Tree Browser to show the dynamic calling relationships of the current procedure, that is, the procedure at the current line pointer, if any. See Chapter 12, “Tree browser”.
File Calls...	Opens a Tree Browser to show the reference relationships of the current file. See Chapter 12, “Tree browser”.
Procedures In File...	Opens a dialog box to let you specify a file name, then bring up a Browse window to show all procedures in the file. See “Browse window for procedures” on page 191.
Type...	Opens a dialog box to let you specify a type name, then bring up a data explorer to show the structure of the type.

Target menu

Target menu (debugger)	
Menu Item	**Meaning**
Connect to Target...	Opens a dialog box to let you specify the target to connect and the corresponding parameters.
Disconnect from Target...	Disconnects from the current remote target debug server.
Show Target Windows	Displays remote target windows. These windows normaly appear upon connecting to the debugger server. They may be closed without disconnecting from the debug server, in which case this option will make them reappear.
1,2,3,4	List the most recently connected debug servers. To connect to any of them, click it.
Load Program	The Load Program sub-menu. See “Target > Load Program sub-menu” below. Loads the debugged program into the target system’s memory.
Refresh Section...	Reloads the specified section of the current program: **text**, **data**, or **all** into the target system’s memory.
IO Buffering	Toggles buffering for the remote I/O window.
Memory Manipulation	The Memory Manipulation sub-menu. See “Memory Manipulation sub-menu” below.

The following is the Target > Load Program sub-menu:

Load Program sub-menu	
Menu Item	**Meaning**
Load Program...	Loads a program into the target systems’s memory.
1, 2, 3, 4	Lists the most recently loaded programs. To load any of these programs into the target system’s memory, click it.

The following is the Target > Memory Manipulation sub-menu:

Memory Manipulation sub-menu	
Menu Item	**Meaning**
Copy...	Copies memory. The copy continues for the specified number of sections of memory chunks. You can copy memory backwards or forwards.
Fill...	Fills the given sections of memory with the given value.
Find...	Finds a value in memory. The search continues for the specified number of sections of memory. Each memory value is bitwise **AND**'ed with the mask before compared.
Compare...	Compares memory. Specify the two starting memory locations to compare, and the number of sections and chunks of memory to compare. You can also specify the comparison operation: **==**, **>**, **>=**, **<**, **<=**, **!=**.
Memory Load...	Copies a section of memory from specified file.
Memory Dump...	Dumps a section of memory to specified file.

Tools menu

Tools menu (debugger)	
Menu Item	**Meaning**
Builder...	Opens the Builder window on the project for the current program. If MULTI cannot find a project for the current program, it will bring up a builder window on a new project.
Rebuild...	Rebuilds the current program if the project for the program can be located.
Editor...	Opens an Edit File dialog box to let you select a file to open in an Editor window.
Notes	Opens an Editor window on a scratch file.
Search...	Opens MULTI's search window. See "Search dialog box for the source pane" on page 41.
Grep...	Launches the **grep** utility with the specified string. **grep** is a program which searches files for a given string. The debugged program's source files and any other open files are searched.

Config menu

Config menu (debugger)	
Menu Item	**Meaning**
Options...	Displays the Options dialog box, which you use to change options that affect the way the Debugger and other MULTI tools look and behave. .
Save Configuration as Default	Saves the current configuration into the default user configuration file, so that it will be automatically executed when MULTI starts in the next session.
Clear Default Configuration...	Deletes the default user configuration file.
Save Configuration...	Opens a file chooser dialog box to let you specify a file and then save the current configuration in it.
Load Configuration...	Opens a file chooser dialog box to let you choose a file and then execute the configuration statements from it.
State	The State sub-menu. See “State sub-menu” below.

The following is the Config > State sub-menu:

State sub-menu	
Menu Item	**Description**
Show Command History	Prints command history. MULTI keeps a history of all the debugger commands. You can use the **!** command to execute a command from the history. You can also use the UpArrow and DownArrow keys to navigate through the history and choose one to execute. You can use the **h** command to do the same thing. See **h** on page 107.
Save State...	Saves the state of the debugger to the specified file. The saved information includes remote connection and status, breakpoints, assertions, and the source directories list. You can use the **save** command to finish the same task. See **save** on page 132.
Restore State...	Restores the state of the debugger from the specified file. You can use the **restore** command to achieve the same result. See **restore** on page 129.
Record Commands...	Records commands into the specified file. You can get the same result with the **>** command.
Record Commands+Output ...	Records commands and their output into the specified file. You achieve the same result with the >> command.
Stop Recording Commands...	Stops recording comands. It's equivalent to the **>c** command. Use this item to stop recording if it's started by "Record Commands...".
Stop Recording Commands+Output ...	Stops recording comands and their output. It's equivalent to the **>>c** command. Use this item to stop recording if it's started by "Record Commands+Output...".
Playback Commands...	Plays back commands recorded in the selected file. You can do the same thing with the **<** command.

Help menu

Help menu (debugger)	
Menu Item	**Description**
Debugger Help...	Opens online help for the debugger.
Manuals	Opens the “Manuals sub-menu”, which will display a list of manuals appropriate to your version of MULTI. Choosing one of these manuals will open the online help to the first page of that manual.
About MULTI...	Opens the About window. It contains the basic information about MULTI, such as its version, and copyright materials. To dismiss it, click in it.

Debugger toolbar

The toolbar appears just below menu bar in the main debugger window. By default, all buttons are shown as icons. If you prefer to use text button as in previous versions of the debugger, do the following:

1. Choose Config > Options... > General Tab.
2. Disable (uncheck) the option “Use icons for buttons”.
3. Press OK.

All these buttons are programmable, except for Quit. You can define each button’s name, the corresponding icon (optional), and its command string in any of the following ways:

- Define them in a configuration file.
- Define them interactively during a debug session with the **debugbutton** command. See **debugbutton** on page 97.
- Choose Config > Options... > Debugger Tab > Configure Debugger Buttons.

When you program a button, its new name (or icon) appears in the button, and the command string goes to the command pane for execution when you click the button. If no icon is specified, a character icon for the first character of the button name will be used (if the buttons are displayed as icons).

By default, the debugger defines 20 buttons. You can change any of the default buttons in a start up database or during run-time, except for Quit.

If you prefer the toolbar to be at the bottom of the debugger window instead of at the top, do the following:

1. Choose Config > Options... > Debugger Tab.
2. Choose “Position of buttons” > Bottom.

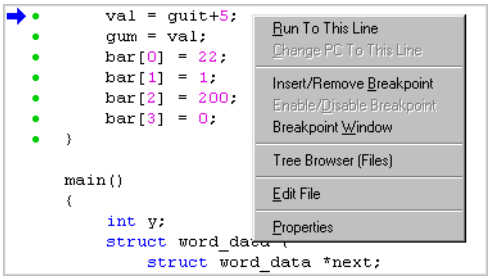
The following table shows the default debugger buttons, their ID numbers, their names, and their equivalent debugger commands (if any):

Debugger toolbar			
Num.	**Button**	**Command**	**Description**
1	 Step	s	Executes one statement. If the statement is a procedure call, it steps into the called procedure. When in interlaced source/assembly mode, a machine instruction is executed instead of a source statement.
2	 Next	n	Executes until the next statement of the current function (i.e. step over procedure calls). When in interlaced source/assembly mode, a machine instruction is executed instead of a source statement.
3	 Return	cU	Continues to end of procedure, and stops in the calling procedure after returning to it.
4	 Go	C	Begins execution of the program. If the program is stopped, it continues execution.
5	 Halt	halt	Interrupts program execution.
6	 Restart	restart	Restarts the program with the same arguments as before.
7	 Reload	debug	Reloads the current executable.
8	 Assem	assem	Toggles between displaying the source code only and source interlaced source with assembly code.
9	 PC	E	Shows the position at the current Program Counter.
10	 Upstack	E+	Views a procedure up one stack frame.
11	 Downstk	E–	Views a procedure down one stack frame.
12	 Calls	callsview	Opens a window displaying a stack trace. See also **callsview** on page 89 and Chapter 10, “Call stack window”.
13	 Stops	breakpoints	Opens the Breakpoints window to add and edit breakpoints.
14	 Memory	memview 0	Opens Memory View window displaying memory starting at address 0x0.
15	 Regs	regview	Opens Register window displaying machine registers.
16	 Locals	view $local$	Creates a data explorer displaying local variables.

Debugger toolbar			
Num.	**Button**	**Command**	**Description**
17	Viewdel	viewdel	Deletes all data explorers, Register windows, Call Stack windows, Breakpoints windows, Memory View windows and Browse windows.
18	Edit	edit	Opens an Editor window on the currently active procedure.
19	Builder	builder	Invokes the Builder window.
None	Quit	quit	Quits MULTI, but if the debugged program is being debugged, MULTI gives you the choices of “Quit and kill Process” and “Detach from process”. You may not re-configure this button to perform another function, but you can configure whether this button appears or not.

Pop-up menus

When you right click in the source pane, a pop-up menu appears:



This is the default behavior. If you have configured the right-click to perform other functions, the pop-up menu will not appear.

This menu is context-sensitive, and depends on the object you click. Different objects in the source pane have different corresponding pop-up menus. Some of the menu items may be grayed out which means they are unavailable given the context.

When we discuss the pop-menus below, we use the term “right-clicked line” to refer to the source line where you’ve just right-clicked.

The pop-up menu has a title to show some basic information about the object.

Pop-up menu for a procedure

When you right click a procedure, a pop-up menu appears with the following items:

Pop-up menu for a procedure	
Menu Item	**Description**
Run To This Line	Runs the program to right-clicked line if there is executable code there. Note: The debugger will set a special breakpoint there, but there is no guarantee that the program will actually stop there, because it is up to the program logic.
Change PC To This Line	If the right-clicked line contains executable code and it is within the same procedure in which the program is currently stopped, changes the program counter to there.
Insert/Remove Breakpoint	Toggles a breakpoint at the right-clicked line if there is executable code there.
Enable/Disable Breakpoint	If there is a breakpoint at the right-clicked line, toggles its status (enabled or disabled).
Breakpoint Window	Opens the Breakpoints window. See Chapter 11, “Breakpoints window”.
Go To Definition	If there is source code for the procedure, views the source code in the source pane.
Browse Callers	Opens a browse window to show the callers of the procedure.
Browse Callees	Opens a browse window to show the callees of the procedure.
Tree Browser (Procedures)	Opens a tree browser window to show the calling relationships of the procedure.
Edit This Procedure	Opens an editor window on the procedure’s source file, if it’s available.
Edit File	If the current file is a source file, brings up an editor window on it.
Properties	Opens a dialog box to show the basic information about the current file (e.g. its name, language type, the right click line number, the current target), and the basic information about the procedure (e.g. its size, address, and whether its scope is static or global).

Pop-up menu for a variable

Even though the pop-up menu title shows different information for global variables and local variables, they have the same pop-up menu options at present.

When you right click a variable, a pop-up menu appears with the following items:

Pop-up menu for a variable	
Menu Item	**Description**
View Value	Opens a data explorer to show the value of the variable.
Properties	Opens a dialog box to show the basic information about the current file (e.g. its name, language type, the right click line number, the current target), and the basic information about the variable (e.g. its size, address, and whether its scope is static or global).
(*other options*)	See “Pop-up menu for a procedure” on page 37.

Pop-up menu for a type

When you right click a type, a pop-up menu appears with the following items:

Pop-up menu for a type	
Menu Item	**Description**
View Struct	Opens a data explorer to show the structure.
Properties	Opens a dialog box to show the basic information about the current file (e.g. its name, language type, the right click line number, the current target), and the basic information about the structure.
(*other options*)	See “Pop-up menu for a procedure” on page 37.

Pop-up menu for other objects

When you right click an object other than a procedure, a variable, or a type, a pop-up menu appears with the following items:

Pop-up menu for other objects	
Menu Item	**Description**
Properties	Opens a dialog box to show the basic information about the current file (e.g. its name, language type, the right click line number, the current target).
(*other options*)	See “Pop-up menu for a procedure” on page 37.

Generic debugger window features

All debugger sub-windows support a base set of features and capabilities including support to customize scroll bars. They include text selection for copying text into the command pane as input to a button command, as well as into other applications that support pasting, and incremental text searching capabilities.

Scroll bars

To customize the mouse behavior in the scroll bar, do one of the following:

- In the command pane, use the **mouse** command.
- Use the **configure** command with the **mouse** option.
- In the configuration file, use the mouse command.

See also **mouse** on page 119 and **configure** on page 92.

Infinite scrolling

You can scroll through all of the target's memory in a Memory View window, or in the source pane when only assembly code is displayed. In such cases, the scrollbar will be in *infinite scrolling* mode. See Chapter 9, "Memory view window" and "Assembly code view" on page 17 for more information.

When the scrollbar is in infinite scrolling mode, the thumb is replaced by a diamond. The thumb is fixed in the center of the scroll bar and cannot be dragged. To scroll through memory, you may use the scroll arrows to scroll one line at a time, or click above or below the thumb to scroll one page at a time.

Selecting text

In the source pane, left-click a word to select it. Drag your mouse over an open or close parenthesis to select all the text in between, including the close or open parenthesis.

You may find when selecting text, the debugger performs a command as soon as you release the mouse button. To prevent the debugger from issuing the command, hold down the Ctrl key while selecting text. Alternatively, you can configure the debugger key bindings to prevent issuing commands.

You can select or highlight the text in the debugger and Editor windows to copy to other windows. To select text in these windows, position the mouse at the

beginning of the text, press the left button, and drag the mouse until the desired text is highlighted. Click twice to select a word.

To deselect the current selection, click in an area without text.

You can also transfer your selection to other window applications that support pasting, such as an xterm; however, different applications may behave differently so consult their reference manuals. Generally, when you select text in one window, any previously selected text is deselected.

Incremental search

The incremental search command searches a window for a string. In the source pane, the search starts at the current line pointer while in other windows the search starts at the beginning or the end of the text currently displayed. To place the debugger in search mode, type Ctrl+f for forward search or Ctrl+b for backward search. The message **"**SSrch" (Source pane Search) appears on the left side of the status bar:

While in search mode, as text is typed, the pattern is matched in the window searched, and the first occurrence of the match is highlighted. To find the next or previous match, type Ctrl+f or Ctrl+b. The search wraps around the entire buffer of displayed text. When it reaches the end or the beginning of the window buffer, the debugger beeps to indicate it is about to wrap.

To end the search, press Enter or click the mouse. The string then becomes the current selection. You can control case sensitivity with the **chgcase** command. See **chgcase** on page 91. The incremental search key strokes include:

Key presses for incremental search	
To do this	**Press**
Turn on incremental search, forward (After the initial search has been performed, it will advance to the next matching pattern.)	Ctrl+f
Turn on incremental search, backward (After the initial search has been performed, it will advance to back to a previous matching pattern.)	Ctrl+b
Reset the search pattern	Ctrl+u
End the search and make matched text the current selection	Enter
Delete the last character in the search pattern	Backspace
Place a character in the search pattern and discard control characters	*any character*

Example:

Your text is:

```
this is a string search.
```

Now, start a search through this string by typing Ctrl+f and the first character of the search, say letter '**i**'. Result: the character '**i**' in the word '**this**' is highlighted.

If you now type '**s**', the two characters '**i**' and '**s**' in '**this**' are highlighted.

To jump to the next occurrence of the pattern '**is**', press Ctrl+f again. This puts the selection on '**is**' in the string.

To search next for the pattern '**in**', press Backspace to reset the search string to '**i**'. Now, type '**n**'. The '**i**' and '**n**' in '**string**' are highlighted.

To terminate the search, press Enter. This leaves you with the current selection.

Search dialog box for the source pane

To control searches in the source pane, bring up the search dialog box in one of the following ways:

- Choose Tools > Search...
- In the command pane, enter: **dialogsearch**

For example, to search for a string such as **tree**:

1. Type **tree** in the search text field.
2. Click Find to search for the next occurrence of the string.

The toggle buttons in this window are the same as those in the Editor's search dialog box.

To initiate or repeat searches without the search dialog box, you can also use the Ctrl+f or Ctrl+b key sequences in the command pane.

Variable lifetime debugging

Registerized variables are those such as function parameters or locals whose values are represented solely within registers. When you refer to a registerized variable outside of its scope (as defined by the compiler), the debugger displays the message "Out Of Register Scope", along with the value. Because the variable is out of scope, this value is probably incorrect.

Multiple .text section debugging

The debugger contains full support for multiple **.text** section debugging. The debugger seamlessly debugs both C source and assembly files which contains procedures located in a different text section.

You can load additional symbol information for a module while debugging.

Other window topics

Mouse clicks

You can customize the mouse operation within the debugger. The debugger supports programming of up to five mouse buttons, and you can assign commands for up to five clicks for each button. Examples:

For a 2-button mouse in source pane		
Button	**One Click**	**Two Clicks**
Button 1 (left)	Selects the word that the mouse is on and, if applicable, prints its value.	Opens a data explorer window on the object represented by the word that the mouse is on. See **view** on page 140.
Button 3 (right)	Opens a pop-up menu on the clicked object.	No operation.

For a 3-button mouse in source pane		
Button	**One Click**	**Two Clicks**
Button 1 (left)	Selects the word that the mouse is on and, if applicable, prints its value.	Opens a data explorer window on the object represented by the word that the mouse is on. See **view** on page 140.
Button 2 (middle)	Sets a breakpoint at the line the mouse is on.	Sets a temporary breakpoint on the line where the cursor is and continues the program.
Button 3 (right)	Opens a pop-up menu on the clicked object.	No operation.

Note: if you configure the mouse operation for the right button, you may affect the default behavior of the right-click pop-up menu.

Kanji character support

MULTI fully supports the kanji character set.

To display Kanji characters appearing in source files, you must specify a Kanji font such as **k14**, in your configure file.

Chapter

3

Expressions, variables, and procedures

This chapter contains:

- Evaluating expressions
- Viewing variables
- Viewing memory addresses
- Variable lifetime
- Special variables
- Examining data
- Wildcards
- Procedure calls
- System variables
- Syntax checking

Evaluating expressions

It is often useful to calculate the value of an expression while debugging. To evaluate an expression, simply enter it into the debugger's command pane. The debugger will print the calculated value of the expression in the command pane. If you want to watch the value of an expression, and have it reevaluated as you step through your program, you should use a data explorer instead. See also Chapter 5, "The data explorer".

When you construct your expressions, beware that:

- If the expression begins the same way as a debugger command, put the expression in parentheses ( ) or explicitly use the **print** command to distinguish it from that command. For example, to look at the value of the variable **c**, enter **(c)** or **print c**. (See **print** on page 123.) If you just entered **c**, the debugger would execute the **c** command.

Expression	Effect
(c)	displays value of variable c
print c	displays value of variable c
c	executes the **c** command

- Do not press Enter in the middle of an expression. An expression is only one line with an Enter at the end.
- Comments begin with **/*** (forward slash + asterisk) and end with either a new line or ***/** (asterisk + forward slash).
- If the program has not been started, then the expression may only contain constants. After the program has started, the expression may contain variables and procedure calls.
- If the program is running, then the expression may only contain constants and, if the system allows, global variables.
- If the program has not been started and was linked against shared objects, then expressions refering to procedures and variables located within these shared objects may not be allowed.
- In C++, the "**.***" and "**->***" operators are not supported.
- In C++, casts to reference types are not supported.

- In C++, the debugger never calls destructors.
- In C++, there are restrictions on procedure calls and overloaded operator calls. See “Procedure calls” on page 58 for more information.
- In Pascal, enclose set constructors in parentheses.

If you are debugging multi-language (e.g. mixed Ada and C++) applications, use the syntax appropriate to the language of the source file that the debugger is currently displaying.

Be careful when using expressions in the debugger that may be evaluated across multiple languages (such as in button definitions). Language changes can cause confusion with the operators. For example, in C, the assignment operator is one equal sign (**=**), and the equality comparison is two equal signs (**==**). In Ada and Pascal, colon equal (**:=**) is the assignment operator, and one equal sign (**=**) is the equality comparison.

The debugger always follows the operator definitions for the current language. This can make definition of language-independent expressions difficult. In order to overcome this problem, the debugger always recognizes the “colon equal” (**:=**) as the assignment operator (in addition to the correct operator in the current language), and two equal signs (==) as the comparison operator. To ensure that expressions are language-independent, these operators should be used to implement features or capabilities (such as button definitions) that may remain in operation over several source languages.

Language keywords

When the debugger evaluates expressions, it understands the following keywords for the current source language:

Language	Keywords
C	char const double enum float int long short signed sizeof struct union unsigned void volatile (The ‘sizeof’ operator behaves the same way as in the C language.)
C++	As above, plus: class namespace
Ada	abs and boolean character false float in int integer mod not null or xor package real rem true
Fortran	.AND. .EQ. .FALSE. .GE. .GT. .LE. .LT. .NE. .NOT. .OR. .TRUE. character complex int integer logical real

Language	Keywords
Pascal	and boolean char chr false fiv in int integer mod nil not or ord real true
Jovial	A B C F P S U V abs and bit boolean character eqv false int integer mod not null or pointer real true xor
SL1	address bitoffset bitwidth byteoffset comment convert_to_int convert_to_pptr convert_to_uptr int integer maxint nil no_op offset ppoi ppointer pstructure size struct structure upoi upointer ustructure wordoffset

Viewing variables

There are several different methods for viewing the value of variables, including the following:

- Click the variable in the source pane.
- Double-click the variable in the source pane.
- Use the **print** or **examine** commands. (See **print** on page 123 and **examine** on page 104.)
- Enter the variable name in the command pane.

MULTI's expression evaluator accepts the following forms of variable notations. You can use them to unambiguously refer to a specific variable even when there are other variables with the same name.

Note: Previous releases of MULTI did not support all of the following forms.

The following list describes the different methods for specifying variable names and memory addresses. Note that an arbitrary variable called **fly** is used in these examples. Spaces before and after the '#' symbol are optional, but the pair of double quotes (" ") around a file name is required. Local variables need to be either static or within a procedure on the stack.

Viewing variables	
Expression	**Meaning**
fly	Performs a scope search for the variable **fly**, starting at the "stop point" in the current procedure and proceeding outwards. Locals, local statics, and parameters are checked, then file statics, globals, and special variables.
$fly	Searches the list of special variables for **$fly**. See "Special variables" on page 51.

Viewing variables	
Expression	**Meaning**
:fly	Searches for a global named **fly**.
::fly	Same as **:fly**.
num **#** **fly**	Uses the *num* procedure on the call stack for the scope search. Caution: if entered directly into the debugger, the debugger will jump to line *num* instead of what you intended. To avoid this, use parentheses to enclose the expression at the beginning of the command line. This is useful if you are debugging a recursive procedure and multiple instances are on the stack. You can then pick the instance and display the value of the variable for that instance. See **e** on page 101.
"foo.c" # *proc* ## *label* # **fly**	Local variable **fly** in the lexical block at label *label* in procedure *proc* in file **foo.c**.
proc ## *label* # **fly**	Local variable **fly** for block at label *label* in procedure *proc*.
stack_depth ## *label* # **fly**	Local variable **fly** in the lexical block at label *label* in procedure at stack depth *stack_depth*.
"foo.c" # *proc* # **fly**	Local variable **fly** for procedure *proc* in file **foo.c**.
proc # **fly**	Local variable **fly** for procedure *proc*.
stack_depth # **fly**	Local variable **fly** for procedure at stack depth *stack_depth*.
stack_depth ## **fly**	Local variable **fly** for procedure at stack depth *stack_depth*.
"foo.c" # fly	Static variable **fly** in file **foo.c**.
. (period)	The period is a symbol that represents the result of the latest expression.

Viewing memory addresses

Expression	**Meaning**
#*line*	Address of the code at line number *line* in the current file.
"foo.c" # *proc* # *line*	Address of line *line* for procedure *proc* in file **foo.c**.
proc # *line*	Address of line *line* for procedure *proc*.
stack_depth # *line*	Address of line *line* for procedure at stack depth *stack_depth*.
"foo.c" # *proc* ## *label*	Address of label *label* for procedure *proc* in file **foo.c**.
proc ## *label*	Address of label *label* for procedure *proc*.

Printing results of a complex statement

The results of a complex statement are not automatically printed. For example:

```
foo
```

results in printing the value of the variable foo, whereas

```
{foo}
```

does not. In this latter case, you can use the **print** command to display the value; that is

```
{print foo}
```

prints the value of `foo`.

Variable lifetime

The Green Hills compilers augment the location description (register number, stack offset, memory location, etc.) for user variables with lifetime information which indicates when the value at the given location is valid.

When you use debugger commands (e.g. **print** or **view**) or data explorers to evaluate expressions, you may see the following messages next to the value of the expression:

Message	Meaning
Uninitialized	The value displayed may represent an uninitialized value.
Out of Register Scope	The value displayed may be invalid because the location used to store the value of this variable may have been reused by the compiler to store the value of a temporary (or another) variable.
Optimized Away	The variable was optimized away by the compiler and does not have any storage. No value will be displayed in conjunction with this message.

Examples:

```
MULTI> print /d my_variable
my_variable = 0 << Uninitialized >>
```

```
MULTI> print /d my_variable
my_variable = 66952  << Out of Register Scope >>
```

```
MULTI> print /d my_variable
my_variable was optimized away
```

Special variables

The debugger maintains a list of special variables which are not a part of your program, but can be used in the debugger as if they were. For example, you could use a special variable in an expression that you evaluate in the debugger. These special variables include machine registers (such as **$r1**), debugger internal variables (such as $_DISPMODE), and user defined variables (such as **$foo**).

When the debugger is evaluating an expression and it finds a variable name (such as **result**), it first performs a scope search in the program to see if the variable exists. If the variable does not exist, then the list of special variables is searched. Variable names beginning with a dollar-sign '**$**' (such as **$result**) are assumed to be special variables.

User-defined special variables are of the same type as the last expression assigned. For example, entering:

```
$mumble=3*4
```

creates the special variable **$mumble**, assigns it the value 12, and makes its type integer. These variables are just like any other variables, except you may not take the address meaningfully.

The processor's registers are included as predefined variables. To find which register names are available on your system, you can list the registers with the **l** (lowercase 'L') command with the **r** option:

```
l r
```

See **l** on page 113.

All registers act as integers of the correct size for the register. Special care should be exercised when modifying the contents of registers while debugging high-level code, since the results of these modifications can often produce unpredictable effects.

The following special predefined variable is also included:

Message	Meaning
$result	Checks the return value of a procedure. This variable is a long-integer type and is an alias for the register on the processor architecture used for returning integers. On most systems, this is also the register to return pointers. It may be written to as well as read from. Caveat: this value is not guaranteed to be correct; it depends on the return type of the function and the processor architecture. The actual return variable may be located elsewhere. As with any register, you must be careful when you change its value.

To list all the other special variables, use the **l** (lowercase 'L') with the **s** option.

```
l s
```

See **l** on page 113.

Examining data

The following commands examine data. You can examine most items in the source pane by double clicking them. See Chapter 5, "The data explorer".

Variables

Variable names are represented exactly the same way they are named in the program. The case sensitivity of the current source language is used when evaluating expressions, but can be overridden with the "exprcasesensitivity" configuration option.

To display the value of a variable in the debugger command pane, do one of the following:

- Click the variable name in the source pane.
- Enter the variable name in the command pane using either the **print** command or parentheses if necessary. (See **print** on page 123.)

Expression formats

An expression format *exp_format* is of the form:

[*count*]*style*[*size*]

where *count* is the number of times to apply the format style *style*, and *size* indicates the number of bytes to format. Both *count* and *size* are optional. For

example, **print/4d2 fly** prints, starting at **fly**, four 2-byte numbers in decimal. *count* defaults to one, and *size* defaults to the size of the type printed.

In addition to a number, *size* can be specified as one of the following values:

b	One byte (byte-integer)
s	Two bytes (short-integer)
l (lowercase **L**)	Four bytes (long-integer)

These are appended to *style*. For example, **print/xb fly** prints a hex byte. The formats which print numbers allow an uppercase version of the character to be synonymous with appending the letter '**l**' to lowercase. For example, **fly/O** prints a long octal, which is the same as typing **print/ol fly**.

The following values are available for *style*:

Values for *style*	
Format	**Meaning**
a	Prints a string using *exp* as the address of the first byte. This prints to the first null character or 128 characters, whatever happens first. The *size* value forces printing of a given number of bytes, regardless of the occurrence of null characters. For example, if the string "hello" is at location 0x40a8, then to see the string, enter: print/a *0x40a8
b	Prints *exp* in decimal as 1 byte.
c	Prints *exp* as a character.
d	Prints *exp* in decimal.
e	Converts *exp* to the style [**-**]*d.ddd***e+***dd* where there is one digit before the radix character and the number after is equal to the precision specification given for *size*. If *size* is not present, then the system default is used.
f	Converts *exp* to the decimal notation in the style [**-**]*ddd.ddd* where the number of *d*'s after the radix character is equal to the precision specification given for *size*. If *size* is not present, then the system default is used. If *size* is explicitly zero, then no digits or radix characters are printed.
g	*exp* prints in style **d**, in style **f**, or in style **e**. The style depends on the converted value. Style **e** is used only if the exponent resulting from the conversion is less than -4 or greater than the precision given for *size*. Trailing zeroes are removed from the result. A radix character appears only if followed by a digit. This is the default for floats and doubles.
i	Using the *exp* as an address, disassembles a machine instruction.

Values for *style*	
Format	**Meaning**
I	(Uppercase ‘i’) Using the *exp* as an address, disassembles a machine instruction. If the address maps evenly to a line number in the source, it prints the source line first. This allows you to see what the compiler generated for a line of source. Using the mixed source/assembly mode in the source pane is an easier way to view the same information. However, this command may be useful if you want to save the information to a file. For example: >> tempfile print/200I myfunc >> c This sequence prints the first 200 instructions of the function, *myfunc*, and saves the output to the file tempfile. See “Record and playback commands” on page 73.
n	Uses the “normal” format based on type. If no format is specified, this is the default.
o	Prints *exp* in octal.
p	Prints the name of the procedure containing address *exp*, along with the filename and the source line or instruction that addresses maps. If size is 1 (**print/pb**), only the procedure name will be printed. If size is 2 (**print/ps**), only the filename and procedure name will be printed.
r	Prints the bounds of a ranged type or variable of a ranged type such as a C bitfield or an Ada subrange.
s	Prints a string using *exp* as a pointer to the first byte of the string. Same as **print /a ***exp*.
S	Creates a formatted dump of a structure. This is the default for items of type struct.
t	The debugger shows the “type” of variable or procedure.
u	Print*s exp* in unsigned decimal.
x	Prints *exp* in hexadecimal.

Viewing expressions

To view previous memory, enter:

^ [*exp_format*]

This causes the debugger to back up and display preceding memory location based on the size and address of last item displayed. Uses a previous format if *exp_format* is not supplied. This may not work if displaying instructions on a machine with variable length instructions.

Eval

The **eval** command evaluates expressions, but does not display the results. This is valuable when dealing with expressions which may refer to volatile memory regions. For example, with the memory cache disabled (**_CACHE = 0**),

```
print *(int *)address = value
```

will perform one write-memory and one read-memory access to the target and will print the value of the expression, whereas

```
eval *(int *)address = value
```

will perform in one write-memory and no read-memory accesses and will not print the value of the expression.

Examine

examine[/*exp_format*] *exp*

If *exp* is a procedure name, then this is equivalent to the **e** *exp* command. See **e** on page 101.

If *exp* is a number followed by a **b**, such as **3b**, then the debugger moves to that breakpoint.

In all other cases, this command is identical to the **print** command (see below).

Print

print[/*exp_format*] *exp*

Displays the value of expression *exp* exactly using *exp_format*.

Examining line numbers

Through command parsing, you can specify procedure-relative versus file-relative line numbers for the following examine commands. Note that the configuration variable **procRelativeLines** controls whether the interpretation of line numbers defaults to being procedure-relative or file-relative.

Examining line numbers	
Expression	**Meaning**
e 10	Examine line number 10 in current procedure of file.
e +10	Examine 10 lines from current position.
e 0x1234	Examine address 0x1234.
e *proc***#4**	Examine (procedure-relative) line 4 of procedure *proc*.

Examining line numbers	
Expression	**Meaning**
e “foo.c”#4	Examine (file relative) line 4 of file **foo.c**.
e “foo.c”#*proc***#4**	Examine (procedure relative) line 4 of procedure *proc* in file **foo.c**.
e (*expression*)	Examine the address which is the value of the expression.
e ($ret())	Examine the return address of the current procedure.
e *	Examine procedure list.
e 2b	Examine breakpoint #2.
e 2_	Examine call stack trace depth 2 (our caller’s caller).

C Labels:

C label	
Expression	**Meaning**
e “foo.c”#*proc***##***label*	Examine C Label *label* in procedure *proc* in file **foo.c**.
e *proc***##***label*	Examine C Label *label* in procedure *proc*.
e ##*label*	Examine C Label *label* in current procedure.

Procedure-relative mode:

Expression	**Meaning**
e *proc***#4**	Examine (procedure-relative) line 4 of procedure *proc*.

File-relative (non-procedure-relative) mode:

Expression	**Meaning**
e *proc***#4**	Examine (file-relative) line 4 of file containing procedure *proc*.

Language dependencies

In C++, when the debugger displays a class it also displays the fields in all the parents of that class, including virtual parents, if that information is available. Static fields associated with a class are also displayed.

In Pascal and Ada, the debugger examines variant tags and only displays the fields of a record that are part of the current variant. If that information is not available, all the fields are displayed.

Wildcards

A few commands specify wildcards for items such as procedure names. A question-mark '**?**' matches any single letter while an asterisk '*****' or an at-sign '**@**' matches any number of letters so that, for example, "**??***" matches all names which are at least two characters long.

There are several different formats when referring to procedures in C++:

Expression	Meaning
class::*func*(*types*)	Wildcard characters may appear in both the *class* or the *func* field, and the character '@' may appear in the *types* list to match an arbitrary number of arguments of arbitrary types.
class::**operator** @(*types*)	Matches all operators of the given *class* and *types*.
class::*func*	Matches all members whose names match *func* of all classes whose names match *class*, regardless of their arguments.
class::**operator** *op*	Matches all operators which match *op*, and are either class members or their first operand is indicated *class*.
::*func*	Matches all functions that are not class members whose names match *func*. Argument types are supplied to restrict the match.
func	Matches all functions, whether class members or not, whose names match *func*.

When using a syntax including *class*::, all base classes of *class* are also searched. Aside from that, there is no other notion of inheritance and this match is purely syntactic.

Procedure calls

From the command pane, you can call procedures in your program if the program being debugged has been compiled with the Debugging Level set to MULTI (which should result in the program being linked with **libmulti.a**).

When you set Debugging Level to **MULTI**, and then do a build, the builder automatically links in a library called **libmulti.a**. This is necessary for doing procedure calls. If you are not using the builder, then to achieve the same result do one of the following:

- Use the build-time option **-G**
- Use the build-time option **-lmulti**

When MULTI detects that **libmulti.a** was not linked in to the executable, and you try to do a procedure call, it will give an error message saying that the **-lmulti** is necessary.

Normally, every program calling a library function has a copy of that function included in its executable.

Procedures are handled from within the expression evaluator. Therefore they are accessed in expressions. For example:

```
fly = AddArgs(1, 2) * 3;
```

In C++, overloaded operators are called, provided they are not inlined, thus the following expression:

```
complex(1,2) + complex(2,3)
```

is converted into the appropriate procedure calls. Constructors are called when appropriate, again provided they are not inlined.

You can make a procedure call to any text label in the file being debugged. For example, assume the procedure **printf** is referenced in the program and thus the code for this is on the target. Enter:

```
printf("Hello, %s!\n", "world")
```

On many systems, it is necessary to print a new line before any of the information appears.

To find out what procedures are available to be called, do a list procedures command:

```
l p *
```

See (lower case **L**) **l** on page 113. To gain access to library routines for debugging purposes that are not referenced anywhere in the program code, and thus are not linked into the program image, add a dummy reference to the program and recompile.

Caveats for procedure calls

- Any breakpoints encountered during command window procedure invocation are handled as usual.
- Return values from procedures are not guaranteed to be correct if a breakpoint is encountered during a procedure call.
- If function prototype information is available, the debugger checks the function prototype and converts each argument expression to the proper type of the corresponding parameter. If it is not available, automatic promotion of arguments and detection of invalid arguments is not supported and you should ensure that function arguments specified are compatible with the function called.
- When evaluating a C expression, the debugger invokes any compiled function, with or without arguments, including both application and operating system functions. However, an OS function on the target system is only called if already linked into your program. You are responsible for linking any system calls that are called from the command line into the program.
- In C++, or any other language with inlined procedures, a procedure only inlined (so there is no stand-alone version of the procedure) may not be called.
- In C++, the expression evaluator is unable to disambiguate overloaded procedure names. In this case, a dialogue will prompt you to identify which function should be used.
- In C++, default arguments are not inserted.
- In C++, the class member **operator()**, the function call operator, and the **new** and **delete operators** are not supported.

System variables

There are a number of system defined variables. Modifying their values changes the way the debugger operates. The following list contains the currently defined system variables. To display the value of a system variable, prepend a dollar-sign to it (for example, $ANSICMODE) and enter it in the command pane.

System variables	
Name	**Meaning**
ANSICMODE	Default value is 1 if **main()** is defined with a prototype such as **main(void)** or **main(int argc, char **argv)**. Otherwise, the default is 0 (zero). If 0 (zero), expressions are evaluated as they are in K&R C. If 1, then they are evaluated as in ANSI C. Generally, this affects how **unsigned shorts**, **unsigned chars** and **unsigned** bit fields are coerced. By default in K&R, they are coerced to **unsigned int**, whereas in ANSI they are coerced to **int**, thus **((unsigned short) 3)/ -3** yields different results in ANSI and K&R. The type of **sizeof** is different, as is the interpretation of the **op=** operators in certain obscure cases.
ARRAYPRINTMAX	Specifies the maximum number of array elements the debugger prints.
CONTINUECOUNT	If this is 0 or 1, the debugger will stop at the next breakpoint. If this is 2, the debugger will stop at the second breakpoint reached by the program, and so on. Use the **c** command to set its value. For example, to set it to 3, enter: c @3
DEBUGSHARED	Enables/disables debugging of shared objects. Only relevant with targets that support shared libraries like certain native UNIX platforms or advanced embedded real-time operating systems.
DEREFPOINTER	Controls whether or not pointers are automatically dereferenced when displayed by the **print** or **examine** commands.
DISNAMELEN	Controls the length of symbols printed when associating program labels to addresses in disassembly mode.
R_SIGNAL	The signal number that caused the current program to stop.
SERVERTIMEOUT	How long (in seconds) MULTI will wait for a debug server to respond before concluding that the server has failed. MULTI will prompt the user to close the connection or keep waiting. If set to zero, MULTI will never time out waiting for a debug server.
SIGNAL	The signal number which is passed back to the target. This is zero if masked by the signal handling code.
TASKWIND	If zero, the task window (for multi-tasking targets) will be disabled.
VERIFYRESTART	Verifies attempts to restart the program by bringing up a confirmation dialog.
VERIFYHALT	Verifies halting a program before setting a breakpoint by bringing up a confirmation dialog.

System variables	
Name	**Meaning**
VIEWARRAYMAX	Maximum number of array elements shown in a data explorer window by default. More array elements can be viewed by changing the type of the array in the data explorer type field.

System special variables beginning with an underscore '**_**' are not normally listed. They represent the internal state of the debugger. To see them, use the **l** command with the **s** option:

```
l s _
```

See (lower case 'L') **l** on page 113.

System special variables	
Name	**Meaning**
_ASMCACHE	When set to one (1) **[default]**, the disassembly of program code in the debugger window is done by reading data from the executable file, not from the debugged program. This allows a faster disassembly print to the screen. Setting _ASMCACHE to zero forces the debugger to read the text to be disassembled from the debugged program, instead of the buffer or executable file. If instruction memory is modified or destroyed, and _ASMCACHE is one, then displays of disassembled instructions continue to show the original unmodified instructions in the executable file. This is confusing since the instructions actually executed are not those shown by the disassembly display. Sometimes, when peculiar behavior occurs on the target system, such as the program stops on an apparently valid instruction or it refuses to single step or continue past a valid instruction, the instruction memory on the target system has been corrupted. Try setting _ASMCACHE to zero and redisplaying the assembly code. You may find invalid instructions at the point of failure. (You may need to turn off **assem** mode and examine another part of the program, turn **assem** mode back on, then return to the point of the entry to clear out the debugger's internal disassembly cache.)
_CACHE	If non-zero, the debugger uses a cache for reading memory from the target. The cache is invalidated every time the program state changes. This speeds up remote debugging. See also **eval** on page 103.
_DATA	Used for PID (position independent data) systems where the executable is linked as if it were at one address while it runs at another. This variable is set to the offset between the location at which the data segment resides and at which it is linked. This is set on the command line with the **-data** option.
_DISPMODE	Determines whether assembly code is interlaced with source code in the source pane. See "Interlaced source view" on page 17.

System special variables	
Name	**Meaning**
_ERRHALT	When the target encounters an exception, then if _ERRHALT is false, the debugger will list the registers, execute any associated exception breakpoint commands, then resume the target process. If TRUE, only the associated commands are executed, leaving the target process halted. This variable defaults to TRUE. See **be** on page 82, **de** on page 96, **l** on page 113 (lowercase 'L') with the **e** and **r** options.
_INIT_SP	Tells the debugger the value of the stack pointer at program start up in certain remote environments where this information is not available.
_LANGUAGE	Shows which expression evaluator is in use. **0** means C, **1** means Fortran, **2** means Pascal, **3** means C++, **4** means Ada, **5** means Jovial, **6** means SL1, **7** means Assembly, and **31** means auto-select based on the type of current file.
_LINES	This shows the number of lines displayed by the **printwindow** command by default. See **printwindow** on page 124.
_NOTIFY	If this is non-zero, then you are notified when new children are forked, when your program performs an exec, and when your program is stopped. This is off by default.
_OPCODE	If non-zero, then disassembly mode displays the hexadecimal value of the instruction. This does not work for 68K.
_TEXT	Used for PIC (position independent code) systems where the executable is linked as if it were at one address, while it runs at another. This variable is set to the offset between the location at which the text segment resides and links. This is set on the command line with the **-text** option.

The following system special variables are read-only: .

Read-only system variables	
Name	**Meaning**
_BREAK	The current breakpoint number.
_FILE	The name of the current file.
_INTERLACE	Indicates whether assembly code is displayed in the source window. This is **1** (one) if there is assembly code currently displayed in the source window, otherwise it is **0** (zero).
_LINE	The current line number.
_MULTI_DIR	The name of the directory that contains the MULTI executable..
_PID	The process ID of the process, as reported by the debug server.
_PROCEDURE	The name of the current procedure.
_PROCESS	The MULTI defined program number of the current program.
_REMOTE	Set to **1** (one) if the debugger is debugging a program on a remote target. Otherwise, it is set to **0** (zero, for native debugging).
_SELECTION	A string variable representing the current selection from the source pane.
_STATE	Process state. See “Process state” table below.

Process state:

Process state			
1 = no child	2 = stopped	3 = running	4 = dying
5 = just fork’ed	6 = just exec’ed	7 = about to resume	n/a

Syntax checking

The syntax checking mechanism checks the validity of a command without actually executing it and thus without requiring target interactions and without changing the system settings.

The debugger command **sc** performs syntax checking. It can be used in two different ways.

To check the syntax of a single command, enter:

sc “*command***”**

To check the syntax of an entire script file and all nested files, enter:

sc < *script_file_name*

See **sc** on page 132.

Syntax checking is also automatically invoked whenever a breakpoint with an associated command or condition is created. The **bpsyntaxchecking** configuration option can be used to disable this automatic checking. The validity of the commands associated with the breakpoint are checked in the context that would exist if the breakpoint were hit. If a syntax error is found in the breakpoint command, a warning message is issued.

For example, entering the command:

```
sc "print abcdef"
```

will echo the error message:

```
Syntax Checking: Unknown name "abcdef".
```

and entering the command:

```
b main { print abcdef; }
```

will echo the the error messages:

```
Syntax Checking: Unknown name "abcdef".
Failed to set breakpoint owing to syntax error.
```

Chapter 4

Debugger commands

This chapter contains:

- Debugger notations
- Command groups
- Debugger commands

The MULTI debugger (“the debugger”) provides commands and features to debug your program, ranging from window related commands to program execution commands. This chapter describes all these commands in detail. The commands in this chapter are listed in alphabetical order. You can execute them from the debugger command line window. Most of these commands are also available from the debugger menus.

Debugger notations

Double quotes “ ”

This is a pair of double quotes.

Format: **“***any_string***”**

Prints the string between the double quotes. The string can contain the standard C language character escapes. For example, this can be used to print comments in breakpoint commands.

%*bp_label*

This is a breakpoint label. It starts with the percent sign (%).

See “Breakpoint label” on page 68.

@bp_count

This is a breakpoint count. It starts with the at-sign (@).

See “Breakpoint commands” on page 71.

{ *cmds* **}**

This is a pair of curly braces that contain a list of commands. See “Command list” on page 69.

Address expressions

An *address_expression* is a flexible MULTI command language construct which allows many ways of referring to a location within your program.

Examples of *address_expression*'s using the **e** command:

Displaying variables	
Expression	**Meaning**
e 10	Examine line number 10 in current procedure or file.
e +10	Examine 10 lines from current position.
e 0x1234	Examine address 0x1234.
e proc2#4	Examine (procedure-relative) line 4 of procedure **proc2**.
e “file3”#4	Examine (file-relative) line 4 of file **file3**.
e “file3”#proc2#4	Examine (procedure-relative) line 4 of procedure **proc2** in file **file3**.
e (*expression*)	Examine the address which is the value of the expression.
e ($ret( ))	Examine the return address (exit point) of the current procedure.
e 1b	Examine breakpoint with id equal to 1.
e %bp_label	Examine the location where breakpoint with label equal to **bp_label**.
e 2_	Examine stack level 2.
e “file3”#proc2##label4	Examine C Label **label4** in procedure **proc2** in file **file3**.
e proc2##label4	Examine C Label **label4** in procedure **proc2**.
e ##label4	Examine C Label **label4** in current procedure.
e *	Examine procedure list (wild card search).

Procedure-relative vs file-relative line numbers

The configuration option **procRelativeLines** controls whether or not a line number given in address expressions is to be interpreted as file-relative or procedure-relative. The default is to use procedure-relative line numbers.

Procedure relative:

Procedure-relative	
Expression	**Meaning**
e proc3#4	Examine (procedure-relative) line 4 of procedure **proc3**.
e 4	Examine source code at line number 4 in the current procedure.
e #4	Examine source code at line number 4 in the current file.

File-relative (Non-procedure relative):

File-relative	
Expression	**Meaning**
e proc3#4	Examine (file-relative) line 4 of file containing procedure **proc3**. (The line must exist within **proc3**).
e 4	Examine source code at line number 4 in the current file.
e #4	Examine source code at line number 4 in the current procedure.

See “Line numbers” on page 15.

Breakpoint label

The **b** commands for setting breakpoints (for example, **b**, **br**, **bx**) accept **%***bp_label* as an argument to specify a name for the breakpoint.

For example:

```
b %foo main#24
```

This command sets a breakpoint labeled foo on line 24 of procedure main.

The **B**, **e**, **d**, and **tog** commands can refer to breakpoint labels by using the percent qualifier (**%**).

For example:

Expression	Meaning
d %foo	Remove breakpoint labeled **foo**.
d %3	Remove breakpoint with ID = 3.
d main#4	Remove breakpoint on line 4 of **main**.
d	Remove breakpoint on current line.

Breakpoint list and ranges

A breakpoint list is a comma separated list of **%** qualified breakpoints. A breakpoint range consists of two colon separated breakpoints. The **B** and **d** commands can refer to breakpoint ranges.

For example:

Expression	Meaning
d %foo,%bar,%gamma	Remove breakpoints labeled **foo**, **bar**, and **gamma**.
d %foo:%gamma	Remove breakpoint **foo** through **gamma**.
d %1,%3:%5	Remove breakpoints with ID's 1, 3, 4, and 5.

*stacklevel_*

A call stack trace level is a number followed immediately by an underscore; it refers to the call stack level relative to the current procedure. For example, if the procedure **main()** calls **foo()** which calls **bar()** which calls **hum()** and in the debugger you are currently debugging **hum()**, then the following command:

```
e 1_
```

will change the current viewing location to **bar()**, because **bar()** is one (1) level up from the current procedure **hum()**. And this command:

```
e 2_
```

will change the current viewing location to **foo()**, because **foo()** is two (2) levels up from the current procedure **hum()**.

Command list

Many debugger commands, assertions, breakpoint commands, and so forth, are given with a list of other commands to perform at specific times. You can use C-style comments:

```
/* a C-style comment, between a forward_slash+asterisk and an asterisk+forward_slash. */
```

These lists may span several lines if they are surrounded by curly braces **{ }**. If a list is not surrounded by curly braces, then it is read to the end of the line. Curly braces can contain other pairs of curly braces as long as they are all paired correctly. These lists may be any combination of debugger commands separated by semicolons "**;**". The syntax for expressions is the same as the C language, with a few exceptions. See "Evaluating expressions" on page 46 for more

information. For example, the following command checks the value of some global variables at a breakpoint:

```
{ "Global variable "; print /d fly; if var<9 {c} else {"error"} }
```

This first prints “Global variable ” followed by “fly = ” with the value of the variable fly in decimal. If the value of variable fly is less than nine it continues to run. Otherwise, it prints “error”.

Executing a command after a continue ({c}) is not supported. For example, do *not* do this:

```
{ if (var < 10) {c;} print var }
```

default search path

MULTI maintains a search path that it uses when locating user specified filenames on the file system. The search path will always contain the current directory (**.**) as its last entry. To change this search path, do one of the following:

- Use the **source** command.
- Use the **-I** command line option to MULTI.
- From the main debugger menu, choose View > Source Path...

MULTI uses the default search path when locating the following types of files:

- source files
- script files
- object files

If a debugger command uses the default search path, it will be indicated in its description.

See the **-I** command line option to MULTI in the *Building and Editing with MULTI 2000* manual.

Printing structs

In C and FORTRAN, the debugger uses a straightforward algorithm when displaying a struct or a union. Since this may not be best for large or complex structs, the debugger allows you to define your own display method. To do this, include a routine in your source code with the same name as the struct or union, preceded by an underscore. This routine should take two arguments, which are passed by the debugger: the address of the struct and the *size* parameter. The

default value is -1, and user passable values are 0 (zero) to intMax. For example, if you have a type called **struct FLY**, then your own routine to dump its contents must be named **_FLY**.

When you attempt to print the contents of a struct, the debugger checks to see if a routine is defined for that struct. For example, if **flyFirst** is a struct of type **FLY** and you enter **print /n2 flyFirst**, then the debugger first checks for a routine named **_FLY**. If there is no such routine, then the debugger uses its own algorithm. However if you define such a routine, then the debugger calls that routine, passing a pointer to the struct and the *size* parameter.

If you define your own routine to print out a struct but prefer to use the debugger's own algorithm in a particular case, use the **N** format. This format works exactly the same as the **n** format, except it overrides your custom definition. See "Expression formats" on page 52 for more information. For example, in the case mentioned above with **FLY** and **flyFirst,** if you want to print **flyFirst** using the debugger's algorithm instead of the one defined in **_FLY**, enter **print/N flyFirst**

Command groups

Breakpoint commands

The debugger provides a number of commands for setting and removing breakpoints. A breakpoint is associated with an address. A breakpoint may also have a count *n*. The program will stop when the breakpoint is encountered for the *n*-th time. To set breakpoints in GUI mode, see Chapter 11, "Breakpoints window". The count is set by adding an **@** followed by the count number after the breakpoint command, but before the command list. For example, the following command sets a breakpoint with a count of four:

```
b @4
```

In all of the two letter breakpoint commands, if the second character is uppercase (for example **bU** instead of **bu**), then the breakpoint is temporary instead of permanent.

All of the commands containing an argument [**{***cmds***}**] may take an optional list of commands that are executed when the breakpoint is hit. See "Command list" on page 69.

Most of the breakpoint commands take an optional address expression which specifies the location of the breakpoint. If an address is not specified in a

command it takes, the current line is used. See “Address expressions” on page 66.

Breakpoint commands
B on page 80
b on page 81
bA on page 81
ba on page 82
be on page 82
bg on page 83
bI on page 83
bi on page 83
bif on page 83
bpload on page 84
bpsave on page 84
bpview on page 84
bR on page 84
br on page 84
bt on page 86
bU on page 86
bu on page 86
bX on page 87
bx on page 87

Continue commands

The **continue** commands (**C**, **c**, **cb**, **cu**, **cU**) all set a **continue count**. The **continue count** is given by a number *num* following an **@** sign. This count causes the debugger to stop at the *num*th breakpoint that stops execution. It is important to note that only breakpoints which stop program execution are counted. A conditional breakpoint whose condition is false or a breakpoint whose commands resume program execution are not counted. The continue count may be viewed with the CONTINUECOUNT system variable. See “CONTINUECOUNT” on page 60.

These commands also take an optional line number. If given, a temporary breakpoint is set at that line number. The breakpoint is removed as soon as it is reached.

History commands

The debugger has a simple history mechanism that remembers the last 60 commands. This can re-examine long expressions. The history syntax has changed from 1.8.9 MULTI. Where 1.8.9 used the '#' character for history, 2.0 MULTI uses the '!' character, similar to most UNIX shells. Some history examples:

History commands	
Expression	**Meaning**
h	Shows the existing history.
!!	Re-executes the last command.
! *number*	Re-executes command number *number*.
! *string*	Re-executes the command starting with the given string.
~	Smart repeat of last command. This increments the last command, if appropriate, before repeating it. For example, if the last command displays a memory location, this command increments the address displayed to show the next location. The following commands are repeated with a **~**:
	Any of the single step commands (s, si, S, Si).
	Any command to display source lines.
	The search commands / and ?.
	The ^ command.
	Any command to display memory.
=	This command was used in 1.8.9 MULTI but has been removed. Its behavior (smart repeating a command 10 times) can be accomplished by making an **alias** that uses **~** 10 times in a row. See **alias** on page 78.
%	Commands involving **%** were used in non-GUI mode in 1.8.9 MULTI. These are no longer supported.

Record and playback commands

The debugger contains a record and playback feature to recreate program states for bugs requiring long setups. The files created are ASCII files, and can be edited by hand later. Only debugger commands are recorded. However, if a GUI

action creates a debugger command, that command is recorded. This includes pressing any button in the debugger window or clicking in the source pane. The target window commands and output cannot be recorded.

The record and playback commands are:

Record and playback commands	
Expression	**Meaning**
> *file*	Sets command recordfile to the given file and turns on command recording.
> (**t** | **f** | **c**)	Turns command recording on (**t**), off (**f**), or closes command recording file (**c**). If no argument is given, give current status.
>> *file*	Sets screen output recordfile to the given file and turns on screen output recording (recording commands and their output).
>> (**t** | **f** | **c**)	Turns screen output recording on (**t**), off (**f**), or closes screen output recording file (**c**). If no argument exists, give current status.
< *file*	Starts command playback from the given file. The filename will be searched for using the default search path. See “default search path” on page 70.
<< *file*	Starts command playback from the given file, using the single stepping feature of playback. This command is not supported in GUI mode. The filename will be searched for using the default search path. See “default search path” on page 70.

Scripts may include other scripts, to a maximum script depth of 25.

Do not place any line beginning with **>** or **<** in the current record file. You can override this by simply beginning the line with a space. Comments are supported in command playback files, as in all debugger input, through the standard C style comments (**/* ... */**). In addition, a comment is always terminated at the end of a line. Hence a **/*** (forward slash+asterisk) with no closing match comments out the remainder of the line, but does not carry over to the next line as in standard C. You may not play back from a file currently open for recording or vice versa, as the result is undefined.

See also the **-p** and **-R** command line options to MULTI, in the *Building and Editing with MULTI 2000* manual.

Note: if you use the command “**>** *file*” when the command recordfile is already set, the old recordfile will be closed and all subsequent commands will be recorded to *file*. The command “**>>** *file*” works similarly when the screen output recordfile is already set.

Search path for scripts

The source file search path (as specified by the **source** command) will be used when searching for scripts for playback (the **<** command). See **source** on page 136.

Search commands

See **fsearch** on page 105, **bsearch** on page 86, and **dialogsearch** on page 100.

Searches wrap around the beginning and end of files and obey the current case sensitivity setting. See **chgcase** on page 91. If a *string* is not given, the previous one is used.

Command	Example	Meaning
/[*string*]	/extern	Searches forward through the current file, from the line after the current line, for *string*. In the example, the cursor will jump forward to the word **extern**. You can then find more occurrences of this word by repeatedly entering the forward slash (***/***). In GUI mode, the **fsearch** command works similarly to ***/***, but also highlights *string* in the source pane.
?[*string*]	?extern	Searches backwards for *string* from the line before the current line. In the example, the cursor will jump backward to the word “extern.” You can then find more occurrences of this word, going backwards, by repeatedly entering the question mark (**?**). In GUI mode, the **bsearch** command acts similarly to **?**, but also highlights *string* in the source pane.

Stack trace commands

The debugger provides the **calls** command and the **callsview** command for listing a stack trace. In GUI mode, **callsview** opens up a new window with the call stack trace. See **calls** on page 89, **callsview** on page 89, and Chapter 10, “Call stack window”.

Debugger commands

! command

This command is obsolete. It is the exclamation point (!).

Used with the history commands. In 1.8.9 MULTI, this command was used to invoke a shell. That use has been replaced by the **shell** command. See **shell** on page 134 and “History commands” on page 73.

+ command

This command is the plus sign (+).

Format: + [*num*]

Moves your current viewing position in the source *num* lines (one(1) if *num* is not specified) towards the end of the file.

– command

This command is the minus sign (–).

Format: **–** [*num*]

Moves your current viewing position in the source *num* lines (one(1) if *num* is not specified) towards the beginning of the file.

/ command

This command is the forward slash (***/***).

Format: ***/***[*string*]

Forward search for string (or the last string used if *string* is not specified). See “Search commands” on page 75.

? command

This command is the questsion mark (?).

Format: **?** [*string*]

Backwards search for string (or the last string used if *string* is not specified). See “Search commands” on page 75.

-> command

Format: **->** *menu_name*

Opens the menu *menu_name*. The **menu** command can be used to list all of the available menu names, as well as to define new menus. See **menu** on page 119. For example:

-> FileMenu

opens the File menu.

^ command

Format: ^ [*num*] [*format*]

Back up *num* preceding memory location (based on size of last item displayed). If *num* is not specified, one (1) is used. If *format* is specified, that format is used, otherwise the previous format is used. Note that backing up for displaying instructions doesn't work very well if the code has variable length instructions.

For example, given an integer array containing square numbers, called `squares`, then the following two commands:

```
print squares[6]; ^ 3
```
might give the following output:

```
*0x21558:  36
*0x2114c:   9      16     25
```

A

Format: **A** [**a** | **s**]

Sets the overall state of the assertions mechanism. If **a** is specified, activates it. If **s** is specified, suspends it (note that suspended assertions continue to exist, but are not "in use"). If nothing is specified, then toggles the state.

a

Format: **a {** *cmds* **}**

(Also Format: **a** *num modifier*) See **a** on page 78.

Creates a new assertion with the given command list *cmds*.

Assertions are lists of commands that are executed before every statement. This means that if there is even one active assertion, the program will be automatically single-stepped. This has a significant impact on the debugger's speed of execution.

This is an example of an assertion:

```
a { if { (foo!=$foo) { $foo=foo;print /d foo; if (foo>9) {x}  } } }
```

This command will create an assertion to report the changing value of some global, `foo`, and stop if it ever exceeds some value. It uses a debugger special variable to keep track of the old value of `foo`.

Another example:

```
a { if (foo > (bar–9)*10) {A;x 1;c} else {bar –= 10} }
```

This assertion checks the condition. If it is false, `bar` is decremented by 10. If it is true, assertions are suspended, assertion mode is exited, and the program continues at normal speed. Without the number **1** after the **x** command, the **c** command would not have been reached. See **x** on page 143.

There are some restrictions on using this command. Using local variables is not recommended since they will most likely go out of scope when a subroutine is entered and can cause unspecified results. Assertions are also not recommended when using shared libraries for similar reasons.

See also **watchpoint** on page 141.

a

Format: **a** *num modifier*

(Also Format: **a {** *cmds* **}**) See **a** on page 77.

Modifies assertion numbered *num*. *modifier* can be one of:

Modifier	Meaning
a	Activates it.
d	Deletes it.
s	Suspends it. Suspended assertions continue to exist, but are not “in use”.

To list the current assertions, use the **l** command (lowercase **L**):

```
l a
```

See also **info** on page 111 and **l** on page 113 (lowercase **L**).

about

Format: **about**

In GUI mode, opens the About dialog box with information such as the current version of MULTI. In non-GUI mode, echoes the same information to the screen.

alias

Format: **alias** [*string1* [*string2*]]

Translates *string1*, when encountered in a command, into *string2*. *string1* only translates as a unit and not a part of a larger word. Substitution is only performed once, so references to other aliases are ignored.

There are three forms of the alias command:

Modifier	Meaning
alias	Lists all aliases.
alias *string1*	Lists alias, if any, for *string*.
alias *string1 string2*	Value of *string2* becomes the alias *string1*.

For example, entering:

```
alias sh showdef
```

allows you to type **sh** instead of **showdef** when using the **showdef** command.

apply

Note: This is a software-update command and may not be available on most systems.

Format: **apply** *dot_Q_archive_name* [*source_search_path*]

Downloads the software update module and updates the debug information symbol table with the information for the update. The update module will be searched for using the default search path. When the debugger searches for source files that contributed to the building of the update module, the *source_search_path*, if specified, will be checked before the default search path. See “default search path” on page 70.

dot_Q_archive_name is an update archive file ending with a “**.Q**” suffix. *source_search_path* is a directory name. For example:

```
apply foo.Q /newsrc/foo/dir1
```

assem

Format: **assem** [**on** | **off** | **tog**]

Turns on/off the interlaced assembly display. **assem on** interlaces the appropriate assembly instructions between the lines of source code. **assem off** shows just the source code. **assem** or **assem tog** switches to the other display mode.

assert

Format: **assert** *logic_expression*

This command is a useful shortcut for the **a** command when a simple assertion is desired. (See **a** on page 77) An assertion is set that will stop the program if *logic_expression* evaluates to true, and print out that the program was stopped by *logic_expression*. For example:

assert foo >= 0

is equivalent to:

a if (foo >= 0) {“Stopped by assertion: foo >= 0\n”; halt}

attach

Format: **attach** *pid* [**pr=***num*]

This command is for attaching to a process running on an RTOS. This command can also be used for native UNIX debugging (via unixserv). Note that unless you are **root**, you can only debug your own native processes.

pid is the operating system’s process number of the process you attach to. If you specify the optional **pr=***num*, then the process is placed in the debugger’s internal process slot number *num*. If no process slot is specified, the process is placed in the first empty slot.

The **detach** command detaches from a process. See **detach** on page 99.

B

Lists all breakpoints. The output format is:

Arguments	[*address_expression* | *breakpoint_list*]
address_expression	See “Address expressions” on page 66.
breakpoint_list	See “Breakpoint list and ranges” on page 68.

ID *bp_label*	*location*:	*address*	count:	*flags*	*commands*

For example:

```
MULTI> b main#5;
MULTI> b %my_bp_name main#6 { print "Here I am"; };
MULTI> b main #7 { print "main#7" };
MULTI> tog main#5

MULTI> B
0                   main#5: 0x10204 count: 1 (inactive)
1 my_bp_name  main#6: 0x01220 count: 1 <{ print "Here I am"; }>
2                   main#8: 0x1022c count: 1 <{ print "main#7" }>

MULTI> B %my_bp_name:%2
1 my_pb_name   main#6: 0x10220 count: 1 <{ print "Here I am"; }>
2                    main#8: 0x1022c count: 1 <{ print "main#7" }>
```

See also **l** on page 113 with the **b** option.

b

Arguments	[**%***bp_label*] [**@***bp_count*] [*address_expression*] [**{***cmds***}**]
%*bp_label*	See “Breakpoint label” on page 68.
@bp_count	See “@bp_count” on page 66.
address_expression	See “Address expressions” on page 66.
{ *cmds* **}**	See “Command list” on page 69.

Sets a breakpoint at the specified location.

If a procedure name is specified, for example, **b Fly**, the breakpoint is not set at the first machine instruction of the procedure, but rather at the first machine instruction after the procedure’s stack set up code if any. This ensures that the arguments and local variables of a procedure are read correctly when you stop in that procedure. Use the **bi** command if you want to stop at the first machine instruction. See **bi** on page 83.

b is the same as the **br** command.

bA

See **ba** on page 82.

ba

Arguments	[*@bp_count*] *wild_card_proc* [**{** *cmds* **}**]
@bp_count	See “@bp_count” on page 66.
wild_card_proc	wild-card procedure names
{ *cmds* **}**	See “Command list” on page 69.

Sets a (temporary for **bA**, permanent for **ba**) breakpoint with commands. In GUI mode, this command opens a dialog box listing all procedures that match the wild card pattern *wild_card_proc*. Either pick some, all, or none of the procedures listed.

backhistory

Gives the previous command in the command pane history list or the target window history list. This command is intended to be bound to a key. (See **keybind** on page 112.) By default, the debugger binds the UpArrow key to this command.

backout

Note: This is a software-update command and may not be available on most systems.

Format: **backout** *dot_Q_archive_name*

This unloads the previously applied software update module and removes the corresponding debug information from the debug information symbol table. *dot_Q_archive_name* is an archive file ending with a “**.Q**” suffix. For example:

```
backout foo.Q
```

The filename will be searched for using the default search path. See “default search path” on page 70.

bat

This command is deprecated, use the **sb** command instead. See **sb** on page 132.

be

Format: **be** *exception_number* **{** *cmds* **}**

Adds a new breakpoint for the hardware exception *exception_number*. When this exception is encountered, the command list *cmds* will be executed like those for any other breakpoint. See also **de** on page 96, **l** on page 113 (lowercase **L**) with the **e** option, and **tog** on page 138.

bg

Arguments	[**%***bp_label*] [**@***bp_count*] [*address_expression*] [**{** *cmds* **}**]
%*bp_label*	See “Breakpoint label” on page 68.
@bp_count	See “@bp_count” on page 66.
address_expression	See “Address expressions” on page 66.
{ *cmds* **}**	See “Command list” on page 69.

Sets a global breakpoint at the specified location.

bI

This is lowercase **b** and uppercase **i**. This command has the same format and arguments as the **bi** command. See **bi** on page 83.

bi

The **bi** and **bI** commands have the same formats and arguments:

Arguments	[**%***bp_label*] [**@***bp_count*] [*address_expression*] [**{** *cmds* **}**]
%*bp_label*	See “Breakpoint label” on page 68.
@bp_count	See “@bp_count” on page 66.
address_expression	See “Address expressions” on page 66.
{ *cmds* **}**	See “Command list” on page 69.

Sets a (temporary for **bI**, permanent for **bi**) breakpoint on an instruction at the location specified.

If a location is not specified, then the address of the last item looked at with the “**/i**” or “**/I**” display mode is used. For example, **printf+0x12;bi** sets a breakpoint 12 bytes into procedure **printf**.

If a procedure name is specified, then the breakpoint is set on the first address of the procedure.

bif

Arguments	[**%***bp_label*] [**@***bp_count*] *address_expression condition*
%*bp_label*	See “Breakpoint label” on page 68.
@bp_count	See “@bp_count” on page 66.
address_expression	See “Address expressions” on page 66.
condition	expression in the current language

Set a conditional breakpoint that will stop if *condition* evaluates to true.

bpload

Format: **bpload** *filename*

Loads breakpoints from the given file. You can also use the “load” button in the Breakpoints window to achieve the same results. See also **bpsave** on page 84.

bpsave

Format: **bpsave** *filename* [*breakpoint_list*]

Saves breakpoints to the given file. The breakpoints are generally preserved in the form file#proc#line, to provide maximal portability between debug sessions. You can also use the “save” button in the Breakpoints window to achieve the same results.

For example, after a debug session, you issue the following command:

```
bpsave brkpts.lst
```

This saves the breakpoints to the file brkpts.lst.

Later, you restart the debugger, and you issue the following command:

```
bpload brkpts.lst
```

This restores the breakpoints from the previous debug session. See also **bpload** on page 84.

bpview

Arguments	none
Button equivalent	
Menu equivalent	View > Breakpoints...

Opens the breakpoints window which allows you to add, change, or delete breakpoints. This window lists all software breakpoints, hardware breakpoints, and signals. See Chapter 11, “Breakpoints window”.

bR

See **br** on page 84.

br

br is identical to the **b** command. **bR** is the same as the **b** command except that it sets a temporary breakpoint. See **b** on page 81.

break

Format: **break**

Breaks out of loops created with the debugger **while** command. See **while** on page 142.. For example:

```
while ($i<20) { $j+=$i; if ($j>50) {$j=50; break;}; $i++; }
```

In this case, if ($j>50) is true, then the while loop will terminate regardless of the value of ($i).

See also **error** on page 103.

breakpoints

This command is deprecated. See **bpview** on page 84.

browse

Format: **browse** *objects*

Allows you to browse through lists of objects. Below are the arguments that this command accepts. If no arguments are given then **proc** is assumed.

browse command's arguments	
Argument	**Meaning**
files | **filelist**	A list of all the files in the program.
procs | **procedures**	A list of all the procedures in the program.
global | **globals**	A list of all the global variables in the program.
classlist	A list of all the classes in the program.
classes	Opens a browser for classes.
calls | **scalls** [*proc*]	Opens a browser for static calls, optionally centered on the procedure *proc*, otherwise centered on the current procedure.
dcalls [*proc*]	Opens a browser for dynamic calls, optionally centered on the procedure *proc*, otherwise centered on the current procedure.
fcalls [*file*]	The Static File browser, optionally centered on the file *file*, otherwise centered on the current file.
filename	A list of all the procedures in the file *filename*.
classname	A list of the data members and functions of class *classname*.
profile	This use is deprecated. Use the **profile** command instead. See **profile** on page 125.

For details on each of the browsers, see Chapter 8, "Browse window".

bsearch

Format: **bsearch** *string*

Searches backward in the source pane for the previous occurrence of *string* and highlights it. If *string* is omitted, then the *string* used in the last **fsearch**, **bsearch**, or incremental search is used. If it reaches the beginning of the file, it beeps and then resumes searching from the end. This is only available in GUI mode. To search backward in non-GUI mode, use the **?** command. See **? command** on page 76. See also **dialogsearch** on page 100. See “Search commands” on page 75.

To search incrementally within the Target Window, press Ctrl+f for a forward search, and use Ctrl+b for a backward search.

bt

Arguments	[*@bp_count*] *proc_name* [**{***cmds***}**]
@bp_count	See “@bp_count” on page 66.
proc_name	procedure name
{*cmds***}**	See “Command list” on page 69.

Displays a message every time the specified procedure enters or exits, and continues automatically. The message says whether the procedure exits or enters. If it’s an exit, the message gives the return value.

bU

See **bu** on page 86.

bu

Arguments	[*@bp_count*] [*stacklevel*] [**{***cmds***}**]
@bp_count	See “@bp_count” on page 66.
stacklevel	call stack trace level
{*cmds***}**	See “Command list” on page 69.

Sets a (temporary for **bU**, permanent for **bu**) up-level breakpoint. The breakpoint is set immediately after the return to the level specified by *stacklevel*. (Note that you specify a numeric value for *stacklevel*, without any underscore. For example, **5**.) See “Stack trace commands” on page 75. If *stacklevel* is not specified, then the breakpoint is set one level up from the current procedure. For example, a common sequence after accidentally single stepping into a procedure is

bU Enter c

to set a temporary, up-level break and continue. The command **cU** accomplishes the same idea.

bX

See **bx** on page 87.

bx

Arguments	[**%***bp_label*] [**@***bp_count*] [*address_expression*] [**{***cmds***}**]
%*bp_label*	See “Breakpoint label” on page 68.
@*bp_count*	See “@bp_count” on page 66.
address_expression	See “Address expressions” on page 66.
{*cmds***}**	See “Command list” on page 69.

If no arguments are specified, sets a (temporary for **bX**, permanent for **bx**) breakpoint at the exit point of the current function. This is at a point which ALL returns of any kind will go through.

If a call stack trace level is specified, sets a breakpoint at the exit point of the function at specified stack level. See “Stack trace commands” on page 75. If a procedure name is specified, sets a breakpoint at the exit point of the procedure. Note that both a stack level and a procedure are address expressions. See “Address expressions” on page 66.

If *cmds* is specified, then the commands will be executed like those for any other breakpoint.

For example:

```
bx foo
bx “foo.c”#a_routine
```

The first command sets a breakpoint at the exit point of procedure foo. The second command sets a breakpoint at the exit point of the procedure a_routine, which is located in file foo.c.

build

Format: **build** [*project_name*]

Invokes the MULTI **build** command to build *project_name*. If no *project_name* is specified, the project that the current program is in is used.

builder

Format: **builder**

Opens the MULTI Builder. See the chapter on the Builder in the *Building and Editing with MULTI 2000* manual. See also **createcontrol** on page 93.

button

This command is deprecated. Use **debugbutton**. See **debugbutton** on page 97. 1.8.9 MULTI users upgrading to 2.0 should note that the syntax for the **debugbutton** command is different than it was for the **button** command.

C

Arguments	[**%***bp_count*] [*line*]
%*bp_count*	See "@bp_count" on page 66.
line	line number
Button equivalent	

This is the capital C.

Continues a suspended program after a breakpoint or an interrupt. If the program stops because of a signal, this command continues without the signal. If *line* is specified, set a temporary breakpoint on line *line*. See also **cu** on page 94.

c

Arguments	[**%***bp_count*] [*line*]
%*bp_count*	See "@bp_count" on page 66.
line	line number

Continues a suspended program after a breakpoint or an interrupt. If the program stops because of a signal, this command continues with or without the signal based on the current signal handling specified for that signal by the **zignal** command. See **zignal** on page 144. If *line* is specified, set a temporary breakpoint on line *line*. See also **cu** on page 94.

ca

Format: **ca**

Resumes a stopped actor. Actors are not supported by every target. Consult your target's *Development Guide* for details specific to your target.

cag

Format: **cag**

Resumes a stopped actor set. Actor sets are not supported by every target. Consult your Target Development Guide for details specific to your target.

calls

Arguments	[*maxdepth*] [**par** | **nopar**] [**pos** | **nopos**] [**local** | **nolocal**]
%*name*	Name for the window
maxdepth	The maximum visible depth of the call stack. If this is not specified, the previously defined value is used. The default value is 20, and the maximum value is 500.
par | **nopar**	Show parameters passed to functions.
pos | **nopos**	Show source positions of functions.
local | **nolocal**	Show local variables used in functions.
Button equivalent	

Lists the stack trace where *maxdepth* specifies the maximum depth of the stack you want to display. *maxdepth* has a default value of 20 and a maximum value of 500. Other options let you to choose whether or not to display the corresponding procedure's parameters (**par**), source position (**pos**), or local variables (**local**). The default choices are display parameter, display source position, and do not display local variables (**par pos nolocal**).

In GUI-mode, this information can be displayed in its own window with the **callsview** command. See **callsview** on page 89.

callsview

Arguments	[**%***name*] [*maxdepth*] [**par** | **nopar**] [**pos** | **nopos**] [**win** | **nowin**] [**local** | **nolocal**]
%*name*	Name for the window
maxdepth	The maximum visible depth of the call stack. If this is not specified, the previously defined value is used. The default value is 20, and the maximum value is 500.
par | **nopar**	Show parameters passed to functions.
pos | **nopos**	Show source positions of functions.
local | **nolocal**	Show local variables used in functions (only applicable with **nowin**).
win | **nowin**	Where to display the callstack. See description below.
Button equivalent	

This command lists all functions on the call stack. In GUI-mode, unless **nowin** is specified, a window displaying the current stack trace is created (the "call stack window"). At the beginning of the user's debug session, this command defaults to **par pos win**. However, subsequent calls to this command default to

the previous configuration of the call stack window. Thus, if you change a setting for the call stack window via the GUI, it will be remembered the next time you open the call stack window. See Chapter 10, “Call stack window”.

Cb

This is the capital ‘C’ with the lowercase ‘b’. See **cb** on page 90.

cb

Arguments	[**%***bp_count*] [*line*]
%*bp_count*	See “@bp_count” on page 66.
line	line number

Continues and blocks the command line input. Signals for **cb** and **Cb** are handled as they are for **c** and **C**, respectively. See **c** on page 88. Use this command in a playback file. No further processing of the playback file occurs while the program is running. The debugger does not accept new commands until it reaches a breakpoint or the program exits. You can interrupt this command with the Esc key.

This command is helpful in non-GUI mode when a program needs to accept input from routines like **gets()** and **scanf()**.

cf

Format: **cf** *address_expression*

Continues a halted process after changing the program counter to the value of the specified *address_expression*. See “Address expressions” on page 66. The following example installs a breakpoint at label `bar` of procedure `foo` whose action is to continue from the return point of the procedure, effectively skipping the rest of the function and returning immediately. The address must be within the current active procedure.

```
b foo##bar { cf ($ret()) }
```

As another example, the following command installs a breakpoint on line 12 of procedure foo whose action is to continue from line 14 of procedure foo. This will effectively skip lines 12 and 13 of procedure foo.

```
b foo#12 { cf foo#14; }
```

cfb

Format: **cfb** *address_expression*

Continues a halted process after changing the program counter to the value of the specified *address_expression.* See “Address expressions” on page 66. The address must be within the current active procedure. This command blocks any command inputs. You can interrupt this command with the Esc key.

chgcase

Format: **chgcase** [**0** | **1**]

Set the case sensitivity of text searches. **chgcase 0** sets searches to case sensitive. **chgcase 1** sets searches to case insensitive. **chgcase** toggles the current case sensitivity.

clearconfig

Format: **clearconfig**

Clears the user’s default configuration for MULTI (not only the debugger but the entire development environment). See the Configuration Chapter for more information.

comeback

Format: **comeback**

Used to bring back all of MULTI’s windows after the **goaway** command has been used. See **goaway** on page 106. **comeback** and **goaway** are only useful when MULTI is being controlled externally via something like a command script.

compare

Format: **compare** [*operation*] *src1 src2 length* [*size*]

Compares two blocks of memory beginning at *src1* and *src2* and continuing for *length* bytes. The compare operation is specified by *operation* and the size of

the value to compare is specified by *size*. *size* is the number of bytes and is either 1, 2, or 4. The default is 4 if *size* is not specified. *operation* may be:

operation	**Meaning**
<=	Less than or equal to
<	Less than
>=	Greater than or equal to
>	Greater than
==	Equal to
!=	Not equal to

If *operation* is not specified, then == (equality) is used.

If the comparison succeeds, the addresses are printed and the values are compared.

The following example compares two overlapping arrays of six 4-byte integers. The first array starts at `0x1000` and the second at 0x1008. The compare command displays only the results of comparisons that succeed:

compare >= 0x10000 0x10008 6 4

0x10000, 0x10008 : 2091264888, 2086935416

0x10004, 0x1000c : 2089100152, 945815572

0x10008, 0x10010 : 2086935416, 1279398274

0x10014, 0x1001c : 1207968893, 1099038740

completeselection

Format: **completeselection**

If the current selection is in the source pane, then this command extends the selection so that it selects an entire word. For example, if the selection starts or ends in the middle of a word, the entire word is selected. Also, it selects an entire expression in parentheses. For instance, if the selection includes an unmatched parenthesis, square bracket, or curly brace, then the selection extends to the matching one.

configoptions

This command is only supported in GUI mode. Opens the Options dialog box.

configure

Format: **configure** *config_item*[=|**:**|]*value*

Format: **configure ?**

This command changes the value of a MULTI configuration option. **configure ?** displays a list of all items you can configure. The *config_item* can be separated from *value* by either an equal sign (‘=’), a colon (‘:’), or a space (‘ ’). For example, to change MULTI’s tab size to 9, enter:

```
configure tabsize=9
```

See the chapter on Configuration in the *Building and Editing with MULTI 2000* manual.

configurefile

configurefile *file*

Configuration options are read and set out from the *file*. This file must be in a special format.

See also **saveconfigtofile** on page 132.

connect

Same as the **remote** command. See **remote** on page 128.

copy

Format: **copy** *src dest length* [*size*] [*direction*]

Copies a block of memory of *length* units of size *size* from *src* to *dest*. Thus, the total size of memory copied is (*length* x *size*). If a size is not specified, it defaults to the size of an integer. The direction of the copy is specified by *direction* with either a **1** (one) or **forw** for forward copying, or a **-1** (negative one) or **backw** for reverse copying. If no size is given, then **forw** or **backw** should be used for the direction to avoid confusion.

Reverse copying is the same as forward copying except the starting location of the copy is *src*+(*length* x *size*) and is decremented down to *src*. The destination of the copy is also started at *dest*+(*length* x *size*) and decremented down to *dest*.

createcontrol

Format: **createcontrol**

Same as the **builder** command. See **builder** on page 87.

CU

This is uppercase ‘C’ and uppercase ‘U’. See **cu** on page 94.

Cu

This is uppercase ‘C’ and lowercase ‘u’. See **cu** on page 94.

cU

This is lowercase ‘c’ and uppercase ‘U’. See **cu** on page 94.

cu

Arguments	[**%***bp_count*] [*line*]
%*bp_count*	See “@bp_count” on page 66.
line	line number

This is all lowercase ‘cu’. If either continue command **c** or **C** is immediately followed by either a **‘u’** or **‘U**’, it sets an up-level breakpoint. The **cu** and **Cu** commands set a permanent breakpoint. The **cU** and **CU** commands set a temporary breakpoint. The breakpoint is set at the address where the current procedure returns. The **cu** and **cU** commands handle signals like the **c** command (see **c** on page 88); the **Cu** and **CU** commands handle signals like the **C** command (see **C** on page 88).

For example, use this command if you have accidentally single-stepped into a procedure you meant to step over, or you want execution to proceed to another place further up the stack.

cvconfig

Format: **cvconfig** [**%***name*] *key* [*key* [...]]

Configures the call stack track window. See Chapter 10, “Call stack window”. The **%***name* option specifies the call stack view window to configure. If the name is omitted, the command configures the call stack view window that was last created or configured. The cvconfig command is mainly useful for scripts and most of the functions it provides are accessible directly from the memory view window.

key can be one of two forms: stand-alone keys that don’t have an assignment, and assignment keys with an assignment. Note that the keys and values are case insensitive.

The following are the stand-alone keys:

Stand-alone keys	
Key	**Meaning**
stop	Freezes the call stack window.
refresh	Unfreezes the call stack window.
help	Opens online help for the call stack window.
par	Shows parameters passed to the functions.
nopar	Hides parameters passed to the functions.
pos	Shows source position of functions.
nopos	Hides source position of functions.
edit	Opens an editor on the function currently selected in the window.
local	Opens a data explorer with all of the locals of the function currently selected in the window.
print	Prints the call stack window.
quit	Closes the call stack window.

The following are the assignment keys and their valid values:

Key values	
Key	**Meaning**
name=*newname*	Renames the window as *newname*.
mdepth=*depth*	Sets the maximum depth of the call stack window to *depth*.
select=*num*	Selects the stack level *num* within the call stack window.

cx

Format: **cx** object

'object' is a required argument and can be one of **t**, **a**, **g**, or **e**.

Resumes a task, actor, or actor group. **t** resumes a task, **a** resumes an actor, **g** resumes an actor group, and **e** resumes every actor. Actors are not supported by every target. Consult your target's *Development Guide* for details specific to your target.

D

Format: **D**

Deletes all breakpoints.

d *

Format: d *

Opens a dialog box listing all current breakpoints and deleting some, all, or none of them.

d

Format: **d** [*address_expression* | *breakpoint_list*]

breakpoint_list breakpoint list. See “Breakpoint list and ranges” on page 68. See “Address expressions” on page 66.

Note: The syntax for the **d** command has changed from version 1.8.9 of MULTI. It now accepts a command syntax very similar to that of the **b** command. In particular, **d** *my_number* will no longer delete the breakpoint with **id=***my_number*, but will delete the breakpoint on line *my_number*.

The d command deletes the breakpoint at *address_expression* or the list of breakpoints specified in *breakpoint_list*. If no arguments are given, the breakpoint at the current line is removed. This command removes breakpoints and all of their associated attributes; if you simply wish to temporarily disable a breakpoint, see the **tog** command. See **tog** on page 138.

dbnew

Format: **dbnew** [**c** | **n**]

Debug a different program. This command will bring up a file chooser to find the new program. If **c** is specified, the program is loaded into the current debugger, replacing what is currently being debugged. If **n** is specified (and by default), the program is loaded in a new debugger. See also the **debug** command. See **debug** on page 97.

dbprint

Format: **dbprint** [**w** | **f**]

Print the source currently being viewed in the debugger. If **f** is specified, print the entire source file. If **w** is specified (and by default), print only the source that is presently visible in the debugger window.

de

Format: **de** *exception_number*

Removes the command associated with the specified exception. The debugger associates “action clauses” with any general target exceptions.

See also **be** on page 82 and **tog** on page 138..

debug

Format: **debug** [*program_name*] [*core_file*] [**pr=***num*]

Replaces one of the debugger's existing internal program slots, given by *num*, with a new program to debug given by *program_name*. If no program slot is given, the current slot is used. If no new program is given, the current program's name is used. The program replaces what is currently in the debugger window. The program to be replaced must halt first. All monitors and monitor windows in GUI mode are deleted, and any child programs from that window which are currently debugging are also killed.

If *core_file* is specified, then the program shows where it died. Otherwise, the main routine is shown.

The filenames will be searched for using the default search path. See "default search path" on page 70.

debugbutton

Format: **debugbutton** [*num*] [*name*] [[**c=**]*command*] [[**i=**]*iconname*] [[**h=**]*helpstring*] [[**t=**]*tooltip*]

This command adds a new icon button to the debugger toolbar.

command, *iconname*, *helpstring*, and *tooltip* are all either single words, or quoted strings. Quoted string are of the form:

```
"This is a quoted string."
```

There are several forms of the command:

Form	Meaning
debugbutton	By itself, the command lists all the defined buttons. Note that the quit button and the spacer before it are never listed. Those buttons are special and can not be modified or deleted.
debugbutton 0	Deletes all buttons (except the quit button and its spacer).
debugbutton *num*	Deletes the button numbered *num*.
debugbutton *num name* [...]	Replaces the button numbered *num*
debugbutton *name*	Deletes the button named *name*
debugbutton *name* [...]	If a button named *name* exists, the button is replaced. Otherwise a new button named *name* is added to the end of the debugger toolbar.

command is the command executed when the button is pressed. You may use semicolons in the command to execute multiple commands. For example: **debugbutton printxy c="print x;print y"**.

iconname is the name of the icon associated with the button. If not specified, then the first letter of the command name will be used as the icon for the button.

iconname may either be the name of one of MULTI's built-in icons (see below for how to obtain a list of these names), or it may be the filename of a bitmap you have created yourself. If the filename is not an absolute filename, it is assumed to be relative to the directory where MULTI is installed.

If you create your own bitmap file, it must end in a **.bmp** extension and must be in the uncompressed 16-color Windows Bitmap format. Other color depths are not supported, and compressed bitmaps are not supported. An easy way to create such bitmaps is to use the Paint accessory under Microsoft Windows, and make sure you choose "16 Color Bitmap" in the "Save as type" drop-down list box of the "Save As" dialog.

The built-in icons in MULTI are 20 pixels wide by 20 pixels tall, so your buttons will look best if you also use this size for your custom bitmaps.

By default, the color light gray in your custom icons will become transparent. You can specify additional color translations for your custom icon by appending a string of the form "oldcolor1=newcolor1&oldcolor2=newcolor2" with a question mark to the end of your bitmap filename. For example:

```
debugbutton Hello c="echo hello"
i="/home/user/hello.bmp?black=fg&dkgray=shadow&white=highlight" h="Say
hello"
```

You can use the following values for oldcolor and newcolor:

Oldcolor (R,G,B values)	Possible values for newcolor
white (255,255,255)	white (default) highlight
ltgray (192,192,192)	ltgray transparent (default)
dkgray (128,128,128)	dkgray (default) shadow
black (0,0,0)	black (default) fg

To access the list of MULTI's built-in icon names along with what they look like, first open the Options dialog box by doing one of the following:

- Choose Config > Options...
- In the command pane, enter: **configoptions**

Then choose the Debugger tab, and click the button “Configure Debugger Buttons...”.

helpstring is the help text that appears at the botton of the window when the mouse moves over the button.

tooltip is the tooltip text that appears when you move your mouse over the button and wait. If you do not specify a tooltip, the name of the button will be used.

define

Format: **define** *name***(**[*arguments*]**) {***body***}**

Creates a macro inside the debugger.

name is the name of the macro followed by a set of arguments to pass to the macro.

The *body* of the macro is a command list which may contain **if** statements and **while** loops. Macros also return a value by using the **return** command in the *body.* (See **if** on page 110, **while** on page 142, **return** on page 130, and “Command list” on page 69.)

The only local variables created in the macro are the given arguments. All other variables refer to either a variable in your program or to debugger special variables. See “Special variables” on page 51. The debugger searches the list of arguments before the registers, special variables, or program variables. As a result, if an argument in a macro has the same name as a register, you cannot examine that register from within that macro.

A trace of the macro call stack is produced with the **macrotrace** command. See **macrotrace** on page 115. If an error occurs inside of a macro, a trace back is printed, and all macros will clear off the stack.

For example, if you define the following macro:

```
define fly(bat1, bat2) {return(bat1 + bat2)}
```
then enter:

```
fly(3,6)
```
The debugger displays:

```
9
```

detach

Format: **detach** [**pr=***num*]

The **detach** command quits the debugger. All breakpoints are removed before detaching. The process number, *num*, refers to MULTI's internal process slot number, not the operating system's pid number for the process. If the process is a child of the debugger, and not attached to in the first place, its parent process id is set to one. If no process slot number is given, then the current process is used. After a process is detached, the window associated with it disappears. See also **attach** on page 80.

dialog

Format: **dialog** *name*

Opens a pre-defined dialog box named *name*. Dialog boxes are loaded into MULTI with the **loaddialogfile** command. See **loaddialogfile** on page 115. A list of the currently defined dialog boxes are given by the **l** command (lowercase 'L') with the **D** option:

```
l D
```

dialogsearch

Format: **dialogsearch**

Opens a dialog box which controls text or regular expression searching in the debugger source pane. This dialog contains options for search direction and case sensitivity. See also **fsearch** on page 105 and **bsearch** on page 86. See "Search commands" on page 75.

For a list of regular expressions, see "Search dialog box" in the *Building and Editing with MULTI 2000* manual.

dialogue

This command is deprecated. It has been replaced by **dialog**.

disconnect

Format: **disconnect**

Forces MULTI to close the current debug server connect. If no connection exists, MULTI issues a warning.

dumpfile

Format: **dumpfile**

Dump the file currently being viewed in the debugger into a text file. This is useful when viewing interlaced source or pure assembly instructions. A file chooser will appear to prompt for the name of the file to be dumped to.

E

Format: **E** [*stack* | *+num* | *–num*]

Shows or changes your current viewing location in the code. It has one of the following forms:

Displaying variables	
Expression	**Meaning**
E	Enters the procedure at the top of the stack. Equivalent to **e 0**_.
E *stack*	Enters the procedure at stack number *stack*. Equivalent to the **e** *stack*_ command.
E *+num*	Increments location in stack by *num*, and enters that procedure. For example, **E +1** moves up one procedure on the stack. This is different than **E 1** which enters the procedure at stack number one.
E *–num*	Decrements location in stack by *num*, and enters that procedure. For example, **E –1** moves down one procedure on the stack. /

e

Format: **e** [*address_expression*]

If *address_expression* is specified, it changes your current viewing location in the code to that address expression. See “Address expressions” on page 66.

With no arguments, it prints your viewing location in the code. Here are several examples of this command:

Displaying variables	
Expression	**Meaning**
e	Shows current file, procedure, and line number. For example: **test.c:PrintLine:28**
e (*proc* | *file*)	Enters procedure *proc* or file *file*. If a wildcard pattern is used while in GUI mode, then a dialog box appears allowing you to choose from the matching procedures or files.
e *stack_*	Enters the procedure at call stack trace level number *stack*. The stack number must be followed by an underscore "_". Use the **calls** command to view the stack. See also "Stack trace commands" on page 75.
e *address_expression*	Enters the procedure at address specified by the address expression.
e +*offset*	Changes the viewing location to (current address + *offset*).
e –*offset*	Changes the viewing location to (current address – *offset*).
e *num*b	Enters the procedure containing breakpoint number *num*. Use the **B** command to view breakpoint numbers. (See **B** on page 80.) For example, **e 1b** enters the procedure containing breakpoint number one.

echo

Format: **echo** *text*

Echoes *text* to the command pane, taking out quotes if there are any. For example, both of the following give the same result:

```
echo foo bar
echo "foo bar"
```

This command is preferable to **print** in cases where you don't want the text to be evaluated. (See also **print** on page 123.)

edit

Format: **edit** [*address_expression*]

Opens an Editor on the file and line of *address_expression*. If no *address_expression* is given, it uses the current viewing location in the code. An example: **edit bar** opens the Editor on the file containing the function bar, with

the cursor positioned at the beginning of the function bar. See “Address expressions” on page 66.

editbutton

Format: **editbutton** [*num*] [*name*] [[**c=**]*command*] [[**i=**]*iconname*] [[**h=**]*helpstring*] [[**t=**]*tooltip*]

This command adds a new icon button to the Editor toolbar. The syntax of this command is identical to the **debugbutton** command. (See **debugbutton** on page 97.)

editfile

This command is deprecated. Use the **edit** command. See **edit** on page 102.

editview

Arguments	**editview** [*expr* | *proc* | *file*]
expr	an expression
proc	procedure name
file	file name

Either opens the MULTI editor or a data explorer window, depending on the arguments passed to it. You can bind this command to a mouse to create a “smart” mouse click that either views or edits anything you click.

error

This command is deprecated. Instead, use the **break** command to abort a while loop, and the **return** command to abort a macro. See **break** on page 85 and **return** on page 130.

eval

Format: **eval** *exp*

exp is an expression in the current language.

This is similar to print, but does not echo the results. This should be used instead of print when performing I/O accesses since printing the result of *exp* may cause an extra read of the I/O address. For example,

```
eval *(int * ) 0xffffa0c0 = 0x123
```

If you are concerned about accessing I/O memory, see also the system variable **_CACHE** on page 61.

examine

Format: **examine** [/*format*] *exp*

If *exp* is a procedure name then this is equivalent to **e exp** which will display the named procedure in the source pane. If *exp* consists of <number>**b** then it will cause MULTI to display that breakpoint. Otherwise it is equivalent to **print exp** which will evaluate the expression and print the result. See **print** on page 123. .

f

Format: **f "***printf_style_format***"**

Sets address printing format using **printf** style formatting specification. See any C reference for more information on **printf**. If no argument exists, this defaults to "**%#lx**", which prints the address in long hex. This is for viewing memory addresses in decimal, octal, or some other format.

For example, entering:

```
print a
```

may by default give:

```
*0x21098:  5
```

You may get the address in octal instead by entering the following:

```
f "0%o"; print a
```

which will show:

```
*0410230:  5
```

All future addresses will also be in this format until **f** is used again.

filedialog

Format: **filedialog** [*buttonlabel windowtitle*]

This command opens a File Chooser and returns the name of the file that is selected from the chooser. By default the button is labeled "Select" and the title

of the window is “Choose File”, but these may be changed with the *buttonlabel* and *windowtitle* parameters.

filedialogue

This command is deprecated. See **filedialog** on page 104.

fill

Format: **fill** *dest length* [*value*] [*size*]

Performs raw memory initialization. Fills the block of memory beginning at *dest* and *length* units of size *size* long with *value*, or zero if *value* is not specified. This, the total size of memory filled is (*length* x *size*). *size* is the number of bytes to place *value* in, and is either 1, 2, or 4. If *size* is not specified, the default is 4. If *value* is larger than *size*, then *value* is truncated.

You can interrupt this command with the Esc key.

find

Format: **find** *src length value* [*size*] [*mask*]

Searches memory starting at *src* for *value* of size *size*. *size* may be 1, 2, or 4 bytes, and defaults to the size of an integer. The search stops when *length* values of the given size are checked. If *mask* is specified, then it is logically **AND**’ed with each memory location before comparing with *value*. Every match found is listed on a separate line with the address of the match.

findleaks

Format: **findleaks**

Find memory leaks within a program that has been compiled with the proper run-time error checking options. See “Finding memory leaks” on page 171 .

forwardhistory

Gives the next command in the command pane history list. This command is intended to be bound to a key (see **keybind** on page 112). By default, MULTI bounds the DownArrow key to this command.

fsearch

Format: **fsearch** *string*

Searches forward in the source pane for the next occurrence of *string*, and highlights it. If *string* is omitted, then the *string* used in the last **fsearch**,

bsearch, or incremental search is used. If it reaches the end of the file, it beeps and then resumes searching from the beginning. This command is only available in GUI mode. To search forward in non-GUI mode, use the ***/*** command (forward slash). See “Search commands” on page 75.. See also **dialogsearch** on page 100.

To search incrementally within the Target Window, press Ctrl+f for a forward search, and use Ctrl+b for a backward search.

g

Format: **g** *line*

This changes the program counter so *line* becomes the next execution point. You cannot set the next execution point to a line outside the current procedure with this command.

getargs

Format: **getargs**

Shows the current arguments that will be passed the next time the program runs. Both **getargs** and **setargs** are only applicable to the debugging of programs which take arguments in the traditional `main(argc,argv)` sense. The **setargs** command sets the program arguments to be used. The **getargs** command prints the current program arguments. The following example shows the use of **setargs**, **getargs**, and **r**:

```
MULTI> setargs abc def ghi
MULTI> getargs
abc def ghi
MULTI> r
running 'a.out abc def ghi'
MULTI>
```

See also **setargs** on page 134.

goaway

Format: **goaway**

Used to hide all of MULTI’s windows, including the debugger. These windows can be brought back with the **comeback** command. See **comeback** on page 91. **goaway** and **comeback** are only useful when MULTI is being controlled externally via something like a command script.

grep

Format: **grep** [[**-i**] [**-w**] [**-F**] *text*]

This command searches all files edited and all files in MULTI's file list for the string entered. The output from this command is put in a temporary window. Double-clicking any of the lines in this window opens an Editor.

If *text* is specified, then that expression is used. Otherwise, the debugger will prompt you for an expression. If the **-i** option is specified, **grep** will search in a case-insensitive way. If the **-w** option is specified, **grep** will only find matches which match as a whole word. If the **-F** option is specified, **grep** will treat *text* as a fixed string to search for, rather than a regular expression.

This command works by running the GNU **grep** utility. For your convenience, a copy of GNU grep is installed along with MULTI. However, GNU grep is not part of MULTI and is not distributed under the same license as MULTI. For more information about the GNU General Public License which GNU grep is distributed under, refer to the file gnugrep.README, which is located in the directory where MULTI is installed.

H

Format: **H**

When the process is halted, this command will give the signal which caused the halt.

h

Format: **h** [*depth*]

Shows the previous *depth* commands in the debugger's command history. *depth* must be between 0 and 61, and defaults to 10 if no value is given. See "History commands" on page 73.

halt

Arguments	[**pr=***num*] [**{** *cmds* **}**]
num	program number
{ *cmds* **}**	See "Command list" on page 69.
Button equivalent	

Halts the program number *num*, or the current program if *num* is not specified. *num* corresponds to MULTI's internal program number, and not the operating system's process ID number. The program halts without sending an interrupt,

allowing you to cleanly continue the program later. If *cmds* are specified, they will be executed as soon as the process halts.

halta

Format: **halta**

Halt an actor. Actors are not supported by every target. Consult your target's *Development Guide* for details specific to your target.

haltag

Format: **haltag**

Halt an actor set. Actor sets are not supported by every target. Consult your target's *Development Guide* for details specific to your target.

haltx

Format: **haltx** object

'object' is a required argument and can be one of **t**, **a**, **g**, or **e**.

Halts a task, actor, or actor group. **t** stops a task, **a** stops an actor, **g** stops an actor group, and **e** stops every actor. Actors are not supported by every target. Consult your target's *Development Guide* for details specific to your target.

hardbrk

Arguments	[**read**] [**write**] [**execute**] [**mask=num**] *exp*[**:***num*] [**{***cmds***}**] [**delete=***num*]
read	attribute
write	attribute
execute	attribute
mask	bit mask
exp	memory address, variable, or pointer name
{*cmds***}**	See "Command list" on page 69.
delete	attribute

Sets, clears, displays, enables, and disables hardware breakpoints, based on a single address with attributes read, write and execute. These breakpoints are implemented through direct hardware support and are only on some targets. MULTI removes all hardware breakpoints when detaching from process.

One advantage to a hardware breakpoint is that you can set it on a specific memory location. This causes a break when accessing that location, regardless of what instruction the program is on.

When a hardware breakpoint is reached, a message displays the breakpoint number and whether the break occurred on a read, write, or execute. For example:

```
Stopped by hardware break 1 on execute
```

Typing **hardbrk** by itself lists all currently set hardware breakpoints.

Any combination of **read**, **write**, and **execute** can be specified. **read** causes the break to occur when reading from the given address. **write** causes the break to occur when writing to the given address. **execute** causes the break to occur if the instruction stored at the given address is executed. Often the break only occurs after the read or write. **read** and **write** are used by default.

If **mask=***num* is specified, then the bitwise complement of *num* is bitwise **AND**'ed with all addresses involved. This can give a range of addresses. For example, including **mask=0xf** ignores the lower four bits of the given address. In effect, this gives a range of 15 memory locations to use. **mask** is set to zero by default.

exp may be a memory address, variable, or pointer name. If **:***num* is specified after *exp*, then *num* bytes after the address is used. The default size is one byte for memory locations, and the size of the object for variables.

If *cmds* is specified, the given commands will be executed each time the hardware breakpoint is hit.

If **delete=***num* is specified, then hardware breakpoint *num* is deleted. Use **hardbrk** to get breakpoint numbers.

An error message appears if the target system cannot support the requested breakpoint.

For example:

```
hardbrk read val
```
Stops on any read from variable **val**.

```
hardbrk mask=0xf 0x10000
```
Stops on any read or write to locations **0x10000** to **0x1000f**.

```
hardbrk write *string:9
```
Stops on any write to the first nine bytes pointed by **string**.

```
hardbrk delete=2
```
Deletes breakpoint number two.

```
hardbrk val {"stopped on val ";c}
```

Prints **stopped on val** in the command window any time the variable **val** is accessed.

```
hardbrk execute 0x100ff
```
Stops anytime when the instruction at address **0x100ff** is executed.

help

Format: **help** [*keyword*]

Opens the help system to look for help on *keyword*. If no keyword is given, brings up general help on the debugger. See Online Help System .

i

This command is obsolete. It has been replaced by the **info** command. See **info** on page 111.

if

Format	**if** *exp* **{***cmds***}** [**else {***cmds***}**]
exp	an expression in the current language
{ *cmds* **}**	See “Command list” on page 69.

This is a conditional command execution. If the expression *exp* evaluates to a non-zero value, the first group of commands is executed, else the second group, if present. This command can be nested.

indexnext

Arguments	none
Button equivalent	

Changes the current viewing location in the code to the next item in Debugger’s history list. See “History navigation buttons” on page 19.

indexprev

Arguments	none
Button equivalent	

Changes the current viewing location in the code to the previous item in the debugger’s history list. See “History navigation buttons” on page 19.

infiniteview

Format: **infiniteview** *lvalue*

Creates a view window that displays memory as an array of the basic type of *lvalue* with every line formatted to that type. The window can be scrolled in either direction until you run out of memory. Some examples:

infiniteview $sp Displays the stack starting where at the stack pointer.

infiniteview $pc Displays the text segment starting at the program counter.

This effect can also be achieved by selecting Format > Infinite from the format menu in a normal view window.

info

Format: **info**

Prints out the following information about the state of MULTI:

- Debugging status
- Core file status
- Child program status
- Assertion status
- Output recording status
- Command recording status
- Case sensitivity status

inspect

Format: **inspect** [*string*]

This command is generally bound to a mouse click. See **mouse** on page 119 This opens a context sensitive menu on *string*, equivalent to the default behavior of right-clicking *string*.

iobuffer

Format: **iobuffer** *state*

Disables or enables buffering for the remote in/out window. Buffering is enabled by default. If buffering is enabled (**on**), then input to the remote in/out window is not sent to the target until a newline is encountered in the input stream. If buffering is disabled (**off**), then every character is sent to the target as

soon as it is typed. Disabling the buffering in MULTI may cause problems on some remote targets if they expect input to buffer.

isearch

Format: **isearch** [+|–]**wid=***num*

This command starts an incremental search in the window specified by *num*, the window id number. If an incremental search is already active in that window, then the current search string is searched again. Putting a plus sign (+) in the command searchs forward. This is the default. Putting a minus sign (–) in the command searchs backward.

This command should not be used from the command window. It should be bound to a key with the **keybind** command, or to a mouse press with the **mouse** command. See **keybind** on page 112 and **mouse** on page 119.

isearchadd

Format: **isearchadd wid=***num text*

Adds *text* (no quotes) to the search string and continues an incremental search in the window pointed by *num*. The window must already be in an incremental search for this command to work.

This command should not be used from the command window. It should be bound to a key with the **keybind** command, or to a mouse press with the **mouse** command. See **keybind** on page 112 and **mouse** on page 119.

k

Format: **k**

Kills the current program. The process must be halted in order to be killed. To access from the menus, choose Debug > Kill Process.

keybind

Format: **keybind** [*location*]

Format: **keybind** *key*[|*modifiers*][**@***location*][**=***command*]

This command is used to bind a key to a command. This command is covered in great detail in the Configuration Command List chapter. See also **backhistory** on page 82, **forwardhistory** on page 105, **isearch** on page 112, and **isearchadd** on page 112.

L

This command was in 1.8.9 MULTI but has been removed. **E** will give the same functionality. See **E** on page 101.

l

This command is the lowercase ‘L’.

Format: **l** [*option*] [*string*]

This command lists most items. If no argument is given, then all locals and parameters of the current procedure are listed. The following is a description of the allowed *option* values for this command:

Values for *option*	
Value	**Meaning**
@	Lists the addresses of local variables. If a *string* is specified, it is interpreted as a procedure name and variables local to that procedure are listed. The procedure must be on the stack. Equivalent to View > List > Local Addresses.
a	Lists assertions. Equivalent to View > List > Assertions.
b	Lists breakpoints. Identical to the **B** command. Equivalent to View > List > Breakpoints. See also **B** on page 80.
d	Lists the directories that will be searched for source. Identical to the **source** command. Equivalent to View > List > Source Paths. See also **source** on page 136.
D	Lists all dialog boxes. Equivalent to View > List > Dialog Boxes.
e	Lists the exceptions with breakpoints.
f	Lists files. This command takes an optional prefix as the *string* argument. If given, all files with *string* are displayed. Otherwise, all files are printed. Equivalent to View > List > Files.
g	Lists globals. This command takes an optional prefix as the *string* argument. If given, all global variables starting with *string* are displayed. Otherwise, all global variables are displayed. Equivalent to View > List > Globals.
m	Lists procedures with their mangled names. This is identical to **l p**, except that in C++ programs it also lists the mangled names of the procedures. Equivalent to View > List > Mangled Procedures.
M	Lists menus defined with the **menu** command. Equivalent to Config > Functionality Settings... > General tab > Configure Menus. See also **menu** on page 119.
p	Lists procedures and their addresses. An asterisk (*) indicates that the procedure has no debug information. This command takes wild cards. If a filename is given as the *string* argument, then all procedures located in that filename are listed. Equivalent to View > List > Procedures.

Values for *option*	
Value	**Meaning**
P	Lists processes. Equivalent to View > List > Processes.
r	Lists registers. This takes an optional prefix as the *string* argument. If given, all registers starting with *string* are listed. Otherwise, all registers are listed. Equivalent to View > List > Registers. See “View menu” on page 26 and “regview” on page 128.
R	Identical to **l r** if *string* is specified. Otherwise, it lists all register synonyms. Equivalent to View > List > Register Synonyms.
s	Lists special variables. This command takes an optional prefix as the *string* argument. If given, all special variables starting with *string* are displayed. Otherwise, all special variables are displayed. Equivalent to View > List > MULTI Variables.
S	Lists statics. This command takes either a filename or a prefix as the *string* argument. If a filename is given, all static variables in that file are displayed. If a prefix is given, all static variables in the current file starting with *string* are displayed. Otherwise, all static variables in the program are displayed. Equivalent to View > List > Statics.
T	Lists tasks. Equivalent to View > Tasks.
t	Lists typedefs. This command takes an optional prefix as the *string* argument. If given, all typedefs starting with *string* are displayed. Otherwise, all typedefs are displayed.
z	Lists signals. Equivalent to View > List > Signals.
?	Lists help on this command.
proc	Lists all locals and parameters of the procedure *proc*. If the procedure name *proc* starts with an **@,** then the address of all locals and parameters are printed. *proc* must be on the stack. Equivalent to View > List > Locals and View > List > Variables in Procedure.

load

Format: **load** *filename*

Loads the current program into the target system’s memory. For some targets this is a rather long process, depending on the size of the program. An implicit **load** is performed when executing a program for the first time. The program is not started automatically. By default, **.bss** is zeroed but this may depend on the debug server.

If *filename* (for example **load a.out**) is specified, the given file will be loaded to the server instead of the image being debugged. Use this option with extreme caution. MULTI will assume that the loaded file contains an adequate subset of the current image, and will attempt to execute and debug it as such, without attempting to download the current image as well.

The filename will be searched for using the default search path. See “default search path” on page 70.

You can interrupt this command with the Esc key.

loadconfigfromfile

Format: **loadconfigfromfile**

Brings up a file dialog allowing the user to select a MULTI configuration file to load into MULTI. See the Configuration Chapter for more information.

loaddialogfile

Format: **loaddialogfile** *file*

Loads dialog box descriptions from file *file*. See also **dialog** on page 100.

loaddialoguefile

This command is deprecated. See **loaddialogfile** on page 115.

loadsym

Format: **loadsym** *filename* [*text_offset* [*data_offset*]]

Loads the debug symbols from the file specified by *filename* and merges them into the symbol table. If the optional text and data offset values are supplied (*text_offset* and *data_offset*, respectively) then the text and data addresses are offset by the given values.

You can use this command in remote environments where new code, typically position independent code, is loaded to the target at runtime. This command does not load executable code from the given file to the target. For example:

```
loadsym a.out 0x20000
```

You can load additional symbol information for a module while debugging.

The filename will be searched for using the default search path. See “default search path” on page 70.

M

This command is obsolete; it has been replaced by the **map** command. See **map** on page 116.

macrotrace

Format: **macrotrace**

Prints the stack of all presently executing macro commands. For example, with the following macros:

```
define a1() {return a2();}
define a2() {return a3();}
define a3() {macrotrace; return 42;}
```

then the following would be output if you enter `a1()`:

```
 0 a3()
 1 a2()
 2 a1()
42
```

See also **define** on page 99.

make

Format: **make** [*string*]

Executes the system command **make** and passes to it the arguments you supply in *string*. If **make** succeeds, it kills the current process, removes all state information, reloads the program you are currently debugging, and allows you to continue debugging.

If you are using the MULTI Editor, then any changes you make are saved before **make** starts. The output of **make** then appears in a special window. You can examine erroneous lines by clicking the appropriate error messages that appear in this window.

map

Format: **map**

Prints the section address map for the current program.

mark

This command is obsolete. It was in 1.8.9 MULTI, but has been removed. It has been replaced by MULTI's automarking capability. See "History navigation buttons" on page 19.

memdump

Arguments	[**srec	raw**] *filename start length*
srec	Motorola S-Record
raw	Raw binary data
filename	The file to load into memory
start	The starting address in memory to load.
length	How many bytes of data to load into memory, starting at *start*.

Copies a section of memory to a file named *name*. *start* specifies the starting address in memory to copy, and *length* specifies the number of bytes to copy. If **srec** is specified, then the file is stored in Motorola S-Record format. If **srec** is not specified, then the file is stored in a MULTI specific binary format.

Note: This MULTI specific binary format is platform specific. You should only use **memload** to load the file from the same platform from which you saved the file. See **memload** on page 117.

You can interrupt this command with the Esc key.

memload

Arguments	[**srec	raw**] [**-w***size*] *filename* [*start* [*length*]]
srec	Motorola S-Record
raw	Raw binary data
filename	The file to load into memory
start	The starting address in memory to load.
length	How many bytes of data to load into memory, starting at *start*.
size	The size in bytes of the individual memory writes. The value must be 1, 2, or 4. The default is 1 byte. *length* must be a multiple of *size*.

If *start* and *length* are omitted, then the values specified when the file was created is used. If **srec** is specified, then *start* and *length* are ignored, and the file is read as a Motorola S-Records file. If **srec** is not specified, then the file is read as a MULTI specific binary file created by memdump. See **memdump** on page 117.

If **raw** is specified, load a binary file, starting at the first byte of the file, and continuing for *length* bytes. *start* must be specified. *length* defaults to the length of the file, but may be overridden.

You can interrupt this command with the Esc key.

memread

Format: **memread** *size addr*

Performs a sized memory read from the target and prints the result. This command is intended to be used to perform low-level writes to regions of memory or memory-mapped I/O registers. This command does not make use of MULTI's memory cache and the read is performed immediately. See **_CACHE** on page 61.

size must be 1, 2, or 4. The units are bytes. *addr* must be aligned correctly to the nearest *size* bytes, and may consist of any expression that the debugger can evaluate.

memview

Format: **memview** [**%***name*] [**@***count*] *address*

The **memview** command opens a memory view window for interactively displaying and modifying memory contents. This window starts at memory address *address* (which can be specified by any expression in the current language). By default, the window will open sized to show 64 bytes on the screen; this can be changed by specifying *count*. Also, a title can be given to the window by specifying *name*. For example:

```
memview %Arguments @128 argv[0]
```

opens a memory view window titled "Arguments", sized to show 128 bytes on the screen, beginning with the address of argv[0]. See Chapter 9, "Memory view window".

memwrite

Format: **memwrite** *size addr value*

Performs a sized memory write to the target. This command is intended to be used to perform low-level writes to regions of memory or memory-mapped I/O registers. This command does not make use of MULTI's memory cache and the write is performed immediately. See **_CACHE** on page 61.

size must be 1, 2, or 4. The units are bytes. *addr* must be aligned correctly to the nearest *size* bytes. Both *addr* and *value* may consist of any expression that MULTI can evaluate. If *value* is larger than can fit in *size* bytes, it will be truncated to fit.

menu

Format: **menu** [*name*] [**{{***label* [**'***rlabel***'**] *cmd***}}**]

This command defines a menu to attach to a menu bar, MULTI button, mouse button, or key from the keyboard. This command is covered in great detail in the Configuration Command List chapter. See also **l** on page 113 (lowercase 'L') with the **M** option.

monitor

Format: **monitor** [**0** | **{***cmds***}** | [*num* [**{***cmds***}**]]]

Saves the command list *cmds* to send to the debugger every time the program stops. An unlimited number of monitors can be active at any time. However, you should be careful when using this command, as the output is quite tedious if your code stops frequently.

This command has five forms:

Form	Meaning
monitor	Lists all of the monitors in order.
monitor *num*	Deletes monitor number *num*. This does not renumber the current monitors so that if you have four monitors and delete number **3**, then the remaining three are numbered **1**, **2**, **4**, creating an "empty slot" where **3** was formerly located.
monitor **{***cmds***}**	Inserts a monitor with the given command list in the first available empty slot.
monitor *num* **{***cmds***}**	Puts a monitor with the given command list in the *num* slot. It replaces any existing monitor in that position.
monitor 0	Deletes all monitors.

mouse

Format: **mouse** [*location*]

Format: **mouse** *button_num*[**AtOnce**][***click** *click_num*] [|*modifiers*] [**@***location*] [**=***command*]

Defines the function of the mouse buttons. See also **inspect** on page 111, and **isearch** on page 112, and **isearchadd** on page 112. This command is covered in great detail in the Configuration Command List chapter.

mprintf

Format: **mprintf(***format_string***, ...)**

This command takes the same syntax as the C library printf() function, except the **%n** format is not supported.

For example, given the following target code:

```
char * my_string = "hello world";
int my_int = 10;
```

And with the following command:

```
mprintf("my_string=\"%s\" and (2*my_int+1)=%d",
my_string, 2*my_int+1);
```

The debugger will output:

```
my_string="hello world" and (2*my_int+1)=21
```

mvc

Format: **mvc** *args*

The *args* are passed to MULTI's version control for the file that is presently displayed in the debugger. For example:

```
mvc co
```

would check the presently displayed file out of MULTI's version control. See the chapter on MULTI version control for more details on mvc arguments.

mvconfig

Format: **mvconfig** [**%***name*] *key*[=*value*] [*key*[=*value*]] [...]

Configures a memory view window. (This is **mv**-config, not **mvc**-config.) See Chapter 9, "Memory view window".The **%***name* option specifies the memory view window to configure. If the name is omitted, the command configures the memory view window that was last created or configured. You may also configure the defaults for the next memory view window to be created by using **%default** as the name. The mvconfig command is mainly useful for scripts and most of the functions it provides are accessible directly from the memory view window.

The following key-value pairs are valid (note that the keys and values are case insensitive):

Key	Meaning
bpr=*n*	Changes the number of bytes per row displayed. Valid values for *n* are **+**, **–**, **4**, **8**, **16**, **32**, **64**, or **128**. The plus sign (**+**) increases the bytes per row to the next higher setting. The minus sign (**–**) decreases the bytes per row to the next lower setting. A number sets the bytes per row to that number.
name=*newname*	Renames the window with the given new name.
endian=*e*	Sets the endianness mode. Valid values for *e* include **big** and **lit** or **little**.
type=*typestr*	Displays memory as the type specified by the type strings. This is equivalent to setting the type pulldown to the specified position. See the table below of valid values for *typestr*.
ascii=*state*	Sets the state of the ASCII column. Valid values for *state* are **on** (show the ASCII column), **off** (hide the ASCII column), and **toggle** (switch the current state of the ASCII column to the opposite setting).
frozen=*state*	Sets the frozen state of the window. Valid values for *state* are **on** (freeze the window), **off** (unfreeze the window), and **toggle** (switch the current freeze state of the window to the opposite setting).
quit	Closes the memory view window.

Valid values for **type=***typestr* include:

Valid values for *typestr*	
Value	**Meaning**
f	float
fd	double
un	unsigned decimal integer of size n bytes. n=1,2,4,8
sn	signed decimal integer of size n bytes. n=1,2,4,8
hn	hexadecimal number of size n bytes. n=1,2,4,8
bn	binary number of size n bytes. n=1,2,4,8

n

Format: **n**

Same as the **S** command. See **S** on page 131. .

new

Format: **new** *program_name* [**pr=***num*] [*core_file*]

Tells the debugger to open a new window to start debugging the program, *program_name*, and put in MULTI's internal program slot numbered *num*. If *program_name* is not specified, the current program name is used. If *num* is not specified, MULTI puts the program in the first empty slot.A core file for the new program may also be specified.

ni

Format: **ni**

Same as the **Si** command. See **Si** on page 135.

nl

This is lowercase 'N' and lowercase 'L'.

Format: **nl**

Same as the **Sl** command. See **Sl** on page 135.

note

Format: **note**

Opens a "notes" file in a normal MULTI Editor window, to which you may add you own free-form notes. This command takes no arguments. The file opened is ".Notes", located in the user's home directory (~). If the notes file already exists, it will be opened so you may edit or add to your previous notes. If the file does not yet exist, it will be created once you save the new notes you have made.

P

This is the uppercase 'P'.

Format: **P** [**pr=***num*] [*subcommand*]

Used exclusively during multi-process debugging. If this command is given by itself, it lists all process slots in use. This command sends the commands given for *subcommand* to the process in MULTI's internal process slot number *num*. For example **P pr=1 b** toggles the state of the **b** flag in process number one.

The following is a list of *subcommands*:

Sub-command	Meaning
b	Toggles breakpoint inheritance after forking. If true, children of the current process inherit all breakpoints set at the time of the fork.
c	Toggles flag causing children to be debugged. If true, children of the current process are added to the list of processes under control of MULTI.
e	Toggles flag causing children to stop upon execution of the **exec** system call. This acts as if a breakpoint were encountered at the first instruction of routine main in the exec'd program.
f	Toggles flag causing children to stop upon execution of the **fork** system call. This acts as if a breakpoint were encountered immediately following the fork. This normally means you are in the middle of the library routine fork.
k	Toggles flag causing tasks to be debugged.
t	Toggles flag causing MULTI to stop upon task-creation.

After a **fork** or **exec** of a process, MULTI prints a message indicating that this has happened, provided that the system variable _NOTIFY is set appropriately. See **_NOTIFY** on page 62.

The following *subcommands* are deprecated in this version, and were left in for compatibility purposes. The commands that supersede them are given.

Deprecated sub-commands	
Sub-command	**Meaning**
s *num*	Sends signal *num* to the current process. Equivalent to the **signal** command. See **signal** on page 135.

p

This is lowercase 'P'. This is the same as the **print** command. See **print** on page 123.

Note for users of 1.8.9 MULTI: The 'p' command in 1.8.9 version would print the line you were on in non-GUI mode. The current 'p' command does not do that any more. To get the functionality of the 1.8.9 version of the 'p' command, use the **printline** command. See **printline** on page 124.

pop

This command is obsolete. It was in 1.8.9. MULTI, but has been replaced by the **indexprev** command. See **indexprev** on page 110.

print

Format: **print**[/*format*] *exp*

Displays the value of *exp* exactly as the current language does with *format*. *exp* can be any expression in the current language. See “Expression formats” on page 52. See also **echo** on page 102 and **examine** on page 104.

printline

Format: **printline** [*count* [*line*]]

Prints *count* lines, starting at the line number *line*. If *count* is not specified, one line is printed. If *line* is not specified, then the current line is the starting point. The current line is updated to the last line printed after this command is executed, which will change the source display if in GUI mode.

Simply typing a line number also proints out that line in non-GUI mode.

printsearch

Format: **printsearch**

Prints out the search string or indicates that there is no search string. If there is a search string, it is printed within square brackets, so beginning and ending whitespace can be seen. For example:

```
printsearch
```
may result with:

```
[ foo  ]
```
meaning that the search string is the word foo preceeded by one space and followed by two spaces. See **isearch** on page 112, **fsearch** on page 105, **bsearch** on page 86, **? command** on page 76, and **/ command** on page 76.

printwindow

Format: **printwindow** [*line*] [*num*]

This command is most useful in non-GUI mode. Prints a window of text, *num* lines long, centered about *line*. The default value for *num* is specified by the system variable _LINES which defaults to 22. The default for *line* is the current line. See “System variables” on page 60. The current line is indicated by a greater-than sign (**>**) in the left most print position. The current viewing position is unchanged.

profdump

Format: **profdump**

Used to retrieve profiling information from a target prior to the program’s exit. See Chapter 7, “The Profiler”.

profile

Format: **profile**

Opens the Profiler window. See Chapter 7, “The Profiler”.

profilegui

This command is obsolete It was in 1.8.9 MULTI, but has been replaced by the **profile** command. See **profile** on page 125. .

profilemode

Format: **profilemode** *command*

Used to control a vast array of Profiler functionality, such as **start**ing the profiler, **range** analysis, **process**ing data, and much more. See Chapter 7, “The Profiler” for all of the commands.

push

This command is obsolete. It was in 1.8.9 MULTI, but has been has been replaced by the **indexprev** command. See **indexprev** on page 110.

pwd

Format: **pwd**

Show MULTI’s current working directory.

Q

Format: **Q** [**0**|**1**]

This is the “quiet” command. **Q 0** (zero) turns off quiet mode (its default), **Q 1** (one) turns on quiet mode, and **Q** alone toggles quiet mode. When the debugger is in quiet mode, many commands are less verbose. For example, when setting or toggling a breakpoint in quiet mode, the breakpoint will not be echoed to the command pane.

q

Format: **q**

This is the prompted quit command in non-GUI mode only. When prompted, answer either:

Answer	Meaning
n	Cancels the exit request. This is the default.
s	Saves breakpoints, assertions, and directory list to the file named **multistate** and then exits.
y	Exits MULTI.

qfst

Note: This is a software-update command and may not be available on most systems.

Format: **qfst**

Lists the status of all applied and backed out modules.

quit

Format: **quit [ask | force | now | all]**

Quits the current debugger window. If this is the last program, and no control panel is present, then MULTI quits. If you do not specify any arguments, you will be prompted as to the disposition of any process you are attached to. You may specify one of the following arguments to modify the behavior of the quit command:

Option	Meaning
ask	Always confirms whether to exit if the **PromptQuitDebugger** config option is true.
force or **now**	Causes the debugged process to be killed without asking.
all	Equivalent to the **quitall** command.

quitall

Format: **quitall**

Causes MULTI to quit without prompting the user.

R

Format: **R**

Runs a new target program with no arguments. If a program already exists, terminate it. When debugging multiple programs, this causes re-running of the

current program if and only if it is a direct child program of MULTI. See also **r** on page 127.

r

Format: **r** *arguments*

Runs a new target program passing *arguments*, a space separated list, to the program. If a program already exists, terminates and restarts it. When debugging multiple programs, this causes re-running of the current program if and only if it is a direct child program of MULTI.

If no *arguments* are given, then the last ones given are used again. If no previous run exists, no arguments are used.

arguments may contain **<**, **>**, **>>**, **>&**, or **>>&** to redirect standard in, standard out, and standard error. Text between quotes, either single (‘ ’) or double (“ ”), are treated as a single argument. Eventually, the quotes are removed. If you are running **csh,** then a **~** expands the same way as the shell. However, other shell processing is not done; no wildcards, pipes, and so forth.

For example, **r fly 3**, runs the program with the two arguments **fly** and **3**.

See also **setargs** on page 134.

Rb

Format: **Rb**

See **rb** on page 127.

rb

Format: **rb** *arguments*

Similar to the **r** and **R** commands, **rb** and **Rb** run and block the command line input until the program terminates, until it hits a breakpoint, or the target stops. While using these commands, you can still perform all interactive operations appropriate with a running program. There are useful when writing scripts that control execution of a remote program. Use **rb** or **Rb** when you want to run until you hit a breakpoint and then read the next line of a script file. See also the Continue and Block command **cb** on page 90, **r** on page 127, and **R** on page 126.

refresh

Format: **refresh** *section*

Reloads a section of the program into the target system's memory. *section* may be either **text**, **data**, or **all**.

If **text** is specified, then the code sections of the debugged program reads from the executable file and reloads into target memory.

If **data** is specified, then the global variables reset to their initial values. Uninitialized global variables are set to zero.

If **all** is specified, then the entire program image reloads.

This command is not supported for all targets.

registers

This command is deprecated. See **regview** on page 128.

regview

Opens a data explorer window displaying all the registers. See Chapter 5,

Arguments	none
Button equivalent	

"The data explorer". In non-GUI mode, the registers are echoed to the screen instead of in a new window.

remote

Format: **remote** [**log**[**=***filename*] | **nolog**] *debug_server* [*arguments*]

Connects to a remote target before any debugging on that target is done. Remote targets include simulators, emulators, and monitors. (See also **connect** on page 93.)

debug_server is the name of the debug server executable for the remote target. This debug server is generally a program that controls the remote target device, and must be designed for the target CPU that your are compiling your program for. *arguments* are specific to each debug server; consult your debug server documentation.

If **log** is specified, then a list of all transactions between MULTI and the debug server is sent to standard error. If a filename is specified after **log=**, then the transaction list is written to the named file instead of standard error. Generally, the output of the **log** option is a debugging feature to aid customers who are

developing their own debug servers. **remote log=***filename* may be specified after having connected to a debug server.

remote nolog	Stops logging and closes log file.
remote log	Enables logging again.

restart

Arguments	none
Button equivalent	

Identical to the **R** command when not used with remote debugging. (See **R** on page 126.) When used with a remote system configuration, this command saves a lot of time when re-executing a program. Instead of completely reloading the entire program, the debugger resets global variables, the program counter, and the stack pointer to their initial values. Uninitialized global variables are set to zero. Then the program begins execution.

restore

Format: **restore** [*filename*]

Restores the state of the debugger from the file *filename*, or from the file **multistate** if *filename* is not given. These files must have been created with the **save** command. (See **save** on page 132.) If you are connected to a debug server when using the **save** command and are not currently connected to the server, then this command also reconnects you to that debug server.

The filename will be searched for using the default search path. See “default search path” on page 70.

resume

Format: **resume** [*address_expression*]

This command is only for use within a breakpoint command list. (See “Command list” on page 69.) It resumes program execution at the given *address_expression*, after all the breakpoint commands have been executed. See “Address expressions” on page 66. If no *address_expression* is specified then the address that the breakpoint is set at is used. For example, to skip over line 5 in your program, you could use the following, which will stop before line 5, and then resume execution at line 6:

```
b 5 {resume 6}
```

resume will continue the program in the same manner that the breakpoint was encountered. For example, if the program was performing an **S** instruction when the breakpoint was encountered, the **S** command will be resumed.

return

Format: **return** [*exp*]

Returns from macro defined in MULTI. See **define** on page 99. If *exp* is given, then *exp* is returned as the macro's value. See also **error** on page 103.

rload

Format: **rload** [*load_symbols*] *executable*

The **rload** command loads an object module while debugging. The symbols for the executable (*executable***.dnm** and *executable***.dla**) are loaded only if *load_symbols* is set to 1. In addition, if *load_symbols* is set to 1, **.bss** is not zeroed and the program is not started automatically.

The filename will be searched for using the default search path. See "default search path" on page 70.

You can interrupt this command with the Esc key.

rom

This is used for ROM debugging, which is not supported on all targets. Consult your target's *Development Guide* for details specific to your target.

rundir

Format: **rundir** [*dir*]

This command changes the directory your program runs in to *dir*. If no argument is given, then the current directory the program runs in is printed. This command is only relevant to native debugging.

runtask

Format: **runtask** *proc* [*args*]

Works only for multitasking remote targets such as VxWorks. This is the standard way to start a task on these systems.

proc is the name of any downloaded procedure, and *args* is a list of space separated arguments to pass to the procedure. Acceptable values for *args* are:

- decimal and hexadecimal numeric constants
- character constants
- string constants enclosed in double quotes
- names of global variables (the **&** operand does not work here)
- I/O redirection operators **<** and **>**

When debugging C++, *proc* may be the member function of a global object, specified as *object.function*. If the requested function is ambiguous, a dialog box presenting all options is displayed.

S

Arguments	none
Button equivalent	
Menu equivalent	Debug > Next
Keyboard equivalent	F10

This is uppercase ‘S’.

Same as the **s** command, but treats procedure calls as normal statements. Thus, it steps over, instead of into, procedures. See also **n** on page 121 and **s** on page 131.

You can interrupt this command with the Esc key.

s

Arguments	none
Button equivalent	
Menu equivalent	Debug > Step
Keyboard equivalent	F11

This is the lowercase ‘s’.

Single steps one statement. If you accidently step into a procedure you do not care about, you can click the Return button (), which is the same as using the **cU** command. See **cU** on page 94.

You can interrupt this command with the Esc key.

save

Format: **save** [*filename*]

Saves the state of the debugger. This writes out the breakpoints, the assertions, the source directory or directories as set by the **source** command, and the remote debug server you are connected to, if any, to the file *filename*, or to **multistate** if no filename is given on the command line. This file is retrieved by the **restore** command. See **restore** on page 129.

saveconfig

Format: **saveconfig**

This command will save out a file which MULTI will read each time it starts to restore your configuration to the state it was in when you saved it.

saveconfigtofile

Format: **saveconfigtofile**

Similar to the **saveconfig** command, but lets you choose a file to save the configuration into. (See **saveconfig** on page 132.) This can be useful in conjunction with **configurefile** command. See **configurefile** on page 93.

sb

Format: **sb** <**a** | **d** | **t** | **u** > <**g** | **e** | **t** | **a** | **s** | **n**> [**l** | **i** | **p** | **x**] *val* [**@***count*] [**{***cmds***}**]

(In 'Format' above, the angular bracket pair **<>** contains arguments that must be supplied.)

This command sets special breakpoints. These breakpoints are target specific. Consult your target's *Development Guide* for more specific details.

In the first group of letters, **a** specifies any task, **d** is on any attached task, **t** is on the current task, and **u** is on any unattached task.

In the second group of letters, **a** specifies to stop the actor, **e** is stop every actor, **g** is stop the actor group, **n** is to notify, **s** is to stop the system, and **t** is to stop the task.

In the third (optional) group of letters, **l** (lowercase 'L') indicates that *val* is a line number, **i** indicates that *val* is an address, and **p** and **x** are target specific.

sc

Format: **sc** [**"***command***"** | **<***filename*]

Performs syntax checking on either a single command or an entire script file and all nested script files. See “Syntax checking” on page 63.

The filename will be searched for using the default search path. See “default search path” on page 70.

scrollcommand

Format 1: **scrollcommand max** [**l** | **c**] [*pixels*] [**wid=***num*]

Format 2: **scrollcommand** *count* [**l** | **c**] [*pixels*] [**wid=***num*]

Scrolls the window indicated by the identification number *num* by *count*, or to the **max**imum, in the given direction. If *count* or **max** are followed by **l**, and by default, the scroll is vertical and *count* corresponds to a number of lines. If *count* or **max** are followed by a **c**, then the scroll is horizontal and *count* corresponds to a number of characters. If *pixels* is also specified, then it scrolls by that many additional pixels. Not all windows scroll on a per pixel basis; some are constrained to full lines. Both *count* and *pixels* may be negative.

The window identification number *num* is obtained by using the special sequence **%w** with either the **mouse** command or the **keybind** command. If no window identification number is supplied, the source window is used. See **mouse** on page 119 and **keybind** on page 112.

For example,

The following example scrolls the source pane one line towards the end of the file:

```
scrollcommand 1
```

This example scrolls the command pane backwards by two lines:

```
scrollcommand -2 wid=-2
```

This scrolls the source pane three characters to the right:

```
scrollcommand 3c
```

And both these commands scroll the source pane to the beginning of the code:

```
scrollcommand -max
scrollcommand -maxl wid=-1
```

See **bsearch** on page 86 and **fsearch** on page 105.

setargs

Arguments	[*program_arguments*]
Menu equivalent	Debug > Set Program Arguments...

Sets the program arguments used with the next **r** command. (See **r** on page 127.) If no arguments are specified, then no arguments are used. The arguments are a space separated list. See also **getargs** on page 106.

setbrk

Format: **setbrk** [**0**]

Toggles the breakpoint set at the current line (pointed to by the current line pointer) or the current address. The current address exists only in GUI mode and specifies the line where the mouse was last clicked in an interlaced text/assembly view. By setting one of the mouse's click commands to **setbrk** (via the **mouse** command), you can toggle a breakpoint on a line in the debugger by clicking anywhere on the line rather than having to touch the break dots at the side of the text. See **mouse** on page 119.

This command has two forms:

Form	Meaning
setbrk	Toggles the break on the current line or the current assembly address in interlaced source/assembly view.
setbrk 0	Sets a temporary breakpoint on the current line or current address and executes the **c** command. (See **c** on page 88.) Once reaching the temporary breakpoint, the program halts and the debugger automatically clears the breakpoint.

shell

Format: **shell** *cmds*

Invokes a shell. If *cmds* is present, then the given commands are executed and immediately returned to the debugger. All windows are put in the background until the commands are completely executed.

Before being passed to the shell, the command string following **shell** is processed and all instances of the escape sequence **%EVAL {***multi_command***}** are replaced by the result of evaluating *multi_command*. This is useful for constructing dynamic arguments (that is, arguments that vary depending on your current debugging context) to shell tools. For instance, to run a tool on the current file, construct a command of the form:

shell *toolname constant_args* **%EVAL{$_FILE}**

showdef

Format: **showdef** [*name1* [*name2* [...]]]

Looks at each name on the line and attempts to find a **#define** macro definition for that name, then prints it out. Without arguments, it prints out the defined and undefined macros in the current file. Only enabled for programs built with MULTI debug information.

showfds

Format: **showfds**

This command is only supported on Solaris. It uses the Solaris **fstat()** function to give information about all of the open file descriptors.

Si

Format: **Si**

See **si** on page 135.

si

Format: **si**

The **Si** and **si** commands are similar to the **S** and **s** commands (see **S** on page 131and **s** on page 131), but cause the process to only advance by one machine instruction. Furthermore, the stop-position is shown as a disassembled instruction, not as a statement.

You can interrupt this command with the Esc key.

signal

Format: **signal** *signal* [**pr=***num*]

Sends the signal *signal* to the process specified by *num*, or the current process if *num* is not specified.

This is not supported for all target environments. Currently, only UNIX targets support this command.

Sl

This is capital ‘S’ and lowercase ‘L’.

Format: **Sl**

See **sl** on page 136.

sl

This is lowercase ‘S’ and lowercase ‘L’.

Format: **sl**

The **Sl** and **sl** commands are similar to the **S** and **s** commands (see **S** on page 131and **s** on page 131), but they cause the process to always advance by one higher language instruction, even when viewing the interlaced assembly.

You can interrupt this command with the Esc key.

source

Format 1: **source** [*num*] [*dir*]

Format 2: **source -** *dir*

This command specifies directories that MULTI will search to find source files for the debugged executable. Typing **source** by itself lists the current directories that will be searched. If *num* is specified, then the directory numbered *num* in the current source path is replaced by the new one given by *dir.* In the listed directories, *num* is zero-based. You can specify multiple directories at once. If a number is specified but no replacement directory is supplied, then the specified entry will be deleted from the list.

source - *dir* discards the old directory list which is replaced with the one given by *dir.*

Directory names may include **~** as an abbreviation for specifying your home directory.

stopif

Format: **stopif** [*file_relative_line_number*] *exp*

exp is an expression in the current language.

Sets a conditional breakpoint at the line number specified. If a line number is not specified, then use the current line number. The program breaks at this point if the condition given in *exp* is true. For example, the following command stops the debugger at line 20 if **y** is equal to five:

```
stopif 20 y==5
```

If you omit the line number, then you should not have expressions beginning with a number, otherwise it will be ambiguous. For example, the following should not be done:

```
stopif 5==y
```

The debugger tries to set a breakpoint on line five that stops on the condition (==y), which does not make sense. If you do this, enclose the expression in parentheses:

```
stopif (5==y)
```

MULTI will do limited syntax checking to be sure "y" exists, but the user needs to use variables which exist.

See also "Procedure-relative vs file-relative line numbers" on page 67 and "Address expressions" on page 66.

stopifi

Format: **stopifi** [*addr*] *exp*

exp is an expression of the current language.

addr is either a procedure name or the address of an instruction.

Identical to the **stopif** command, except the breakpoint is placed on the machine instruction at address *addr.* If *addr* is not specified, then the address of the last item you saw with the "**/i**" or "**/I**" display mode is used. *addr* may also be a procedure name, in which case the breakpoint is set on the first address of the procedure.

syncolor

Format: **syncolor** [**0**] [**1**] [**a**] [**C**] [**k**] [**d**] [**n**] [**s**] [**c**]

Set syntax coloring options.

Option	Meaning
0 (zero)	Turns off syntax coloring for all options.
1	Turns on syntax coloring for all options.
a	Toggles syntax coloring for all options.
C	Toggles syntax coloring for comments.
k	Toggles syntax coloring for language keywords.
d	Toggles syntax coloring for deadcode.
n	Toggles syntax coloring for numbers.
s	Toggles syntax coloring for string constants.
c	Toggles syntax coloring for character constants.

For example, **syncolor 0Ck** will turn on syntax coloring for only comments and language keywords; **syncolor 1d** will turn on syntax coloring for everything except deadcode. Without any arguments, **syncolor** will echo the present state of all options.

T

This command existed in 1.8.9 MULTI, but has been removed. To get the same functionality, enter: **calls local**. See **calls** on page 89.

t

This command is obsolete. It was in 1.8.9 MULTI, but has been replaced by the **calls** command. See **calls** on page 89.

target

Format: **target** *string*

Transmits commands to the target. Identical to **xmit** command. See **xmit** on page 143.

targetwindow

Format: **targetwindow**

Opens the Target and I/O windows used by some debug servers and simulators for direct communication with the target. These windows open automatically when you connect to a debug server. See Using the Builder section for more information about these windows. For a command line interface to these windows, see **xmit** on page 143 and **xmitio** on page 143.

taskwindow

Format: **taskwindow**

Opens the Task Window which displays the current tasks that are run on an embedded multi-tasking target, such as **rtserv**, **vxserv**, and **tornserv**. This window contains columns of information about each of the tasks. The contents of these columns differ depending on the target. See your target's *Development Guide* for information specific to your target. Clicking on a task name will automatically attach to and begin a debug session on any task (equivalent to the command **attach** *tid*. See **attach** on page 80).

tog

Format: **tog** [**on**|**off**|**tog**] [**e** *exception_number* | **hbp** *hbp_id* | [**b**] [*address_expression* | *breakpoint_list*]]

Toggles the active status of *address_expression* or a breakpoint. See "Address expressions" on page 66.

Only existing breakpoints/hardware breakpoint/exceptions can be modified with this command. If no such breakpoint/hardware breakpoint/exception exists, an error message is displayed.

See also **b** on page 81**, be** on page 82, **de** on page 96, **hardbrk** on page 108.

unalias

Format: **unalias** *string*

Unassigns an **alias**. It disassociates *string* from its substitution. For example, if you had aliased **sh** to **showdef**, then typing:

```
unalias sh
```
unassigns **sh**.

update

Format: **update** [*interval*]

This command forces all currently open and non-frozen view and monitor windows to be re-evaluated, halting the process, if necessary, to get the information. If it halts the process, it will resume it after refreshing the windows. This provides a quick and easy way to update your view windows to their current values without having to manually halt the process and then resume it.

If *interval* is specified, then MULTI will automatically do an update approximately every *interval* seconds while the program is running. This is a useful way to monitor the value of a variable continuously while the program is running. To deactivate the automatic update, specify 0 for *interval*.

uptosource

Format: **uptosource**

Moves up the stack until it finds a procedure with source code, and shows the corresponding source. Note this does not change the program counter or execute any program instructions on the target.

view

This command creates a data explorer window to display an item. There are several ways to open a data explorer window:

Form	Meaning
view *exp*	Creates a view displaying the expression *exp*.
view *type*	Creates a view displaying the type *type*.
view $locals$	Displays all local variables.
view *filename*	Displays all procedures and any special variables in the file *filename*.
view **address*	Creates a window displaying the contents of the given location in memory. An asterisk (*) must be in front of the viewed address.
view *exp1,exp2,exp3*	Creates a view displaying the multiple expressions.

See “View command” on page 150.

viewcommand

Format: **viewcommand** *cmds* [=y[**,**x]] [**press**|**release**] [**wid=***num*]

This command is not meant to be used from the command line, but rather is expected to be an argument to the **mouse** or **keybind** command. See “View command” on page 150.

viewdel

Format: **viewdel**

Closes all of the current view windows.

viewlist

Format: **viewlist** *structptr nextptr* [*links*]

Bring up a number of items from a list type of structure, where *structptr* is the pointer to the structure, *nextptr* is the name of the next pointer within the structure, and *links* is the number of items in the list to show (default value is 25). For example, given the following C code:

```
struct S {int a; struct S *next; }; struct S *ptr;
```

The command **viewlist ptr next 3** would bring up view windows on the first three structures in this list. In this case, the **viewlist** command is equivalent to entering:

```
view ptr; view ptr->next; view ptr->next->next;
```

W

This command is obsolete. It was in 1.8.9 MULTI, but has been replaced by the **printwindow** command. See **printwindow** on page 124.

w

This command was in 1.8.9 MULTI but has been removed. To get the same functionality, enter:

```
printwindow (1+_LINES/2)
```

See **printwindow** on page 124.

wait

Format: **wait**

Blocks command processing until the program is halted. This is useful in playback files or breakpoint command lists. (See “Command list” on page 69.) For example, if you want to step the program three times after a certain breakpoint is encountered, but you don’t want to enable **blockStep**, enter:

```
b {s; wait; s; wait; s}
```

This assures each **s** comand is complete before executing the next one. Since Esc halts the process, it in effect cancels this command.

watchpoint

Format: **watchpoint** *exp*

Set a watchpoint on the address indicated by *exp*, which causes the program to halt when the address is written do.

This command is implemented in one of three ways:

- On systems which support it (emulators, simulators, etc.) a hardware breakpoint is set at the given address. (See **hardbrk** on page 108.)
- If you compiled your program with the **-check=watch** option of the Green Hills compiler then you may establish one watchpoint which will operate fairly efficiently. (See the Builder chapter on how to set this.)
- Otherwise, the debugger will create an assertion to check when that address changes value. This will slow down your program considerably. (See **a** on page 77.)

while

Format: **while (***exp***) {***cmds***}**

exp is an expression in the current language.

This command list *cmds* continues to execute as long as *exp* evaluates to a non-zero value. This is similar to the **while** loop in C. See “Command list” on page 69.

You can interrupt this command with the Esc key.

window

Format: **window** [*num*] [**{***cmds***}**]

This command creates, deletes, lists, or changes the contents of a monitor window. A monitor window captures the output of a command or command list. See “Command list” on page 69. The commands are executed every time the program stops, and the output of these commands is printed in the window. There is a limit of 100 windows per program that are defined. The command list may contain multiple commands separated by a semicolon (**;**). Multiple commands must be surrounded by curly braces (for example, **window {calls; B}**). Monitor windows have the same standard window features as data explorer windows.

This command has several forms:

Form	Meaning
window	Lists all existing windows and their assigned commands in order.
window *num*	Deletes window number *num*. The number is displayed on the window border. For example, window 1 is titled **MONITOR 1**, therefore entering **window 1** removes that window.
window **{***cmds***}**	Creates a window displaying the results of given command list.
window *num* **{***cmds***}**	Replaces the command list for monitor number *num* with *cmds*. The command list also changes by left-clicking the command’s name in the window in the upper left hand corner.
window 0	Deletes all existing windows.

For example, the command **window calls** displays a stack trace. To change the window to display the breakpoints, use the command **window 1 B**.

windowcopy

Format: **windowcopy wid=***num*

Copies the current selection in the window specified by *num*, its window identification number, to the clipboard. Many commands that affect specific windows require this window identification number. You can get this number

by using the special sequence **%w** as part of a **mouse** or **keybind** command. See **mouse** on page 119 and **keybind** on page 112. See “Other window topics” on page 42.

windowpaste

Format: **windowpaste wid=***num*

Takes the clipboard and places it in the input buffer of the window specified by *num*, its window id number. This command is typically used as part of a **mouse** or **keybind** command. See **mouse** on page 119 and **keybind** on page 112. Note that this slightly differs from the **windowspaste** command in that this uses the clipboard, where as **windowspaste** uses the selection. See **windowspaste** on page 143.

windowspaste

Format: **windowspaste wid=***num*

Takes the selection and places it in the input buffer of the window specified by *num*, its window id number. This command is typically used as part of a **mouse** or **keybind** commands. See **mouse** on page 119 and **keybind** on page 112. Note that this slightly differs from the **windowpaste** command in that this uses the selection, where as **windowpaste** uses the clipboard. See **windowpaste** on page 143.

x

This command was in 1.8.9 MULTI but has been removed. Use the **halt** command instead. See **halt** on page 107.

xmit

Format: **xmit** *string*

Transmits commands to the target. Identical to **target** command. See **target** on page 138.

xmitio

Format: **xmitio** *string*

Transmits *strin*g immediately to the remote debug server, if one exists, without being processed any further by the debugger. *string* is sent exactly as typed, meaning that no form of local substitutions will work such as aliases or local symbol names. The commands available with the remote debug server are listed

in the manual for the particular remote debug server used. **xmit** has no effect with some remote debug servers.

Z

This is the capital Z. This command is obsolete. It was in 1.8.9 MULTI, but has been replaced by the **chgcase** command. See **chgcase** on page 91.

z

This command is obsolete. It was in 1.8.9 MULTI, but has been replaced by the **zignal** command. See **zignal** on page 144.

zignal

Format: **zignal** [*signal*] [**s**] [**i**] [**r**] [**b**] [**C**] [**Q**] [**c** [**{***cmds***}**]]

Sets up the signal handling table. If *signal* is not specified, the "current" signal is used. The optional flags are:

- **s** Toggles stop. If stop is on, then the program stops execution when the signal occurs.
- **i** Toggles ignore. If ignore is on, then the debugger does not send the signal to the process.
- **r** Toggles report. If report is on, then a message displays every time the signal occurs.
- **b** Toggles bell. If bell is on, then a beep sounds every time the signal occurs.
- **C** Clears the signal by setting all four of the above flags to false.
- **Q** Does not print the new state of the signal.
- **c** The signal's command list is set to *cmds* and is executed every time the signal is encountered. If no commands are supplied, any existing commands are removed. If you wish to continue from a signal that has commands, end the commad list with the **c** command (see **c** on page 88).

The **l z** command (see **l** on page 113) to list the current handling of signals. For example, assuming a start up state of (**don't stop**, **don't ignore**, **don't report**, **no bell**), the command **z 14 sr** sets the alarm clock signal to **stop** (but still **don't ignore**) and **report** it occurred, but don't beep. Doing **z 14 sr** again toggles these flags back to the other state. Doing **z 14 Csb**, no matter what the previous state of the signal, will set the alarm clock signal to **stop** (but still **don't ignore**) and beep, but **don't report**.

WARNING: It is highly recommended that you do not tamper with the state of the “breakpoint” signal.

Chapter

5

The data explorer

This chapter contains:

- The data explorer
- Data explorer basics
- View command
- Related commands
- Data explorer autosizing
- Data explorer messages
- Working with data explorers
- Data explorer format menu
- Data explorers with an infinite view
- Updating data explorer windows

The data explorer

The data explorer allows you to view one or more variables of any type in a separate window and is one of MULTI's most powerful features. This window can be moved around and resized. The data explorer updates the values of variables each time the program stops.

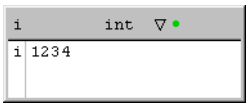
To open a data explorer, do one of the following:

- Double-click a variable in the source pane.
- In the command pane, enter: **view** *variable_name*.

For more information on the **view** command, see "View command" on page 150.

Data explorer basics

Here is an example of a data explorer:



Title bar

The title bar of a data explorer is active. Here is a description of the items on the title bar from left to right:

Data explorer title bar	
Item	**Meaning**
name	This is the name or address of the variable or variables displayed. In the above example, **i** is the name of the data explorer. You can change the name. For more information, see "Modifying values" on page 158.
type	This is the type of the variable being displayed. In the above example, **int** refers to the C type integer. You can change the type. For more information, see "Modifying values" on page 158.
▽	This is the Format button. When you click it, the data explorer Format menu appears. For a description of the menu items, see "Data explorer format menu" on page 160

 	Next to the down arrow is a bullet called the freeze dot. Clicking the bullet freezes and unfreezes the representation of the data explorer. In a frozen state, a stop sign appears and the contents of the window are preserved. The window is no longer being updated, and you cannot change the contents. Once a window is unfrozen (click the stop sign), it will be updated to reflect the current state of the program. Typing Ctrl+d in the window opens a duplicate window and freezes the original, replacing the bullet with a stop sign. This is useful if you want to continue execution and compare a future value of a variable to its current value.
	The pop arrow pops the data explorer back up one level. This only occurs when there are derived data explorers present. See also “Working with data explorers” on page 156.
	When you click the close button, this closes the data explorer window. You can also press Ctrl+q in the window to close it.

Hot keys

You can access many of data explorer’s different formatting options via hot keys, so an experienced user can quickly and easily control which information is displayed and how it is displayed. See “Data explorer format menu” on page 160 for a list of formatting options and their hot keys.

Searching and selections

To search for a value or label in a data explorer, use Ctrl+f or Ctrl+b. The search string will be displayed in the “name” area of the data explorer. See also “Incremental search” on page 40.

Text can only be selected in a data explorer window using the search feature. Selections cannot be made with the mouse unless the default mouse bindings are removed. See also “Selecting text” on page 39.

Mouse bindings

Many mouse actions are bound to useful data explorer commands. See “Working with data explorers” on page 156 for a list of mouse clicks and behaviors.

View command

The **view** command opens a data explorer to display an item. You can monitor an item and modify it, if it is a changeable variable (as opposed to a constant). The different items you can view are the following:

View items	
Command	**Description**
view *expr*	Opens a data explorer displaying the given expression.
view *type*	Opens a data explorer displaying the given type. This is useful for viewing items in a structure or class.
view $locals$	Displays all local variables in the current scope.
view *filename*	Displays all procedures and any special variables in the given file.
view **address*	Opens a data explorer displaying the contents of the given location in memory. An asterisk (*****) must be in front of the viewed address. See also “Data explorers with an infinite view” on page 164.

Viewing multiple items

You can view several items at once in a data explorer by specifying a comma separated list of items to the **view** command. For example, this command:

```
view fly, bat, cheese
```

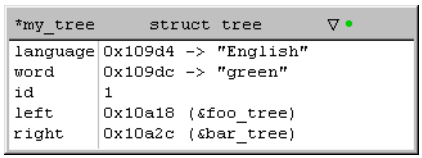
displays the three variables—**fly**, **bat**, and **cheese**—in the same data explorer.

To view an individual variable from the list in the same window, click the variable. To view an individual variable from the list in a new window, double-click the variable. For a list of mouse clicks and behaviors, see “Working with data explorers” on page 156.

Viewing structures

If the item being viewed is a structure, the data explorer will show each field of the struct separately.

The following is a data explorer showing a struct. It was generated with the command **view my_tree**:



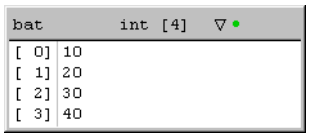
In this example, the data explorer value is ***my tree**, and its type is **struct tree**, which contains five fields: **language**, **word**, **id**, **left**, and **right**.

Fields are displayed in their natural format (except unexpanded fields that are structures or arrays), pointers to simple items are tracked and the value of the item pointed to is shown. By default, fields are highlighted whenever their values change. See also “ShowChanges” on page 164.

Viewing arrays

If the item being viewed is an array, the data explorer will show each element of the array separately.

The following is a data explorer window showing an array. It was generated by the command **view bat**, where **bat** is an array of 4 integers:



To display pointer or address types as arrays, do one of the following:

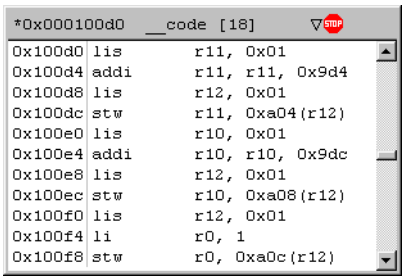
- Right-click the type field. In this example, the **int [4]**.
- Click the Format button (▽)and choose MakeArray.

Each subsequent use of MakeArray will increase the size of the array displayed. See also “Data explorer format menu” on page 160.

To view a char pointer (in C and C++) as an array, click the Format button (▽)and choose View Alternate.

Viewing disassembled code

Using a procedure name as the argument displays disassembled code. For example, **view** *procedure_name* opens a frozen data explorer of the disassembled code for that routine:



Viewing C++ classes

In C++, classes are displayed including base classes and virtual base classes. Static fields are displayed inside square brackets. Members of anonymous unions are displayed with a greater-than sign (**>**) preceding the member names.

Related commands

infiniteview

This command creates a data explorer with an “infinite” view at a specified address. See “Data explorers with an infinite view” on page 164.

update

This command opens data explorer windows once or at a specified interval. See “Updating data explorer windows” on page 164.

viewlist

This command provides a convenient way to view any number of elements of a linked list. Its usage is:

viewlist *structptr nextptr* [*links*]

Where *structptr* is the name of the linked list head, *nextptr* is the name of the element of the structure that points to the next link, and *links* is the maximum number of links to follow. See “viewlist” on page 140.

viewdel

This command deletes all data explorer windows associated with the debugger. It also deletes all memory view windows, call stack windows, and breakpoints windows.

viewcommand

This command can be used to manipulate the data explorer. This command is event driven and therefore is only useful for mouse or key bindings.

Format: **viewcommand** *cmds* [=y[**,**x]] [**press**|**release**] [**wid=***num*]

Requests a data explorer, monitor, or remote window to perform some action. The *num* is a window identification number obtained by the **%w** command. See “scrollcommand” on page 133 for more information on *num*.

Much of the information needed for certain **viewcommand**’s is dynamic and difficult to obtain, thus there are a few variables available which the debugger will automatically assign when a **viewcommand** is run. They can be used to dynamically assign some of these values:

Variables automatically assigned by the debugger	
%m	If the event is a mouse button press or release, this will be replaced by the word **press** or **release** respectively.
%w	This will automatically be replaced with the window manager assigned identification number of the window in which the event took place.
%x	This will be replaced with the current mouse X-coordinant.
%y	This will be replaced with the current mouse Y-coordinant.
Examples	
viewcommand IncrField=%y,%x %m wid=%w	
viewcommand Pop wid=%w	

For more examples, choose Config > Options... > General tab > Mouse Bindings... and look at the bindings in the Mouse Commands window.

The following is a list of *cmds*:

Values for *cmds*	
Beep	The data explorer beeps to indicate an error. If no command matches a **press**, this is the default.
Noop	Does nothing. (Short for “No operation”.)

EditType	Opens a dialog box to change the type of data displayed in the data explorer. This command needs a window number.
EditAddress	Opens a dialog box to change the address displayed by the data explorer. This command needs a window number.
AddVariable	Opens a dialog box to change the variable(s) displayed by the data explorer. This command needs a window number.
AddVarOrAdr	Opens a dialog box to change either the variables or the address displayed by the data explorer. This command needs a window number.
EditField	Opens a dialog box to change the value of a field, or element of an array. This command needs a y value and a window number.
IncrField	Increases the value of the field by the integer value one. This command needs a window number, x, y, and a **press/release** field.
DecrField	Decreases the value of the field by the integer value one. This command needs a window number, x, y, and a **press/release** field.
MakeArray	Changes the type of item displayed in an array of the current type. If the current type is an array, it makes a bigger one. This command needs a window number.
FindTypeAndCast	For C++ only: determines the most derived type of current object, casts the data explorer to that, and displays it. This command needs a window number.
ViewField	Changes the data explorer to look at a field. This command needs a y value and a window number.
NewViewField	Opens a new data explorer to look at a field. This command needs a y value and a window number.
FormatMenu	Opens the format menu. This command needs a window number and a **press/release** field.
ToggleFreeze	Toggles the data explorer between being frozen or not. This command needs a window number.
Duplicate	Opens another copy of this data explorer. This command needs a window number.
DuplicateFreeze	Opens another copy and freezes the current data explorer. This command needs a window number.
CloseView	Pops to a previous data explorer if it exists, or removes the data explorer. This command needs a window number.
KillView	Removes the data explorer. This command needs a window number.
PopView	Pops to a previous data explorer. This command needs a window number.
Help	Pops up a small help window describing the behavior of the data explorer.

Data explorer autosizing

The debugger will try to pick reasonable and convenient values for certain data explorer window sizes. In order they do not override user sizing, auto-sizing will only occur when the data explorer is created and when the number of rows changes.

The width and height of the window will be set to the best value between the specified minimums and maximums. You may configure these minimums and maximums in the Configuration Options dialog box (Choose Config > Options...). For uniformity, the width minimum and maximum are both set to 40 characters, by default. The height ranges from 3 rows minimum to a function of the display height for the maximum.

The column divider will resize itself based on the length of the longest string in the first column.

Data explorer messages

A number of messages may be displayed in the data explorer at various times. Some warn that the data being displayed might be untrustworthy, others describe why the data cannot be displayed.

Data explorer messages	
Message	**Meaning**
Infinite views must look at memory, not registers	The data explorer is in infinite view mode, but the variable is in a register. See also “Infinite” on page 163.
NaN	Short for “not a number”. For floating-point variable types, the value is not a legal representation of any number.
na	The data is too complex to show on this line. To expand the data to the current window, left-click the line. To show the data in a new data explorer window, double-left-click the line.
No process	No process is currently being debugged.
No symbols for this procedure	Debug symbol information does not exist for the current procedure.
Optimized away	The variable does not exist because a compiler optimization decided it was not necessary.
Original procedure not on stack	The original procedure in which the variable was in scope is no longer on the call stack. This message will only be displayed in “evaluate in context” mode. See “In Context” on page 162.

Out of register scope	The variable was assigned to a register, but is no longer assigned to any register. If the register has been overloaded, it may now represent a different variable. The data explorer will show the current value of the register along with this message, but the value may now be meaningless.
Out of scope	The variable no longer exists in the current lexical scope. The data explorer will show the current value of the register along with this message, but the value is most likely meaningless.
Process running	The process being debugged is currently running, so no value is known for the variable.
Uninitialized	The variable has most likely not been assigned a value and could be a random value in memory. The data explorer will show the current value of the memory, but it is most likely meaningless.
Unreadable memory	For a pointer or address type, the debugger does not have read access to the memory pointed to, or it does not exist.

Working with data explorers

Data Explorers are interactive, and are manipulated to display associated information or to change the values of variables. Most of the mouse actions listed below refer to the default bindings. The bindings have been provided where applicable. To change the default mouse bindings, choose Config > Options... > General tab and press the “Mouse Bindings...” button, or use the mouse command from the debugger. See **mouse** on page 119.

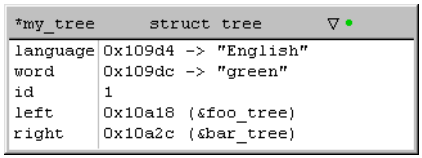
Configuring the maximum complexity of displayed data

Depending on your target, it may be desirable to minimize reading data from the target which has not been explicitly requested. Three configuration options are provided to limit the complexity of information displayed in data explorer windows. The **FormatStringMaxLength** configuration option allows you to specify a maximum length for the string representation of your data in the data explorer window. Once the data accumulated reaches this length, no further data will be read from the target for the display of that variable in the data explorer window without pushing into it. The **FormatStringMaxDepth** configuration option allows you to specify how many levels of nested structures the data explorer window attempts two display on one line. Nested structures below this depth will have their values displayed as “< na >”. Finally, the **LoadLongArraysOnViewWinCreation** configuration option allows you to specify whether arrays of more than 1,000 elements will initially have all elements displayed or only their first 1,000 elements.

Changing views

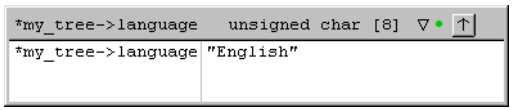
Pushing views

To display more information on a field in a data explorer window, click the desired field. This pushes the currently displayed data onto the window's stack, displaying the requested information. An up arrow (↑) will appear to indicate that this data explorer contains other view(s) on its stack. To return to the previous data explorer view, click this arrow. In our "view my_tree" example above:



*my_tree	struct tree
language	0x109d4 -> "English"
word	0x109dc -> "green"
id	1
left	0x10a18 (&foo_tree)
right	0x10a2c (&bar_tree)

When we click "language", we get the item pointed to by "language" in the same data explorer window:



*my_tree->language	unsigned char [8]
*my_tree->language	"English"

The default binding for this action is:

```
Mouse1*Click1@View=viewcommand ViewField=%y wid=%w
```

Popping views

To return to the first window, click the Pop button (↑). The Pop button may be pressed once for each view that has been pushed.

New views

To open a new data explorer window to display a field, double click the field. This leaves the original window unchanged. The view stack is not transferred, so the new explorer will not have any views on its stack even if the original one did.

The default binding for this action is:

```
Mouse1*Click2@View=viewcommand NewViewField=%y wid=%w
```

Modifying values

You can modify the information displayed in the data explorer window. Here are the values you can modify:

- type of variable displayed
- value of a variable
- name of variable displayed
- address of data displayed

Modifying the name or address

To change the name or address of the data being displayed, left-click the name field. You cannot change addresses in frozen data explorers. If you are changing the variable name, you can enter a comma separated list of items in this dialog box to view multiple items at once.

The default binding for this action is:

```
Mouse1*Click1@Name=viewcommand AddVarOrAdr wid=%w
```

Modifying the type

To change the type of the data being displayed, left click the type field. You cannot change types in frozen data explorers.

The default binding for this action is:

```
Mouse1*Click1@Type=viewcommand EditType wid=%w
```

Modifying the data

To increment the value of the data in a field, middle click the number to be incremented. The value will be increased by the integer one.

To decrement the value of the data in a field, Shift middle click the number to be decremented. The value will be decreased by the integer one.

To change a data value more generally, right click the data to be changed. A dialogue box will pop up prompting you to type in a new value.

The default binding for this action is:

```
Mouse1*Click1@Values=viewcommand EditField wid=%w
```

Changing view style

To change the type into an array, right click the type field. If it is already an array, a larger array is created. This is useful for looking at items of unknown size such as Ada unconstrained typed variables, C strings, or C++ virtual tables.

Right clicking the address is only relevant in C++. If an object is examined through a pointer to one of its base classes, then the actual type is hidden. With multiple inheritance, its address is also altered. The debugger attempts to find the actual type by looking up the name of the virtual table.

Note: to view a C or C++ char pointer as an array first use the ViewAlternate command.

The default binding for this action is:

```
Mouse3*Click1@Type=viewcommand MakeArray wid=%w
```

Default mouse bindings

The following are the default mouse clicks.

Default mouse clicks		
Mouse Click	**Object**	**Effect**
left click	Freeze button	Toggles data explorer frozen state
left click	Close button	Closes the data explorer
left click	Format button	Opens the Format menu
left click	Name field	Edit the item(s) to be viewed
left click	Type field	Edit the type
right click	Type field	Makes the data explorer an array, or increases the size of the current array
left click	Data field	Pushes the current view, showing the data value
double left click	Data field	Opens a new data explorer showing the data value
middle click	Data field	Increments the value in the field
shift+middle click	Data field	Decrements the value in the field
left click	Data field	Edit the value in the field

Data explorer format menu

To open the data explorer format menu, click the Format button (▽)on the title bar.

A dot beside an option indicates it is currently set. Selecting an option toggles its state. The following sections contain an explanation of each option. Hotkeys are shown for options where they are available. You can type these hotkeys anywhere in a data explorer window.

Display address or type

Show Address

Hotkey: S

The name field in the upper left corner of the data explorer will either display the actual name of the viewed variable, or the address. This option toggles between these two states.

Show Type

Hotkey: T

When you view multiple variables, classes, or structures, to display the type of each member, choose Show Type.

Number bases

The following pertain to all types except string types (character pointers) which are always displayed as quotes strings unless “View Alternate” is selected, in which case they are displayed as an array of characters.

Natural

Hotkey: N

In natural mode all numbers are shown in their default state. If “Hexadecimal” mode is selected, all numbers are shown in hexadecimal, otherwise addresses are displayed in hexadecimal, characters in ASCII, and other numbers in decimal.

Decimal

Hotkey: D

All numbers are displayed in base 10.

Hexadecimal

Hotkey: H

All numbers are displayed in base 16.

Binary

Hotkey: B

All numbers are displayed in base 2.

Octal

Hotkey: O

All numbers are displayed in base 8.

Alternate viewing methods

View Alternate

Hotkey: V

In addition to the standard way of displaying a value, an alternate is available. The definition of the alternate depends on the type of item displayed. For example, an integer is also displayed in hex, a field with an enum type is also displayed in decimal. Character pointers, which are normally displayed as a string, will be displayed as a character array. Other pointers have no alternate display type.

Memory View

Hotkey: M

Opens a Memory View window for interactively displaying and modifying memory contents. This window initially displays memory at the address specified. See **memview** on page 118.

Make Array

Hotkey: A

Displays pointer and address types as an array. If the displayed item is already an array, this will increase the size of the array displayed. This will also be executed by a mouse right-click on the type field of the data explorer window. **Note:** character pointer types (in C and C++) must be in “view alternate” mode to be viewed as an array, see View Alternate above.

Evaluate sub-menu

Only one of these four can be selected at a time. Normally, “As Global” is used for all expressions involving only global variables, “By Address” is used if the expression is a static variable, and “In Context” is used for most others. If the value to create the data explorer involves a procedure call, such as “array[fly()]”, then “By Address” is employed.

In Context

Hotkey: C

Every time the debugger is about to update the data explorer (for instance when the target process hits a breakpoint) it reevaluates the expression named in the title bar. It attempts to evaluate the expression in the same context where it was first evaluated. For example, if a procedure is called since the data explorer was created, the debugger walks up the stack until it finds a stack frame with the right procedure and evaluates the expression there. If it cannot find such a stack frame, it displays an error.

As Local

Hotkey: L

Similar to “In Context”, except it always evaluates to the current procedure at the top of the stack.

As Global

Hotkey: G

The debugger reevaluates the expression, ignores all procedure scopes, and only looks for variables in the global scope. This is useful if an expression involves only global variables.

By Address

Hotkey: A

The debugger pays no attention to the expression, and instead uses the last valid address for this data explorer to display the data. This is useful for examining local variables before and afer they are in scope.

Format sub-menu

You can only select one of the following:

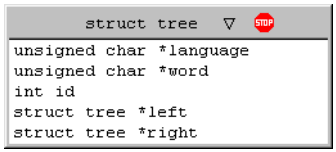
Formatted

The data explorer shows the data according to the given type. This is the normal mode.

Type

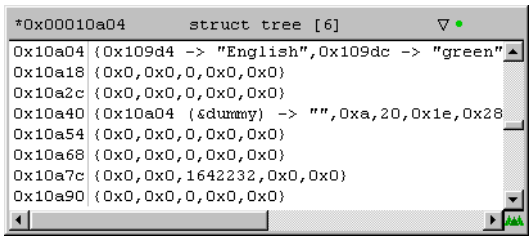
Only the type information is displayed. In C++, a list of all member functions of the type is displayed. Left clicking a member function causes the source pane to display that function. **Warning:** Inlined functions are not shown in the list.

Using the same "my_tree" example above, we choose Format > Type:



Infinite

The address of the viewed object displays memory in an infinitely scrollable fashion, limited only by the memory of the machine. In this mode, the scroll thumb is fixed in the center of the scrollbar. You can click above or below the fixed scroll thumb, or on the scroll arrows, to scroll the windows.



See also "Data explorers with an infinite view" on page 164.

Advanced Sub-menu

Expand Value

This indicates that pointers to simple items show what they point to (if in readable memory), rather than displaying only the value of the pointer. Also, simple arrays and structures show their first few elements. This option is on by default.

Open Pointer

If this item is selected, the data explorer automatically dereferences all pointers. Otherwise it displays the value of the pointer. This option is on by default.

ShowChanges

If this item is selected, then fields changing in the data explorer windows are highlighted. This option is on by default.

Print

This item will invoke the Print Dialog, allowing you to print the contents of the window.

Make Default

If this item is selected, then the current setting of the items above will be the default when the next data explorer is created. This command does not pertain to the Format menu items. These settings are saved for the remainder of this MULTI session only.

Reset Type

Resets the type field to the original setting if it has changed.

Refresh

Reloads the data explorer from memory, and redraws it.

Data explorers with an infinite view

**infiniteview ****address*

This is almost identical to the **view** command. See “View command” on page 150. The difference is that this command opens a data explorer window to scroll through all of memory, starting at the location specified by *address*. The memory address must be preceded by an asterisk (*****). To get the same effect with a normal data explorer, choose Format menu > Format > Infinite. See also “Format sub-menu” on page 162.

Updating data explorer windows

Format: **update** [*interval*]

This command forces all currently open and non-frozen data explorer and monitor windows to re-evaluate, halting the process, if necessary, to get the information. If the process halts, it resumes after refreshing the windows. This provides a quick and easy way to update your data explorer windows to their current values without having to manually halt the process and resume. This feature may not work under all circumstances.

If *interval* is specified, then the debugger automatically updates data explorer windows approximately every *interval* seconds while the program is running. This is a useful way to monitor the value of a variable continuously while the program is running. To deactivate the automatic update, specify *interval* to zero.

Chapter

6

Run-time error checking

This chapter contains:

- Run-time error checking
- Run-time Error tab check boxes
- Memory checking drop-down list
- Finding memory leaks

MULTI provides run-time error checking for many different classes of program errors, using a combination of compiler checks, special libraries, and debugger commands. You can enable several run-time error checking capabilities in the Run-time Checking tab (or the **-check=** build-time command line option). Building the program with run-time checking enabled makes the error checking available to the debugger.

Run-time error checking

(Builder: Project > Options for Selected Files... > Run-time Error tab)

The Memory Checking drop-down list box at the top has four choices: Default, None, Allocation, or Memory, explained below. Below this list box is a row of check boxes. Select the check boxes to enable the desired error checks. Most of these checks occur at run-time, although some occur completely at compile time, indicated below. To support the run-time checks, the compiler generates extra code at compile time which will increase the size of the resulting program.

Run-time Error tab check boxes

The following are the check boxes in the Run-time Error tab.

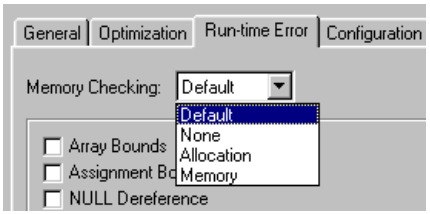
Run-time Error tab check boxes	
Check box	**Description**
Array Bounds	Checks array bound indexes. For constant indexes, this check occurs at compile-time; for other expressions at run-time. Equivalent to the **-check=bounds** build-time command line option. The error message is: “Array index out of bounds”
Assignment Bounds	When assigning a value to a variable or field which is a small integral type such as a bit field, this checks if the value is within the range of the type. Equivalent to the **-check=assignbound** build-time command line option. The error message is: “Assignment out of bounds” or “Value outside of type”
NULL Dereference	Generates an error message for all dereferences of NULL pointers. Equivalent to the **-check=nilderef** build-time command line option. The error message is: “NULL pointer dereference”

Run-time Error tab check boxes	
Check box	**Description**
Case/Switch Statement	Generates a warning if the **case/switch** expression does not match any of the **case/switch** labels. This does not apply when using a default **case/switch** label. Equivalent to the **-check=switch** build-time command line option. The error message is: “Case/switch index out of bounds”
Divide by Zero	Generates an error message indicating a divide by zero. Equivalent to the **-check=zerodivide** build-time command line option. The error message is: “Divide by 0”
Unused Variables	Generates an error message at compile-time for declared variables never used. Equivalent to the **-check=usevariable** build-time command line option. The error message is: “Unused variable”
Pascal Variants	Checks that the tag field of a variable declared as a variant record type matches one of the case selectors in the record. This applies only to Pascal. Equivalent to the **-check=variant** build-time command line option. The error message is: “Bad variant for reference”
Watchpoint	Enables the debugger’s **watchpoint** command to create one watchpoint without using an assertion. Equivalent to the **-check=watchpoint** build-time command line option. See “watchpoint” on page 141. The error message is: “Write to watchpoint”

Return	Generates a warning if a non-void procedure ends without an explicit return. For example, the following procedure generates a warning when exiting:
	`int func() {`
	`  for (int x = 0; x< 10; x++) {`
	`  if (x == 10)`
	`  return x;`
	`  }`
	`}`
	This option only applies to C and C++. Equivalent to the **-check=return** build-time command line option.
	The error message is: “No value returned from function”

Memory checking drop-down list

(Builder: Project > Options for Selected Files... > Runtime-Error tab > Memory checking drop-down list)



Memory checking is not available for use on all systems. It is not supported for use with an RTOS with non-standard memory allocation primitives.

Memory checking drop-down list	
Item	**Description**
Default	Maintains the previous or inherited setting. Originally, the default is None.
None	Disengages memory checking.
Allocation	Checks for the following memory errors. Equivalent to the **-check=alloc** build-time command line option. It also enables the debugger's **findleaks** command. See "Finding memory leaks" on page 171. To support this allocation memory checking the program is linked with an instrumented version of the malloc() library, usually located in the library **libdbmem.a**. If the program attempts to free memory not previously allocated, this error is reported: "Attempt to free something not allocated" If the program attempts to free memory already free, sometimes the previous error message is reported here. Otherwise, this error is reported: "Attempt to free something already free" If the program attempts to allocate memory after various other errors occurred, this error report appears: "Malloc internals (free list?) corrupted"
Memory	Generates an error message when the program tries to access memory that is not yet allocated. Equivalent to the **-check=memory** build-time command line option. Compiling a source file with this level of checking will cause the generated code to be both larger and slower. You may wish to link the application with Allocation checking and only compile a few selected modules with Memory checking. This level of checking displays the appropriate Allocation error messages, above, in addition to the following: "Attempt to read/write memory not yet allocated"

Finding memory leaks

Command-line format: **findleaks**

If your program is built/linked with either allocation or memory level memory checking, then this command finds chunks of memory that were allocated but are not reachable by pointer in the application.

You must halt the process you are debugging to use this command, but invoke the command before the process terminates. Often, this is most easily accomplished by setting a breakpoint on the last line of your program.

This command creates a window showing the following information for each chunk of memory found:

- The address of the chunk of memory allocated.
- The size of the allocated block.
- The procedure and line number (or address) of the routine calling **malloc** and the routine calling that routine and so on up to five levels.

If you click a line in this window, the debugger source pane display moves to the procedure which called **malloc**, while double clicking a line shows the procedure which called that one, and so on up to five clicks.

Chapter 7

The Profiler

This chapter contains:

- Introduction to the profiler
- Using the profiler
- Profiling targets
- The profdump command
- The protrans utility

Introduction to the profiler

The MULTI profiler (“the profiler”) is a tool that gathers important data about the execution of your program. This data can greatly improve the performance of existing programs.

To compile a program with one or more of the profiling options:

1. From the Builder window, choose Project > Options for Selected Files...
2. Set the Performance Analysis and the Coverage Analysis drop-down list boxes. They set the profiling options for the compiler.

The Performance Analysis drop-down list box contains the Functions and Graph options mentioned below. These options are equivalent to the **-p** and **-pg** compiler options. Coverage Analysis is equivalent to compiling with the **-a** compiler option.

Support for profiling and the steps necessary to collect profile data may vary depending on your target environment. Please consult the Profiling Targets section of this Chapter and your target’s *Development Guide* for additional target-specific information on using the profiler.

Execution time

During execution, the runtime environment collects samples of the program counter (PC) at various times. The exact sampling mechanism is target-specific. See “Profiling targets” on page 182. The resulting collection of PC data points gives a profile of where the program spends its time. The profiler shows you how much time is spent in:

- the program
- each function
- each basic block
- each source line
- each assembly instruction.

You may see large improvements in execution time by focusing and improving small amounts of code corresponding to a large percentage of total execution time. The compiler does not need to change your code to gather an execution sampling.

Standard calls

Compiling with the Functions option places calls to special profiling routines in your program to see how many times each function is called.

Call graph

Compiling with the Graph option also places calls in your program. It shows the number of calls made to each function, which "child" functions are called by each "parent" function, and how many times each child is called.

To bring up a graphical representation of the call graph, do one of the following:

- In the command pane, enter: **browse dcalls**
- Choose Browse > Dynamic Calls...

See also "Browsing dynamic calls, by function" on page 218.

Block coverage

Compiling with Coverage Analysis gathers a profile of basic block executions. If a given basic block is executed zero times, then the group of instructions making up this block is considered dead code for the given sample input where the program runs. You may remove the dead code from the application or try to discover if there is functionality missing from the application since it never reaches the dead code.

Using the profiler

Before using the Profiler, you must do the following:

1. From the Builder window, choose Project > Options for Selected Files... .
2. *Before* you compile your program, choose the desired profiler options.
3. *Before* you run your program, open the debugger and start profiling. You can do this one of two ways: either open the profiler window or in the command pane, enter: **profilemode start**.
4. Let your program run at least once (or halt it and use the **profdump** command) *before* generating reports in the profiler. See "The profdump command" on page 183.

Note: Some targets require additional preparation to collect profiling data.

To open the profiler window, do one of the following:

- Choose View > Profile...
- In the command pane, enter **profile**.

Let your program run at least once before generating profiler reports. This allows MULTI to generate profile information files from the program's directory that the profiler can use.

The following are the menus in the profiler window.

File menu	
Item	**Description**
Save Report...	Save the report currently in the profiling window to a file.
Append Report...	Append the report currently in the profiling window to a pre-existing file.
Print Report...	Print the report currently in the profiling window.

The following shows the "Config > New Data" and the "Config > Data Processing" sub-menu items and their command line equivalents:

Config menu		
Item	**Description of Configuration**	**Command**
New Data > Added to old	When a new set of profile data is processed, such as data from a new run of the program, this new data will be added to the old data. This is the default.	profilemode add
New Data > Replaces old	When a new set of profile data is processed, such as data from a new run of the program, this new data will replace the old data.	profilemode replace
Data Processing > Automatic	After a run completes, the new profile data is automatically processed and stored in the profile database. This is the default.	profilemode automatic
Data Processing > Manual	After a run completes, the data must be manually processed. Helpful if a program is to be run multiple times prior to analyzing profiling results.	profilemode manual

The following are the buttons in the profiler window and their command-line equivalents, if any:

Profiler window buttons		
Button	**Description of action**	**Command**
	Begin collecting profiling information. Bringing up the profiling window automatically executes this command.	profilemode start
	Stop collecting profiling information.	profilemode stop
	Delete any existing profiling data.	profilemode clear
	Displays in the debugger the percentage of time spent in each source line, to the left of each source code line. In assembly mode, it is the percentage of time spent on each instruction. This is the default. (Note that "~0%" means that line/instruction took very close to 0% of the total execution time.)	profilemode percent
	Highlights "dead code", lines which were never executed during the profiling run. This is only available if the program was compiled with the Coverage option.	profilemode coverage
	If the program was compiled with the Coverage option, then this displays in the Debugger the total number of times each line (or instruction) is executed. If the program was not compiled with the Coverage option, but was compiled with either the Functions or Graph options, then this displays in the Debugger (at the beginning of each function) the number of times each function was called. If the program was not compiled with any of the Coverage, Functions, or Graph options, then this view is not available.	profilemode count
	Displays in the profiler the Status Report. See “Status report” on page 179.	n/a
	Displays in the profiler the Standard Calls Report. See “Standard calls report” on page 179.	n/a
	Displays in the profiler the Call Graph Report. See “Call graph report” on page 180.	n/a
	Displays in the profiler the Summary of Coverage Information Report. See “Block coverage summary” on page 181.	n/a
	Displays in the profiler the Detailed Coverage Information Report. See “Detailed block coverage” on page 181.	n/a
	Displays in the profiler the Source Lines Report. See “Source lines report” on page 182.	n/a
	Manually processes the profiling information. See “Processing data” on page 178.	profilemode process

	Opens Range Analysis window. See “Range analysis” on page 178.	profilemode range *addr1 addr2*
	Opens Dynamic Call Graph centered on the present function being examined in the debugger. See “Browsing dynamic calls, by function” on page 218.	browse dcalls
	Dump the currently available profiling information from the target. See “The profdump command” on page 183.	profdump
	Close the profiler window. Note that this does not deactivate or in any other way affect the state of profiling. You can configure whether or not to have this button.	n/a

Processing data

To set the processing of data to manual, do one of the following:

- Choose Config > Data Processing > Manual
- In the command pane, enter: **profilemode manual**

When the processing of data is set to manual, to force processing of profiling data, do one of the following:

- Click the Process data button ().
- In the command pane, enter: **profilemode process**

When debugging natively, data processing is also used when first entering the profiler to process **mon.out**, **gmon.out** files, etc. previously generated with the program.

When debugging on an embedded target, the data processing is only used when at least one run of the program is completed in the current profiler session. The sampling data generated by the debug server, unlike **mon.out**, **gmon.out** files etc. produced by the program itself, are internal to the profiler and are deleted after processing. The profile data file written by the debug server also contains information such as the target endianness which must be known by the profiler in order to read the other profile data files such as **mon.out**. Thus, the profiler cannot read in profile data generated during a different session.

Range analysis

Improving the performance of an application often requires isolation of small portions of large subroutines, such as computationally intensive nested loops, which account for the majority of the subroutine’s execution time.

After profile data becomes available, you can get profile information for a particular section of code by clicking the Range Analysis button ().

You may specify a range of hexadecimal addresses with these fields. When the debugger displays the program in assembly mode, hexadecimal addresses are located to the left of their corresponding instructions. You can click these addresses to input the number into the range text fields, thereby giving a convenient way to specify ranges.

When a range is specified in the text fields, click the Calculate Range button to display the amount of time in seconds as well as a percentage of total execution time for this range in the bottom of the Range Analysis window. If no range or an inappropriate range is specified when you select the Calculate Range button, the valid range of sampling addresses for the program is displayed in the Range Analysis window. The window can be dismissed by pressing the Close button.

To obtain Range Analysis from the debugger command pane, enter:

```
profilemode range start_addr end_addr
```

where *start_addr* and *end_addr* are the beginning and end of your range, respectively. The result will appear in the debugger command pane.

The profiling reports

Six profile reports are available, depending on whether the appropriate data exists. The reports generally consist of several columns of information, and a status bar at the bottom of the report. Most of the columns can be sorted by clicking the header. Clicking the same header again will sort in the opposite direction. All of the columns can be resized by dragging the seperator to the left or right. All of the columns can also be moved around by dragging the header to the appropriate location. Within each of the reports, clicking a function name (or other program component) moves you to that location in the Debugger source pane; a double click invokes an editing window. You can save, append, or print any report from the File menu.

Status report

This is the report that appears when you first open the profiling window. It is always available. It gives general information about profiling, such as what type of data has been collected. The status bar indicates whether or not profiling is active.

Standard calls report

This report gives a summary of program execution time per function. It is available when any type of program counter sampling is done such as **mon.out** or **gmon.out**.

Standard calls	
Header	**Meaning**
Percent time	The percentage of the total program time spent in each function.
Time	The amount of time spent in each function. The status bar indicates if this is measured in seconds or milliseconds
# calls	The number of times the function is called.
Time(ms)/call	The number of milliseconds spent on each call to the function. Note that this is always measured in milliseconds, regardless of the contents of the status bar.
Function	The name of the function.

Note that the percentages may not add to 100%, as the profiler only monitors how much time is actually spent in the function.

Call graph report

This report gives a summary of program execution time per function, including functions' descendants. Descendants of a function are all routines called by that function, and all routines called by those routines, and all routines called by those routines, etc. This report is available when the program is compiled with the Graph option and **gmon.out** file(s) are created.

Call graph	
Header	**Meaning**
Function	The name of the function.
Calls by Function	The routines called by the function. This lists each function called by the function, and the percent of the total calls made by this function to each of the listed routines.
Calls to Function	The routines that called the function. This lists all the routines that called the function, and the percent of the total calls made to this function by each of the listed routines.
Self Time	The actual time spent in each function
Self %	The percentage of total time spent in each function.
Child Time	The actual time spent in the all of the children of each function
Child %	The percentage of total time that the children of each function represent

Self+Child Time	The total time spent in each function including its children.
Self+Child %	The percentage of total time spent in each function and its children.

The status bar indicates if the times are listed in seconds or milliseconds.

Block coverage summary

The status bar gives a summary of the coverage of the entire program. It is available when the program is compiled with the Coverage Analysis option on.

Block coverage summary	
Header	**Meaning**
Function	The function names.
Blocks	The number of basic blocks in each function.
% Covered	The percentage of basic blocks executed.

The status bar gives a summary of the coverage of the entire program. For more information about coverage within each function, see the Block Coverage Detailed section below.

Detailed block coverage

This report gives the program code coverage per basic block. It is available when the program is compiled with the Coverage Analysis option on.

Block coverage (detailed)	
Header	**Meaning**
Function	The function names.
Address	The starting address of each block.
Line Number	The (file-relative) source line corresponding to each block.
Executions	The number of times that block was entered.
Time	The total time spent in each block.
% Total Runtime	The percentage of total time spent in each block

The status bar indicates if the times are listed as seconds or milliseconds.

Source lines report

This report is a listing of all the source lines of the program, along with how long each took to execute. A source line is uniquely determined by the filename and (file-relative) line number. Only lines with positive times are displayed. The time it takes to process this report is proportional to the size of the program and hence may take longer for very large programs. This report is available when any type of program counter sampling is done such as **mon.out** or **gmon.out**.

Source lines	
Header	**Meaning**
Filename	The file that the source line is in.
Line Number	The (file-relative) line number of the source line.
Function	The function that the source line appears in.
Time	The time spent on the given line. The status bar indicates if this is displayed in seconds or milliseconds.

Profiling targets

You can use the Profiler with both remote and native targets. It works with simulators, monitors, and emulators.

Profiling native targets

Profiling uses a regular interrupt, typically 60 Hz, to obtain the location of the program counter.

Profiling with simulators

Profiling with a simulator is often much more accurate than native profiling. Normally, profiling information is obtained by periodically halting the program and recording the location of the program counter. This method is purely statistical and is subject to errors. Besides knowing which instruction it is simulating, a simulator also has a concept of how many machine clocks passed. This information tells the length of each instruction. This gives exact profiling information, subject to the accuracy of the simulator's model of the target processor.

Profiling with monitors

When profiling, monitors use a regular interrupt, typically 60 hertz, to obtain the location of the program counter. These samples are stored in an internal buffer, the contents of which are sent to the host as soon as it is full.

Profiling with emulators

When profiling, emulators provide a mechanism where trace information is interpreted as profiling data. Depending on the emulator, this is done either automatically or manually.

The profdump command

profdump

This command is primarily used when debugging a remote program that does not terminate normally such as an operating system, since normally profiling information is dumped upon exiting the program. This command also obtains timing information prior to complete program execution.

To write and clear current timing buffers, do one of the following:

- Click the Profdump button ().
- In the command pane, enter the command **profdump**

After the command executes, you must process the data (see “Processing data” on page 178). Because this command clears the buffers, subsequent uses of it dump profile information not contained in the previous dump.

The protrans utility

Format: **protrans** *options program*

Options:

Protrans options	
Option	**Meaning**
-a	This switch adds the profile data for the current execution to the summary profile. Without this switch, a new profile is generated with each execution, overwriting the previous profile.
-q	This switch suppresses message printing. Without this switch, **protrans** prints a small message for each execution describing the type of profile data found.
-m *file*	Specifies *file* as a file containing calls profile data.
-g *file*	Specifies *file* as a file containing call graph profile data.
-b *file*	Specifies *file* as a file containing coverage analysis profile data.

protrans is a utility that reads profile data produced when a program built for profiling is executed. **protrans** accumulates profile data over multiple runs of a program. **protrans** translates the data into an intermediate format that the profiler uses when displaying profile data.

The **protrans** utility is also used outside of the MULTI environment. This is desirable if you want to automate the acquisition of large amounts of profile data, and then use the profiler to display the data once all the runs are complete. Suppose you have a program called **dylan**, built with some combination of calls, call graph, and coverage profiling (see “Standard calls” on page 175, “Call graph” on page 175, and “Block coverage” on page 175), and a sample input to the program: **sampleinput1**, **sampleinput2**, etc. Now, consider the following **csh** shell script:

```
#!/bin/csh
foreach p (sampleinput*)
    $RUN dylan $p
    protrans -a -q dylan
end
rm -f mon.out gmon.out bmon.out
```

This script repeatedly runs the **dylan** program with the sample input and calls the **protrans** utility to read in the generated profile data for each execution. The argument **dylan** specifies the profiled program from the generated data. The default is **a.out**.

After the script finishes, an intermediate profile data file is generated which contains the summary profile. The intermediate file has a **.pro** extension appended to the program name if there is no current extension. Otherwise, the **.pro** extension replaces the current extension. For example, the program **fly.bat** generates the file **fly.pro**. In the example above, the file **dylan.pro** is generated in the same directory we are executing.

Now you can process the data (see “Processing data” on page 178) in the debugger to read in the summary profile stored in **dylan.pro** and then use the various features of the Profiler to view the information.

The last line in the script above deletes any profile data files left around after executions of the profiled program. The **mon.out** file is produced after running a program built for calls profiling. See “Standard calls” on page 175 for more information. The **gmon.out** file is produced after running a program built for call graph profiling. See “Call graph” on page 175 for more information. A program can only produce one or the other of these two files. A **bmon.out** file is produced after running a program built for coverage analysis (see “Block coverage” on page 175); this type of profiling is either done alone or in conjunction with either calls or call graph profiling. By default, the **protrans** utility looks for files with the names mentioned above. However, you can specify certain data files to **protrans**.

Thus, the shell script is rewritten:

```
#!/bin/csh
foreach p (sampleinput*)
    $RUN dylan $p
    mv gmon.out gmon.$p
    mv bmon.out bmon.$p
end
foreach p (sampleinput*)
    protrans -q -a -m mon.$p -b bmon.$p dylan
    rm -f mon.$p bmon.$p
end
```

The first loop runs **dylan** with the sample input and stores the generated data files into uniquely named temporary files. The second loop then calls **protrans** to read in the data from these files and produces a summary profile. The **-m** switch specifies a calls profile data file; the **-g** switch specifies a call graph profile data file; the **-b** switch specifies a coverage analysis data file. You can specify multiple files of a single profile data type with multiple uses of these switches. For example:

```
protrans -m mon.1 -m mon.2 -m mon.3 ...
```

When using the profiler to run a profiled program to collect the resulting data, the actions of **protrans** are transparent to you. The information in this section is only provided for those who want to run **protrans** separately from MULTI.

Chapter 8

Browse window

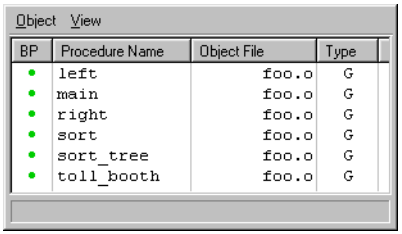
This chapter contains:

- Browse window
- Browse window for procedures
- Browse window for globals
- Browse window for source files
- Dialog box for procedures

This chapter shows you how to use the Browse window to explore procedures, globals, and source files, and the Dialog box to choose procedures.

Browse window

The debugger provides a Browse window for browsing procedures, global variables, and source files.



Once you have opened a browse window, you can change the object you're browsing with the Object menu.

Browse window > Object menu

Browse window > Object menu	
Item	**Description**
Globals	Browses all globals of the debugged program in the current browse window.
Procedures	Browses all procedures of the debugged program in the current browse window.
Files	Browses all source files of the debugged program in the current browse window.
Print	Prints the text contents of the current browse window.
Help	Opens a browser for HTML file and show MULTI's online help for browse window in it. You can navigate and search in the HTML browser.
Close	Closes the browse window.

The debugger provides a set of predefined filters in a browse window. For each type of object, some of the predefined filters are applicable, while others are grayed out. The debugger also enables by default some predefined filters for your convenience when globals are first loaded into a browse window. You can choose which objects to display by toggling the pre-defined filters, and/or by

defining filters yourself. User-defined filters are applied to the displayed object names, which are always shown in the browse window.

Browse window > View menu

Browse window > View menu	
Item	**Description**
User-defined Filter	Opens a dialog box so you can define your filters (see Filter Dialog Box below).
Hide C++ VTBLs	Enables or disables displaying Virtual Tables in C++ programs. Applicable only to globals.
Hide C++ Type Identifiers	Enables or disables displaying Type Identifiers in C++ programs. Applicable only to globals.
Hide C++ Type Info	Enables or disables displaying Type Information in C++ programs. Applicable only to globals.
Hide C++ Initialization Names	Enables or disables displaying Initialization names in C++ program. Applicable only to globals.
Hide C++ std::*	Enables or disables displaying names matching pattern “std::*” in C++ program. Applicable to globals and procedures.
Hide .*	Enables or disables displaying names matching pattern “.*”. Applicable only to globals and procedures.
Hide _*	Enables or disables displaying names matching pattern “_*”.
Hide __*	Enables or disables displaying names matching pattern “__*”.
Hide Globals from Shared Library	Enables or disables displaying globals from shared library.
Hide Files without Procedure	Enables or disables displaying source files which don’t contain any procedures.
Hide Procedures without Source	Enables or disables displaying procedures which don’t have source code.
Hide Inlined Procedure	Enables or disables displaying procedures which are inlined.
Hide Static Names	Enables or disables displaying objects which are defined as static. Applicable only to globals and procedures.
Hide Non-Static Names	Enables or disables displaying objects which are not defined as static. Applicable only to globals and procedures.

Other than User-defined Filter, all the other menu items are predefined filters, and they affect the objects shown in the browse window only when they are enabled.

User-defined Filter dialog box

From a browse window, to open the User-defined Filter dialog box, choose View > User-defined Filter.

In the Show and Hide text fields, you can type in a set of patterns separated by a space or semicolon.

The debugger uses the following algorithm to determine the set of objects to display in a browse window:

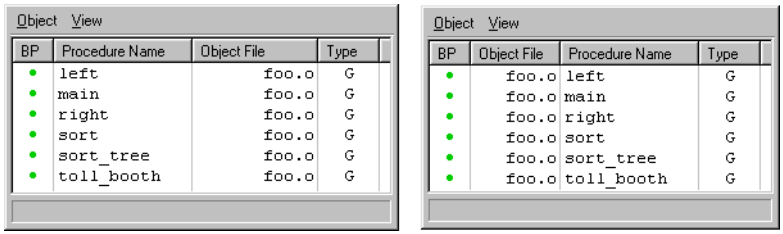
1. Determine the "base" set of objects in the browse window. When you first open a browse window, the base set of objects are those that are initially loaded in the browse window. (For example, if you enter the command **e f***, then the base objects are all the procedures whose names begin with the letter **f**. Another example: if you enter the command **browse procs**, then the base objects are all the procedures.) After that, each time you use the Object menu to change the object type to browse, the newly loaded objects now become the new base set. (For example, you open a browser with **browse procs**, the base set is now all the procedures. Now if you choose Object > Globals, the base set is now all the global variables.)
2. Use the user-defined filters to select objects from the base object set, and then remove those whose names match the user-defined hiding patterns. For example, if the user-defined filters include **fa*** as a selection pattern and **fab*** as a hiding pattern, then only those objects which come from the base object set and whose names start with **fa** but not **fab** will be selected.
3. Remove those specified by the enabled predefined filters.

Note: the base set is not affected by the View menu or any menu items under the View menu. The View menu only affects what is displayed in the browse window, depending what filters there are to operate on the base set. As noted above, the base set does change if you use the Object menu to change the objects to view.

The mouse clicks in the browse pane (just below the menu bar) are associated with some actions. Right clicking in the browse pane or the column header area opens a pop-up menu. The right click pop-up menu usually contains the following information:

- The formats in which to display the names of the objects.
- Switches for showing and hiding some attributes of the objects.
- Actions applicable to the clicked object.

You can switch the relative positions of existing columns by dragging the corresponding column header and dropping it at the desired position.



To sort the objects according to a column, click the corresponding column header.

Browse window for procedures

To open a browse window for procedures, do one of the following:

- From the debugger, choose Browse > Procedures...
- In the command pane, use the **e** command with a pattern. For example, "**e f***".
- In the command pane, enter **browse procedures** or **browse procs**.
- From the status bar of the debugger, open the Procedure drop-down list box, and choose "Browse procedures in program..." or "Browse procedures in current file...".
- In a browse window for source file, double click a file or right click a file and choose "Show procedures of File" from the pop-up menu.
- In the debugger source pane, right click a procedure and choose "Browse Callers" or "Browse Callees" from the pop-up menu.

When you open a browse window, it may contain all the procedures in the debugged program or a subset of all the procedures selected by certain criteria (for example, a pattern, or "callers of a procedure"). But whenever you choose a different object type and then choose Object > Procedures again, the browse window will contain all procedures.

By default, a browse window for procedures shows four attributes of a procedure. The following table lists all the attributes of a procedure which can be shown in a browse window for procedures.

Procedure attributes	
Attribute	**Information shown in column**
Procedure Name	The name or mangled name of the procedure, depending on what kind of name is chosen.
BP	If a breakpoint is set at the prologue address of the procedure, the icon for the corresponding breakpoint type will be shown, otherwise, a green dot is shown.
Object File	The object file from which the procedure comes.
Source File	The source file from which the procedure comes.
Module	Name of the module from which the procedure comes, if any.
Library	Name of the library from which the procedure comes, if any.
Address	Address of the procedure.
Size	Size of the procedure.
Type	GI: if the procedure is an inlined non-static procedure; SI: if the procedure is an inlined static procedure; G: if the procedure is a not-inlined non-static procedure; S: if the procedure is a not-inlined static procedure.

A procedure's information is displayed as follows:

- Grayed out if it has no source code.
- Displayed in the color for "dead code" in syntax coloring if it is an inlined procedure.
- Displayed in the color for "comment" in syntax coloring if it is a static and not fall in the above categories.
- Displayed in the normal foreground color otherwise.

The following table lists the operations when you click in the browse pane.

Mouse action	Description
Left click	Displays the clicked procedure in the debugger source pane. If you click in the BP column, the debugger will either insert a breakpoint at the clicked procedure if no breakpoint is there, or remove the breakpoint there if one already exists. If it sets a breakpoint at the procedure, it is at the first instruction after the prologue, if any.
Double left click	Opens an Editor window for the clicked procedure. If the clicked procedure has no source code, the debugger will issue a beep as a warning.
Right click	Opens a pop-up menu. See “Pop-up menu for procedure” below for detail.

Pop-up menu for a procedure	
Menu Item	**Description**
Name	Shows normal (unmangled) names of procedures.
Mangled Name	Shows mangled names of procedures.
Show in Debugger	Loads the clicked procedure into the debugger source pane.
Show in Editor	Opens an Editor window for the clicked procedure.
Show in Tree Browser	Opens a tree browser window to show the clicked procedure’s calling relationships.
(*other options*)	Enables or disables displaying the corresponding attribute.

The following table lists the operations when you click a column header in the browser pane:

Mouse action	Function Description
Left click	Sorts the objects according to the column (the order toggles).
Right click	Opens a pop-up menu. The menu items’ functions are the same as those specified in table "Pop-up menu for procedure" above.

Browse window for globals

To bring up a browse window for globals, do one of the following:

- From the debugger, choose Browse > Globals...
- In the command pane, enter: **browse globals**.

By default, a browse window for globals shows three attributes of a global variable. The following table lists all the attributes of a global variable which can be shown in a browse window for globals.

Procedure attributes	
Attribute	**Information shown in column**
Global Name	The name or mangled name of the global variable, depending on what kind of name is chosen.
Module	Name of the module from which the global variable comes from, if any.
Object File	Name of the object file in which the global variable is defined if it is static, or name of a object file in which the global variable is defined or referred to if it is not static.
Library	Name of the library from which the global variable comes from, if any.
Address	Address of the global variable.
Size	Size of he global variable.
Type	G: if the global variable is non-static; S: if the global variable is static.

A global variable's information is displayed as follows:

- Displayed in the color for "comment" in syntax coloring if it is static.
- Displayed in the normal foreground color otherwise.

The following table lists the operations when you click in the browse pane.

Mouse action	**Function Description**
Left click	Prints the global's value in the command pane if applicable, otherwise, the debugger issues a beep as a warning (for reasons such as the process is running, etc.).
Double left click	Opens a data explorer to show the global's value if applicable, otherwise, the debugger issues a beep as a warning (for reasons such as the process is running, etc.).
Right click	Opens a pop-up menu. See "Pop-up menu for a global" below for details.

Pop-up menu for a global	
Menu Item	**Description**
Name	Shows normal names for globals.
Mangled Name	Shows mangled names for globals.
Print Value	Prints the clicked global's value in the command pane.
View Value	Opens a data explorer for the clicked global.
(*other options*)	Enables or disables displaying the corresponding attribute.

The following table lists the operations when you click a header column in the browse pane:

Mouse action	**Function Description**
Left click	Sorts the objects according to corresponding column (the order toggles).
Right click	Opens a pop-up menu. The menu items' functions are the same as those specified in the table "Pop-up menu for a global" above.

Browse window for source files

To open a browse window for source files, do one of the following:

- From the debugger, choose Browse > Files...
- From the status bar of the debugger, open the File drop-down list box, and choose "Browse all source files in program...".
- In the command pane, enter: **browse files**.

A source file's information is displayed as follows:

- Grayed out if there is no procedure defined in the source file.
- Displayed in the normal foreground color otherwise.

The following table lists the operations when you click in the browse pane.

Mouse action	Function Description
Left click	Displays the clicked source file into the debugger source pane.
Double left click	Opens a browse window to show the procedures defined in the clicked source file if it is not grayed out, otherwise issues a beep as a warning.
Right click	Opens a pop-up menu, see “Pop-up menu for a source file” below.

Pop-up menu for a source file	
Menu Item	**Description**
Full Name	Shows full names for source files.
Base Name	Shows base names for source files.
Show Procedures of File	Opens a browse window to show all procedures defined in the source file.
Show in Debugger	Displays the source file in the debugger source pane.
Show in Editor	Opens an Editor window on the source file.
Show in Tree Browser	Opens a tree browser to show the reference relationships of the source file.

The following table lists the operations when you click a column header in the browse pane:

Action	Function Description
Left click	Sorts the objects according to the corresponding column (the order toggles).
Right click	Opens a pop-up menu. The menu items’ functions are the same as those specified in the table “Pop-up menu for a source file” above.

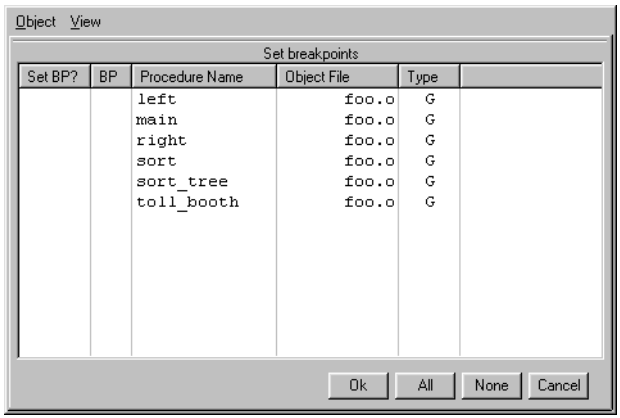
Dialog box for procedures

The dialog box for procedures is similar to a browse window for procedures except for the following:

- The dialog box is modal. (A browse window is modeless.)
- The dialog box does not let you change anything in the debugger until you dismiss the dialog box.
- The dialog box has an extra attribute column.
- The dialog box has a set of buttons at the bottom.

The debugger opens the dialog box for procedures one of two ways:

- Case 1: In the command pane, you use the **b** command with a pattern as parameter, for example, "**b ***".
- Case 2: The debugger tries to resolve an overloaded procedure in a C++ program.



The extra attribute column is always shown. This column header is one of two different names depending on which way the dialog is opened. In Case 1, the name is "Set BP?". In the Case 2, the name is Choice.

In Case 1, you can select multiple procedures. In Case 2, you can only select one procedure. If you select a procedure, a check mark appears under the extra attribute column.

The following table shows the buttons in the dialog box.

Button Name	Description
OK	Accepts the current selection(s).
All	Marks all procedures shown in the dialog box as selected. The item is only applicable in Case 1.
None	Marks all procedures shown in the dialog box as unselected. The item is only applicable in Case 1.
Cancel	Cancels what has been done in the dialog box and closes it.

Chapter 9

Memory view window

This chapter contains:

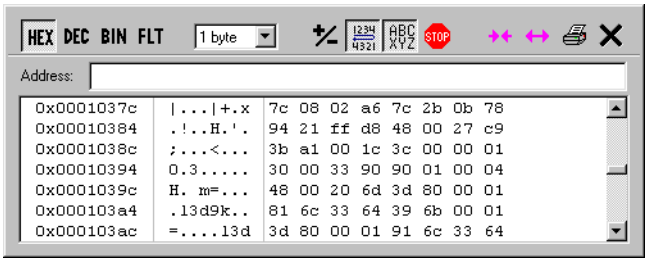
- Opening a memory view window
- Configuring a memory view window
- Changing the address in a memory view window
- Editing memory in a memory view window

The Memory View window is useful for examining large buffers, strings, and other data that do not display well in the normal data explorer. The window can be configured to display memory in a variety of formats. Additionally, the memory may be modified from this window.

Opening a memory view window

To open a memory view window, do one of the following:

- Click the Memory View button ( ) on the tool bar.
- Choose View > Memory.
- In a data explorer, type m, or click the Format button ( ) and choose Memory View... . This will bring up a memory view window examining the same memory location as the data explorer.
- In the command pane, use the **memview** command. See “memview” on page 118.



The memory view window consists of a memory pane and several controls which configure how the contents of memory are displayed. In the memory pane, there are three columns. The left column displays the address of memory being viewed. The middle column displays the memory contents in ASCII format. The right column displays the memory contents in a customized format, based on the configuration you have currently selected. When the memory view window is not stopped, it is updated every time the program being debugged stops. Bytes in the memory view which have been changed since the last time the program was stopped will appear highlighted.

Configuring a memory view window

The first four buttons control the basic format of the memory display.

Call stack window tool bar	
Button	**Display format**
HEX	hexadecimal
DEC	decimal
BIN	binary
FLT	floating point

Click one of the buttons to set the display to that format. The button will remain depressed to show that the memory view is currently configured to that format. By default, the memory view window appears in hexadecimal mode.

The *size* drop-down list box controls the unit size of the memory elements displayed. For hexadecimal, decimal, and binary, the choices are: 1 byte, 2 bytes, 4 bytes, and 8 bytes. For floating point, the choices are single precision and double precision. The default is 1 byte and single precision.

The signed button () affects decimal display only. It controls whether the decimal display should show signed values or unsigned values. When the button is depressed, the memory displays as signed values. When the button is raised, the memory is displayed as unsigned values. There is no effect unless the basic format is set to decimal. The default is unsigned.

The endian button () controls the endianness of the displayed memory. There are two choices for endianness. In big endian mode, the most significant byte is first and the least significant byte is last. In little endian mode, the least significant byte is first, and the most significant byte is last. When the button is depressed, the memory is displayed in big endian mode. When the button is raised, the memory is displayed in little endian mode. Note that in 1 byte view, endianness has no effect, since both modes are identical. The default is big endian.

The ASCII button () toggles the state of the ASCII column. The ASCII column may be hidden or shown. When the button is depressed, the ASCII column is shown. When the button is raised, the ASCII column is hidden. The default is to show the ASCII column.

The freeze button () controls the refreshing of the memory view window. When the button is depressed, the contents of the window are frozen. This

means that the window will not be updated when the memory contents change, and will continue to display the same information until the window is unfrozen. While the window is frozen, several features are disabled. You may not edit the contents of memory, scroll to a different memory location, or change the size of the view. You can still change the display format, however. When the button is raised, the window is unfrozen and will update normally. The default is unfrozen.

The shrink () and expand () buttons control the width of each line of memory. You may shrink the window to as few as 4 bytes per row, or expand the window to as many as 128 bytes per row. The default is 8 bytes per row.

The print button () allows you to print the contents of the memory view window. Only the visible area will be printed, so you should resize the window and scroll to the correct location before printing.

The first memory view window will appear with the default settings for the configuration options. Subsequent memory view windows will appear with the same settings as the previous memory view window. You may also use the **mvconfig** command to configure the memory view window and to change the default settings of the memory view window. See “mvconfig” on page 120.

Changing the address in a memory view window

To change the address being viewed, enter an address or expression in the Address text field. (Note that this does not work if you are editing memory. If you are editing, click anywhere in the column of addresses to get out of editing mode before entering an address expression.) The memory view window will jump to the specified address and display it in the top row of the memory pane. Note that the starting memory address will be aligned according to the number of bytes per row. For example, if you entered 0x00010007, and the window is currently displaying 8 bytes per row, the memory view will actually jump to 0x00010000.

Valid entries in the Address text field include an absolute address (0x10000), an expression (pointer + 0x200), or a variable name (mybuffer). For absolute addresses and expressions, the memory view jumps to the specified address. For variable names, the memory view will behave differently depending on the type of the variable. If the type is a pointer type (eg. pointer, array, function name, string), the memory view will take the value of the pointer and show the contents of the memory at that location. If the type is not a pointer type (eg. integer, floating point, structure), the memory view will take the address of the variable and show the contents of memory at that location. If the variable is

being stored in a register (and therefore does not reside in memory), the memory view will reject the variable and not change the view.

You may also use the scrollbar to change the address being viewed. Because the contents of memory are so large, the scrollbar is set to a special “infinite mode”. In this mode the scroll thumb is deactivated and is fixed in the center of the scrollbar. You may still scroll the window line by line or page by page. To scroll one line at a time, click on the up or down scroll arrow. To scroll a page at a time, click on the scrollbar above or below the scroll thumb. You may scroll continuously by holding down the mouse button instead of releasing it.

Editing memory in a memory view window

To edit the contents of memory, do the following:

1. Click a row of memory you want to edit.
2. Choose the ASCII (middle) column to edit in ASCII or the formatted (right-most) column to edit in the currently displayed format. The contents of the row will appear in the textfield above the memory pane.
3. Edit the contents of the textfield.
4. Press Enter to write the contents back into memory.

The format you use to edit memory will be the same as the format of the column selected. For example, if you clicked in the ASCII column, you must edit memory as ASCII characters. If you are in hexadecimal 2-byte mode, and you click the formatted column, you must edit memory as 2 byte hexadecimal values.

A non-printing character is normally represented by a period (**.**) in the ASCII column. When editing in ASCII, however, non-printing characters are both displayed and modified using a special backslash sequence “*nn*”, where *nn* is the hexadecimal value for the character. The backslash character has the special sequence “\\” (a double backslash).

When you edit the contents of the memory in the text field, you are allowed to modify memory beyond the end of the current row. You can do this by simply adding more values to the end. For example, if you are currently displaying 8 bytes per row, and you click in the ASCII column, you might see “abcdefgh” in the textfield. If you change the contents to “1234567890”, you will not only change the original 8 bytes from “abcdefgh” to “12345678”, but you will also change the next 2 bytes to “90”.

Chapter

10

Call stack window

This chapter contains:

- Call stack window

This chapter shows you how to open a call stack window and how to use it.

Call stack window

To open a call stack window, do one of the following:

- Click the call stack button ().
- Choose View > Call Stack... .
- In the command pane, use **callsview**. See **callsview** on page 89.

The following are the buttons on the toolbar.

Call stack window tool bar	
Button	**Description**
 Parameter	Enables or disables displaying parameters in function calls.
 Position	Enables or disables displaying the position of the function call, that is, the filename, the file-relative and proc-relative line numbers.
 Freeze	Enables or disables refreshing the window.
 Edit	Opens an Editor window on the selected function if it has source code.
 Locals	Opens a data explorer to show all the local variables of the selected function.
 Print	Prints the ASCII text contents of the call stack window.
 Close	Closes the call stack window. You can configure whether or not to have this button.

To the far right of the toolbar is the “Max Depth” field, which defines the maximum depth the debugger will display the calls stack. You can change it according to your preference. For example, if you’re debugging a program on a very slow target and you only care about the first few levels of the call stack, you can decrease the number so that the window is refreshed more quickly.

Below the toolbar is the call stack pane, where the call stack is displayed up to the maximum depth. The following table lists the mouse and keyboard operations in the pane:

To do this	Do this
Display a function in the source pane.	Click the function
Open an Editor window on a function.	Double click the function
Search forward in the call stack pane, if it has the focus (click in it to put focus there.)	Press Ctrl+f
Search backward in the call stack pane, if it has the focus (click in it to put focus there.)	Press Ctrl+b
Reset the search pattern, if you are searching in the call stack pane.	Press Ctrl+u

In a debugging session, whenever you change a call stack window's attributes (that is, displaying parameters, displaying location), the changes will affect subsequently created call stack windows until you change them the next time. You can also change these attributes with the **cvconfig** command. See **cvconfig** on page 94.

Call stack window and command-line function call

Suppose from the command pane, you make a function call into the debugged program, and suppose the program is stopped before the function returns to you, for example, because it hits a breakpoint. If you open a call stack window now, you will see two parts in the call stack pane. The bottom part is the call stack before you call the function, the top part is the call stack starting from the function.

Caveat

At the beginning and end of every function is a region called the prologue and epilogue. Inside this region of code, various registers may be saved and restored, and the stack pointer may be modified. Full source-level debugging is not possible within these regions. This is why no source level breakpoints are displayed here. You may single step at the machine level through this code, but you cannot trace the stack, or examine variables, or perform many other tasks until you are outside this region.

Chapter 11

Breakpoints window

This chapter contains:

- Opening the Breakpoints window
- Breakpoint types
- Using the Breakpoints window

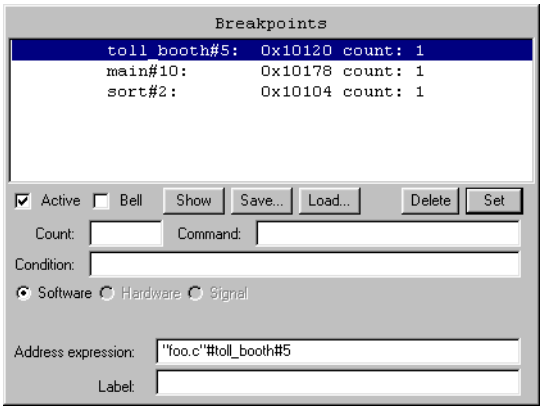
The Breakpoints window provides a graphical interface for examining and manipulating various kinds of breakpoints.

Opening the Breakpoints window

To open the Breakpoints window from the debugger, do one of the following:

- Choose View > Breakpoints.
- From the command pane, enter the **bpview** command.

When you bring up the Breakpoints window, it will initially appear in software breakpoint mode. (See “bpview” on page 84.)



Breakpoint types

There are three kinds of breakpoints which can be examined and manipulated with the Breakpoints window:

- Software breakpoints
- Hardware breakpoints
- Signals

To work with software breakpoints

Click the Software radio button.

This is the default type whenever the Breakpoints window appears.

To work with hardware breakpoints

Click the Hardware radio button. If hardware breakpoints are unavailable on the current target, the Hardware radio button will be greyed out.

To work with signals

Click the Signals radio button. If signals are unavailable on the current target, the Signals radio button will be greyed out.

Using the Breakpoints window

To toggle a breakpoint

This function is not available for Signals.

1. Choose the breakpoint from the list.
2. Click the Active check box.

To toggle whether a bell will sound when a breakpoint is hit

1. Choose the breakpoint from the list.
2. Click the Bell check box.

To change the count for a breakpoint

This function is only available for Software breakpoints.

1. Choose the breakpoint from the list.
2. Type the new count into the Count text field.
3. Click the Set button.

To change the list of commands associated with a breakpoint

1. Choose the breakpoint from the list.
2. Type the new command list into the Command text field.
3. Click the Set button.

To make a breakpoint conditional

1. Choose the breakpoint from the list.
2. Type the new condition into the Condition text field.

3. Click the Set button.

To examine a breakpoint

This function is only available for Software breakpoints.

1. Choose the breakpoint from the list.
2. Click the Show button.

This action can also be performed using the **e** command. See **e** on page 101.

To delete a breakpoint

This function is not available for Signals.

1. Choose the breakpoint from the list.
2. Click the Delete button.

This action can also be performed using the **d** command. See **d** on page 96.

To set a new software breakpoint

1. Choose the Software radio button.
2. Type the address expression where the breakpoint should be set into the Address Expression text field.
3. Type the count into the Count text field.
4. Type the command list into the Command text field.
5. Type the condition into the Condition text field.
6. Click the Set button.

This action can also be performed using the **b** command. See **b** on page 81.

To save the current list of software breakpoints to a file

1. Choose the Software radio button.
2. Click the Save button.
3. Choose a filename to save to in the file chooser which appears.

This action can also be performed using the **bpsave** command.

To load a list of software breakpoints from a file

1. Choose the Software radio button.

2. Click the Load button.
3. Choose a filename to load from in the file chooser which appears.

This action can also be performed using the **bpload** command.

To set a new hardware breakpoint

1. Choose the Hardware radio button. Hardware breakpoints are not available on some targets.
2. Type an expression into the Expression text field.

— or —

Type the address at which the breakpoint is to be set into the Address text field.

3. Type the size, in bytes, of the region on which the breakpoint is to be set into the Size text field. The default depends on how the address to set the breakpoint at was specified. If the name of a variable is given in the Expression text field, the default size is the size of the variable. If an address is given in the Address text field, the default size is one byte.
4. Type the mask to be applied to all addresses into the Mask text field. The default is 0.
5. Choose one of the Read, Write, Read/Write, and Execute radio buttons.
6. Type the command list into the Command text field. The default is no command list.
7. Type the condition into the Condition text field. The default is unconditional.
8. Click the Set button.

This action can also be performed with the **hardbrk** command. See “hardbrk” on page 108.

To change the actions performed when a signal is received

1. Choose the signal to be modified from the list.
2. Choose the desired combination of the Stop, Report, and Ignore check boxes. Note that changes to these check boxes take effect immediately.
3. Type the condition into the Condition text field. The default is unconditional.

4. Type the command list into the Command text field. The default is no command list.
5. Click the Set button.

This action can also be performed with the **zignal** command. See **zignal** on page 144.

Chapter 12

Tree browser

This chapter contains:

- Opening a tree browser
- Using a tree browser

The debugger tree browser is a graphical tool which allows you to examine the structure of your program in several ways.

Opening a tree browser

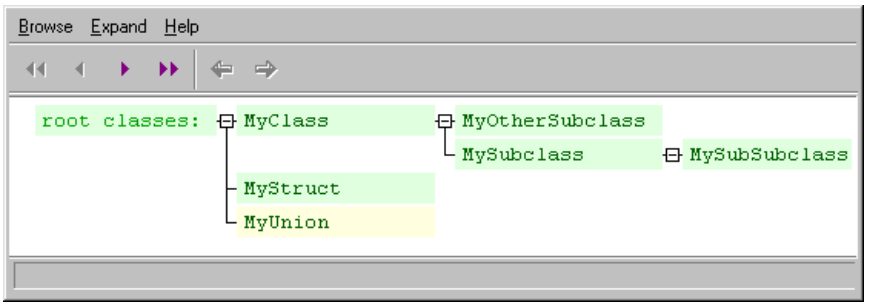
To use a tree browser, you must be debugging a program. You can open a tree browser in several ways, depending on what type of information you want to view.

Browsing classes

To use a tree browser to browse your class hierarchy, do one of the following:

- Choose Browse > Classes...
- In the debugger command pane, enter **browse classes**.

A window that looks something like this will appear:



The children of the ‘root classes’ node are all of your classes (including structs and unions) which do not inherit from another class. A class which is a subclass of another class is shown as a child of its parent class. Colors are used to distinguish classes and structs, shown in one color, from unions, shown in another.

To view the members in a class, do one of the following:

- Double-click the class.
- Right-click the class and choose Browse Members in Class.

Browsing static calls, by function

The tree browser can use information from your program’s symbol table to show you which functions your functions call, or are called by. These are potential, or “static”, paths solely based on the build-time symbol table of your

program, and not the actual run-time paths taken by your program during execution.

To browse static calls by function, do one of the following:

- Choose Browse > Static Calls...
- In the debugger command pane, enter: **browse scalls**.

A tree browser will then start up centered on the function you are currently looking at in the debugger source window.

To open a tree browser on a specific function, for example, foo:

- In in the debugger command pane, enter: **browse scalls foo**.

Color is used to provide information about the function represented by a given node. Separate colors are used for functions with debug information, functions without debug information, functions which may be recursive, and nodes used to represent functions whose address is taken and may therefore be called via function pointers.

To view a function in the debugger source pane, click the function node.

To open an editor window on a function, double-click the function node.

Both features are also available from the right-click menu.

Browsing static calls, by file

Besides being able to view the static call graph as functions, you can also view it as files. This will let you see the other source files whose functions are called from a particular source file.

To browse static calls by file, do one of the following:

- Choose Browse > File Calls...
- In the debugger command pane, enter **browse fcalls**.

A tree browser will then open on the file you are currently looking at in the debugger.

To start a tree browser on a specific file, for example, foo.c:

- In the debugger command pane, enter: **browse fcalls foo.c**.

To view a file in the debugger, click its node. To edit a file, double-click its node. Right-click a file node to bring up a menu which allows you to edit the file, view the file in the debugger source pane, or browse a list of functions in the file.

Browsing dynamic calls, by function

The dynamic call graph uses profiling information to display which functions a function actually called during run-time, unlike the static call graph which shows potential calls.

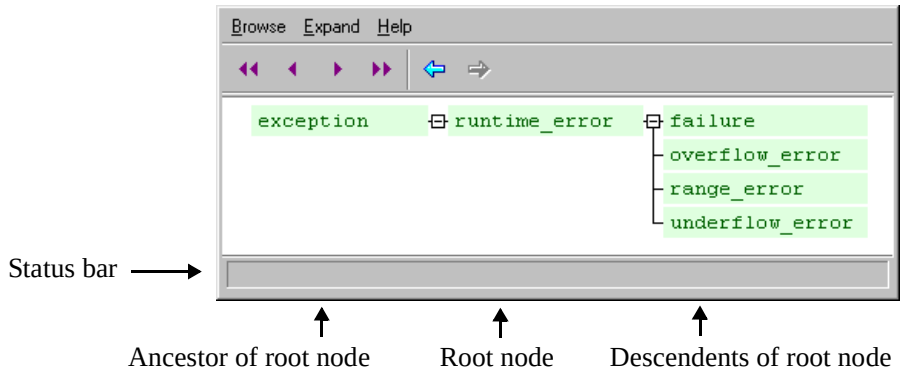
To browse the dynamic call graph, do one of the following:

- Choose Browse > Dynamic Calls...
- In the debugger command pane, enter: **browse dcalls**.
- To view a particular function specified by *function_name*, in the debugger command pane, enter: **browse dcalls** *function_name*.

This will only work if you have collected profiling information. See “Call graph” on page 175.

Using a tree browser

Regardless of what kind of information you are using the tree browser to view, the interface is basically the same.



The main part of the tree browser window is a tree graph. There is one node in the graph which is the ‘root node’; it is the particular function (or other object) which you are examining. The name of the root node is displayed in the title bar of the tree browser window. You can expand ancestors (i.e. callers, if you are looking at a function, or superclasses, if you are looking at a class) of the root node towards the left, and descendents (callees, or subclasses) of the root node towards the right. You may go as many levels as you want away from the root node. However, if you wish to look at an ancestor of a descendent of the root node, for example, you will need to reroot your graph. (See more on rerooting, below.)

To expand ancestors or descendents of a node, click the plus sign next to the node. If there is no plus sign, it means there is nothing you can expand. To contract something you have expanded, click the minus sign.

If you wish to expand many things at once, there are four ways to do it, and they are available from the Expand menu, or from the four purple expansion buttons on the toolbar.

To expand all of the nodes on the descendent side of the graph, until there are no more descendents, or until recursion is detected, do one of the following:

- Choose Expand > All Descendents.
- Click Expand All Descendents ( ).

Note that performing this expansion on a large program may take an extremely long time, and may yield unmanagable results anyway. To cancel this operation, press Esc.

To do the similar expansion for the ancestor side of the graph, do one of the following:

- Expand > All Ancestors
- Click Expand All Ancestors ( ).

Perhaps more useful than expanding all the nodes is the ability to expand one level of nodes.

To expand one more level of descendents, do one of the following:

- Choose Expand > One Level of Descendents.
- Click Expand Descendents One Level ( ).

This is basically equivalent to clicking the plus sign on every node in the descendent side of the graph.

To do the same thing to ancestors, do one of the following:

- Choose Expand > One Level of Ancestors.
- Click Expand Ancestors One Level ( ).

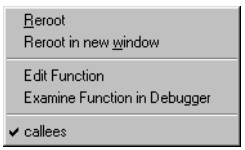
Node operations

Each node is labeled with a short name which describes what it is. In C++, the short name does not include class or namespace names which come before the final double colon (**::**). To view the entire name, point the mouse cursor over the node, and a tooltip will appear with the entire name.

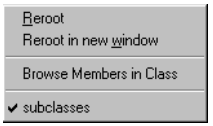
To view more information about a node, click it. Information about the node, including its full name, will be displayed in the status bar of the tree browser window. For certain types of nodes, such as function and file nodes, this will also cause the source code for the node to be displayed in the debugger source pane.

To open a right-click menu for a node, right-click it.

For example, the right-click menu for a function node will look something like this:



For a class node, its right-click menu will look something like this:



The first two operations, Reroot and Reroot in New Window, are described in the section on rerooting, below. The middle portion of the menu contains various actions you can perform on the node; these actions vary depending on the type of node, and are documented in the section on invoking the tree browser, above. The bottom portion of the menu lists the type(s) of ancestors or descendents a node may have, and allows you to select whether they are expanded; this is equivalent to the plus/minus box on the side of the node.

Rerooting

If there is a node on the graph that you would like to make the root node (so that you can examine both its ancestors and its descendents), you can reroot on that node.

To reroot on a node, do one of the following:

- Left-click the node and choose Browse > Reroot Selected Node.
- Right-click the node and choose Reroot.
- Middle-click the node.

Once you reroot on a node, everything which was previously in your window disappears. However, your previous window contents are stored in a history mechanism much like a web browser.

To access the history, do one of the following:

- Choose Browse > Back, or click Back ().
- Choose Browse > Forward, or click Forward ().

To reroot a node in a new window, rather than replacing the current window contents, do one of the following:

- Left-click the node and choose Browse > Reroot Selected Node(s) in New Window(s).
- Right-click the node and choose Reroot in New Window.
- Double-middle click the node.

Window operations

To close a tree browser:

Choose Browse > Close Window, or click Close (). (You can configure whether or not to have this button on the toolbar. See also “Display close (x) buttons” on page 241.)

To open a new tree browser which has the exact same contents as an existing tree browser, choose Browse > Clone Window.

For help on the tree browser:

Choose Help > Help.

Configuring tree browser colors

A number of configuration options exist which allow you to control how nodes of various types are displayed:

Foreground color configuration option	Background color configuration option	Node type
TBFunctionNormalFG	TBFunctionNormalBG	Functions in the static calls browser with debugging information.
TBFunctionNoInfoFG	TBFunctionNoInfoBG	Functions in the static calls browser without debugging information.
TBFunctionRecursiveFG	TBFunctionRecursiveBG	Functions which may be recursive in the static calls browser.
TBFunctionAdrTakenFG	TBFunctionAdrTakenBG	Nodes representing the possibility of calls to a function through a function pointer.
TBDynNormalFG	TBDynNormalBG	Functions with debugging information in the dynamic calls browser.
TBDynNoInfoFG	TBDynNoInfoBG	Functions without debugging information in the dynamic calls browser.
TBFileNormalFG	TBFileNormalBG	Files with debugging information.
TBFileNoInfoFG	TBFileNoInfoBG	Files without debugging information.
TBClassUnionFG	TBClassUnionBG	Unions.
TBClassStructFG	TBClassStructBG	Classes and structs.
TBClassNoInfoFG	TBClassNoInfoBG	Types without debugging information.

Index

Symbols

Numerics

A

Index

B

C

D

Index

Index

E

F

G

H

I

K

L

M

N

O

P

Index

Q

R

S

T

U

V

W

X

Z

Index